Mordechai Ben-Ari
Francesco Mondada

# Elements of Robotics

Springer Open

# Elements of Robotics

Mordechai Ben-Ari · Francesco Mondada

# Elements of Robotics

Mordechai Ben-Ari
Department of Science Teaching
Weizmann Institute of Science
Rehovot
Israel

Francesco Mondada
Laboratoire de Systèmes Robotiques
Ecole Polytechnique Fédérale de Lausanne
Lausanne
Switzerland

ISBN 978-3-319-62532-4 ISBN 978-3-319-62533-1 (eBook)
https://doi.org/10.1007/978-3-319-62533-1

Library of Congress Control Number: 2017950255



Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

*For Itay, Sahar and Nofar.*

Mordechai Ben-Ari

*For Luca, Nora, and Leonardo.*

Francesco Mondada

# Preface

Robotics is a vibrant field which grows in importance from year to year. It is also a subject that students enjoy at all levels from kindergarten to graduate school. The aim of learning robotics varies with the age group. For young kids, robots are an educational toy; for students in middle- and high-schools, robotics can increase the motivation of students to study STEM (science, technology, engineering, mathematics); at the introductory university level, students can learn how the physics, mathematics, and computer science that they study can be applied to practical engineering projects; finally, upper level undergraduate and graduate students prepare for careers in robotics.

This book is aimed at the middle of the age range: students in secondary schools and in their first years of university. We focus on robotics algorithms and their mathematical and physical principles. We go beyond trial-and-error play, but we don't expect the student to be able to design and build robots and robotic algorithms that perform tasks in the real world. The presentation of the algorithms without advanced mathematics and engineering is necessarily simplified, but we believe that the concepts and algorithms of robotics can be learned and appreciated at this level, and can serve as a bridge to the study of robotics at the advanced undergraduate and graduate levels.

The required background is a knowledge of programming, mathematics, and physics at the level of secondary schools or the first year of university. From mathematics: algebra, trigonometry, calculus, matrices, and probability. Appendix B provides tutorials for some of the more advanced mathematics. From physics: time, velocity, acceleration, force, and friction.

Hardly a day goes by without the appearance of a new robot intended for educational purposes. Whatever the form and function of a robot, the scientific and engineering principles and algorithms remain the same. For this reason, the book is not based on any specific robot. In Chap. 1 we define a generic robot: a small autonomous mobile robot with differential drive and sensors capable of detecting the direction and distance to an object, as well as ground sensors that can detect markings on a table or floor. This definition is sufficiently general so that students should be able to implement most of algorithms on any educational robot.

The quality of the implementation may vary according to the capabilities of each platform, but the students will be able to learn robotics principles and how to go from theoretical algorithms to the behavior of a real robot.

For similar reasons, we choose not to describe algorithms in any specific programming language. Not only do different platforms support different languages, but educational robots often use different programming approaches, such as textual programming and visual programming using blocks or states. We present algorithms in pseudocode and leave it to the students to implement these high-level descriptions in the language and environment for the robot they are using.

The book contains a large number of *activities*, most of which ask you to implement algorithms and to explore their behavior. The robot you use may not have the capabilities to perform all the activities, so feel free to adapt them to your robot.

This book arose from the development of learning materials for the Thymio educational robot (https://www.thymio.org). The book's website http://elementsofrobotics.net contains implementations of most of the activities for that robot. Some of the more advanced algorithms are difficult to implement on educational robots so Python programs are provided. Please let us know if you implement the activities for other educational robots, and we will post a link on the book's website.

Chapter 1 presents an overview of the field of robotics and specifies the generic robot and the pseudocode used in the algorithms. Chapters 2–6 present the fundamental concepts of autonomous mobile robots: sensors, reactive behavior, finite state machines, motion and odometry, and control. Chapters 7–16 describe more advanced robotics algorithms: obstacle avoidance, localization, mapping, fuzzy logic, image processing, neural networks, machine learning, swarm robotics, and the kinematics of robotic manipulators. A detailed overview of the content is given in Sect. 1.8.

## Acknowledgements

This book arose from the work on the Thymio robot and the Aseba software system initiated by the second author's research group at the Robotic Systems Laboratory of the Ecole Polytechnique Fédérale de Lausanne. We would like to thank all the students, engineers, teachers, and artists of the Thymio community without whose efforts this book could not have been written.

Open access to this book was supported by the Ecole Polytechnique Fédérale de Lausanne and the National Centre of Competence in Research (NCCR) Robotics.

We are indebted to Jennifer S. Kay, Fanny Riedo, Amaury Dame, and Yves Piguet for their comments which enabled us to correct errors and clarify the presentation.

We would like to thank the staff at Springer, in particular Helen Desmond and Beverley Ford, for their help and support.

Rehovot, Israel — Moti Ben-Ari
Lausanne, Switzerland — Francesco Mondada

# Contents

# Chapter 1
# Robots and Their Applications

Although everyone seems to know what a robot is, it is hard to give a precise definition. The Oxford English Dictionary gives the following definition: "A machine capable of carrying out a complex series of actions automatically, especially one programmable by a computer." This definition includes some interesting elements:

- "Carrying out actions automatically." This is a key element in robotics, but also in many other simpler machines called automata. The difference between a robot and a simple automaton like a dishwasher is in the definition of what a "complex series of actions" is. Is washing clothes composed of a complex series of actions or not? Is flying a plane on autopilot a complex action? Is cooking bread complex? For all these tasks there are machines that are at the boundary between automata and robots.
- "Programmable by a computer" is another key element of a robot, because some automata are programmed mechanically and are not very flexible. On the other hand computers are found everywhere, so it is hard to use this criterion to distinguish a robot from another machine.

A crucial element of robots that is not mentioned explicitly in the definition is the use of sensors. Most automata do not have sensors and cannot adapt their actions to their environment. Sensors are what enable a robot to carry out sanscomplex tasks.

In Sects. 1.1–1.5 of this introductory chapter we give a short survey of different types of robots. Section 1.6 describes the generic robot we use and Sect. 1.7 presents the pseudocode used to formalize the algorithms. Section 1.8 gives a detailed overview of the contents of the book.



M. Ben-Ari and F. Mondada, *Elements of Robotics*,
https://doi.org/10.1007/978-3-319-62533-1_1

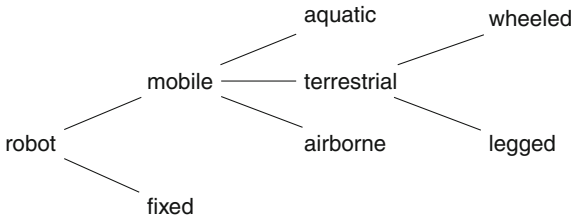



**Fig. 1.1** Classification of robots by environment and mechanism of interaction

## 1.1 Classification of Robots

Robots can be classified according to the environment in which they operate (Fig. 1.1). The most common distinction is between *fixed* and *mobile* robots. These two types of robots have very different working environments and therefore require very different capabilities. Fixed robots are mostly industrial robotic manipulators that work in well defined environments adapted for robots. Industrial robots perform specific repetitive tasks such soldering or painting parts in car manufacturing plants. With the improvement of sensors and devices for human-robot interaction, robotic manipulators are increasingly used in less controlled environment such as high-precision surgery.

By contrast, mobile robots are expected to move around and perform tasks in large, ill-defined and uncertain environments that are not designed specifically for robots. They need to deal with situations that are not precisely known in advance and that change over time. Such environments can include unpredictable entities like humans and animals. Examples of mobile robots are robotic vacuum cleaners and self-driving cars.

There is no clear dividing line between the tasks carried out by fixed robots and mobile robots—humans may interact with industrial robots and mobile robots can be constrained to move on tracks—but it is convenient to consider the two classes as fundamentally different. In particular, fixed robots are attached to a stable mount on the ground, so they can compute their position based on their internal state, while mobile robots need to rely on their perception of the environment in order to compute their location.

There are three main environments for mobile robots that require significantly different design principles because they differ in the mechanism of motion: aquatic (underwater exploration), terrestrial (cars) and aerial (drones). Again, the classification is not strict, for example, there are amphibious robots that move in both water and on the ground. Robots for these three environments can be further divided into subclasses: terrestrial robots can have legs or wheels or tracks, and aerial robots can be lighter-than-air balloons or heavier-than-air aircraft, which are in turn divided into fixed-wing and rotary-wing (helicopters).

Robots can be classified by intended application field and the tasks they perform (Fig. 1.2). We mentioned industrial robots which work in well-defined environments

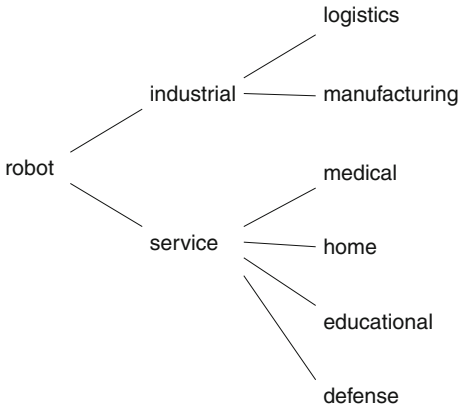



**Fig. 1.2** Classification of robots by application field

on production tasks. The first robots were industrial robots because the well-defined environment simplified their design. Service robots, on the other hand, assist humans in their tasks. These include chores at home like vacuum clears, transportation like self-driving cars, and defense applications such as reconnaissance drones. Medicine, too, has seen increasing use of robots in surgery, rehabilitation and training. These are recent applications that require improved sensors and a closer interaction with the user.

## 1.2 Industrial Robots

The first robots were industrial robots which replaced human workers performing simple repetitive tasks. Factory assembly lines can operate without the presence of humans, in a well-defined environment where the robot has to perform tasks in a specified order, acting on objects precisely placed in front of it (Fig. 1.3).

One could argue that these are really automata and not robots. However, today's automata often rely on sensors to the extent that they can be considered as robots. However, their design is simplified because they work in a customized environment which humans are not allowed to access while the robot is working.

However, today's robots need more flexibility, for example, the ability to manipulate objects in different orientations or to recognize different objects that need to be packaged in the right order. The robot can be required to transport goods to and from warehouses. This brings additional autonomy, but the basic characteristic remains: the environment is more-or-less constrained and can be adapted to the robot.

Additional flexibility is required when industrial robots interact with humans and this introduces strong safety requirements, both for robotic arms and for mobile robots. In particular, the speed of the robot must be reduced and the mechanical

**Fig. 1.3** Robots on an assembly line in a car factory. *Source* https://commons.wikimedia.org/wiki/File:AKUKA_Industrial_Robots_IR.jpg by Mixabest (Own work). CC BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0) or GFDL (http://www.gnu.org/copyleft/fdl.html), via Wikimedia Commons

design must ensure that moving parts are not a danger to the user. The advantage of humans working with robots is that each can perform what they do best: the robots perform repetitive or dangerous tasks, while humans perform more complex steps and define the overall tasks of the robot, since they are quick to recognize errors and opportunities for optimization.

## 1.3 Autonomous Mobile Robots

Many mobile robots are remotely controlled, performing tasks such as pipe inspection, aerial photography and bomb disposal that rely on an operator controlling the device. These robots are not autonomous; they use their sensors to give their operator remote access to dangerous, distant or inaccessible places. Some of them can be semi-autonomous, performing subtasks automatically. The autopilot of a drone stabilizes the flight while the human chooses the flight path. A robot in a pipe can control its movement inside the pipe while the human searches for defects that need

to be repaired. Fully *autonomous mobile robots* do not rely on an operator, but instead they make decisions on their own and perform tasks, such as transporting material while navigating in uncertain terrain (walls and doors within buildings, intersections on streets) and in a constantly changing environment (people walking around, cars moving on the streets).

The first mobile robots were designed for simple environments, for example, robots that cleaned swimming pools or robotic lawn mowers. Currently, robotic vacuum cleaners are widely available, because it has proved feasible to build reasonably priced robots that can navigate an indoor environment cluttered with obstacles.

Many autonomous mobile robots are designed to support professionals working in structured environments such as warehouses. An interesting example is a robot for weeding fields (Fig. 1.4). This environment is partially structured, but advanced sensing is required to perform the tasks of identifying and removing weeds. Even in very structured factories, robot share the environment with humans and therefore their sensing must be extremely reliable.

Perhaps the autonomous mobile robot getting the most publicity these days is the self-driving car. These are extremely difficult to develop because of the highly complex uncertain environment of motorized traffic and the strict safety requirements.

**Fig. 1.4** Autonomous mobile robot weeding a field (Courtesy of Ecorobotix)

An even more difficult and dangerous environment is space. The Sojourner and Curiosity Mars rovers are semi-autonomous mobile robots. The Sojourner was active for three months in 1997. The Curiosity has been active since landing on Mars in 2012! While a human driver on Earth controls the missions (the routes to drive and the scientific experiments to be conducted), the rovers do have the capability of autonomous hazard avoidance.

Much of the research and development in robotics today is focused on making robots more autonomous by improving sensors and enabling more intelligent control of the robot. Better sensors can perceive the details of more complex situations, but to handle these situations, control of the behavior of the robot must be very flexible and adaptable. Vision, in particular, is a very active field of research because cameras are cheap and the information they can acquire is very rich. Efforts are being made to make systems more flexible, so that they can learn from a human or adapt to new situations. Another active field of research addresses the interaction between humans and robots. This involves both sensing and intelligence, but it must also take into account the psychology and sociology of the interactions.

## 1.4 Humanoid Robots

Science fiction and mass media like to represent robots in a humanoid form. We are all familiar with R2-D2 and 3-CPO, the robotic characters in the *Star Wars* movies, but the concept goes far back. In the eighteenth century, a group of Swiss watchmakers—Pierre and Henri-Louis Jaquet-Droz and Jean-Frédéric Leschot—built humanoid automata to demonstrate their mechanical skills and advertise their watches. Many companies today build humanoid robots for similar reasons.

Humanoid robots are a form of autonomous mobile robot with an extremely complex mechanical design for moving the arms and for locomotion by the legs. Humanoid robots are used for research into the mechanics of walking and into human-machine interaction. Humanoid robots have been proposed for performing services and maintenance in a house or a space station. They are being considered for providing care to the elderly who might feel anxious in the presence of a machine that did not appear human. On the other hand, robots that look very similar to humans can generate repulsion, a phenomenon referred to as the *uncanny valley*.

Humanoid robots can be very difficult to design and control. They are expensive to build with multiple joints that can move in many different ways. Robots that use wheels or tracks are preferred for most applications because they are simpler, less expensive and robust.

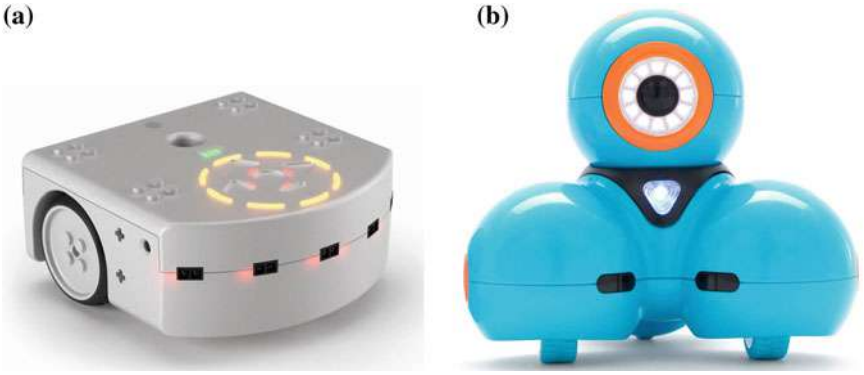



**Fig. 1.5** **a** Thymio robot. *Source* https://www.thymio.org/en:mediakit by permission of École Polytechnique Fédérale de Lausanne and École Cantonale d'Art de Lausanne. **b** Dash robot. *Source* https://www.makewonder.com/mediakit by permission of Wonder Workshop

## 1.5 Educational Robots

Advances in the electronics and mechanics have made it possible to construct robots that are relatively inexpensive. Educational robots are used extensively in schools, both in classrooms and in extracurricular activities. The large number of educational robots makes it impossible to give a complete overview. Here we give few examples that are representative of robots commonly used in education.

### Pre-Assembled Mobile Robots

Many educational robots are designed as pre-assembled mobile robots. Figure 1.5a shows the Thymio robot from Mobsya and Fig. 1.5b shows the Dash robot from Wonder Workshop. These robots are relatively inexpensive, robust and contain a large number of sensors and output components such as lights. An important advantage of these robots is that you can implement robotic algorithms "out of the box," without investing hours in mechanical design and construction. However, pre-assembled robots cannot be modified, though many do support building extensions using, for example, LEGO® components.

### Robotics Kits

The LEGO® Mindstorms robotics kits (Fig. 1.6a) were introduced in 1998.[1] A kit consists of standard LEGO® bricks and other building components, together with motors and sensors, and a programmable brick which contains the computer that controls the components of the robot. The advantage of robotics kits is that they are

[1]The figure shows the latest version called *EV3* introduced in 2014.

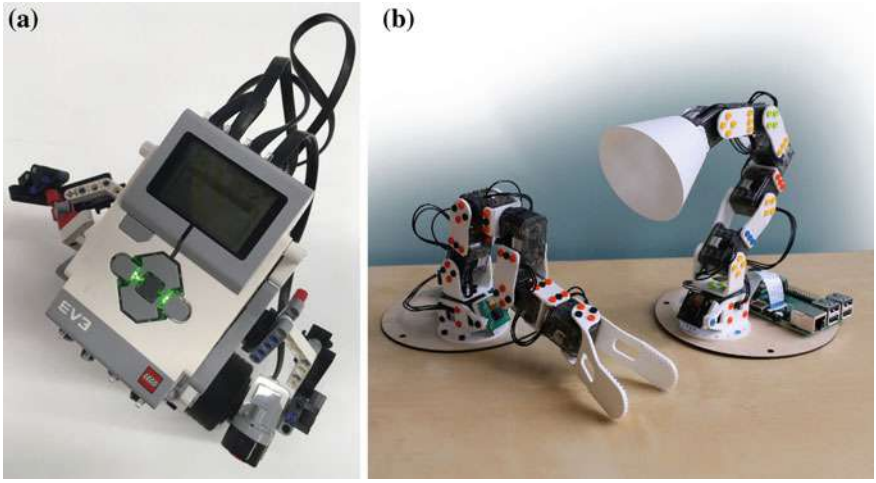



**Fig. 1.6** **a** LEGO® Mindstorms EV3 (Courtesy of Adi Shmorak, Intelitek), **b** Poppy Ergo Jr robotic arms (Courtesy of the Poppy Project)

flexible: you can design and build a robot to perform a specific task, limited only by your imagination. A robotics kit can also be used to teach students mechanical design. The disadvantages of robotics kits are that they are more expensive than simple pre-assembled robots and that exploration of robotics algorithms depends on the one's ability to successfully implement a robust mechanical design.

A recent trend is to replace fixed collections of bricks by parts constructed by 3D printers. An example is the Poppy Ergo Jr robotic arm (Fig. 1.6b). The use of 3D printed parts allows more flexibility in the creation of the mechanical structure and greater robustness, but does require access to a 3D printer.

### Robotic Arms

To act on its environment, the robot needs an *actuator* which is a component of a robot that affects the environment. Many robots, in particular robotic arms used in industry, affect the environment through *end effectors*, usually grippers or similar tools (Figs. 1.3, 14.1 and 15.5b). The actuators of mobile robots are the motors that cause the robot to move, as well as components such as the vacuum pump of a vacuum cleaner.

Educational robots are usually mobile robots whose only actuators are its motors and display devices such as lights, sounds or a screen. End effectors can be built with robotics kits or by using additional components with pre-assembled robots, although educational robotic arms do exist (Fig. 1.6b). Manipulation of objects introduces

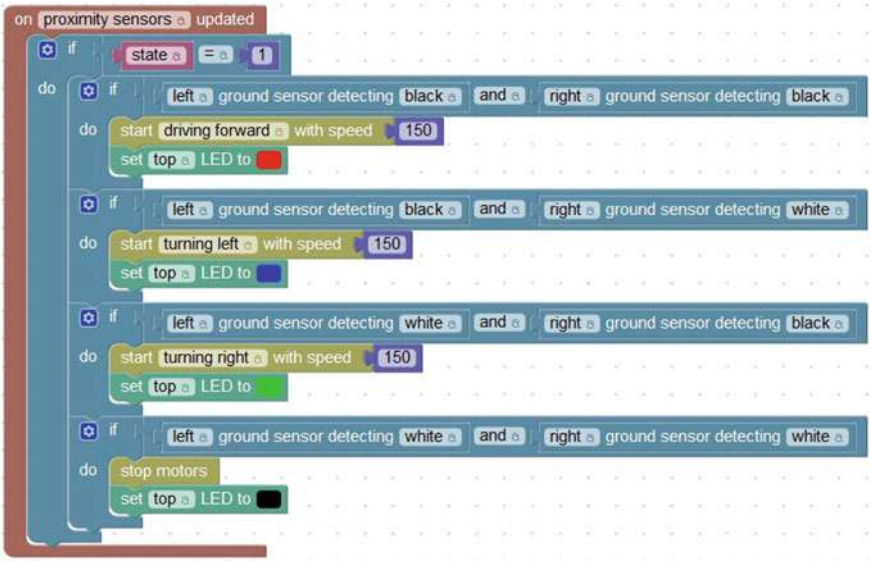



**Fig. 1.7** Blockly software for the Thymio robot

complexity into the design; however, since the algorithms for end effectors are similar to the algorithms for simple mobile robots, most of the activities in the book will assume only that your robot has motors and display devices.

### Software Development Environments

Every educational robotics system includes a *software development environment*. The programming language can be a version of a standard programming language like Java or Python. Programming is simplified if a block-based language is used, usually a language based upon Scratch or Blockly (Fig. 1.7).

To further simplify programming a robot by young students, a fully graphical programming notation can be used. Figure 1.8 shows VPL (Visual Programming Language), a graphical software environment for the Thymio robot. It uses event-action pairs: when the event represented by the block on the left occurs, the actions in the following blocks are performed.

Figure 1.9 shows the graphical software environment for the Dash robot. It also uses events and actions, where the actions are represented by nodes and events are represented by arrows between nodes.

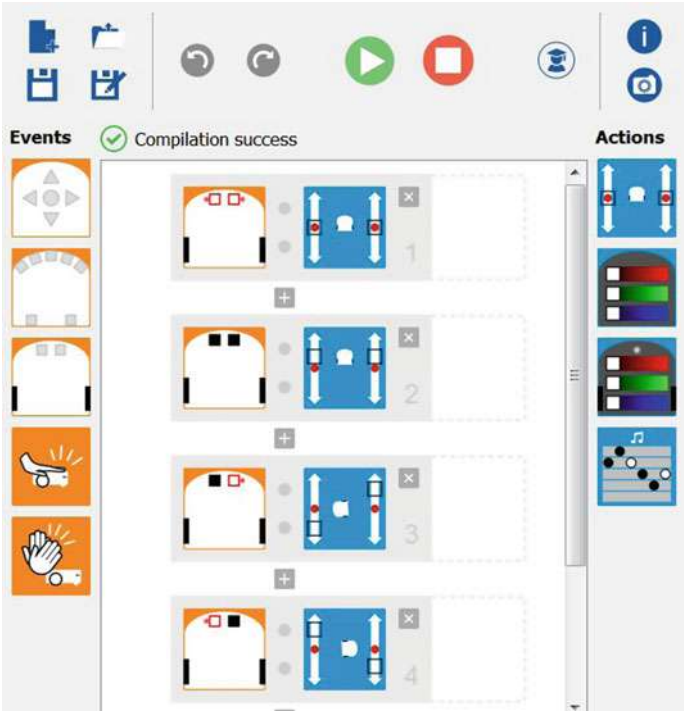



**Fig. 1.8** VPL software for the Thymio robot


**Fig. 1.9** Wonder software for the Dash robot. *Source* https://www.makewonder.com/mediakit by permission of Wonder Workshop

## 1.6 The Generic Robot

This section presents the description of a generic robot that we use to present the robotics algorithms. The capabilities of the generic robot are similar to those found in educational robots, but the one you use may not have all the capabilities assumed in the presentations so you will have to improvise. You may not understand all the terms in the following description just yet, but it is important that the specification be formalized. Further details will be given in later chapters.

### *1.6.1 Differential Drive*

The robot is a small autonomous vehicle with *differential drive*, meaning that it has two wheels that are driven by independent motors (Fig. 1.10). To cause the robot to move, set the motor power to a value from −100 (full power backwards) through 0 (stopped) to 100 (full power forwards). There is no predefined relationship between the motor power and the velocity of robot. The motor can be connected to the wheels through different gear ratios, the type of tires on the wheels affects their traction, and sandy or muddy terrain can cause the wheels to slip.

Figure 1.10 shows a view of the robot from above. The front of the robot is the curve to the right which is also the forward direction of the robot's motion. The wheels (black rectangles) are on the left and right sides of the rear of the robot's body. The dot is the point on the axle halfway between the wheels. When the robot turns, it turns around an axis vertical to this point. For stability, towards the front of the robot there is a support or non-driven wheel.

**Mechanical Drawing**

A broken line is the standard notation in mechanical engineering for the *axis of symmetry* in a component such as a wheel. When the side view of a wheel is displayed, the intersection of the two axes of symmetry denotes the axis of rotation that is perpendicular to the plane of the page. To avoid cluttering the diagrams, we simplify the notation by only showing broken lines for an *axis of rotation* of a component such as a wheel. In addition, the intersection

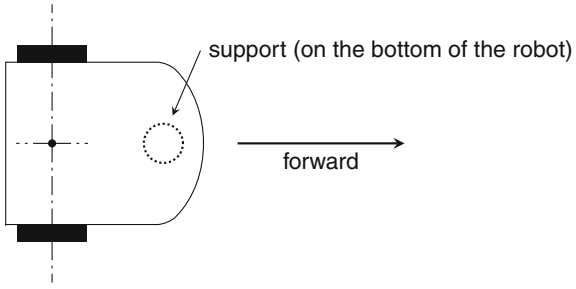



**Fig. 1.10** Robot with differential drive

denoting a perpendicular axis is usually abbreviated to a cross, possibly contained within the wheel or its axle.

Differential drive has several advantages: it is simple since it has only two motors without additional components for steering and it allows the robot to turn in place. In a car, two wheels are driven together (or four wheels are driven in pairs) and there is a separate complex mechanism for steering called *Ackermann steering*. Since a car cannot turn in place, drivers must perform complicated maneuvers such as parallel parking; human drivers readily learn to do this, but such maneuvers are difficult for an autonomous system. An autonomous robot needs to perform intricate maneuvers with very simple movements, which is why differential drive is the preferred configuration: it can easily turn to any heading and then move in that direction.

The main disadvantage of a differential drive system is that it requires a third point of contact with the ground unlike a car which already has four wheels to support it and thus can move easily on difficult terrain. Another disadvantage is that it cannot drive laterally without turning. There are configurations that enable a robot to move laterally (Sect. 5.12), but they are complex and expensive. Differential drive is also used in tracked vehicles such as earth-moving equipment and military tanks. These vehicles can maneuver in extremely rough terrain, but the tracks produce a lot of friction so movement is slow and not precise.

**Setting Power or Setting Speed**

The power supplied by a motor is regulated by a *throttle*, such as a pedal in a car or levers in an airplane or boat. Electrical motors used in mobile robots are controlled by modifying the voltage applied to the motors using a technique called *pulse width modulation*. In many educational robots, control algorithms such as those described in Chap. 6, are used to ensure that the motors rotate at a specified *target speed*. Since we are interested in concepts and algorithms for designing robots, we will express algorithms in terms of supplying power and deal separately with controlling speed.

### *1.6.2 Proximity Sensors*

The robot has *horizontal proximity sensors* that can detect an object near the robot. There exist many technologies that can be used to construct these sensors, such as infrared, laser, ultrasound; the generic robot represents robots that use any of these technologies. We do specify that the sensors have the following capabilities: A horizontal proximity sensor can measure the distance (in centimeters) from the robot to an object and the angle (in degrees) between the front of the robot and the object. Figure 1.11a shows an object located at 3 cm from the center of the robot at an angle of 45° from the direction in which the robot is pointing.[2]

In practice, an educational robot will have a small number of sensors, so it may not be able detect objects in all directions. Furthermore, inexpensive sensors will not be able to detect objects that are far away and their measurements will not be

[2]See Appendix A on the conventions for measuring angles.

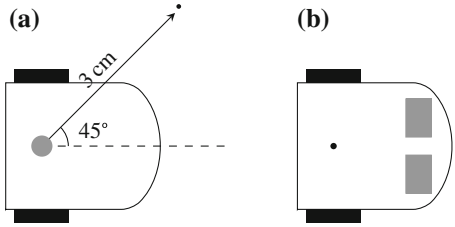



**Fig. 1.11** **a** Robot with a rotating sensor (*gray dot*), **b** Robot with two ground sensors on the *bottom* of the robot (*gray rectangles*)

accurate. The measurements will also be affected by environmental factors such as the type the object, the ambient light, and so on. To simplify our algorithms, we do not assume any predefined limitations, but when you implement the algorithms you will have to take the limitations into account.

### *1.6.3 Ground Sensors*

*Ground sensors* are mounted on the bottom of the robot. Since these sensors are very close to the ground, there is no meaning to distance or angle; instead, the sensor measures the brightness of the light reflected from the ground in arbitrary values between 0 (totally dark) and 100 (totally light). The generic robot has two ground sensors mounted towards the front of the robot (Fig. 1.11b), though sometimes we present algorithms that use only one sensor. The figure shows a top view of the robot although the ground sensors are on the *bottom* of the robot.

### *1.6.4 Embedded Computer*

The robot is equipped with an *embedded computer* (Fig. 1.12). The precise specification of the computer is not important but we do assume certain capabilities. The computer can read the values of the sensors and set the power of the motors. There is a way of displaying information on a small screen or using colored lights. Signals and data can be input to the computer using buttons, a keypad or a remote control.

Data is input to the computer by *events* such as touching a button. The occurrence of an event causes a procedure called an *event handler* to be run. The event can be detected by the hardware, in which case the term *interrupt* is used, or it can be detected by the software, usually, by *polling*, where the operating system checks for events at predefined intervals. When the event handler terminates, the previous computation is started up again.

Event handlers are different from sequential programs that have an initial instruction that inputs data and a final instruction that displays the output, because event handlers are run in response to unpredictable events. Event handling is used to implement graphic user interfaces on computers and smartphones: when you click on or touch an icon an event handler is run. On a robot, an event can be a discrete input like touching a key. Events can also occur when a continuous value like the value read by a sensor goes above or below a predefined value called a *threshold*.

The computer includes a *timer* which functions like a stopwatch on a smartphone. A timer is a variable that is *set* to a period of time, for example, 0.5 s, which is represented as an integer number milliseconds or microseconds (0.5 s is 500 ms). The hardware clock of the computer causes an interrupt at fixed intervals and the operating system decrements the value of the timer. When its value goes to zero, we say that the timer has *expired*; an interrupt occurs.

Timers are used to implement repeated events like flashing a light on and off. They are also used for *polling*, an alternative to event handlers: instead of performing a computation when an event occurs, sensors are read and stored periodically. More precisely, polling occurs as an event handler when a timer expires, but the design of software using polling can be quite different from the design of event-based software.

## 1.7 The Algorithmic Formalism

Algorithms that are implemented as computer programs are used by the embedded computer to control the behavior of the robot. We do not give programs in any specific programming language; instead, algorithms are presented in *pseudocode*, a structured format using a combination of natural language, mathematics and programming structures. Algorithm 1.1 is a simple algorithm for integer multiplication using repeated addition. The input of the algorithm is a pair of integers and the output is the product of the two input values. The algorithm declares three integer variables `x`, `a`, `b`. There are five statements in the executable part. Indentation is used (as in the Python programming language) to indicate the scope of the loop. An arrow is used

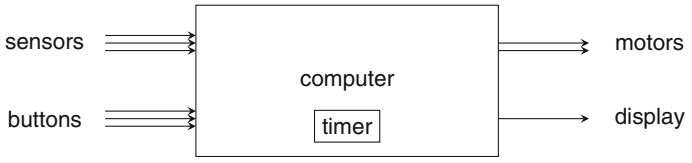



**Fig. 1.12** Embedded computer

for assignment so that the familiar symbols $=$ and $\neq$ can be used for equality and inequality in mathematical formulas.[3]

| Algorithm 1.1: Integer multiplication | |
|---|---|
| integer x $\leftarrow$ 0<br>integer a, b | |
| 1: a $\leftarrow$ input an integer | |
| 2: b $\leftarrow$ input a non-negative integer | |
| 3: while b $\neq$ 0 | |
| 4: x $\leftarrow$ x + a | // Add the value a to x |
| 5: b $\leftarrow$ b − 1 | // for each b |

The motor power is set using assignment statements:

```
left-motor-power ← 50
right-motor-power ← −50
```

We have defined our proximity sensors as returning the distance to a detected object and its angle relative to the forward direction of the robot, but it will often be more convenient to use natural language expressions such as:

```
when object detected in front
when object not detected in back
```

## 1.8 An Overview of the Content of the Book

The first six chapters form the core of robotics concepts and algorithms.

**Chapter 1 Robots and Their Applications** This chapter surveys and classifies robots. It also specifies the generic robot and formalisms used to present algorithms in this book.

**Chapter 2 Sensors** Robots are more than remotely controlled appliances like a television set. They show autonomous behavior based on detecting objects in their environment using sensors. This chapter gives an overview of the sensors used by robots and explains the concepts of range, resolution, precision and accuracy. It also discusses the nonlinearity of sensors and how to deal with it.

**Chapter 3 Reactive Behavior** When an autonomous robot detects an object in its environment, it reacts by changing its behavior. This chapter introduces robotics algorithms where the robot directly changes its behavior based upon input from its sensors. Braitenberg vehicles are simple yet elegant examples of reactive behavior. The chapter presents several variants of line following algorithms.

[3] Many programming languages use = for assignment and then == for equality and `!=` for inequality. This is confusing because equality $x = y$ is symmetrical, but assignment is not `x=x+1`. We prefer to retain the mathematical notation.

**Chapter 4 Finite State Machines** A robot can be in different states, where its reaction to input from its sensors depends not only on these values but also on the current state. Finite state machines are a formalism for describing states and the transitions between them that depend on the occurrence of events.
**Chapter 5 Robotic Motion and Odometry** Autonomous robots explore their environment, performing actions. Hardly a day goes by without a report on experience with self-driving cars. This chapter reviews concepts related to motion (distance, time, velocity, acceleration), and then presents odometry, the fundamental method that a robot uses to move from one position to another. Odometry is subject to significant errors and it is important to understand their nature.
The second part of the chapter gives an overview of advanced concepts of robotic motion: wheel encoders and inertial navigation systems that can improve the accuracy of odometry, and degrees of freedom and holonomy that affect the planning of robotic motion.
**Chapter 6 Control** An autonomous robot is a closed loop control system because input from its sensors affects its behavior which in turn affects what is measured by the sensors. For example, a self-driving car approaching a traffic light can brake harder as it gets close to the light. This chapter describes the mathematics of control systems that ensure optimal behavior: the car actually does stop at the light and the braking is gradual and smooth.

An autonomous mobile robot must somehow navigate from a start position to a goal position, for example, to bring medications from the pharmacy in a hospital to the patient. Navigation is a fundamental problem in robotics that is difficult to solve. The following four chapters present navigation algorithms in various contexts.

**Chapter 7 Local Navigation: Obstacle Avoidance** The most basic requirement from a mobile robot is that it does not crash into walls, people and other obstacles. This is called *local* navigation because it deals with the immediate vicinity of the robot and not with goals that the robot is trying to reach. The chapter starts with wall following algorithms that enable a robot to move around an obstacle; these algorithms are similar to algorithms for navigating a maze. The chapter describes a probabilistic algorithm that simulates the navigation by a colony of ants searching for a food source.
**Chapter 8 Localization** Once upon a time before every smartphone included GPS navigation, we used to navigate with maps printed on paper. A difficult problem is localization: can you determine your current position on the map? Mobile robots must solve the same localization problem, often without the benefit of vision. The chapter describes localization by trigonometric calculations from known positions. This is followed by sections on probabilistic localization: A robot can detect a landmark but there may be many similar landmarks on the map. By assigning probabilities and updating them as the robot moves through the environment, it can eventually determine its position with relative certainty.
**Chapter 9 Mapping** But where does the map come from? Accurate street maps are readily available, but a robotic vacuum cleaner does not have a map of your apartment. An undersea robot is used to explore an unknown environment. To perform

localization the robot needs a map, but to create a map of an unknown environment the robot needs to localize itself, in the sense that it has to know how far it has moved from one point of the environment to another. The solution is to perform simultaneous localization and mapping. The first part of the chapter describes an algorithm for exploring an environment to determine the locations of obstacles. Then an simplified algorithm for simultaneous localization and mapping is presented.

**Chapter 10 Mapping-Based Navigation** Now that the robot has a map, suppose that it is assigned a task that requires it to move from a start position to a goal position. What route should it take? This chapter presents two algorithms for path planning: Dijkstra's algorithm, a classic algorithm for finding the shortest path in a graph, and the A* algorithm, a more efficient version of Dijkstra's algorithm that uses heuristic information.

The following chapters present advanced topics in robotics. They are independent of each other so you can select which ones to study and in what order.

**Chapter 11 Fuzzy Logic Control** Control algorithms (Chap. 6) require the specification of a precise target value: a heating system needs the target temperature of a room and a cruise control system needs the target speed of a car. An alternate approach called fuzzy logic uses imprecise specifications like cold, cool, warm, hot, or very slow, slow, fast, very fast. This chapter presents fuzzy logic and shows how it can be used to control a robot approaching an object.

**Chapter 12 Image Processing** Most robotic sensors measure distances and angles using lasers, sound or infrared light. We humans rely primarily on our vision. High quality digital cameras are inexpensive and found on every smartphone. The difficulty is to process and interpret the images taken by the camera, something our brains do instantly. Digital image processing has been the subject of extensive research and its algorithms are used in advanced robots that can afford the needed computing power. In this chapter we survey image processing algorithms and show how an educational robot can demonstrate the algorithms even without a camera.

**Chapter 13 Neural Networks** Autonomous robots in highly complex environments cannot have algorithms for every possible situation. A self-driving car cannot possibly know in advance all the different vehicles and configurations of vehicles that it encounters on the road. Autonomous robots must learn from their experience and this is a fundamental topic in artificial intelligence that has been studied for many years. This chapter presents one approach to learning: artificial neural networks modeled on the neurons in our brains. A neural network uses learning algorithms to modify its internal parameters so that it continually adapts to new situations that it encounters.

**Chapter 14 Machine Learning** Another approach to learning is a statistical technique called machine learning. This chapter describes two algorithms for distinguishing between two alternatives, for example, distinguishing between a traffic light that is red and one that is green. The first algorithm, called linear discriminant analysis, is based on the means and variances of a set of samples. The second algorithm uses perceptrons, a form of neural network that can distinguish between alternatives even when the samples do not satisfy the statistical assumptions needed for linear discriminant analysis.

**Chapter 15 Swarm Robotics** If you need to improve the performance of a system, it is often easier to use multiple instances of a component rather than trying to improve the performance of an individual component. Consider a problem such as surveying an area to measure the levels of pollution. You can use a single very fast (and expensive) robot, but it can be easier to use multiple robots, each of which measures the pollution in a small area. This is called swarm robotics by analogy with a swarm of insects that can find the best path between their nest and a source of food. The fundamental problem in swarm robotics, as in all concurrent systems, is to develop methods for coordination and communications among the robots. This chapter presents two such techniques: exchange of information and physical interactions.

**Chapter 16 Kinematics of a Robotic Manipulator** Educational robots are small mobile robots that move on a two-dimensional surface. There are mobile robots that move in three-dimensions: robotic aircraft and submarines. The mathematics and algorithms for three-dimensional motion were developed in another central field of robotics: manipulators that are used extensively in manufacturing. This chapter presents a simplified treatment of the fundamental concepts of robotic manipulators (forward and inverse kinematics, rotations, homogeneous transforms) in two dimensions, as well as a taste of three-dimensional rotations.

There are two appendices:

**Appendix A Units of Measurement** This appendix contains Table A.1 with units of measurements. Table A.2 gives prefixes that are used with these units.

**Appendix B Mathematical Derivations and Tutorials** This chapter contains tutorials that review some of the mathematical concepts used in the book. In addition, some of the detailed mathematical derivations have been collected here so as not to break the flow of the text.

## 1.9 Summary

Robots are found everywhere: in factories, homes and hospitals, and even in outer space. Much research and development is being invested in developing robots that interact with humans directly. Robots are used in schools in order to increase students' motivation to study STEM and as a pedagogical tool to teach STEM in a concrete environment. The focus of this book is the use of educational robots to learn robotic algorithms and to explore their behavior.

Most educational robots have a similar design: a small mobile robot using differential drive and proximity sensors. To make this book platform-independent, we defined a generic robot with these properties. The algorithms presented in this book for the generic robot should be easy to implement on educational robots, although different robots will have different capabilities in terms of the performance of their motors and sensors. The algorithms are presented in a language-independent pseudocode that should be easy to translate into any textual or graphics language that your robot supports.

## 1.10 Further Reading

For a non-technical overview of robotics with an emphasis on biologically-inspired and humanoid robotics, see Winfield [8].

The International Organization for Standardization (ISO)[4] publishes standards for robotics. On their website https://www.iso.org/ you can find the catalog of robotics (ISO/TC 299) and formal definitions of robotics concepts: ISO 8373:2012 Robots and robotic devices—Vocabulary and ISO 19649:2017 Mobile robots—Vocabulary.

The topics in this book are presented in greater detail in advanced textbooks on robotics such as [3, 6]. Their introductory chapters give many examples of robots.

Educational robots come with documentation of their capabilities and software development environments. There are also textbooks based upon specific robots, for example, [7] on programming the LEGO® Mindstorms robotics kits and [4] on using Python to program the Scribbler robots. The design of the Visual Programming Language (VPL) is presented in [5].

Pseudocode is frequently used in textbooks on data structures and algorithms, starting from the classic textbook [1]. The style used here is taken from [2].

## References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley, USA (1974)
2. Ben-Ari, M.: Principles of Concurrent and Distributed Programming, 2nd edn. Addison-Wesley, USA (2006)
3. Dudek, G., Jenkin, M.: Computational Principles of Mobile Robotics, 2nd edn. Cambridge University Press, UK (2010)
4. Kumar, D.: Learning Computing with Robots. Lulu (2011). Download from http://calicoproject.org/Learning_Computing_With_Robots
5. Shin, J., Siegwart, R., Magnenat, S.: Visual programming language for Thymio II robot. In: Proc. of the 2014 Conference on Interaction Design and Children (IDC) (2014)
6. Siegwart, R., Nourbakhsh, I.R., Scaramuzza, D.: Introduction to Autonomous Mobile Robots, 2nd edn. MIT Press, USA (2011)
7. Trobaugh, J.J., Lowe, M.: Winning LEGO MINDSTORMS Programming. Apress (2012)
8. Winfield, A.: Robotics: A Very Short Introduction. Oxford University Press, USA (2012)

[4]No, this is not a mistake! ISO is the official abbreviated name of the organization and not an acronym in any of its three official languages: English, French and Russian.

# Chapter 2
# Sensors

A robot cannot move a specific distance in a specific direction just by setting the relative power of the motors of the two wheels and the period of time that the motors run. Suppose that we want the robot to move straight ahead. If we set the power of the two motors to the same level, even small differences in the characteristics of the motors and wheels will cause the robot turn slightly to one side. Unevenness in the surface over which the robot moves will also cause the wheels to turn at different speeds. Increased or decreased friction between the wheels and the surface can affect the distance moved in a specific period of time. Therefore, if we want robot to move towards a wall 1 m away and stop 20 cm in front of it, the robot must *sense* the existence of the wall and stop when it *detects* that the wall is 20 cm away.

A *sensor* is a component that measures some aspects of the environment. The computer in the robot uses these measurements to control the actions of the robot. One of the most important sensors in robotics is the distance sensor that measures the distance from the robot to an object. By using multiple distance sensors or by rotating the sensor, the angle of the object relative to the front of the robot can be measured. Inexpensive distance sensors using infrared light or ultrasound are invariably used in educational robots; industrial robots frequently use expensive laser sensors because they are highly accurate.

Sound and light are also used for *communications* between two robots as described in Chap. 15.

More extensive knowledge of the environment can be obtained by analyzing images taken by a camera. While cameras are very small and inexpensive (every smartphone has one), the amount of data in an image is very large and image-processing algorithms require significant computing resources. Therefore, cameras are primarily used in complex applications like self-driving cars.

Section 2.1 introduces the terminology of sensors. Section 2.2 presents distance sensors, the sensors most often used by educational robots. This is followed by Sect. 2.3 on cameras and then a short section on other sensors that robots use



M. Ben-Ari and F. Mondada, *Elements of Robotics*,
https://doi.org/10.1007/978-3-319-62533-1_2

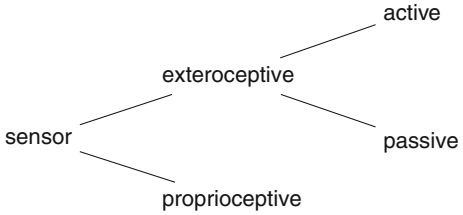



**Fig. 2.1** Classification of sensors

(Sect. 2.4). Section 2.5 defines the characteristics of sensors: range, resolution, precision, accuracy. The chapter concludes with a discussion of the nonlinearity of sensors (Sect. 2.6).

## 2.1 Classification of Sensors

Sensors are classified as *proprioceptive* or *exteroceptive*, and exteroceptive sensors are further classified as *active* or *passive* (Fig. 2.1). A proprioceptive sensor measures something internal to the robot itself. The most familiar example is a car's speedometer which measures the car's speed by counting rotations of the wheels (Sect. 5.8). An exteroceptive sensor measures something external to the robot such as the distance to an object. An active sensor affects the environment usually by emitting energy: a sonar range finder on a submarine emits sound waves and uses the reflected sound to determine range. A passive sensor does not affect the environment: a camera simply records the light reflected off an object. Robots invariably use some exteroceptive sensors to correct for errors that might arise from proprioceptive sensors or to take changes of the environment into account.

## 2.2 Distance Sensors

In most applications, the robot needs to measure the distance from the robot to an object using a *distance sensor*. Distance sensors are usually *active*: they transmit a signal and then receive its reflection (if any) from an object (Fig. 2.2). One way of determining distance is to measure the time that elapses between sending a signal and receiving it:

$$s = \frac{1}{2} vt \,, \tag{2.1}$$

where $s$ is the distance, $v$ is the velocity of the signal and $t$ is the elapsed time between sending and receiving the signal. The factor of one-half takes into account that the

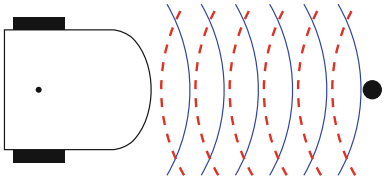


**Fig. 2.2** Measuring distance by transmitting a wave and receiving its reflection

signal travels the distance twice: to the object and then reflected back. Another way of reconstructing the distance is by using triangulation as explained in Sect. 2.2.4.

Low-cost distance sensors are based on another principle: since the intensity of a signal decreases with distance, measuring the intensity of a reflected signal gives an indication of the distance from the sensor to an object. The disadvantage of this method is that the intensity of the received signal is influenced by factors such as the reflectivity of the object.

### *2.2.1 Ultrasound Distance Sensors*

Ultrasound is sound whose frequency is above 20,000 hertz, higher than the highest frequency that can be heard by the human ear. There are two environments where sound is better than vision: at night and in water. Bats use ultrasound for navigating when flying at night because after the sun sets there is little natural light for locating food. Ships and submarines use ultrasound for detecting objects because sound travels much better in water than it does in air. Check this yourself by going for a swim in a muddy lake or in the ocean: How far away can you see a friend? Now, ask him to make a sound by hitting two objects together or by clapping his hands. How far away can you hear the sound?

The speed of sound in air is about 340 m/s or 34,000 cm/s. If an object is at a distance of 34 cm from a robot, from Eq. 2.1 it follows that an ultrasound signal will travel to the object and be reflected back in:

$$\frac{2 \cdot 34}{34000} = \frac{2}{1000} = 2 \times 10^{-3} \text{ s} = 2 \text{ ms}.$$

An electronic circuit can easily measure periods of time in milliseconds.

The advantage of ultrasound sensors is that they are not sensitive to changes in the color or light reflectivity of objects, nor to the light intensity in the environment. They are, however, sensitive to texture: fabric will absorb some of the sound, while wood or metal will reflect almost all the sound. That is why curtains, carpets and soft ceiling tiles are used to make rooms more comfortable.

Ultrasound sensors are relatively cheap and can work in outdoor environments. They are used in cars for detecting short distances, such as to aid in parking. Their

main disadvantage is that the distance measurement is relatively slow, since the speed of sound is much less than the speed of light. Another disadvantage is that they cannot be focused in order to measure the distance to a specific object.

### 2.2.2 Infrared Proximity Sensors

*Infrared light* is light whose wavelength is longer than red light, which is the light with the longest wavelength that our eyes can see. The eye can see light with wavelengths from about 390 to 700 nm (a nanometer is one-millionth of a millimeter). Infrared light has wavelengths between 700 and 1000 nm. It is invisible to the human eye and is therefore used in the remote control of TV sets and other electronic appliances.

Proximity sensors are simple devices that use light to detect the presence of an object by measuring the intensity of the reflected light. Light intensity decreases with the square of the distance from the source and this relationship can be used to measure the approximate distance to the object. The measurement of the distance is not very accurate because the reflected intensity also depends on the reflectivity of the object. A black object reflects less light than a white object placed at the same distance, so a proximity sensor cannot distinguish between a close black object and a white object placed somewhat farther away. This is the reason why these sensors are called *proximity* sensors, not distance sensors. Most educational robots use proximity sensors because they are much less expensive than distance sensors.

### 2.2.3 Optical Distance Sensors

Distance can be computed from a measurement of the elapsed time between sending a light signal and receiving it. The light can be ordinary light or light from a laser. Light produced by a laser is *coherent* (see below). Most commonly, lasers for measuring distance use infrared light, although visible light can also be used. Lasers have several advantages over other light sources. First, lasers are more powerful and can detect and measure the distance to objects at long range. Second, a laser beam is highly focused, so an accurate measurement of the angle to the object can be made (Fig. 2.3).

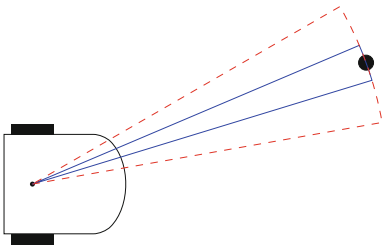


**Fig. 2.3** Beam width of laser light (*solid*) and non-coherent light (*dashed*)

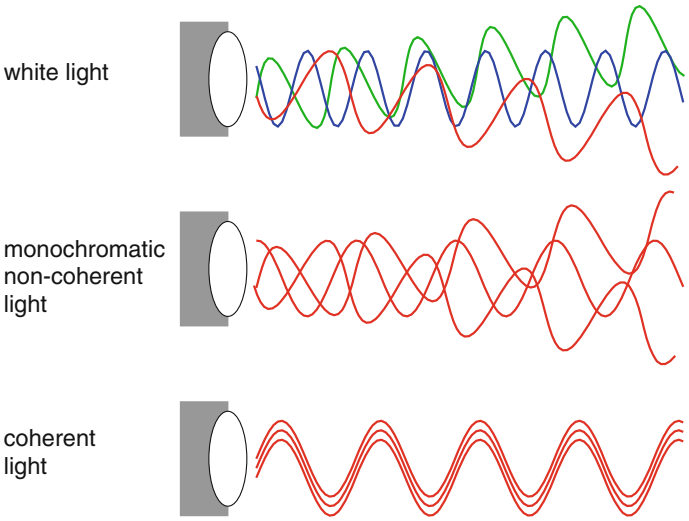



**Fig. 2.4** White, monochromatic and coherent light

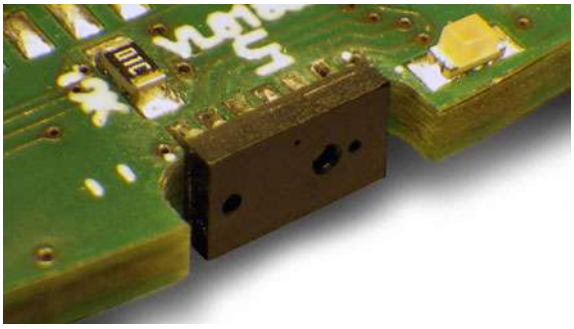


**Fig. 2.5** A time of flight distance sensor (*black*) mounted on a 1.6 mm thick printed circuit (*green*)

**Coherent light**

Three types of light are shown in Fig. 2.4. The light from the sun or a light bulb is called *white light* because it is composed of light of many different colors (frequencies), emitted at different times (phases) and emitted in different directions. Light-emitting diodes (LED) emit *monochromatic light* (light of a single color), but they are non-coherent because their phases are different and they are emitted in different directions. Lasers emit *coherent light*: all waves are of the same frequency and the same phase (the start of each wave is at the same time). All the energy of a light is concentrated in a narrow beam and distance can be computed by measuring the time of flight and the difference in phase of the reflected light.

Suppose that a pulse of light is transmitted by the robot, reflected off an object and received by a sensor on the robot. The speed of light in air is about 300,000,000 m/s, which is $3 \times 10^8$ m/s or $3 \times 10^{10}$ cm/s in scientific notation. If a light signal is directed at an object 30 cm from the robot, the time for the signal to be transmitted and received is (Fig. 2.5):

$$\frac{2 \cdot 30}{3 \times 10^{10}} = \frac{2}{10^9} = 2 \times 10^{-9}\,\text{s} = 0.002\,\text{ms}$$

This is a very short period of time but it can be measured by electronic circuits.

The second principle of distance measurement by a light beam is triangulation. In this case the transmitter and the receiver are placed at different locations. The receiver detects the reflected beam at a position that is function of the distance of the object from the sensor.

### 2.2.4 Triangulating Sensors

Before explaining how a *triangulating sensor* works, we have to understand how the reflection of light depends on the object it hits. When a narrow beam of light like the coherent light from a laser hits a shiny surface like a mirror, the light rays bounce off in a narrow beam. The angle of reflection relative to the surface of the object is the same as the angle of incidence. This is called *specular* reflection (Fig. 2.6a). When the surface is rough the reflection is *diffuse* (Fig. 2.6b) in all directions because even very close areas of the surface have slightly different angles. Most objects in an environment like people and walls reflect diffusely, so to detect reflected laser light the detector need not be placed at a precise angle relative to the transmitter.

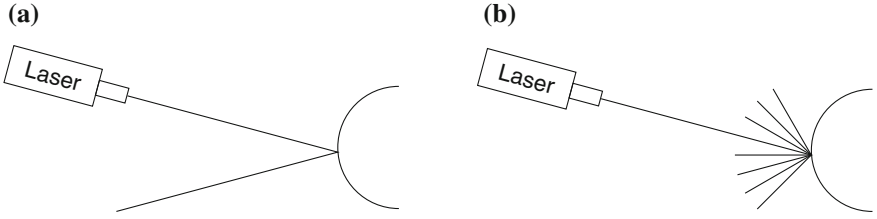



**Fig. 2.6** **a** Specular reflection, **b** Diffuse reflection

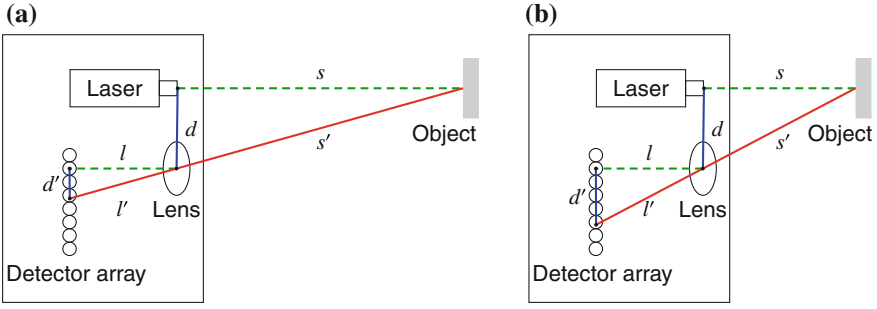



**Fig. 2.7** **a** Triangulation of a far object, **b** Triangulation of a near object

Figure 2.7a–b show a simplified triangulating sensor detecting an object at two distances. The sensor consists of a laser transmitter and at a distance $d$ away a lens that focuses the received light onto an array of sensors placed at a distance $l$ behind the lens. Assuming that the object reflects light diffusely, some of the light will be collected by the lens and focused onto the sensors. The distance $d$ along the sensor array is inversely proportional to the distance $s$ of the object from the laser emitter.

The triangles $\triangle ll'd'$ and $\triangle ss'd$ are similar, so we have the formula:

$$\frac{s}{d} = \frac{l}{d'} .$$

Since $l$ and $d$ are fixed by construction, by measuring $d'$ from the index of the sensor which detects the focused light, we can compute $s$, the distance of the object from the sensor. The sensor has to be calibrated by measuring the distance $s$ corresponding to each sensor within the array, but once a table is stored within the computer, the distance $s$ can be performed by a table lookup.

There are many design parameters that affect the performance of a triangulating distance sensor: the power of the laser, the optical characteristics of the lens, the number of sensors in the array and their sensitivity. In addition to the usual trade-off of performance and cost, the main trade-off is between the range and the minimal distance at which an object can be measured. For a very short distance $s$, the size of the detector array $d'$ becomes very large and this puts a practical limit on the minimal distance. The minimal distance can be made shorter by increasing the distance between the laser emitter and the detector array, but this reduces the range. A triangulating sensor can be characterized by the distance $s_{opt}$ for optimal performance, the minimal distance and the range around $s_{opt}$ at which measurements can be made.

### 2.2.5 *Laser Scanners*

When ultrasound or proximity sensors are used, a small number of sensors can be placed around the robot in order to detect objects anywhere in the vicinity of the robot (Fig. 2.8a). Of course, the angle to the object cannot be measured accurately, but at least the object will be detected and the robot can approach or avoid the object.

With a laser sensor, the width of the beam is so small that a large number of (expensive) lasers would be needed to detect objects at any angle. A better design is to mount a single laser sensor on a rotating shaft to form a *laser scanner* (Fig. 2.8b). An angular sensor can be used to determine the angle at which an object is detected. Alternatively, the computer can measure the period of time after the rotating sensor passes a fixed direction. A full rotation of 360° enables a laser scanner to generate a profile of the objects in the environment (Fig. 2.9).

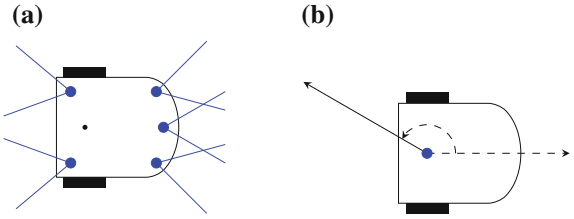



**Fig. 2.8** **a** Five separate sensors, **b** A rotating sensor

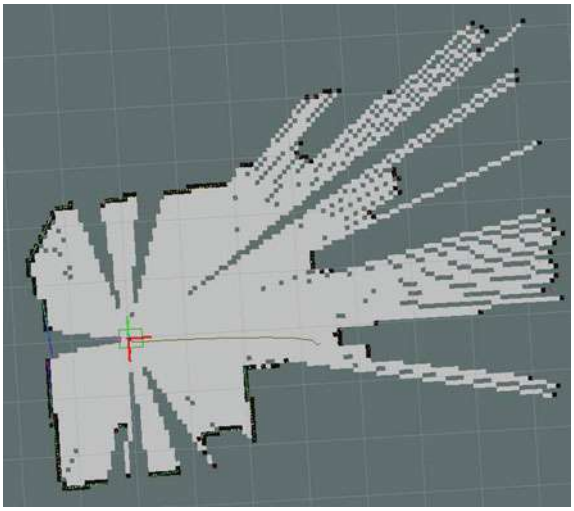


**Fig. 2.9** A map of the environment obtained by a laser scanner

**Activity 2.1: Range of a distance sensor**

- Determine the maximum range at which the proximity sensors on your robot can detect an object. Is there also a minimum range or can objects be detected even if they are placed in direct contact with the sensor? If there is a minimum range, explain why closer objects cannot be detected.
- Your software may enable you to measure numerical values returned by the sensor. If so, are these values distances or are they just arbitrary values that need to be converted into distance? If they are arbitrary values, find a formula for the conversion or construct a table that gives the distances for different values returned by the sensor.
- A sensor that does not use coherent laser light can detect an object to its left or right, not only objects that are directly in front of it. Measure the angle

at which it is possible to detect objects. Can objects be detected at the same angle to the left and to the right of the center of the sensor?
- How many sensors would you need to be able to detect an object placed anywhere around your robot?

**Activity 2.2: Thresholds**

- A mobile robot like a self-driving car does not stop exactly in front of an obstacle; it leaves some extra space for safety, perhaps 1 m or 50 cm. Define a *threshold*, the minimum safe distance to an object, and program your robot so that it stops at this distance from an object.
- If your software does not enable you to measure numerical values returned by the sensor, it may enable you to take an action when the returned value passes one or more thresholds (for example, when the object is "close," "middle," "far"). Measure the distances corresponding to these thresholds: place an object close to the sensor and slowly move it away. Record the distances at which the thresholds are crossed.

**Activity 2.3: Reflectivity**

- Since an infrared proximity sensor works by measuring the light reflected by an object, it is reasonable to assume that the measured values depend on the characteristic of the object. Repeat the experiments in Activity 2.1 for objects of different shapes, colors, and materials. Summarize your conclusions.
- Try to extend the range at which your sensor can detect an object: use an object with a polished metal surface, attach a mirror to the object or paste reflecting tape used by joggers and cyclists onto the object.
- If your robot has an ultrasound sensor, perform these experiments for this sensor and compare different textures of the surface of an object.

**Activity 2.4: Triangulation**

- Use a laser pointer to create a beam toward an object placed about 50 cm away. Dim the lights or close the curtains on the windows so that you can see the reflection of the beam on the object. Then place a camera on a table or tripod about 10 cm to the side of the laser and point it at the spot; now take a

picture. Move the object farther away and take another picture. What do you observe when you compare the two pictures?
- Move the object further and further away from the laser and the camera, and write down the distances and the place of the spot on the picture from the edge of the image. Plot the data. Explain your observations. What defines the minimal and maximal distance that this sensor can measure?

## 2.3 Cameras

Digital cameras are widely used in robotics because a camera can provide much more detailed information than just the distance and the angle to an object. Digital cameras use an electronic component called a *charge-coupled device* which senses light waves and returns an array of *picture elements*, or *pixels* for short (Fig. 2.10).

Digital cameras are characterized by the number of pixels captured in each frame and by the content of the pixels. A small camera used in one educational robot contains 192 rows of 256 pixels each for a total of 49,152 pixels. This is a very small picture: the sensors of digital cameras in smartphones record images of millions of pixels.

A camera can return values for each pixel as black and white (1 bit per pixel), shades of gray called grayscale (8 bits per pixel) or full color red-green-blue (RGB)

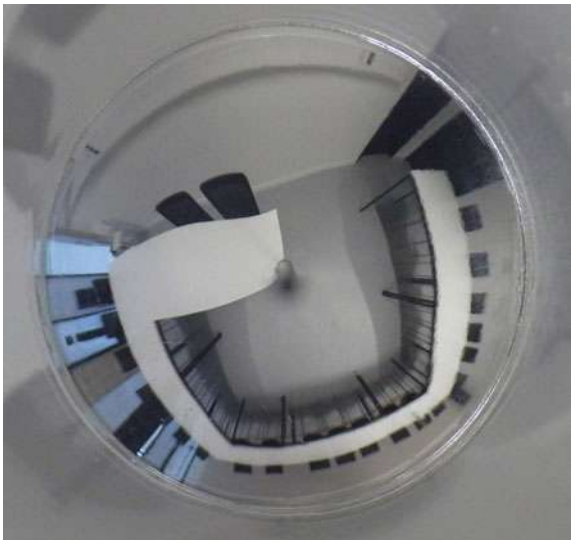


**Fig. 2.10** An image captured by an omnidirectional camera with a field of view of 360 degrees

($3 \times 8 = 24$ bits per pixel). The small $256 \times 192$ camera thus needs about 50 kbyte for a single grayscale image or 150 kbyte for a color image. Since a mobile robot such as a self-driving car will need to store several images per second (movies and TV display 24 images per second), the memory needed to store and analyze images can be very large.

An important characteristic in the design of a camera for a robot is the *field of view* of its lens. Given the position of the camera, what portion of the sphere surrounding the camera is captured in the image? For a given sensor in a camera, a lens with a narrow field of view captures a small area with high-resolution and little distortion, whereas a lens with a wide field of view captures a large area with lower resolution and more distortion. The most extreme distortion arises from an *omnidirectional camera* which captures in a single image (almost) the entire sphere surrounding it. Figure 2.10 shows an image of a conference room taken by an omnidirectional camera; the camera's position is indicated by the black spot at the center. Cameras with a wide field of view are used in mobile robots because the image can be analyzed to understand the environment. The analysis of the image is used for navigation, to detect objects in the environment and to interact with people or other robots using visual properties like color.

The fundamental issue with cameras in robotics is that we are not interested in an array of "raw" pixels, but in identifying the objects that are in the image. The human eye and brain instantly perform recognition tasks: when driving a car we identify other cars, pedestrians, traffic lights and obstacles in the road automatically, and take the appropriate actions. Image processing by a computer requires sophisticated algorithms and significant processing power (Chap. 12). For that reason, robots with cameras are much more complex and expensive than educational robots that use proximity sensors.

## 2.4 Other Sensors

A *touch sensor* can be considered to be a simplified distance sensor that measures only two values: the distance to an object is zero or greater than zero. Touch sensors are frequently used as safety mechanisms. For example, a touch sensor is mounted on the bottom of small room heaters so that the heater runs only if the touch sensor detects the floor. If the heater falls over, the touch sensor detects that it is no longer in contact with the floor and the heating is shut off to prevent a fire. A touch sensor can be used on a mobile robot to apply an emergency brake if the robot approaches too close to a wall.

Buttons and switches enable the user to interact directly with the robot.

A *microphone* on the robot enables it to sense sound. The robot can simply detect the sound or it can use algorithms to interpret voice commands.

An *accelerometer* measures acceleration. The primary use of accelerometers is to measure the direction of the gravitational force which causes an acceleration of about 9.8 m/sec$^2$ towards the center of the earth. With three accelerometers mounted

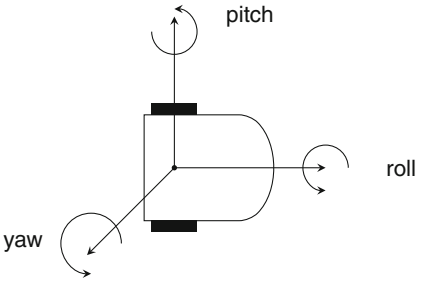



**Fig. 2.11** Three-axis accelerometer

perpendicular to each other (Fig. 2.11), the *attitude* of the robot can be measured: the three angles of the robot, called *pitch*, *yaw* and *roll*. Accelerometers are discussed in greater detail in Sect. 5.9.1 and a task using accelerometers is presented in Sect. 14.4.1.

**Activity 2.5: Measuring the attitude using accelerometers**

- Write a program that displays the attitude of your robot when you pick it up and rotate it around all three axes.
- Implement a game of your choice using the robot as a controller.
- Write a program that causes the robot to move forwards, stopping if an incline is reached. Use the accelerometer that measures pitch.

## 2.5 Range, Resolution, Precision, Accuracy

Whenever a physical quantity is measured, the measurement can be characterized by its range, resolution, precision and accuracy, concepts that are often confused.

The *range* is the extent of the set of values that can be measured by a sensor. An infrared proximity sensor might be able to measure distances in the range 1 cm–30 cm. Since laser beams focus a lot of power into a narrow beam they have a much larger range. The range needed by a distance sensor for a robot moving in a building will be about 10 m, while a distance sensor for a self-driving car needs to measure distances of about 100 m.

*Resolution* refers to the smallest change that can be measured. One distance sensor may return distances in centimeters (1 cm, 2 cm, 3 cm, 4 cm, …), while a better sensor returns distances in hundredths of a centimeter (4.00 cm, 4.01 cm, 4.02 cm, …). For a self-driving car, a resolution of centimeters should be sufficient: you wouldn't park a car 1 cm from another, to say nothing of parking it 0.1 cm away. A surgical robot

needs a much higher resolution since even a millimeter is critical when performing surgery.

*Precision* refers to the consistency of the measurement. If the same quantity is measured repeatedly, is the same value returned? Precision is very important because inconsistent measurements will lead to inconsistent decisions. Suppose that a sensor of self-driving car measures distances to the nearest 10 cm, but successive measurements return a wide range of values (say, 250 cm, 280 cm, 210 cm). When trying to maintain a fixed separation from a vehicle it is following, the car will speed up and slow down for no good reason, resulting in an uncomfortable and energy-wasting ride.

Very often a sensor will have a high resolution but low precision; in that case, the resolution cannot be trusted. For example, a distance sensor might return values in millimeters, but if the precision is not sufficiently high, returning, say, 45 mm, 43 mm, 49 mm, the sensor should only be trusted to return values within the nearest centimeter or half-centimeter.

**Activity 2.6: Precision and resolution**

- What is the resolution of the distance sensors on your robot?
- Place an object at a fixed distance from your robot and repeatedly record the distance measured. What is the precision of the measurement?
- Measure the distance to an object under different circumstances such as changes in temperature and light. Turn the heater or air-conditioner on and off; turn the lights on and off. Do the measurements change?

*Accuracy* refers to the closeness of a measurement to the real-world quantity being measured. If a distance sensor consistently claims that the distance is 5 cm greater than it actually is, the sensor is not accurate. In robotics, accuracy is not as important as precision, because a sensor measurement does not directly return a physical quantity. Instead, a computation is performed to obtain a physical quantity such as distance or speed from a measured electronic value. If the inaccuracy is consistent, the sensor value can be calibrated to obtain the true physical quantity (Sect. 2.6). A distance sensor using light or sound computes the distance from the time of flight of a signal $s = vt/2$. If we know that the sensor consistently returns a value 5 cm too large, the computer can simply use the formula $s = (vt/2) - 5$.

**Activity 2.7: Accuracy**

- Place an object at various distances from the robot and measure the distances returned by the sensor. Are the results accurate? If not, can you write an function that transforms the sensor measurements into distances?

## 2.6 Nonlinearity

Sensors return electronic quantities such as potential or current which are proportional to what is being measured. The analog values are converted into digital values. For example, a proximity sensor might return 8 bits of data (values between 0 and 255) that represent a range of distances, perhaps 0–50 cm. An 8-bit sensor cannot even return angles in the range 0°–360° at a resolution of one degree. The computer must translate the digital values into measurements of a physical quantity. Discovering the mapping for this translation is called *calibration*. In the best case, the mapping will be linear and easy to compute; if not, if the mapping is nonlinear, a table or nonlinear function must be used. Tables are more efficient because looking up an entry is faster than computing a function, but tables require a lot of memory.

### *2.6.1 Linear Sensors*

If a horizontal distance sensor is *linear*, there is a mapping $x = as + b$, where $x$ is the value returned by the sensor, $s$ is the distance of an object from the sensor and $a, b$ are constants ($a$ is the slope and $b$ is the intercept with the sensor axis). Suppose that the sensor returns the value 100 for an object at 2 cm and the value 0 for an object at 30 cm (Fig. 2.12).

Let us compute the slope and the intercept:

$$slope = \frac{\Delta x}{\Delta s} = \frac{0 - 100}{30 - 2} = -3.57\,.$$

When $s = 30, x = 0 = -3.57 \cdot 30 + b$, so $b = 107$ and $x = -3.57s + 107$. Solving for $s$ gives a function that the robot's computer can use to map a sensor value to the corresponding distance:

$$s = \frac{107 - x}{3.57}\,.$$

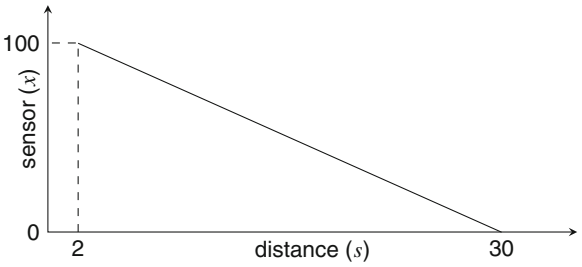



**Fig. 2.12** Value returned as a linear function of distance

**Activity 2.8: Linearity**

- Tape a ruler on your table and carefully place the robot so that its front sensor is positioned next to the 0 mark of the ruler. Place an object next to the 1 cm mark on the ruler. Record the value returned by the sensor. Repeat for 2 cm, 3 cm, …, until the value returned goes to zero.
- Plot a graph of value returned vs. distance. Is the response of the sensor linear? If so, compute the slope and the intercept.
- Repeat the experiment with objects of different shapes and materials. Does the linearity of the graph depend on the characteristics of the object?

### *2.6.2 Mapping Nonlinear Sensors*

Figure 2.13 shows a possible result of the measurements in Activity 2.8. The measurements are shown as dots together with the linear function from Fig. 2.12. The function is reasonably linear in the middle of its range but nonlinear outside that range. This means that it impossible to use a linear function of the raw values of the sensor to obtain the distance of an object from the robot.

We can construct a table to map sensor values to distances. Table 2.1 is a table based upon real measurements with an educational robot. Measurements were made every two centimeters from 2 cm to 18 cm; at 20 cm the sensor no longer detected the object. The second column shows the value of the sensor for each distance. The third column shows the values $x_l$ that would be returned by the sensor if it were linear with the function $x = -2s + 48$. We see that the actual values returned by the sensor do not deviate too much from linearity, so it would not be unreasonable to use a linear function.

Obviously, it would be better if we had a table entry for each of the possible values returned by the sensor. However, this would take a lot of memory and may be

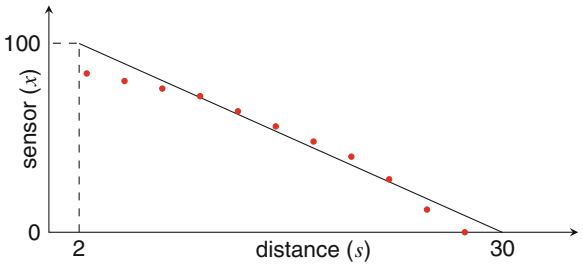



**Fig. 2.13** Experimental values returned as a function of distance

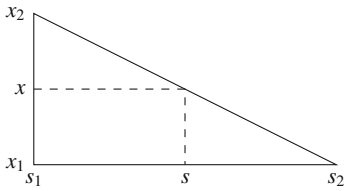



**Fig. 2.14** Interpolation of sensor values

impractical if the range of values returned by the sensor is much larger, say, from 0 to 4095 (12 bits). One solution is to take the nearest value, so that if the value 27 is returned by the sensor whose mapping is given in Table 2.1, the distance would be 12.

A better solution is to use interpolation. If you look again at the graph in Fig. 2.13, you can see that the segments of the curve are roughly linear, although their slopes change according to the curve. Therefore, we can get a good approximation of the distance corresponding to a sensor value by taking the relative distance on a straight line between two points (Fig. 2.14). Given distances $s_1$ and $s_2$ corresponding to sensor values $x_1$ and $x_2$, respectively, for a value $x_1 < x < x_2$, its distance $s$:

$$s = s_1 + \frac{s_2 - s_1}{x_2 - x_1}(x - x_1)\,.$$

## 2.7 Summary

When designing a robot, the choice of sensors is critical. The designer needs to decide *what* needs to the measured: distance, attitude, velocity, etc. Then the designer has

**Table 2.1** A table mapping sensor values to distances

| $s$ (cm) | $x$ | $x_l$ |
|---|---|---|
| 18 | 14 | 12 |
| 16 | 18 | 16 |
| 14 | 22 | 20 |
| 12 | 26 | 24 |
| 10 | 29 | 28 |
| 8 | 32 | 32 |
| 6 | 36 | 36 |
| 4 | 41 | 40 |
| 2 | 44 | 44 |

to make trade-offs: larger range, finer resolution, higher precision and accuracy are always better, but come at a price. For educational robots, price is the overriding consideration, so don't expect good performance from your robot. Nevertheless, the algorithmic principles are the same whether the sensors are of high quality or not, so the trade-off does not affect the ability to learn with the robot.

Any sensor connected to the robot's computer is going to return discrete values within a fixed range. The computer must be able to map these sensor values to physical quantities in a process called calibration. If the sensor is linear, the calibration results in two values (slope and intercept) that determine a linear function. If the sensor is nonlinear, a table or a non-linear function must be used.

## 2.8 Further Reading

For an overview of sensors used in mobile robots see [2, Sect. 4.1]. The book by Everett [1] is devoted entirely to this topic.

## References

1. Everett, H.: Sensors for Mobile Robots. A.K, Peters (1995)
2. Siegwart, R., Nourbakhsh, I.R., Scaramuzza, D.: Introduction to Autonomous Mobile Robots, 2nd edn. MIT Press, Cambridge (2011)



# Chapter 3
# Reactive Behavior

We are now ready to write our first algorithms for robots. These algorithms demonstrate *reactive behavior*: an event (such as the detection of a nearby object by the robot) causes the robot to react by performing an action that changes its behavior (such as stopping the motors). Purely reactive behavior occurs when the action is related only to the occurrence of an event and does not depend on data stored in memory (state).

The reactive behaviors are those of *Braitenberg vehicles* which are appropriate for introducing robotics because complex behavior arises from simple algorithms. Sections 3.1–3.3 describe Braitenberg vehicles that demonstrate reactive behavior; in Chap. 4 we present Braitenberg vehicles that have non-reactive behavior with states. Section 3.4 presents several algorithms for the classic reactive behavior of line following. Line following is an interesting task because the algorithms are sensitive to the characteristics of the sensors and the lines. Calibration is necessary to determine the optimum thresholds for fast robust motion of the robot. Section 3.5 gives a brief overview of Braitenberg's original formulation of the vehicles in a biological approach with sensors connected directly to the motors, not through a computer. Later in the book (Sect. 13.3) we discuss the implementation of Braitenberg vehicles using neural networks.

## 3.1 Braitenberg Vehicles

Valentino Braitenberg was a neuroscientist who described the design of virtual vehicles that exhibited surprisingly complex behavior. Researchers at the MIT Media Lab developed hardware implementations of the vehicles from *programmable bricks* that



M. Ben-Ari and F. Mondada, *Elements of Robotics*,
https://doi.org/10.1007/978-3-319-62533-1_3

were the forerunner of the LEGO® Mindstorms robotics kits.[1] This chapter describes an implementation of most of the Braitenberg vehicles from the MIT report. The MIT hardware used light and touch sensors, while our generic robot is based upon horizontal proximity sensors.

To facilitate comparison with the MIT report (and indirectly with Braitenberg's book), the names of their vehicles have been retained, even though it may be difficult to understand their meanings in our implementations.

Two vehicles are presented in detail by giving all of the following:

- The specification of the behavior of the robot;
- A formalized algorithm for the specified behavior;
- An activity that asks you to implement the algorithm on your robot.

The other vehicles are presented in activities that specify the behavior and ask you to develop an algorithm and to implement it on your robot.

## 3.2 Reacting to the Detection of an Object

**Specification (Timid)**: When the robot does not detect an object, it moves forwards. When it detects an object, it stops.

Algorithm 3.1 implements this behavior.

**Algorithm 3.1: Timid**

```
1: when object not detected in front
2:     left-motor-power ← 100
3:     right-motor-power ← 100
4:
5: when object detected in front
6:     left-motor-power ← 0
7:     right-motor-power ← 0
```

The algorithm uses two event handlers, one for the event of detecting an object and one for the event of not detecting an object. The event handlers are written using the when statement whose meaning is:

when the event *first occurs*, perform the following actions.

Why do we use this construct and not the more familiar while statement (Algorithm 3.2)?

[1] The MIT report uses the term *Braitenberg creatures* but we retain the original term.

If we use the while statement, *as long as* the object is not detected the motors will be turned on, and *as long as* the object is detected the motors will be turned off. Since the sensor will detect the object over a range of distances, the motors will be repeatedly turned on or off. It is likely that no harm will be done if a motor that is already off is turned off and a motor that is already on is turned on, but these repeated commands are not necessary and may waste resources. Therefore, we prefer to turn the motors off only when the object is first detected and to turn them on only when the object is first not detected. The when statement gives the semantics that we want.

**Algorithm 3.2: Timid with while**

```
1: while object not detected in front
2:     left-motor-power ← 100
3:     right-motor-power ← 100
4:
5: while object detected in front
6:     left-motor-power ← 0
7:     right-motor-power ← 0
```

**Activity 3.1: Timid**

- Implement the Timid behavior.

**Activity 3.2: Indecisive**

- Implement the Indecisive behavior.

  **Specification (Indecisive)**: When the robot does not detect an object, it moves forwards. When it detects an object, it moves backwards.

- At just the right distance, the robot will *oscillate*, that is, it will move forwards and backwards in quick succession. Measure this distance for your robot and for objects of different reflectivity.

**Activity 3.3: Dogged**

- Implement the Dogged behavior.

  **Specification (Dogged)**: When the robot detects an object in front, it moves backwards. When the robot detects an object in back, it moves forwards.

**Activity 3.4: Dogged (stop)**

- Implement the Dogged (stop) behavior.

  **Specification (Dogged (stop))**: As in Activity 3.3, but when an object is not detected, the robot stops.

**Activity 3.5: Attractive and repulsive**

- Implement the Attractive and repulsive behavior.

  **Specification (Attractive and repulsive)**: When an object approaches the robot from behind, the robot runs away until it is out of range.

## 3.3 Reacting and Turning

A car turns by changing the angle of its front wheels relative to the frame of the vehicle. The motor power is not changed. A robot with differential drive has no steering mechanism (like the steering wheel of a car or the handlebars of a bicycle). Instead, it turns by setting different levels of power to the left and right wheels. If one wheel turns faster than the other, the robot turns in the direction opposite that of the faster wheel (Fig. 3.1a). If one wheel turns backwards while the other turns forwards, the turn is much sharper (Fig. 3.1b). In the figures, the arrows denote the direction and speed of each wheel. The *turning radius* is the radius of the circle that is the path of the robot. We say that a turn is tighter if the radius is smaller. At the extreme, if one wheel turns forwards and the second turns backwards at the same speed, the robot will turn in place and the turning radius is zero.

Let us now implement a Braitenberg vehicle whose specification requires the robot to turn.

**Specification (Paranoid)**: When the robot detects an object, it moves forwards (colliding with the object). When it does not detect an object, it turns to the left.

Algorithm 3.3 implements this behavior.

**Algorithm 3.3: Paranoid**

```
1: when object detected in front
2:     left-motor-power ← 100
3:     right-motor-power ← 100
4:
5: when object not detected in front
6:     left-motor-power ← −50        // Left motor backwards
7:     right-motor-power ← 50        // Right motor forwards
```

**Activity 3.6: Paranoid**

- Implement the Paranoid behavior.
- In the algorithm, the left and right motors are set to equal but opposite powers. Experiment with these power levels to see their influence on the turning radius of the robot.

**Activity 3.7: Paranoid (right-left)**

- Implement the Paranoid (right-left) behavior.

  **Specification (Paranoid (right-left))**: When an object is detected in front of the robot, the robot moves forwards. When an object is detected to the right of the robot, the robot turns right. When an object is detected to the left of the robot, the robot turns left. When no object is detected the robot does not move.

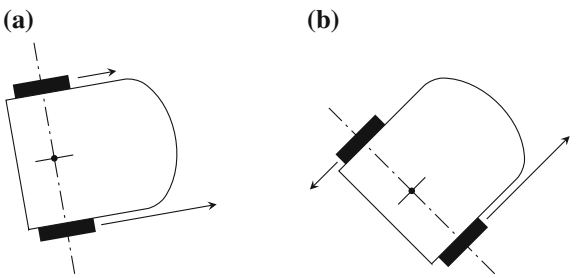



**Fig. 3.1** **a** Gentle left turn. **b** Sharp left turn

**Activity 3.8: Insecure**

- Implement the Insecure behavior.

  **Specification (Insecure)**: If an object is not detected to the left of the robot, set the right motor to rotate forwards and set the left motor off. If an object is detected to the left of the robot, set the right motor off and set the left motor to rotate forwards.

- Experiment with the motor settings until the robot follows a wall to its left.

**Activity 3.9: Driven**

- Implement the Driven behavior.

  **Specification (Driven)**: If an object is detected to the left of the robot, set the right motor to rotate forwards and set the left motor off. If an object is detected to the right of the robot, set the right motor off and set the left motor to rotate forwards.

- Experiment with the motor settings until the robot approaches the object in a zigzag.

## 3.4 Line Following

Consider a warehouse with robotic carts that bring objects to a central dispatching area (Fig. 3.2). Lines are painted on the floor of the warehouse and the robot is instructed to follow the lines until it reaches the storage bin of the desired object.

*Line following* is a task that brings out all the uncertainty of constructing robots in the real world. The line might not be perfectly straight, dust may obscure part of the line, or dirt may cause one wheel to move more slowly than the other. To follow

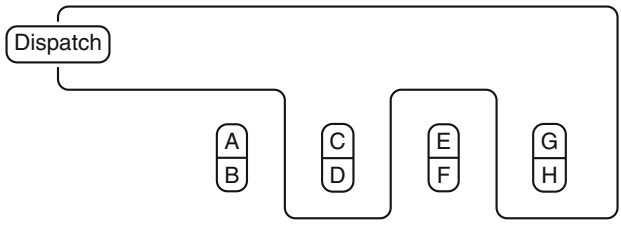



**Fig. 3.2** A robotic warehouse

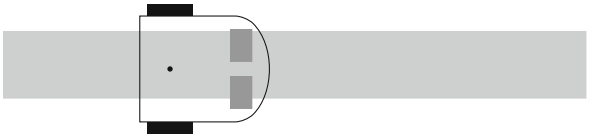


**Fig. 3.3** A robot with two ground sensors over a line

**Fig. 3.4** Leaving the line

a line, the robot must decide whether it is on the line or not, and if it starts to leave the line on one side, it must turn in the correct direction to regain the line.

### *3.4.1 Line Following with a Pair of Ground Sensors*

To follow a line, a pair of ground sensors can be used (Fig. 3.3).[2] A ground sensor on a light-colored floor will detect a lot of reflected light. If a dark line is painted on the floor, the sensor will detect very little reflected light when it is over the line.[3] The line should be black for increased contrast with the white floor, but the figure displays the line in light gray so as not to obscure the robot and its sensors. Thresholds are used to determine when the event occurs of a sensor moving from detecting the line to detecting the floor or conversely.

The line must be wide enough so that both ground sensors will sense dark when the robot is directly above the line. The sensors do not have to be entirely over the line; it is sufficient that the amount of light reflected from line onto the sensor be below the threshold defined for black.

To implement line-following, the robot must move forward whenever both sensors detect a dark surface, indicating that it is on the line. If the robot starts to leave the line, either the left or the right ground sensor will leave the line first (Fig. 3.4):

- If the robot moves off the line to the *left*, the *left* sensor will not detect the line while the *right* sensor is still detecting it; the robot must turn to the *right*.
- If the robot moves off the line to the *right*, the *right* sensor will not detect the line while the *left* sensor is still detecting it; the robot must turn to the *left*.

[2]The figure shows a top view although the ground sensors are on the bottom of the robot.

[3]Our presentation of the line following algorithms assumes that the floor is light-colored. If your floor is dark-colored, a white line should be used.

For now, we specify that the robot stops whenever neither sensor detects the line.

Algorithm 3.4 formalizes the above informal description.

**Algorithm 3.4: Line following with two sensors**

```
1: when both sensors detect black
2:     left-motor-power ← 100
3:     right-motor-power ← 100
4:
5: when neither sensor detects black
6:     left-motor-power ← 0
7:     right-motor-power ← 0
8:
9: when only the left sensor detects black
10:    left-motor-power ← 0
11:    right-motor-power ← 50
12:
13: when only the right sensor detects black
14:    left-motor-power ← 50
15:    right-motor-power ← 0
```

**Activity 3.10: Line following with two sensors**

- Implement Algorithm 3.4.
- Use black electrician's tape or gaffer tape to create a line on the floor. (Gaffer tape is used on stage and film sets to bind cables or fix them to the floor. Gaffer tape is usually less reflective than electrician's tape and thus a better choice for implementing line-following algorithms.)
- The line should have angles or curves which will cause the robot to start running off the line. Run the program and check that the robot can successfully follow the line with its angles and curves.
- Experiment with motor powers for turning back onto the line. If the turn is too gentle, the other sensor might also run off the line before the robot turns back. If the turn is too sharp, it might cause the robot to run off the other end of the line. In any case, sharp turns can be dangerous to the robot and cause whatever it is carrying to fall off.
- The forward speed of the robot on the line is also important. If it is too fast, the robot can run off the line before the turning actions can affect its direction. If the speed is too slow, no one will buy your robot to use in a warehouse. How fast can your robot follow the line without running off it?

**Activity 3.11: Different line configurations**

- What is the sharpest turn that your robot can follow? Can it follow a line that has a 90° turn?
- Experiment with different configurations of the line: gentle turns, sharp turns and zigzag lines.
- Experiment with the width of the line. What happens if the line is much wider or narrower than the distance between the sensors?

**Activity 3.12: Regaining the line after losing it**

- Modify the algorithm so that the robot returns to the line after it loses it, that is, when both sensors no longer detect black.
- Algorithm 1: Before both sensors no longer detect the line, one of them will be the first that no longer detects the line. Use a variable to remember the sensor that first lost the line; when the line is lost, turn in the opposite direction of the value of this variable.
- Algorithm 2: The previous algorithm might not work if the robot is moving too fast and runs off the line before it detects that only one sensor lost the line. Instead, if both sensors no longer detect black, cause the robot the search for the line for a short distance, first in one direction and then in the other direction.

**Activity 3.13: Sensor configuration**

- Discuss what effect the following modifications to the robot would have on its ability to follow a line:
  - Ground sensing events occur more often or less often.
  - The sensors are further apart or closer together.
  - There are more than two ground sensors on the bottom of the robot.
  - The sensors are on the back of the robot or the robot moves backwards.
- Experiment with these changes if they can be done on your robot.

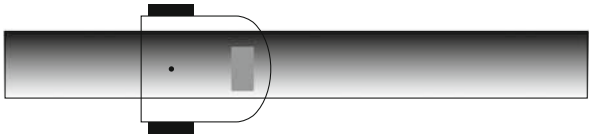


**Fig. 3.5** A line with a grayscale gradient

### *3.4.2 Line Following with only One Ground Sensor*

A robot can follow a line with only one ground sensor if the reflectivity of the line varies across its width. Figure 3.5 shows a grayscale line whose shade varies continuously from fully black to fully white across its width. The ground sensor will return values from 0 to 100 depending on which part of the line it is over.

When the robot is directly over the center of the line, the sensor will return the value 50 that is midway between black and white. Of course, we don't expect the value to be exactly 50, so there is no reason for the robot to turn left or right unless the value approaches 0 or 100. We define two thresholds:

- black_threshold: below this value, the robot is leaving the left side of the line.
- white_threshold: above this value, the robot is leaving the right side of the line.

Algorithm 3.5 changes the motor power settings when the value returned by the sensor crosses the thresholds.

**Algorithm 3.5: Line following with one sensor**

```
integer black-threshold ← 20
integer white-threshold ← 80
when black-threshold ≤ sensor value ≤ white-threshold
    left-motor-power ← 100
    right-motor-power ← 100

when sensor value > white-threshold
    left-motor-power ← −50
    right-motor-power ← 50

when black-threshold < sensor value
    left-motor-power ← 50
    right-motor-power ← −50
```

**Activity 3.14: Line following with one sensor**

- Implement Algorithm 3.5.
- Experiment with the thresholds until the robot can follow the line.
- Modify the algorithm so that the robot detects when it has run completely off the line. Hint: The only problem is when the robot runs off the black side of the line, because there the algorithm doesn't distinguish between that case and running off the white side of the line. One solution is to use a variable to remember the previous value of the sensor, so that the two cases can be distinguished.

**Activity 3.15: Line following with proportional correction**

- Since the sensor values are proportional to the grayscale of the line, the approximate distance of the sensor from the center of the line can be computed. Modify the algorithm to compute this distance.
- Modify the algorithm so that the motor setting is proportional to this distance.
- Run experiments for various constants of proportion and explain the results.

### *3.4.3 Line Following Without a Gradient*

The receiver of a proximity sensor has an *aperture*, an opening through which the light is collected. Apertures are often wide in order to allow more light to fall on the sensor so that it will be responsive to low levels of light. Cameras have *f-stops*: the lower the value of the f-stop, the wider the aperture, so pictures can be taken in relatively dark environments. If the aperture of the ground proximity sensor is relatively wide, a gradient on the line is not needed. A single sensor can follow one *edge* of a line (Fig. 3.6). The figure assumes that the robot is supposed to follow the right edge of the line.

- Left image: If the sensor of the robot is over the line, little light will be sensed, so the robot is too far to the left of the right edge that it should be following. The robot must turn right.
- Center image: If the sensor is off the line, a lot of light will be sensed, so the robot is too far to the right of the right edge that it should be following. The robot must turn left.
- Right image: If the sensor is over the right edge of the line (as it should be), the amount of light sensed will be midway between the two extreme values. The robot can continue to move forwards.

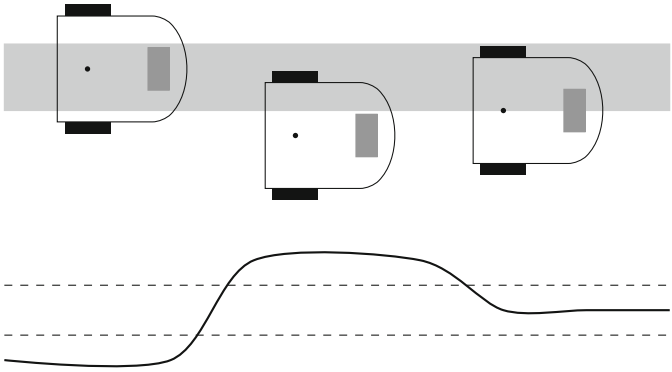


**Fig. 3.6** Line following with a single sensor and without a gradient. *Above*: robot moving over the line, *below*: plot of sensor value vs. distance

At the bottom of the figure is a plot of the values returned by the sensors. The dashed lines are the thresholds between the three states: on the line, off the line, on the edge.

**Activity 3.16: Line following without a gradient**

- Write the algorithm in detail.
- Experiment to determine the thresholds.
- Implement the algorithm.
- Compare the performance of the algorithm with the line following algorithms using two sensors and one sensor with a gradient. Which algorithm is more *robust*, that is, which algorithm performs better at higher speeds and is able to follow sharper turns of the line?

**Activity 3.17: Line following in practice**

- Our presentation of line following is based on the use of sensors that detect a line painted or taped to the floor. What other technologies could be used to portray the line and to detect it?
- The algorithms cause the robot to follow the line but for a warehouse robot there needs to be a way of identifying when the robot has reached the required bin and a way of locating a specific item in the bin. How can these tasks be implemented?
- Suppose that the warehouse adds new bins. What changes need to be made to the robot? How could the algorithms be designed to facilitate change?

## 3.5 Braitenberg's Presentation of the Vehicles

Valentino Braitenberg's vehicles were constructed as thought experiments not intended for implementation with electronic components or in software. The vehicles had sensors directly connected to the motors as in the nervous system of living creatures. Some vehicles were designed with memory similar to a brain.

Figures 3.7a–b show robots that demonstrate Braitenberg's presentation. They have light sensors (the semicircles at the front of the robots) that are connected directly to the motors of the wheels. The more light detected, the faster each wheel will turn, as indicated by the + signs on the connections. If a strong light source is directly ahead of the robot, both sensors will return the same value and the robot will move rapidly forwards. Suppose now that the light source is off to the left. For the robot in Fig. 3.7a, the left wheel will turn rapidly and the right wheel will turn slowly. This results in the robot turning sharply right away from the light source. Braitenberg called this vehicle *coward*. For the robot in Fig. 3.7b, the right wheel turns rapidly and the left wheel turns slowly so the robot turns towards the light source, eventually crashing into it. This behavior is *aggressive*.

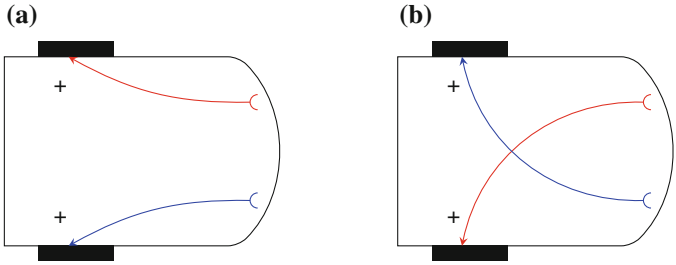



**Fig. 3.7** **a** Coward vehicle **b** Aggressive vehicle

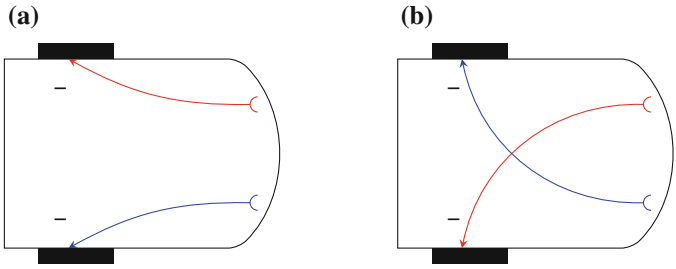



**Fig. 3.8** **a** Loves vehicle **b** Explorer vehicle

**Activity 3.18: Braitenberg's presentation of the vehicles**

- Implement the *coward* and *aggressive* vehicles.
- Use proximity sensors in place of Braitenberg's light sensors and the detection or non-detection of an object in place of stronger or weaker light sources.
- The robots in Fig. 3.8a–b are the same as the robots in Fig. 3.7a–b, respectively, except that the values of the sensors are negated (the − signs on the connections): the more light detected, the slower the wheel will turn. Assume that there is a fixed bias applied to the motors so that the wheels turn forwards when no light source is detected.
- Implement the robot in Fig. 3.8a. Why is it called *loves*?
- Implement the robot in Fig. 3.8b. Why is it called *explorer*?

## 3.6 Summary

A robot exhibits reactive behavior when its actions depend only upon the current values returned by its sensors. This chapter presented two families of reactive behavior. Braitenberg vehicles implement reactive behavior by changing the setting of the motors in response to the events of proximity sensors detecting or not detecting an object at some position relative to the robot. The vehicles demonstrate that complex behavior can result from relatively simple reactive algorithms.

Line following is a fundamental task in robotics. Because of uncertainties of the robot's motion and its environment, robots use landmarks such as lines to ensure that they move to their intended destination. Line following is a reactive behavior because the robot modifies its behavior in response to values returned by the ground sensors. We have given three configurations for the robot and the line, and developed algorithms for each case. The performance of the algorithms depends on the sensor thresholds and the motor speeds, which must be determined by experimentation in order to ensure that the robot moves rapidly while remaining robust to changes in the environment.

## 3.7 Further Reading

Braitenberg's book [1] is interesting because he writes from the perspective of a neuroscientist. The book describes vehicles using imagined technology, but they are nevertheless thought-provoking. The Braitenberg vehicles described here are adapted from the hardware implementations described in [2]. An implementation of the Braitenberg vehicles in Scratch by the first author can be found at: https://scratch.mit.edu/studios/1452106.

## References

1. Braitenberg, V.: Vehicles: Experiments in Synthetic Psychology. MIT Press, Cambridge (1984)
2. Hogg, D.W., Martin, F., Resnick, M.: Braitenberg creatures. Technical report E&L Memo No. 13, MIT Media Lab (1991). http://cosmo.nyu.edu/hogg/lego/braitenberg_vehicles.pdf



# Chapter 4
# Finite State Machines

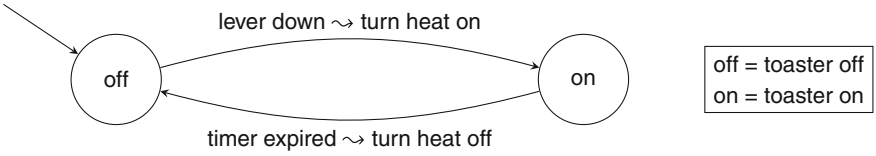
The Braitenberg vehicles and the line following algorithms (Chap. 3) demonstrate reactive behavior, where the action of the robot depends on the *current* values returned by robot's sensors, not on events that happened previously. In Sect. 4.1 we present the concept of state and finite state machines (FSM). Sections 4.2 and 4.3 show how certain Braitenberg vehicles can be implemented with FSMs. Section 4.4 discusses the implementation of FSMs using state variables.

## 4.1 State Machines

The concept of state is very familiar. Consider a toaster: initially, the toaster is in the off state; when you push the lever down, a transition is made to the on state and the heating elements are turned on; finally, when the timer expires, a transition is made back to the off state and the heating elements are turned off.

A *finite state machine (FSM)*[1] consists of a set of *states* $s_i$ and a set of *transitions* between pairs of states $s_i$, $s_j$. A transition is labeled *condition/action*: a condition that causes the transition to be taken and an action that is performed when the transition is taken.

FSMs can be displayed in *state diagrams*:




A state is denoted by a circle labeled with the name of the state. States are given short names to save space and their full names are given in a box next to the state

[1] Finite state machines are also called finite automata.



M. Ben-Ari and F. Mondada, *Elements of Robotics*,
https://doi.org/10.1007/978-3-319-62533-1_4

diagram. The incoming arrow denotes the initial state. A transition is shown as an arrow from the *source* state to the *target* state. The arrow is labeled with the condition and the action of the transition. The action is not continuing; for example, the action turn left means set the motors so that the robot turns to the left, but the transition to the next state is taken without waiting for the robot to reach a specific position.

## 4.2 Reactive Behavior with State

Here is the specification of a Braitenberg vehicle whose behavior is non-reactive:

> **Specification (Persistent)**: The robot moves forwards until it detects an object. It then moves backwards for one second and reverses to move forwards again.

Figure 4.1 shows the state diagram for this behavior.

Initially, when the system is turned on, the motors are set to move forwards. (The condition true is always true so this is done unconditionally.) In state fwd, if an object is detected, the transition to state back is taken, the robot moves backwards and the timer is set. When one second has passed the timer expires; the transition to state fwd is taken and the robot moves forwards. If an object is detected when the robot is in the back state, *no* action is performed, because there is no transition labeled with this condition. This shows that the behavior is not reactive, that is, it depends on the current state of the robot as well as on the event that occurs.

**Activity 4.1: Consistent**

- Draw the state diagram for the Consistent Braitenberg vehicle.

> **Specification (Consistent):** The robot cycles through four states, changing state once every second: moving forwards, turning left, turning right, moving backwards.

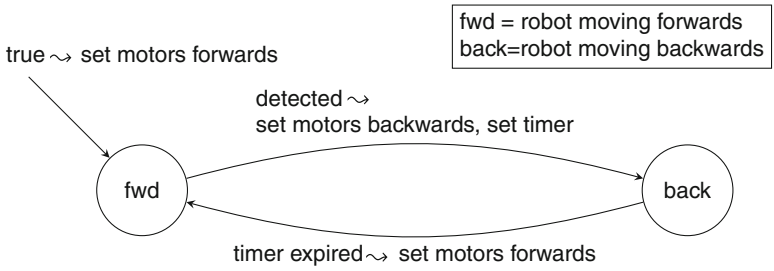



**Fig. 4.1** FSM for the persistent Braitenberg vehicle

## 4.3 Search and Approach

This section presents a more complex example of a robotic behavior that uses states.

**Specification (Search and approach)**: The robot searches left and right (±45°). When it detects an object, the robot approaches the object and stops when it is near the object.

There are two configurations of the sensors of a robot that enable it to search left and right as shown in Fig. 2.8a, b. If the sensors are fixed to the body of the robot, the robot itself has to turn left and right; otherwise, if the sensor is mounted so that it can rotate, the robot can remain still and the sensor is rotated. We assume that the robot has fixed sensors.

Figure 4.2 shows the state diagram for search and approach. The robot is initially searching left. There are two new concepts that are illustrated in the diagram. There is a *final state* labeled found and denoted by a double circle in the diagram. A finite state machine is finite in the sense that it contains a finite number of states and transitions. However, its behavior can be finite or infinite. The FSM in Fig. 4.2 demonstrates finite behavior because the robot will stop when it has found an object and approached it. This FSM also has infinite behavior: if an object is never found, the robot continues indefinitely to search left and right.

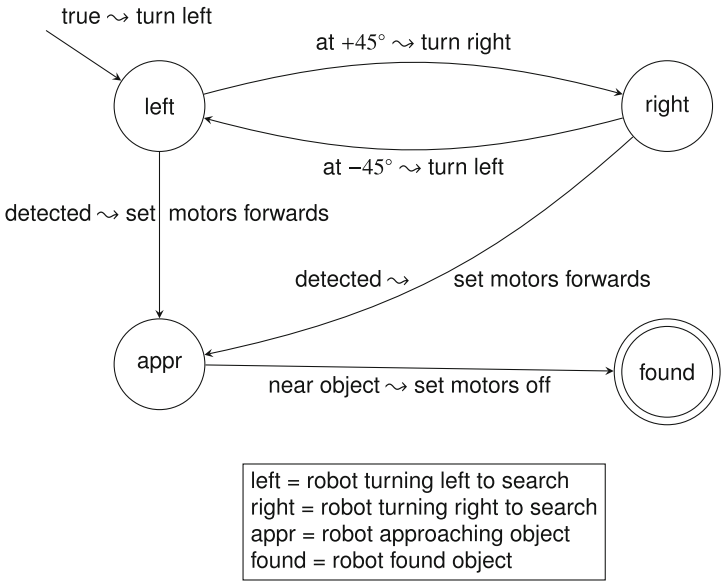



**Fig. 4.2** State diagram for search and approach

The Persistent Braitenberg vehicle (Fig. 4.1) has infinite behavior because it continues to move without stopping. A toaster also demonstrates infinite behavior because you can keep toasting slices of bread forever (until you unplug the toaster or run out of bread).

The second new concept is *nondeterminism*. States left and right each have *two* outgoing transitions, one for reaching the edge of the sector being searched and one for detecting an object. The meaning of nondeterminism is that any of the outgoing transitions may be taken. There are three possibilities:

- The object is detected but the search is not at an edge of the sector; in this case, the transition to appr is taken.
- The search is at an edge of the sector but an object is not detected; in this case, the transition from left to right or from right to left is taken.
- The search is at an edge of the sector exactly when an object is detected; in this case, an arbitrary transition is taken. That is, the robot might approach the object or it might change the direction of the search.

The nondeterministic behavior of the third case might cause the robot to fail to approach the object when it is first detected, if this event occurs at the same time as the event of reaching $\pm 45°$. However, after a short period the conditions will be checked again and it is likely that only one of the events will occur.

## 4.4 Implementation of Finite State Machines

To implement behaviors with states, variables must be used. The Persistent vehicle (Sect. 4.2) needs a timer to cause an event after a period of time has expired. As explained in Sect. 1.6.4, a timer is a variable that is set to the desired period of time. The variable is decremented by the operating system and when it reaches zero an event occurs.

Algorithm 4.1 describes how the FSM of Fig. 4.1 is implemented. The variable current contains the current state of the robot; at the conclusion of the processing of an event handler, the value of the variable is set to the target state of the transition. The values of current are named fwd and back for clarity, although in a computer they would be represented by numerical values.

**Activity 4.2: Persistent**

- Implement the Persistent behavior.

**Algorithm 4.1: Persistent**

```
integer timer                                          // In milliseconds
states current ← fwd
 1: left-motor-power ← 100
 2: right-motor-power ← 100
 3: loop
 4:     when current = fwd and object detected in front
 5:         left-motor-power ← −100
 6:         right-motor-power ← −100
 7:         timer ← 1000
 8:         current ← back
 9:
10:     when current = back and timer = 0
11:         left-motor-power ← 100
12:         right-motor-power ← 100
13:         current ← fwd
```

**Activity 4.3: Paranoid (alternates direction)**

- Draw the state diagram for the Paranoid (alternates direction) Braitenberg vehicle:

  **Specification (Paranoid (alternates direction)):**

  - When an object is detected in front of the robot, the robot moves forwards.
  - When an object is detected to the right of the robot, the robot turns right.
  - When an object is detected to the left of the robot, the robot turns left.
  - If the robot is turning (even if it no longer detects an object), it alternates the direction of its turn every second.
  - When no object is detected and the robot is not turning, the robot stops.

- Implement this specification. In addition to a variable that stores the current state of the robot, use a variable with values left and right to store the direction that the robot turns. Set a timer with a one-second period. In the event handler for the timer, change the value of the direction variable to the opposite value and reset the timer.

Algorithm 4.2 is an outline of the implementation of the state diagram in Fig. 4.2.

**Algorithm 4.2: Search and approach**

```
        states current ← left
 1: left-motor-power ← 50                  // Turn left
 2: right-motor-power ← 150
 3: loop
 4:     when object detected
 5:         if current = left
 6:             left-motor-power ← 100     // Go forwards
 7:             right-motor-power ← 100
 8:             current ← appr
 9:         else if current = right
10:             ...
11:     when at +45°
12:         if current = left
13:             left-motor-power ← 150     // Turn right
14:             right-motor-power ← 50
15:             current ← right
16:     when at −45°
17:         ...
18:     when object is very near
19:         if current = appr
20:             ...
```

**Activity 4.4: Search and approach**

- Fill in the missing lines in Algorithm 4.2.
- Implement Algorithm 4.2.

## 4.5 Summary

Most robotics algorithms require that the robot maintain an internal representation of its current state. The conditions that the robot uses to decide when to change state and actions taken when changing from one state to another are described by finite state machines. State variables are used to implement state machines in programs.

## 4.6 Further Reading

The classic textbook on automata was published in 1979 by John Hopcroft and Jeffrey D. Ullman; its latest edition is [2]. For a detailed explanation of *arbitrary* choice among alternatives in nondeterminism (Sect. 4.3), see [1, Sect. 2.4].

## References

1. Ben-Ari, M.: Principles of Concurrent and Distributed Programming, 2nd edn. Addison-Wesley, Boston (2006)
2. Hopcroft, J., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, 3rd edn. Pearson, Boston (2007)



# Chapter 5
# Robotic Motion and Odometry

The robotics algorithms in the previous chapters react to data from their sensors by changing the speed and direction of their motion, but the changes were not quantitative. We didn't require the robots to move twice as fast or to turn 90° to the right. Robots in the real world have to move to specific locations and may have engineering constraints on how fast or slow they can move or turn. This chapter presents the mathematics of robotic motion.

Sections 5.1 and 5.2 review the concepts of distance, time, velocity and acceleration that should be familiar from introductory physics. The physics of motion is usually taught using calculus, but a computer cannot deal with continuous functions; instead, discrete approximations must be used as described in Sect. 5.3.

Sections 5.4–5.6 present *odometry*, the fundamental algorithm for computing robotic motion. An approximation of the location of a robot can be obtained by repeatedly computing the distance moved and the change direction from the velocity of the wheels in a short period of time. Unfortunately, odometry is subject to serious errors as shown in Sect. 5.7. It is important to understand that errors in direction are much more significant than errors in distance.

In the simplest implementation, the speed of the wheels of a robot is assumed to be proportional to the power applied by the motors. Section 5.8 shows how the accuracy of odometry can be improved by using *wheel encoders*, which measure the actual number of revolutions of the wheels.

Section 5.9 presents an overview of *inertial navigation*, which is a sophisticated form of odometry based upon measuring linear and angular acceleration and then integrating to obtain velocity and position. The sensors for inertial navigation (accelerometers and gyroscopes) were once very expensive, limiting its application to aircraft and rockets, but new technology called *microelectromechanical systems* has made it possible to build robots with inertial navigation.



M. Ben-Ari and F. Mondada, *Elements of Robotics*,
https://doi.org/10.1007/978-3-319-62533-1_5

Cars cannot move up and down unlike helicopters and submarines which have greater freedom of movement. This is expressed in the concept *degrees of freedom (DOF)* which is the subject Sect. 5.10. Section 5.11 discusses the relation between the DOF and number of *actuators* (motors) in a robotics systems.

The number of DOF of a system does not mean that a system such as a vehicle can move freely in all those directions. A car can move to any point in the plane and orient itself in any direction, but it cannot move sideways, so a difficult maneuver is needed during parallel parking. This is due to the difference between the DOF and the *degrees of mobility (DOM)*, a subject explored in Sect. 5.12, along with the concept of *holonomic motion* that relates DOF and DOM.

## 5.1 Distance, Velocity and Time

Suppose that a robot moves with a constant *velocity* of 10 cm/s for a period of *time* of 5 s.[1] The *distance* it moves is 50 cm. In general, if a robot moves at a constant velocity $v$ for a period of time $t$, the distance it moves is $s = vt$. When power is applied to the motors it causes the wheels to rotate, which in turn causes the robot to move at some velocity. However, we cannot specify that a certain power causes a certain velocity:

- No two electrical or mechanical components are ever precisely identical. A motor is composed of magnets and electrical wiring whose interaction causes a mechanical shaft to rotate. Small differences in the properties of the magnet and wire, as well as small differences in the size and weight of the shaft, can cause the shafts of two motors to rotate at slightly different speeds for the same amount of power.
- The environment affects the velocity of a robot. Too little friction (ice) or too much friction (mud) can cause a robot to move slower in comparison with its movement on a dry paved surface.
- External forces can affect the velocity of a robot. It needs more power to sustain a specific velocity when moving uphill and less power when moving downhill, because the force of gravity decreases and increases the velocity. Riding a bicycle at a constant velocity into the wind demands more effort than riding with the wind, and a cross-wind makes the relation between power and velocity even more complicated.

Since $s = vt$ it is sufficient to measure any two of these quantities in order to compute the third. If we measure distance and time, we can compute the velocity as $v = s/t$. Relatively short distances (up to several meters) can be measured accurately (to within 1 cm) using a ruler or a tape measure. The stopwatch application on a smartphone can measure time accurately (hundredths of a second).

[1] Velocity is *speed* in a *direction*. A robot can be moving 10 cm/s forwards or backwards; in both cases, the speed is the same but the velocity is different.

**Activity 5.1: Velocity over a fixed distance**

- Write a program that sets your robot to a constant forward power setting.
- Mark two lines 1 m apart on the floor. Use a stopwatch to measure the time it takes the robot to move between the lines. Compute the velocity of the robot. Run the program ten times and record the velocities. Do the velocities vary?
- Place the robot on the floor and run it for 5 s. Measure the distance that it moves. Compute the velocity. Run the program ten times and record the velocities. Do the velocities vary?
- Which method gives more precise results?
- Repeat this experiment on different surfaces and discuss the results.

Activity 5.1 shows that for a constant power setting the velocity of a robot can vary significantly. To accurately navigate within an environment, a robot needs to sense objects in its environment, such as walls, marks on the floor and objects.

## 5.2 Acceleration as Change in Velocity

Activity 5.1 specified constant power settings and thus the velocity of the robot will be (more or less) constant. What happens when the velocity is varied?

**Activity 5.2: Change of velocity**

- Run the first program from Activity 5.1 varying the distance between the marks: 0.25, 0.5, 1, 1.5, 2 m. For each distance, run the program several times and take the average of the computed velocities. Are the velocities the same for each distance?
- To improve the accuracy of the measurement, place marks on the floor at these distances and use the robot's timer to record the times at which the marks are detected.

In Activity 5.2, you will find that for the longer distances the velocities will be close to each other, but for the shorter distances the velocities will differ considerably. The reason is that the formula $v = s/t$ assumes that the velocity is constant over the entire distance. In reality, a vehicle must *accelerate*—change its velocity—in order to go from standing still to a constant velocity. Similarly, a vehicle must *decelerate* in order to stop.

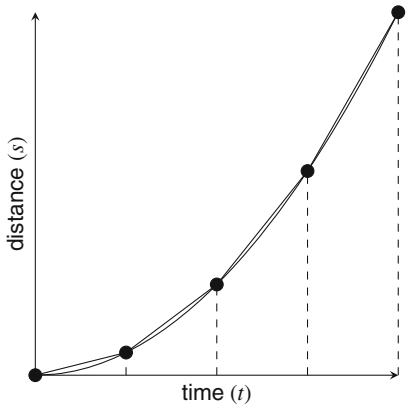
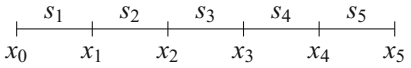



**Fig. 5.1** An accelerating robot: distance increase as the square of time

To get a true picture of the motion of a robot, we need to divide its motion into small segments $s_1, s_2, \ldots$:




and measure the distance and time for each segment individually. Then, we can compute the velocities for each segment. In symbols, if we denote the length of the segment $s_i$ by $\Delta s_i = x_{i+1} - x_i$ and the time it takes the robot to cross segment $s_i$ by $\Delta t_i = t_{i+1} - t_i$, then $v_i$, the velocity in segment $s_i$ is given by:

$$v_i = \frac{\Delta s_i}{\Delta t_i}\,.$$

Figure 5.1 is a graph of distance versus time for an accelerating robot. The time axis has been divided into segments and the slopes $\frac{\Delta s_i}{\Delta t_i}$ show the average velocity in each segment which increases with time.

*Acceleration* is defined as the change in velocity over a period of time:

$$a_i = \frac{\Delta v_i}{\Delta t_i}\,.$$

When the power setting of the robot is set to a fixed value, the force applied to the robot is constant and we expect that the acceleration remains constant, increasing the velocity. However, at a certain point the acceleration is reduced to zero, meaning that the velocity no longer increases, because the power applied to the wheels is just sufficient to overcome the friction of the road and the wind resistance.

Let us see what happens if the power setting is increased with time.

**Activity 5.3: Acceleration**

- Write a program that causes the robot to accelerate by increasing the power setting periodically. For example, start the robot at power 20 and increase to 40 after 1 s, then to 60 after 2 s, to 80 after 3 s, and finally to 100 after 4 s.
- Place the robot on the track and run the program.
- Record the distances between each change of the power setting. Compute and plot the velocities in each of these segments.

## 5.3 From Segments to Continuous Motion

As the size of the segments becomes smaller, we obtain the instantaneous velocity of the robot at a single point in time, expressed as a derivative:

$$v(t) = \frac{ds(t)}{dt}\,.$$

Similarly, the instantaneous acceleration of the robot is defined as:

$$a(t) = \frac{dv(t)}{dt}\,.$$

For constant acceleration the velocity can be obtained by integrating the derivative:

$$v(t) = \int a\,dt = a\int dt = at\,,$$

and then the distance can be obtained by integrating again:

$$s(t) = \int v(t)dt = \int a\,t\,dt = \frac{at^2}{2}\,.$$

**Example** An average car accelerates from 0 to 100 km/h in about 10 s. First, we convert units from km/h to m/s:

$$v_{max} = 100\,\text{km/h} = \frac{100 \cdot 1000}{60 \cdot 60}\,\text{m/s} = 27.8\,\text{m/s}\,.$$

Assuming constant acceleration, $v_{max} = 27.8 = at = 10a$, so the acceleration is 2.78 m/s$^2$ (read, 2.78 meters per second per second, that is, every second the speed increases by 2.78 meters per second). The distance the car moves in 10 s is:

$$s(10) = \frac{at^2}{2} = \frac{2.78 \cdot 10^2}{2} = 139\,\text{m}\,.$$

**Activity 5.4: Computing distance when accelerating**

- For various vehicles (racing cars, motorcycles) look up the time required to accelerate from 0 to 100 km/h. Compute the distance moved.
- Assume that the acceleration of a vehicle increases linearly, that is, $a = kt$ for a constant $k$. What are $v(t)$ and $s(t)$?
- For several values of $k$ and $t$, compute the final velocities and distances.

**Activity 5.5: Measuring motion at constant acceleration**

- Write a program that applies the maximum power setting to a robot.
- Place the robot on a surface and run the program.
- When the robot seems to have reached full speed record the time from the start of the run.
- Compare the measured distance to $s = at^2/2$ (Fig. 5.2b).
- Run again and measure the distances at fixed intervals of time. Compute the speeds from the distances divided by the time and compare to $v = at$ (Fig. 5.2a).
- In some robots you can set a target speed and read the actual speed. If your robot can do this, compare the measured speeds with the computed speeds.

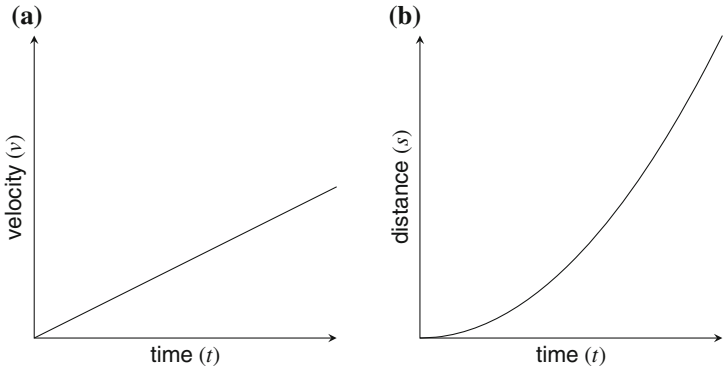



**Fig. 5.2** **a** Velocity for constant acceleration. **b** Distance for constant acceleration

## 5.4 Navigation by Odometry

Suppose that you are in a car and your navigation system issues the following instruction: "In 700 m turn right." Now your task is very simple: Make observations of your car's odometer which measures how far you have traveled. When its value approaches 700 m beyond its initial reading, look for a street on the right. An odometer in a car measures speed and time, and multiplies the two values to compute the distance traveled.

*Odometry*—the measurement of distance—is a fundamental method used by robots for navigation. Measuring time is easy using the internal clock of the embedded computer. Measuring speed is more difficult: in some educational robots wheel encoders are used to count the rotations of the wheels (Sect. 5.8), while in others speed is estimated from properties of the motors. From the distance moved $s = vt$, the new position of the robot can be computed. In one dimension, the computation is trivial, but it becomes a bit more complex when the motion involves turns. This section presents the computation of distance by odometry, first for a robot moving linearly and then for a robot making a turn.

Section 5.7 shows how errors in heading are more serious than errors in distance.

A disadvantage of odometry (with or without wheel encoders) is that the measurements are indirect, relating the power of the motors or the motion of the wheels to changes in the robot's position. This can be error-prone since the relation between motor speed and wheel rotation can be very nonlinear and vary with time. Furthermore, wheels can slip and skid so there may be errors in relating the motion of the wheels to the motion of the robot. Improved estimates of position can be obtained by using an inertial navigation system, which directly measures acceleration and angular velocity that can be used to determine the robot's position (Sect. 5.9).

Odometry is a form of *localization*: the robot must determine its position in the environment. In odometry we determine position by measuring the change from the robot's known initial position, while localization (Chap. 8) refers to the determination of the position of a robot relative to the known positions of other objects such as landmarks or beacons.

## 5.5 Linear Odometry

Before studying the mathematics of odometry you should try the following Activity:

**Activity 5.6: Distance from speed and time**

- Run the robot at a constant power setting for a specific period of time and measure the distance moved.

- Repeat the measurement several times. Is the distance constant? If not, how much does it vary as a percentage of the distance?
- Repeat the measurement several times for different power settings. Is the distance measured linear in the power setting? Does the *variation* in the distance measurement on multiple runs depend on the power setting?
- Repeat the measurement for a fixed power setting but for different periods of time and analyze the results.

When a relation between motor power and velocity $v$ has been determined, the robot can compute the distance moved by $s = vt$. If it starts at position $(0, 0)$ and moves straight along the $x$-axis, then after $t$ seconds its new position is $(vt, 0)$.

This activity should demonstrate that it is possible to measure distance by odometry with reasonable precision and accuracy. A self-driving car can use odometry to determine its position so that it doesn't have to analyze its sensor data continuously to check if the required street has been reached. Given the uncertainties of motion and of the road, the car should not depend only on odometry to decide when to turn, but the error will not be large and the sensor data can be analyzed to detect the turn when odometry indicates that the car is in the vicinity of the intersection.

Activity 5.6 asked you to measure the distance moved in one dimension. Three items of information need to be computed if the motion is in two dimensions: the robot's *position* $(x, y)$ relative to a fixed origin and its *heading* $\theta$, the direction in which the robot is pointing (Fig. 5.3). The triple $(x, y, \theta)$ is called the *pose* of the robot. If the robot starts at the origin $(0, 0)$ and moves in a straight line at angle $\theta$ with velocity $v$ for time $t$, the distance moved is $s = vt$. Its new position $(x, y)$ is:

$$x = vt \cos\theta$$
$$y = vt \sin\theta\,.$$

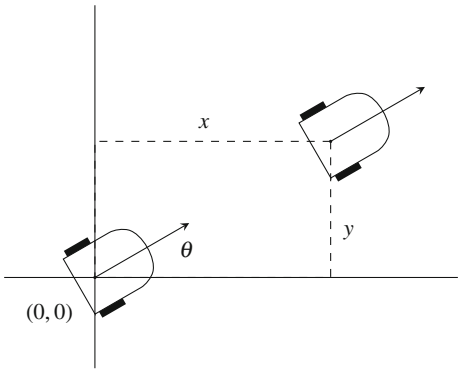



**Fig. 5.3** Position and heading

## 5.6 Odometry with Turns

Suppose that the robot turns slightly left because the right wheel moves faster than the left wheel (Fig. 5.4). In the figure, the robot is facing towards the top of the page; the blue dot is the left wheel, the red dot is the right wheel, and the black dot is the center of the robot which is halfway between the wheels. The *baseline* $b$ is the distance between the wheels, and $d_l, d_r, d_c$ represent the distances moved by the two wheels and the center when the robot turns. We want to compute the new position and heading of the robot.

We can measure $d_l$ and $d_r$, the distances moved by the two wheels using the method described in Activity 5.6: relating motor power to rotational speed and then multiplying by time. Alternatively, we can use the number of rotations counted by the wheel encoders. If the radius of a wheel is $R$ and the rotational speeds of the left and right wheels are $\omega_l, \omega_r$ revolutions per second, respectively, then after $t$ seconds the wheel has moved:

$$d_i = 2\pi R\omega_i t, \quad i = l, r\,. \tag{5.1}$$

The task is to determine the new pose of the robot after the wheels have moved these distances.

Figure 5.4 shows the robot initially at pose $(x, y, \phi)$, where the robot is facing north ($\phi = \pi/2$). After turning $\theta$ radians, what is the new pose $(x', y', \phi')$? Clearly, the heading of the robot is now $\phi' = \phi + \theta$, but we also have to compute $x'$, $y'$.

The length of an arc of angle $\theta$ radians is given by its fraction of the circumference of the circle: $2\pi r\,(\theta/2\pi) = \theta r$. For small angles, the distances $d_l, d_c, d_r$ are approximately equal to the length of the corresponding arcs, so we have:

$$\theta = d_l/r_l = d_c/r_c = d_r/r_r\,, \tag{5.2}$$

where $r_l, r_r, r_c$ are the distances from $P$, the origin of the turn.

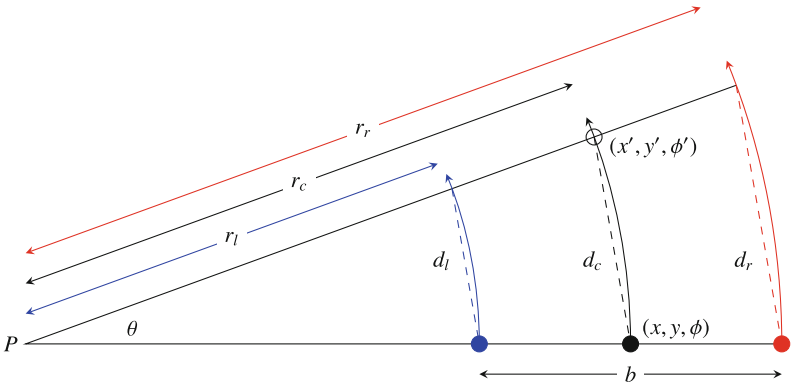



**Fig. 5.4** Geometry of a left turn by a robot with two wheels

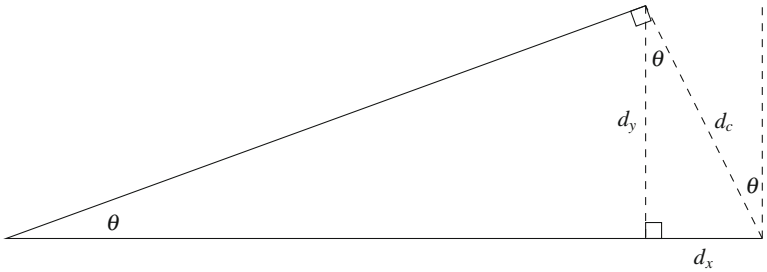



**Fig. 5.5** Change in heading

The distances $d_l$ and $d_r$ are obtained from the rotations of the wheels (Eq. 5.1) and the baseline $b$ is a fixed physical measurement of the robot. From Eq. 5.2, the angle $\theta$ can be computed:

$$\begin{aligned} \theta r_r &= d_r \\ \theta r_l &= d_l \\ \theta r_r - \theta r_l &= d_r - d_l \\ \theta &= (d_r - d_l)/(r_r - r_l) \\ \theta &= (d_r - d_l)/b\,. \end{aligned}$$

The center is halfway between the wheels $r_c = (r_l + r_r)/2$, so again by Eq. 5.2:

$$\begin{aligned} d_c &= \theta r_c \\ &= \theta \left( \frac{r_l + r_r}{2} \right) \\ &= \frac{\theta}{2} \left( \frac{d_l}{\theta} + \frac{d_r}{\theta} \right) \\ &= \frac{d_l + d_r}{2}\,. \end{aligned}$$

If the distance moved is small, the line labeled $d_c$ is approximately perpendicular to the radius through the final position of the robot. By similar triangles, we see that $\theta$ is the change in the heading of the robot (Fig. 5.5). By trigonometry[2]:

$$\begin{aligned} dx &= -d_c \sin\theta \\ dy &= d_c \cos\theta\,, \end{aligned}$$

[2] You were probably expecting cos for $dx$ and sin for $dy$. That would be the case if the robot were facing along the $x$ axis. However, the initial pose is $\phi = \pi/2$ and we have $\sin(\theta + \pi/2) = \cos\theta$ and $\cos(\theta + \pi/2) = -\sin\theta$.

so the pose of the robot after the turn is:

$$(x', y', \phi') = (-d_c \sin\theta, d_c \cos\theta, \phi + \theta) .$$

The formulas show how to compute the changes $dx$, $dy$ and $\theta$ when the robot moves a short distance. To compute odometry over longer distances, this computation must be done frequently. There are two reasons why the intervals between the computations must be short: (a) the assumption of constant speed holds only for short distances, and (b) the trigonometric calculation is simplified by assuming that the distance moved is short.

**Activity 5.7: Odometry in two dimensions**

- Write a program that causes the robot to make a gentle left turn for a specific period of time.
- Compute the pose $(-d_c \sin\theta, d_c \cos\theta, \theta)$ and compare the result with the values measured using a ruler and a protractor. Run the program several times and see if the measurements are consistent.
- Run the program for different periods of time. How does this affect the accuracy and precision of the odometry computation?

## 5.7 Errors in Odometry

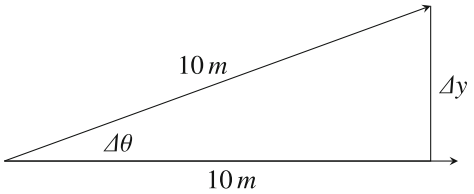
We have already noted that odometry is not accurate because inconsistent measurements and irregularities in the surface can cause errors. In this section we show that even small changes in the direction of the robot's movement can cause errors that are much larger than those caused by changes in its linear motion.

To simplify the presentation, let us assume that a robot is to move 10 m from the origin of a coordinate system along the $x$-axis and then check its surroundings for a specific object. What is the effect of an error of *up to* $p\%$? If the error is in the measurement of $x$, the distance moved, then $\Delta x$, the error in $x$ is:

$$\Delta x \leq \pm 10 \cdot \frac{p}{100} = \pm \frac{p}{10} \text{ m} ,$$

where the value is negative or positive because the robot could move up to $p\%$ before or after the intended distance.

Suppose now that there is an error $p\%$ in the *heading* of the robot, and, for simplicity, assume that there is no error in the distance moved. The geometry is:




The robot intended to move 10 m along the $x$-axis, but instead it moved slightly to the left at an angle of $\Delta\theta$. Let us compute the left-right deviation $\Delta y$. By trigonometry, $\Delta y = 10 \sin \Delta\theta$. An error of $p\%$ in heading is:

$$\Delta\theta = 360 \cdot \frac{p}{100} = (3.6p)^\circ ,$$

so the left-right deviation is:

$$\Delta y \leq \pm 10 \sin(3.6p) .$$

The following tables compare the difference between a linear error of $p\%$ (left) and an error in heading of $p\%$ (right):

| $p\%$ | $\Delta x$ (m) |
|---|---|
| 1 | 0.1 |
| 2 | 0.2 |
| 5 | 0.5 |
| 10 | 1.00 |

| $p\%$ | $\Delta\theta$ (°) | $\sin \Delta\theta$ | $\Delta y$ (m) |
|---|---|---|---|
| 1 | 3.6 | 0.063 | 0.63 |
| 2 | 7.2 | 0.125 | 1.25 |
| 5 | 18.0 | 0.309 | 3.09 |
| 10 | 36.0 | 0.588 | 5.88 |

For a very small error like 2%, the distance error after moving 10 m is just 0.2 m, which should put the robot in the vicinity of the object it is searching for, but a heading error of the same percentage places the robot 1.25 m away from the object. For a more significant error like 5% or 10%, the distance error (50 or 100 cm) is still possibly manageable, but the heading error places the robot 3.09 or 5.88 m away, which is not even in the vicinity of the object.

The accumulation of odometry errors as the distance moved gets longer is displayed in Fig. 5.6. The initial position of the robot is denoted by the dot at the origin.

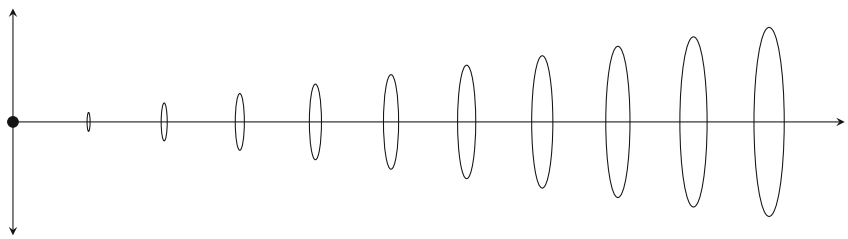


**Fig. 5.6** Odometry errors

Assuming an error of at most $\pm 4\%$ in both the linear direction and the heading, the possible positions of the robot after moving $d = 1, 2, \ldots, 10$ m are displayed as ellipses. The minor radii of the error ellipses result from the linear errors:

$$0.04s = 0.04, 0.08, \ldots, 0.4 \text{ m},$$

while the major radii of the error ellipses result from the angular errors:

$$d \sin(0.04 \cdot 360^\circ) = d \sin 14.4^\circ \approx 0.25, 0.50, \ldots 2.5 \text{ m}.$$

Clearly, the angular errors are much more significant than the linear errors.

Since error is unavoidable, periodically the pose of the robot as computed by odometry must be compared with an absolute position; this becomes the new initial position for further computation. Methods for determining the absolute position of the robot are presented in Chap. 8.

**Activity 5.8: Odometry errors**

- Write a program to cause the robot to move in a straight line for 2 m. Make sure that the surface is smooth so that it doesn't turn off course and calibrate the motor settings so that the robot moves as straight as possible.
- Vary the motor power of both wheels together so that the robot runs somewhat slower or somewhat faster than before. Plot its position at fixed intervals and see if the error remains linear over the course.
- Vary the motor power of one wheel so that the robot turns slightly to one side. Plot its position at fixed intervals and see if the errors are proportional to the sine of the difference between the original heading and the new heading.

**Activity 5.9: Combined effect of odometry errors**

- Write a program that causes the robot to move in a straight line for 2 m and then turn 360°. What is the error in the robot's position?
- Write a program that causes the robot to turn 360° and then move in a straight line for 2 m. What is the error in the robot's position? Is there a difference between this error and the error of the previous experiment?
- Write a program that causes the robot to move in a straight line for 2 m, turn 180° and then move in a straight line for 2 m. How far is it from its starting position?

**Activity 5.10: Correcting odometry errors**

- Modify the program that you wrote for Activity 5.8 to introduce *jitter*, random variation in the power supplied to the motor. Check that the distance that the robot moves in a fixed time is not constant, but changes slightly from run to run.
- Mark a goal line on the floor and compute the time it should take the robot to reach the goal.
- When the robot has moved for that period of time, see if it can find the goal by moving forwards and backwards in small steps until it detects the goal.

## 5.8 Wheel Encoders

Odometry in a wheeled vehicle like a car can be improved by measuring the rotation of the wheels instead of mapping motor power into velocity. The circumference of a wheel is $2\pi r$, where $r$ is the radius of the wheel in cm, so if $n$ rotations are counted, we know that the robot has moved $2\pi nr$ cm. Wheel encoders can be built that measure fractions of a revolution. If a signal is generated 8 times per revolution, the distance moved is $2\pi nr/8$ cm, which $n$ is now the number of signals counted by the computer.

There are many different ways of implementing wheel encoders. A popular design is to use a light source such as a light-emitting diode (LED), a light sensor and an encoding disk that is attached to the axis of the wheel (Fig. 5.7a). The disk is perforated with holes (Fig. 5.7b) so that whenever the hole is opposite the light source, the sensor generates a signal.

The support for wheel encoders in educational robots varies:

- If a robot does not have wheel encoders it must be calibrated;
- The robot may have wheel encoders that are used internally;
- Some robots like the LEGO® Mindstorms enable the user to read the encoders.

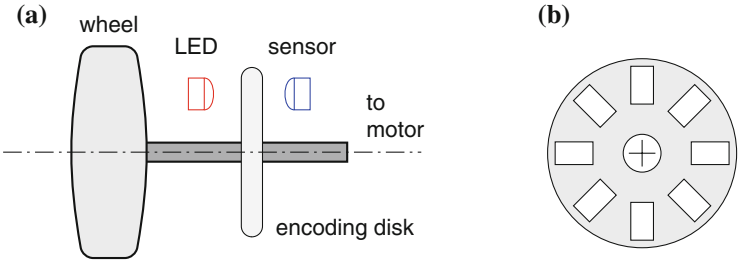



**Fig. 5.7** **a** Optical wheel encoder. **b** Encoding disk

The following activity proposes an experiment to measure the distance moved by counting revolutions of a wheel. It can be carried out even if your robot does not have wheel encoders or they are not accessible.

**Activity 5.11: Wheel encoding**

- Make a mark at the top of a wheel of the robot using chalk or by attaching a narrow piece of colored tape. Write a program that causes the robot to move straight ahead for a fixed period of time. Run the program and take a video of the side of the robot using the camera on your smartphone.
- View the video and determine the number of revolutions by counting the number of times the mark is at the top of the wheel.
- Measure the radius of the wheel and compute the distance moved. How close is the result to the actual distance measured on the floor?
- Repeat the measurement using $n = 2$ and then $n = 4$ equally spaced marks on the wheel. Determine the number of revolutions by counting the number of times that a mark is at the top of the wheel and divide by $n$. Compute the distance.

## 5.9 Inertial Navigation Systems

An *inertial navigation system (INS)* directly measures linear acceleration and angular velocity and uses them to calculate the pose of a vehicle. The term *inertial measurement unit (IMU)* is also used, but we prefer the term INS which refers to the entire system. Integrating acceleration from the initial pose to the current time $\tau$ gives the current velocity:

$$v = \int_0^\tau a(t)\, dt\ .$$

Similarly, integrating angular velocity gives the change in heading:

$$\theta = \int_0^\tau \omega(t)\, dt\ .$$

In an INS, we are not given continuous functions to integrate; instead, the acceleration and angular velocity are sampled and summation replaces integration:

$$v_n = \sum_{i=0}^{n} a_n \Delta t, \quad \theta_n = \sum_{i=0}^{n} \omega_n \Delta t\ .$$

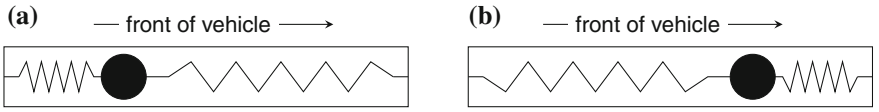



**Fig. 5.8** **a** Forward acceleration. **b** Deceleration (braking)

INSs are subject to errors caused by inaccuracies in the measurement itself as well as by variations caused by environmental factors such as temperature, and by wear and tear of the unit. Inertial measurement is often combined with GPS (Sect. 8.3) to update the position with an absolute location.

INSs for robots are constructed with *microelectromechanical systems (MEMS)*, which use integrated circuit manufacturing techniques that combine mechanical elements with electronics that interface with the robot's computer.

### *5.9.1 Accelerometers*

If you have ever flown on an airplane you have experienced a force pushing you back into your seat as a result of the rapid acceleration of the airplane upon takeoff. Upon landing you are pushed away from your seat. You can also experience this in a car that accelerates rapidly or makes an emergency stop. Acceleration is related to force by Newton's second law $F = ma$, where $m$ is the mass. By measuring the force on an object, we measure the acceleration.

Figures 5.8a, b show how an accelerometer can be built from an object (called a *mass*) connected to a spring. The greater the acceleration, the greater the force exerted by the mass upon the spring, which in turn causes the spring to be compressed. The direction that the mass moves gives the sign of the acceleration: forwards or backwards. The magnitude of the force is measured indirectly by measuring the distance that the mass moves. You can see that the diagrams correspond to our experience: when a car accelerates, you are pushed back into the seat, but when it decelerates (brakes) you continue forward.

### *5.9.2 Gyroscopes*

A *gyroscope* ("gyro") uses the principle of Coriolis force to measure angular velocity. This concept is explained in textbooks on physics and we will not go into it here. There are many types of gyros:

- Classical gyros have spinning mechanical disks which are mounted on gimbals so that the axis of rotation remains fixed in space. These gyros are extremely accurate but are very heavy and consume a lot of power. They are found on high-value vehicles such as aircraft and rockets.
- Ring laser gyros (RLG) have (almost) no moving parts and are preferred over mechanical gyros for most applications. They are based on sending two laser beams in opposite directions around a circular or triangular path. If the gyro is rotating, the path followed by one laser beam will be longer than the path followed by the other beam. The difference is proportional to the angular velocity and can be measured and transferred to the navigation computer.
- Coriolis vibratory gyroscopes (CVG) manufactured using MEMS techniques are found in smartphones and robots. They are inexpensive and extremely robust, although their accuracy is not as good are the gyros previously discussed. We now give an overview of how they work.

Figure 5.9 shows a CVG called a *tuning fork gyroscope*. Two square masses are attached by flexible beams to anchors that are mounted on the base of the component. Drivers force the masses to vibrate left and right. If the component rotates, the masses move upwards and downwards a distance proportional to the angular velocity. The masses and the electrodes form the plates of capacitors whose capacitance increases or decreases as the plates come together or move apart.

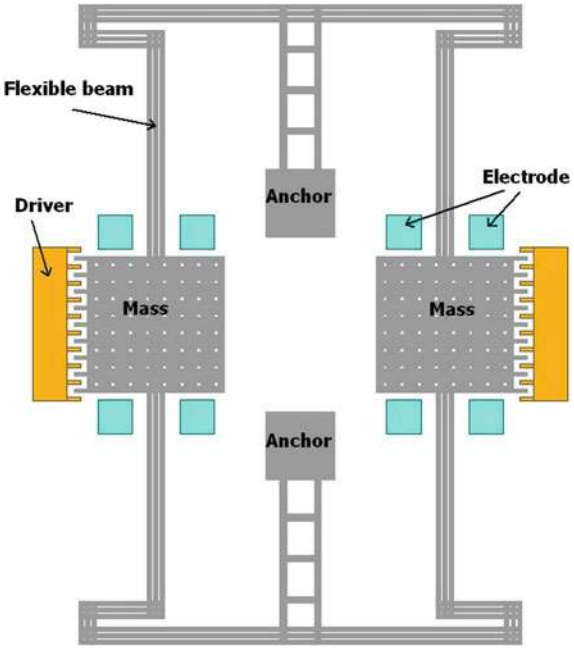



**Fig. 5.9** Tuning fork gyroscope (Courtesy of Zhili Hao, Old Dominion University)

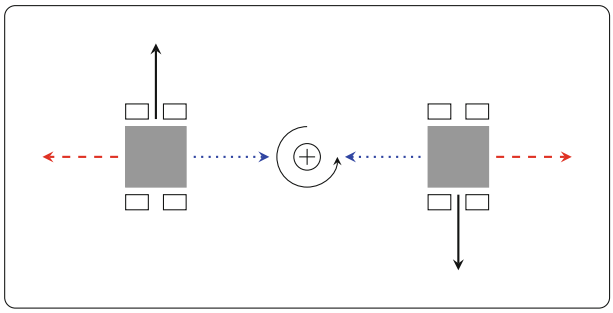


**Fig. 5.10** Physics of a tuning fork gyroscope: *red dashed arrows* and *blue dotted arrows* indicate the direction of the vibration; *solid black arrows* indicate the direction of the Coriolis force

The theory of operation of the tuning fork gyroscope is shown in Fig. 5.10. The masses (gray squares) are forced to vibrate at the same frequency like the two prongs of a tuning fork. They vibrate in different directions, that is, they either approach each other (blue dotted arrows) or they move away from each other (dashed red arrows). The component rotates around an axis perpendicular to its center (the circle with a cross denotes the rotational axis which is perpendicular to the plane of the paper). The Coriolis force is a force whose direction is given by the vector cross product of the axis of the rotation and the movement of the mass, and whose magnitude is proportional to the linear velocity of the mass and the angular velocity of the gyro. Since the masses are moving in different directions, the resulting forces will be equal but in opposite directions (solid arrows). The masses approach or recede from the electrodes (small rectangles) and the change in capacitance can be measured by a circuit.

### *5.9.3 Applications*

An inertial navigation system has three accelerometers and three gyroscopes so that the pose of the vehicle can be computed in three dimensions. This is necessary for robotic aircraft and other robotic vehicles. Airbags use an accelerometer that detects the rapid deceleration in the front-back direction that occurs when a car crashes. This causes an explosive expansion of the airbag. One can conceive of more applications for these components in cars. An accelerometer in the up-down direction can detect if the car has fallen into a pothole. A gyroscope measuring rotation around the vertical axis can detect skidding, while the gyroscope measuring rotation around the front-rear axis can detect if the car is rolling over.

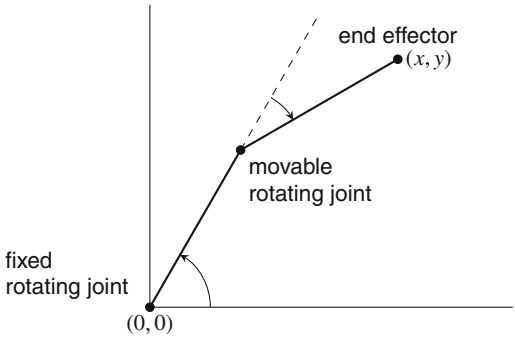



**Fig. 5.11** A two-link robotic arm with two DOF

## 5.10 Degrees of Freedom and Numbers of Actuators

The number of *degrees of freedom (DOF)* of a system is the dimensionality of the coordinates needed to describe a pose of a mobile robot or the pose of the end effector of a robotic manipulator.[3] For example, a helicopter has six DOF because it can move in the three spatial dimensions and can rotate around the three axes. Therefore, a six-dimensional coordinate $(x, y, z, \phi, \psi, \theta)$ is needed to describe its pose.

**The terms used to describe rotations**

A helicopter can rotate around all three of its axes. The rotations are called: (a) pitch: the nose moves up and down; (b) roll: the body rotates around its lengthwise axis; (c) yaw: the body rotates left and right around the axis of its rotor.

The two-link robotic arm in Fig. 5.11 has only two DOF because its end effector moves in a plane and does not rotate; therefore, it can be described by a two-dimensional coordinate $(x, y)$. By examining Fig. 5.3 again, you can see that a mobile robot moving on a flat surface has three DOF, because its pose is defined by a three-dimensional coordinate $(x, y, \theta)$. A train has only one DOF since it is constrained by the tracks to move forwards (or occasionally backwards) along the track. It only takes one coordinate $(x)$, the train's distance from an arbitrary origin of the track, to specify the pose of the train.

We need more information than the degrees of freedom to describe robotic motion. Consider a vehicle like a car, a bicycle or an office chair. Although three coordinates $(x, y, \theta)$ are needed to describe its pose, we cannot necessarily move the vehicle directly from one pose to another. An office chair can be moved *directly* to any point of the plane and oriented in any direction. A car or a bicycle at $(2, 0, 0°)$ (pointed along the positive $x$-axis) cannot be moved directly up the $y$-axis to position $(2, 2, 0°)$. A more complex maneuver is needed.

[3]This section and the following ones are more advanced and can skipped during your first reading. Furthermore, some of the examples are of robotic manipulators described in Chap. 16.

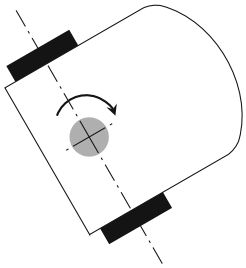


**Fig. 5.12** A robot that can only rotate around an axis (*gray dot*)

We need to know the number of its *actuators* (usually motors) and their configuration. A differential drive robot has two actuators, one for each wheel, although the robot itself has three DOF. The motors move the robot along one axis forwards and backwards, but by applying unequal power we can change the heading of the robot. The two-link arm in Fig. 5.11 has two motors, one at each rotating joint, so the number of actuators equals the number of DOF. Finally, a train has only one actuator, the motor that moves it forwards or backwards in its single DOF.

**Activity 5.12: Robot that can only rotate**

- Figure 5.12 shows a differential drive robot with a fixed rod through its center of rotation. The rod prevents the robot changing its position, allowing it only to rotate around its vertical axis. Characterize this configuration: the number of actuators and the number of DOF.
- What types of tasks could this robot perform? What are the advantages and disadvantages of this configuration?

## 5.11 The Relative Number of Actuators and DOF

Let us analyze systems where:

- The number of actuators equals the number of DOF;
- The number of actuators is fewer than the number of DOF;
- The number of actuators is greater than the number of DOF.

**The Number of Actuators Equals the Number of DOF**

A train has one actuator (its engine) that moves the train along its single DOF. The two-line robotic arm in Fig. 5.11 has two actuators and two DOF. A robotic gripper can be built with three motors that rotate the gripper in each of the three orientations

(roll, pitch, yaw). The advantage of having an equal number of actuators and DOF is that the system is relatively easy to control: each actuator is individually commanded to move the robot to the desired position in the DOF it controls.

**The Number of Actuators is Fewer than the Number of DOF**

Mobile robots will usually have fewer actuators than DOF. A robot with differential drive and a car have only two actuators, but they can reach all possible three-dimensional poses in the plane. Having fewer actuators makes the system less expensive, but the problems of planning and controlling motion are much more difficult. Parallel parking a car is notorious for its difficulty: two rotations and a translation are needed to perform a simple lateral move (Fig. 5.21a, b).

An extreme example is a hot-air balloon which has only a single actuator (a heater) that injects more or less hot air into the balloon and thus controls its altitude. However, winds can cause the balloon to move in any of the three spatial directions and even to rotate (at least partially) in three orientations, so the operator of the balloon can never precisely control the balloon. A hot-air balloon therefore differs from an elevator: both have a single actuator, but the elevator is constrained by its shaft to move in only one DOF.

For another example of the complex relationship between the DOF and number of actuators, the reader is invited to study flight control in helicopters. Helicopters are highly maneuverable (even more so than airplanes which can't fly backwards), but a pilot controls the helicopter's flight using only three actuators:

- The *cyclic* controls the pitch of the main rotor *shaft* which determines if the helicopter moves forwards, backwards or to either side.
- The *collective* controls the pitch of the *blades* of the main rotor which determines if the helicopter moves up or down.
- The *pedals* control the speed of the tail rotor which determines the direction in which the nose of the helicopter points.

**The Number of Actuators is Greater than the Number of DOF**

It doesn't seem to be a good idea to have *more* actuators than DOF, but in practice such configurations are often useful. The systems in Fig. 5.13a, b have more actuators than DOF. The robotic manipulator arm in Fig. 5.13a has four links rotating in the plane with actuators (motors) `a1`, `a2`, `a3`, `a4` at the joints. We assume that the end effector is fixed to the link from `a4` and cannot rotate, so its pose is defined by its position $(x, y)$ and a fixed orientation. Therefore, although the arm has four actuators, it has only two DOF because it can move the end effector only horizontally and vertically.

The mobile robot with an arm (Fig. 5.13b) has three actuators: a motor that moves the robot forwards and backwards, and motors for each of the two rotating joints. However, the system has only two DOF since the pose of its end effector is defined by a two-dimensional $(x, y)$ coordinate. Systems with more actuators than DOF are called *redundant systems*.

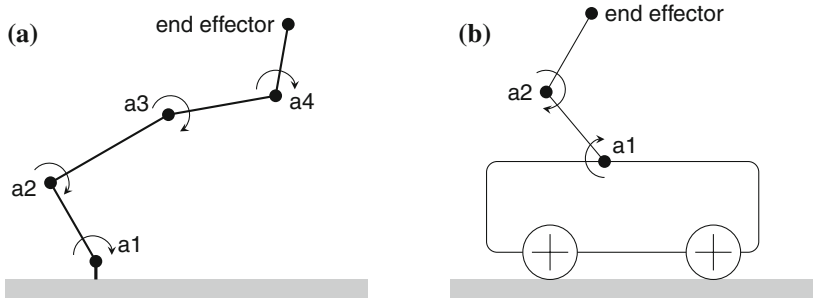



**Fig. 5.13** **a** Robot arm: two DOF and four actuators. **b** Mobile robot and arm: two DOF and three actuators

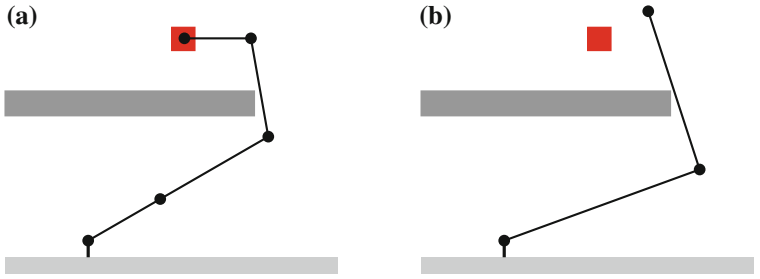



**Fig. 5.14** **a** Arm with four actuators can reach a hidden position. **b** Arm with two actuators is blocked by an obstacle

If possible, engineers avoid using more than one actuator acting on the same DOF because it increases the complexity and cost of a system. The inverse kinematics (Sect. 16.2) of a redundant system results in an infinite number of solutions which complicate the operation of the system. For the mobile robot with the arm (Fig. 5.13b), there are an infinite number of positions of the base and arm that bring the end effector to a specific reachable position.

There are situations where a redundant system is required because the task could not be performed with fewer actuators. Figure 5.14a shows how the four-link robotic arm of Fig. 5.13a can move the end effector to a position that is blocked by an obstacle and thus unreachable by a two-link arm (Fig. 5.14b), even though in both configurations the total length of the links is equal.

An important advantage of redundant systems arises from actuators with different characteristics. The mobile robot in Fig. 5.13b can approach the target quickly, although its final position might not be accurate because of errors like uneven terrain. Once the mobile robot stops, the motors in the joints which do not have to deal with the terrain can be precisely positioned. While the positioning is precise, these joints do not have the broad range of the mobile base.

### An Example of a System with More Actuators than DOF

Figure 5.15 (top and side views) shows a configuration with *two actuators and one DOF*. The system models a robotic crane that moves a heavy weight to a specific

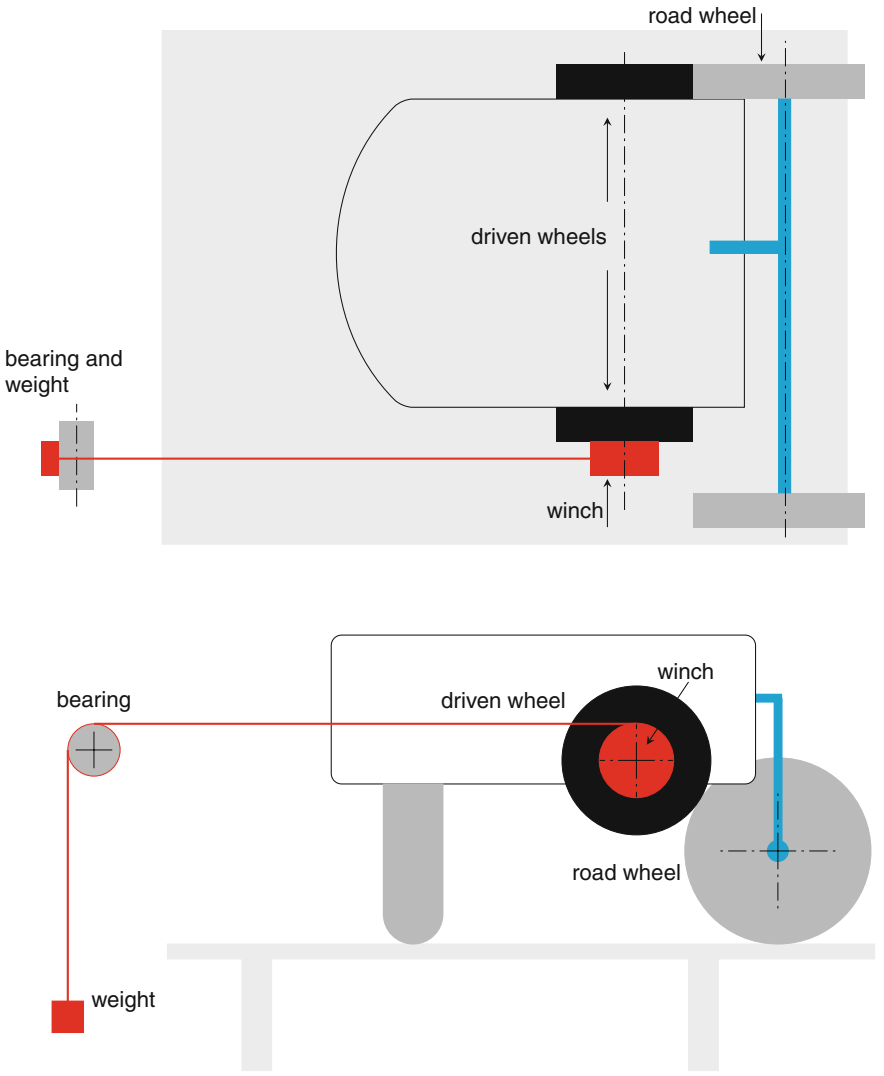



**Fig. 5.15** Robotic crane built from a mobile robot and a winch (top view *above*, side view *below*); in the side view the left wheel is not shown

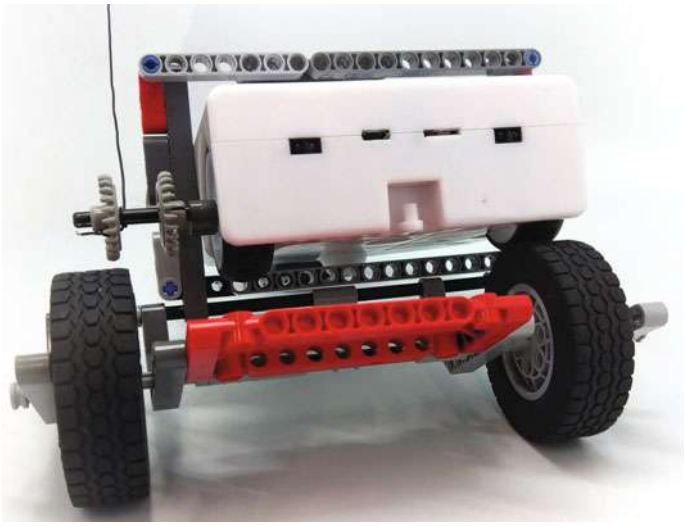


**Fig. 5.16** Robotic crane built from a Thymio robot and LEGO® components

vertical position. Figure 5.16 shows a crane built from a Thymio robot and LEGO® components.

The system is build from a mobile robot with differential drive, but the wheels are not directly used to control the motion of the system. Instead, each wheel is an independent actuator. (Recall that the power to each wheel of a differential drive robot can be set independently to any value in a range such as −100 to 100.)

The robot faces left. The right driven wheel in Fig. 5.15 (the black rectangle at the top of the top view and hidden behind the robot in the side view) controls a pair of (gray) road wheels that move the robot rapidly forwards and backwards. In turn, this causes the cable to move the weight rapidly up and down.

The road wheels are mounted on a structure (in blue) that is fixed to the robot body. There are several options for transferring power from the right driven wheel to the road wheels: friction, pulleys and belts, and gears. Each option has its own advantages and disadvantages, and all three are used in cars: the clutch uses friction, belts are used for timing and to run auxiliary components like water pumps, and gears are used in the transmission to control the torque applied to each wheel.

The left driven wheel (the black rectangle at the bottom of the top view and at the front of the side view) controls a winch (red) that rolls or unrolls a cable attached to the weight that moves up or down over a fixed bearing. The winch has a diameter much smaller than the diameter of the driven wheels, so it can move the weight in small increments as the left driven wheel rotates. The design goal is to be able to perform precise positioning of the weight even though the winch moves the cable at a much slower speed than does the robot body.

There are two activities for this section. This activity is for readers who have good construction skills and an appropriate robotics kit. The second activity suggests alternate ways of demonstrating the concept of two actuators in one DOF.

**Activity 5.13: Robotic crane**

- Construct the robotic crane shown in Fig. 5.15. Explain your choice of mechanism for connecting the driven wheel to the road wheels.
- Write a program that given the current position of the weight and a goal position moves the weight to the goal position. Alternatively, send commands to the motors using a remote control device or a computer connected to the robot.
- Experiment with the relative rotational speeds of the left and right driven wheels that control the road wheels and the winch, respectively. Should you move the two actuators separately or simultaneously?

**Activity 5.14: Robotic crane (alternatives)**

- Write a program that causes a mobile robot to move forwards and backwards. Place a piece of black tape relatively far from the initial position of the robot. The goal is to cause the robot to stop as near as possible to the start of the tape *without* continuously checking the sensor.
- The program has three modes of operation. (1) The robot moves fast, checking its sensor occasionally, and stopping when it detects the tape. (2) As in (1) but the robot moves slowly, checking its sensor relatively often. (3) As in (1) but when the tape is detected, the robot moves backwards using the speed and sampling period as in (2).
- Run the program and compare the results of the three modes: the time until the robot stops and error between the robot's final position and the start of the tape.
- Alternatively, run the program with the three modes on a computer and experiment with the motion parameters and the sampling periods. You will need to choose a model for the motion: constant velocity, constant acceleration, or (more realistically) acceleration then constant velocity and finally deceleration when the tape is detected.

## 5.12 Holonomic and Non-holonomic Motion

Section 5.10 presented the concept of degree of freedom (DOF) and the role of the number of actuators. There is another concept that links the DOF and the actuators in the case of mobile robot: the *degree of mobility (DOM)*. The degree of mobility $\delta_m$ corresponds to the number of degrees of freedom that can be *directly accessed* by the actuators. A mobile robot in the plane has at most three DOF ($(x, y)$ position and heading), so the maximal degree of mobility of a mobile robot is $\delta_m = 3$.

Let consider the DOM of various vehicles. A train has one DOF because it can only move forwards along the tracks, and it has one actuator, its engine, that directly affects this single degree of freedom. Therefore, a train has a degree of mobility of $\delta_m = 1$, meaning that the single DOF can be directly accessed by the actuator.

A robot with differential drive has three DOF. The two actuators are the two motors which act on the wheels. They can directly access two DOF: (a) if both wheels turn at the same speed, the robot moves forwards or backwards; (b) if the wheels have speeds in opposite directions, the robot rotates in place. Therefore, we can directly access the DOF along the forward axis of translation and the DOF of the heading, but we cannot directly access the DOF of the lateral axis of translation (Fig. 5.17a). A differential drive mobile robot has a degree of mobility $\delta_m = 2 < 3 = \#\text{DOF}$.

A car, like a robot with differential drive, has only two actuators for three DOF: one actuator, the motor, gives direct access to the degree of freedom along the longitudinal axis of the car, enabling it to move forwards and backwards. The other actuator, the steering wheel, does *not* give direct access to any additional DOF, it can only orient the first DOF. The car cannot rotate around the vertical axis and it cannot move laterally (Fig. 5.17b). Therefore, a car has only one degree of mobility, $\delta_m = 1$. Intuitively, you can see the lower degree of mobility of a car compared with a robot with differential drive by noting that the robot can rotate in place while the car cannot.

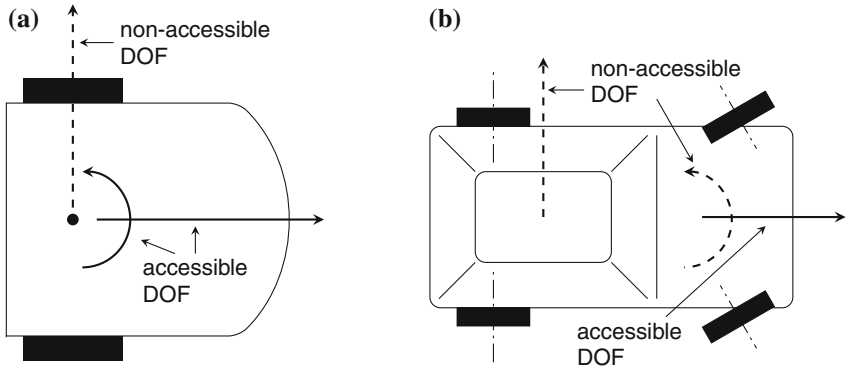



**Fig. 5.17** **a** Accessible and non-accessible DOF for a robot with differential drive. **b** Accessible and non-accesible DOF for a robot with Ackermann steering

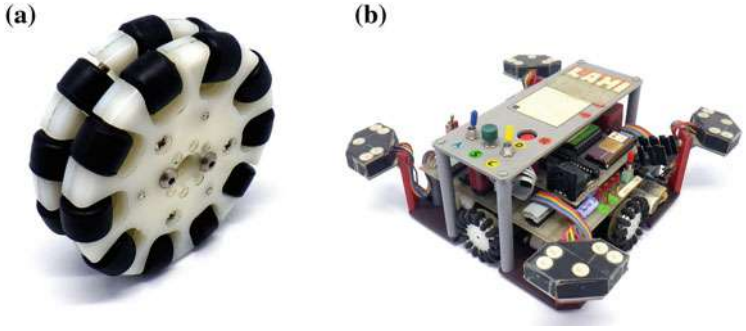



**Fig. 5.18** **a** Swedish wheel. **b** Omnidirectional robot (Courtesy LAMI-EPFL)

By itself, a standard wheel has $\delta_m = 2$: it can roll forwards and backwards and it can rotate around the vertical axis that goes through the point of contact of the wheel with the ground. A wheel cannot move sideways, which is actually a good thing because it prevents the vehicle from skidding off the road during a turn. In the car, the degree of mobility is reduced even further to $\delta_m = 1$, because there are two pairs of wheels, one in the front and one in the rear of the car. This configuration makes it impossible for the car to rotate around its vertical axis, even though the individual wheels can do so (usually only the front wheels). The limitation to $\delta_m = 1$ gives stability to the car—it cannot skid laterally and it cannot rotate—making it easy and safe to drive at high speeds. In fact, an accident can occur when rain or snow reduce the friction so that the car can skid or rotate.

An autonomous mobile robot can profit if it has a greater DOM $\delta_m = 3$. To directly access the third DOF, the robot needs to be able to move laterally. One method is to have the robot roll on a ball or a castor wheel like an office chair. Another method is to use *Swedish wheels* (Fig. 5.18a). A Swedish wheel is a standard wheel that has small free wheels along its rim so that it can move laterally, enabling direct access to the third DOF.

Mobile robots that can directly access all three DOF ($\delta_m = 3$) are called *omnidirectional robots*. Figure 5.18b shows an omnidirectional robot constructed with four Swedish wheels. The two pairs of wheels on opposite sides of the robot can directly move the robot left, right, forwards and backwards. This configuration is redundant but very easy to control. To avoid redundancy, most omnidirectional robots have three Swedish wheels mounted at an angle of 120° from each other (Fig. 5.19). This configuration has $\delta_m = 3$ but is not easy to control using the familiar $x$, $y$ coordinates.

The relative values of the DOF and the DOM of a robot define the concept of holonomic motion. A robot has *holonomic motion* if $\delta_m = \#\text{DOF}$ and it has *non-holonomic motion* $\delta_m < \#\text{DOF}$. A holonomic robot like the one in Fig. 5.18b can directly control all its DOF without difficult maneuvers. Figure 5.20 shows how easy it is for the omnidirectional robot with Swedish wheels (Fig. 5.19) to perform parallel parking.

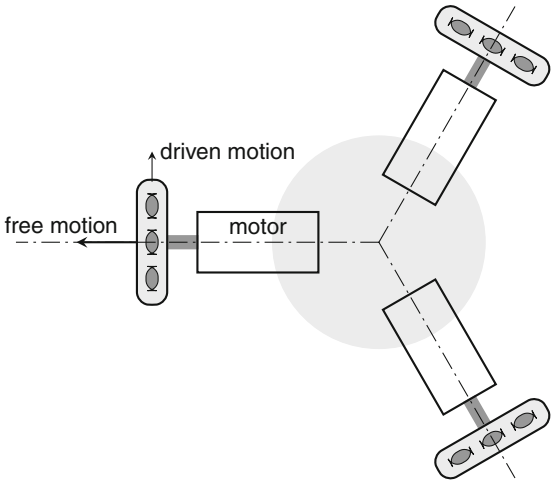



**Fig. 5.19** Omnidirectional robot with three Swedish wheels

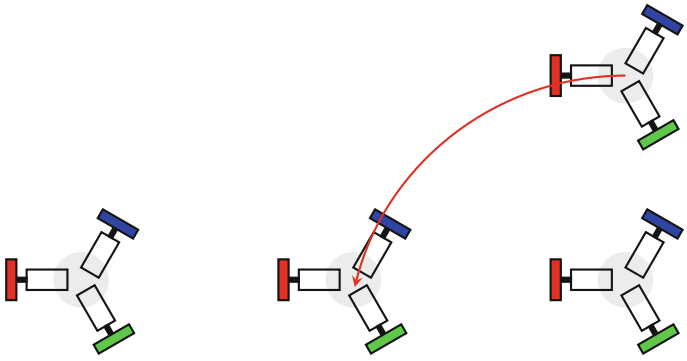


**Fig. 5.20** Parallel parking by an omnidirectional robot

A car and a robot with differential drive are non-holonomic because their DOM ($\delta_m = 1$ and 2, respectively) are lower than their DOF which is three. Because of this limited degree of mobility, these vehicles need complex steering maneuvers, for example, to perform parallel parking. There is a significant difference between the two vehicles. The differential drive robot needs three separate movements, but they are very simple (Fig. 5.21a): rotate left, move backwards, rotate right. The car also needs three separate movements, but they are extremely difficult to perform correctly (Fig. 5.21b). You have to estimate where to start the maneuver, how sharp to make each turn and how far to move between turns. The higher DOM of the differential drive robot is advantageous in this situation.

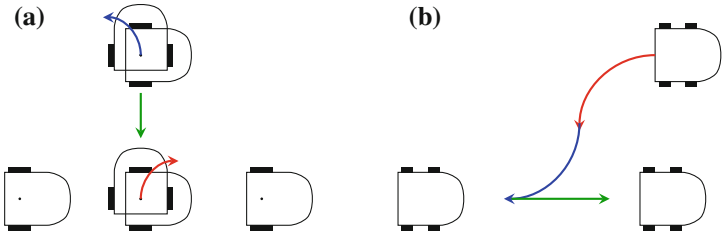



**Fig. 5.21** **a** Parallel parking for a non-holonomic differential drive robot. **b** Parallel parking for a non-holonomic car

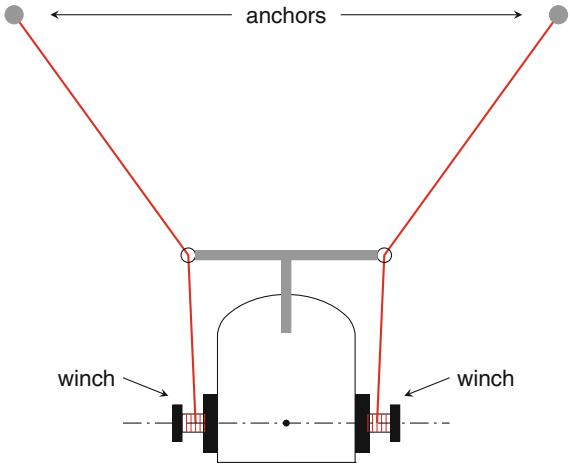



**Fig. 5.22** Robot for cleaning a wall

**Activity 5.15: Holonomic and non-holonomic motion**

- Look again at the mobile robot which is constrained to rotational motion only (Fig. 5.12). What is its degree of mobility $\delta_m$? It is holonomic or not?
- Figure 5.22 shows a robot for cleaning the walls of a building. There are two anchors from which cables descend, passing through eyes fixed to the robot's body and then to winches powered by the robot's wheels. By rolling and unrolling the cables, the robot moves up and down the wall. However, if the two motors do not cause the wheels move precisely at the same rotational velocity, the robot will swing from side to side. How many DOF and how many DOM does this robot have? Is it holonomic or not?

## 5.13 Summary

A mobile robot like a self-driving car or a Mars explorer will not have landmarks always available for navigation. Odometry is used to bring the robot to the vicinity of its goal without reference to the environment. The robot estimates its speed and rotational velocity from the power applied to its motors. Odometry can be improved by using wheel encoders to measure the number of revolutions of the wheels, rather than inferring the velocity from the motor power. The change in the position of an inexpensive robot moving in a straight line can be computed by multiplying speed by time. If the robot is turning, trigonometric calculations are needed to compute the new position and orientation. Even with wheel encoders, odometry is subject to errors that can be very large if the error is in the heading.

Inertial navigation uses accelerometers and gyroscopes to improve the accuracy of odometry. Integrating acceleration gives velocity and integrating angular velocity gives the heading. Microelectromechanical systems have made inertial navigation inexpensive enough for use in robotics.

The DOF of a system is the number of dimensions in which it can move—up to three dimensions on a surface and up to six dimensions in the air or underwater—but a robot may be constrained to have fewer than the maximum number of DOF. An additional consideration is the number and configuration of the actuators of a robot which define its degree of mobility. If the DOM is equal to the number of DOF, the robot is holonomic and it can move directly from one pose to another, although it may be difficult to control. If the DOM is less than the number of DOF, the robot is non-holonomic; it cannot move directly from one pose to another and will require complex maneuvers to carry out some tasks.

## 5.14 Further Reading

A detailed mathematical treatment of odometry errors in two dimensions is given in [5, Sect. 5.24]. For an overview of inertial navigation see [3, 4]. Advanced textbooks on robotics present holonomy [1, 2, 5, 6].

## References

1. Correll, N.: Introduction to Autonomous Robots. CreateSpace (2014). https://github.com/correll/Introduction-to-Autonomous-Robots/releases/download/v1.9/book.pdf
2. Craig, J.J.: Introduction to Robotics: Mechanics and Control, 3rd edn. Pearson, Boston (2005)
3. King, A.: Inertial navigation—forty years of evolution. GEC Rev. **13**(3), 140–149 (1998). http://www.imar-navigation.de/downloads/papers/inertial_navigation_introduction.pdf
4. Oxford Technical Solutions: What is an (INS) inertial navigation system? http://www.oxts.com/what-is-inertial-navigation-guide/

5. Siegwart, R., Nourbakhsh, I.R., Scaramuzza, D.: Introduction to Autonomous Mobile Robots, 2nd edn. MIT Press, Cambridge (2011)
6. Spong, M.W., Hutchinson, S., Vidyasagar, M.: Robot Modeling and Control. Wiley, New York (2005)



# Chapter 6
# Control

Robotics algorithms make decisions. The robot is given a task, but in order to perform the task it must take actions and these actions depend on the environment as detected by the sensors. For example, if a robot is to bring an object from a shelf in a warehouse to a delivery track, it must use sensors to navigate to the proper shelf, to detect and grasp the object, and then to navigate back to the truck and load the object. Only robots that act in extremely well-defined environments can carry out such tasks without information from sensors. An example is a robotic arm assembling a device in a factory; if the parts are precisely positioned on the work surface, the robot can manipulate the parts without sensing them. But in most environments sensors must be used. In a warehouse there may be obstacles on the way to the shelf, the object will not be precisely positioned on the shelf and the truck is never parked in exactly the same place. The robot needs to adapt to these small variations using *control algorithms* to make decisions: based upon data from the sensors what actions does the robot need to perform in order to accomplish the task? There is a sophisticated mathematical theory of control that is fundamental in robotics. In this chapter, we present the basic concepts of control algorithms.

Section 6.1 explains the difference between two control models: open loop control where the parameters of the algorithm are set in advance and closed loop control where data from sensors influences the behavior of the algorithm. Sections 6.2–6.5 present four increasingly sophisticated closed loop control algorithms. The designer of a robot must choose among these and similar algorithms to select the one that gives adequate performance for the least computational cost.



M. Ben-Ari and F. Mondada, *Elements of Robotics*,
https://doi.org/10.1007/978-3-319-62533-1_6

## 6.1 Control Models

There are two ways that a control algorithm can decide upon an action. In an open loop system, the parameters of the control algorithm are preset and do not change while the system runs. In a closed loop system, sensors measure the error between the desired state of the system and its actual state, and this error is used to decide what action to take.

### *6.1.1 Open Loop Control*

A toaster is a machine that performs actions semi-autonomously. You place slices of bread in the toaster, set the timer and push the lever down to start the toasting action. As we all know, the results are not guaranteed: if the duration of the timer is too short, we have to toast the bread again; if the duration of the timer is too long, the scent of burnt toast floats through the kitchen. The outcome is uncertain because a toaster is an *open loop control system*. It does not check the outcome of the toasting action to see if the required result has been achieved. Open loop systems are very familiar: on a washing machine you can set the temperature of the water, the duration of the cycle and the amount of detergent used, but the machine does not measure the "cleanness" of the clothes (whatever that means) and modify its actions accordingly.

A mobile robot that moves to a target position based on odometry alone (Sect. 5.4) is also using open loop control. By keeping track of the motor power and the duration that the motors run, the robot can compute the distance it has moved. However, variations in the speed of the wheels and in the surface on which the robot moves will cause uncertainty in the final position of the robot. In most applications, odometry can be used to move the robot to the vicinity of the goal position, at which point sensors are used to move the robot to the precise goal position, for example, by using sensors to measure the distance to an object.

### *6.1.2 Closed Loop Control*

To achieve autonomous behavior, robots use *closed loop control systems*. We have already encountered closed loop systems in the Braintenberg vehicles (Activity 3.5):

> **Specification (Attractive and repulsive)**: When an object approaches the robot from behind, it runs away until it is out of range.

The robot must *measure* the distance to the object and stop when this distance is sufficiently large. The power setting of the motors depends on the measurement of the distance, but the robot moves at a speed that depends on the power of the motors,

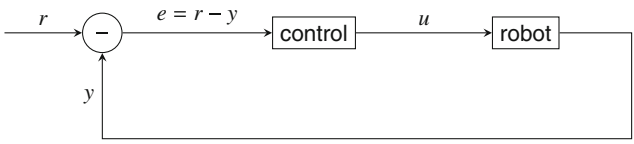



**Fig. 6.1** A closed loop control system

which changes the distance to the object, which again modifies the power setting, which …. This circular behavior is the origin of the term "closed loop."

We now formalize the specification of a closed loop control system for a robot (Fig. 6.1). The variable $r$ represents the *reference value*, the specification of the robot's task. In a warehouse robot, reference values include the position of the robot relative to a stack of shelves and the distance of the gripper arm from the object to be picked up. A reference value cannot be used directly by the robot; instead, it must be transformed into a *control value* $u$. For example, if the reference value is the position of the robot relative to a shelf, the control value will be power settings of the motors and the duration that the motors are running. The variable $y$ represents the output, that is, the actual state of the robot, for example, the distance to an object.

The model in Fig. 6.1 is also called a *feedback control system* because the output value $y$ is fed back to the control algorithm and used to compute the control value. The output is compared with the reference value to compute $e = r - y$, the *error*. The control algorithm uses the error to generate the control signal $u$ that is the input to the robot.

### 6.1.3 The Period of a Control Algorithm

Control algorithms are run periodically (Algorithm 6.1). The robot's software initializes a *timer variable* to the required duration of the period to run the algorithm, for example, every 20 ms. The embedded computer has a *hardware clock* that "ticks" at fixed intervals causing an interrupt. The interrupt is handled by the operating system which decrements the value of the timer variable. When the value of this variable goes to zero, the timer has expired, and an event is raised in the software causing the control algorithm to be run.

The period of the algorithm is an important parameter in the design of a control system. If the period is too short, valuable computing resources will be wasted and the computer can become overloaded to the point that commands to the robot arrive too late. If the period is too long, the robot will not respond in time to correct errors in its motion.

**Algorithm 6.1: Control algorithm outline**

```
        integer period                    // Duration of timer period
        integer timer                     // Timer variable
 1: period ← ···                          // Period in milliseconds
 2: timer ← period                        // Initialize the timer
 3: loop
 4:     when timer-expired-event occurs
 5:         control algorithm             // Run the algorithm
 6:         timer ← period                // Reset the timer

    // Operating system
 7: when hardware-clock-interrupt occurs
 8:     timer ← timer − 1                 // Decrement the timer
 9:     if timer = 0                      // If the timer expires
10:         raise timer-expired-event     //     raise an event
```

**Example** Consider a robot approaching an object that is 10 cm away at 2 cm/s. A control period of 1 ms would waste computing resources because the robot will move only 0.002 cm (0.02 mm) during *each* 1 ms cycle of the control algorithm. Changes in motor power over such small distances won't affect the ability of the robot to fulfill its task. At the opposite extreme, a control period of 2 s is even worse: the robot will move 4 cm during this period and will likely to crash into the object. A control period of roughly 0.25 s (250 ms) during which the robot moves 0.5 cm seems a reasonable value to start with since 0.5 cm is a distance that is meaningful in terms of approaching an object. You can experiment with periods around this value to determine the optimum period: one that achieves satisfactory behavior with a period that is as long as possible to reduce the cost of computation.

**Activity 6.1: Setting the control period**

- In the example, we came to the conclusion that the optimum period of the control algorithm was of the order of magnitude of tenths of a second. In this activity we ask you consider what the optimum period should be for other control algorithms.
- A home heating system contains a thermostat for controlling the temperature. What would be an optimum period for the control algorithm? The period depends on the engineering parameters of the heating system and on the physical properties of how the heat is transferred to the rooms. Explain how you would measure these factors and how they affect the control period.
- Consider a self-driving car trying to park. What assumptions do you need to make to design a control period? What would be a reasonable period?

- How do the properties of the sensors affect the control period? For the example of a robot approaching an object, how would the period change if the sensor can detect the object at 2 cm, 5 cm, 10 cm, 20 cm, 40 cm?

We now define a sequence of four control algorithms, each one building on the previous one and providing more accurate control, at the price of more computational complexity. In practice, the system designer should choose the simplest algorithm that enables the robot to fulfill its task.

The algorithms are presented in the context of a robot which must approach an object and stop at a distance $s$ in front of it. The distance is measured by a proximity sensor and the speed of the robot is controlled by setting the power of the motors.

## 6.2 On-Off Control

The first control algorithm is called the *on-off* or *bang-bang* algorithm (Algorithm 6.2). We define a constant reference that is the distance in front of the object at which the robot is to stop. The variable measured is the actual distance measured by the proximity sensor. The error is the difference between the two:

$$\text{error} \leftarrow \text{reference} - \text{measured},$$

**Algorithm 6.2: On-off controller**

```
integer reference ← ···     // Reference distance
integer measured            // Measured distance
integer error               // Distance error
error ← reference − measured
if error < 0
    left-motor-power ← 100      // Move forwards
    right-motor-power ← 100
if error = 0
    left-motor-power ← 0        // Turn off motors
    right-motor-power ← 0
if error > 0
    left-motor-power ← −100     // Move backwards
    right-motor-power ← −100
```

which is negative if the robot is too far away from the object and positive if it is too close to the object. The motor powers are turned to full forwards or full backwards depending on the sign of the error. For example, if the reference distance is 10 cm and the measured distance is 20 cm, the robot is too far away and the error is −10 cm. Therefore, the motors must be set to move forwards.

The robot approaches the object at full speed. When the robot reaches the reference distance from the object, it takes time for the sensor to be read and the error to be computed. Even if the robot measures a distance exactly equal to the reference distance (which is unlikely), the robot will not be able to stop immediately and will overrun the reference distance. The algorithm will then cause the robot to back up at full speed, again passing the reference distance. When the timer causes the control algorithm to be run again, the robot will reverse direction and go forwards at full speed. The resulting behavior of the robot is shown in Fig. 6.2: the robot will oscillate around the reference distance to the object. It is highly unlikely that the robot will actually stop at or near the reference distance.

A further disadvantage of the on-off algorithm is that the frequent and abrupt reversal of direction results in high accelerations. If we are trying to control a gripper arm, the objects that it is carrying may be damaged. The algorithm generates high levels of wear and tear on the motors and on other mechanical moving parts.

**Activity 6.2: On-off controller**

- Implement the on-off algorithm on your robot for the task of stopping at a reference distance from an object.
- Run it several times starting at differences distances from the object.

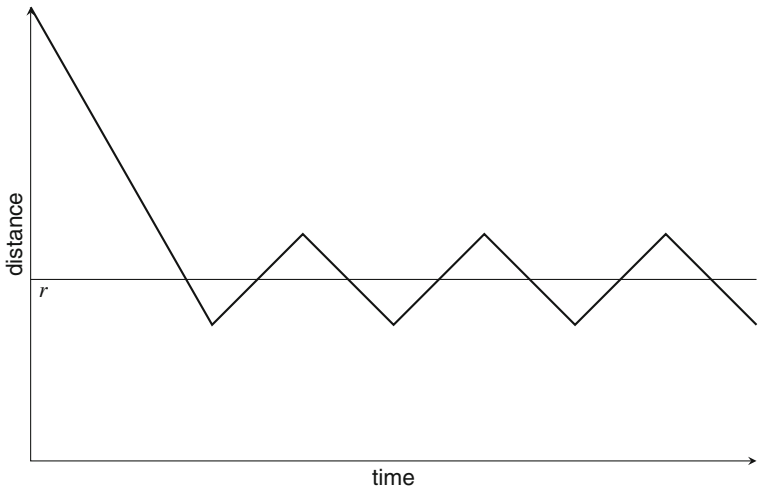



**Fig. 6.2** Behavior of the on-off algorithm

- Algorithm 6.2 stops the robot when the error is exactly zero. Modify the implementation so that the robot stops if the error is within a small range around zero. Experiment with different ranges and see how they affect the behavior of the robot.

## 6.3 Proportional (P) Controller

To develop a better algorithm, we take inspiration from riding a bicycle. Suppose that you are riding your bicycle and see that the traffic light ahead has turned red. You don't wait until the last moment when you are at the stop line and then squeeze hard on the brake lever; if you do so, you might be thrown from the bicycle! What you do is to slow your speed gradually: first, you stop pedaling; then, you squeeze the brake gently to slow down a bit more; finally, when you are at the stop line and going slowly, you squeeze harder to fully stop the bicycle. The algorithm used by a bicycle rider can be expressed as:

> Reduce your speed more as you get closer to the reference distance.

The decrease in speed is (inversely) *proportional* to how close you are to the traffic light: the closer you are, the more you slow down. The factor of proportionality is called the *gain* of the control algorithm. An alternate way of expressing this algorithm is:

> Reduce your speed more as the error between the reference distance and the measured distance gets smaller.

Algorithm 6.3 is the *proportional control algorithm* or a *P-controller*.

**Algorithm 6.3: Proportional controller**

```
integer reference ←···   // Reference distance
integer measured         // Measured distance
integer error            // Error
float gain ←···          // Proportional gain
integer power            // Motor power
1: error ← reference − measured // Distances
2: power ← gain * error         // Control value
3: left-motor-power ← power
4: right-motor-power ← power
```

**Example** Suppose that the reference distance is 100 cm and the gain is −0.8. When the robot is 150 cm away from the object, the error is $100-150 = -50$ and the control algorithm will set the power to $-0.8 \cdot -50 = 40$. Table 6.1 shows the errors and power settings for three distances. If the robot overruns the reference distance of 100 cm and a distance of 60 cm is measured, the power will be set to −32 causing the robot to move backwards.

Figure 6.3 plots the distance of the robot to the object as a function of time when the robot is controlled by a P controller. The line labeled $r$ is the reference distance. The change in the motor power is smooth so the robot doesn't experience rapid accelerations and decelerations. The response is somewhat slow, but the robot does approach the target distance.

Unfortunately, the robot does not actually reach the reference distance. To understand why this happens, consider what happens when the robot is very close to the reference distance. The error will be very small and consequently the power setting will be very low. In theory, the low power setting should cause the robot to move slowly, eventually reaching the reference distance. In practice, the motor power may become so low that it is not able to overcome the internal friction in the motors and their connection to the wheels, so the robot stops moving.

It might seem that increasing the gain of the P controller could overcome this problem, but a high gain suffers from a serious disadvantage. Figure 6.4 shows the effect of the gain on the P controller. Higher gain (dashed red line) causes the robot to

**Table 6.1** Proportional controller for gain of −0.8

| Distance | Error | Power |
|---|---|---|
| 150 | −50 | 40 |
| 125 | −25 | 20 |
| 60 | 40 | −32 |

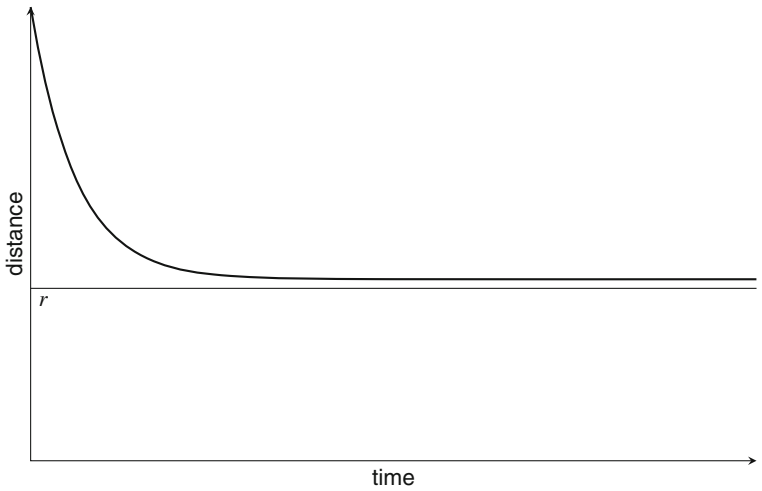



**Fig. 6.3** Behavior of the P controller

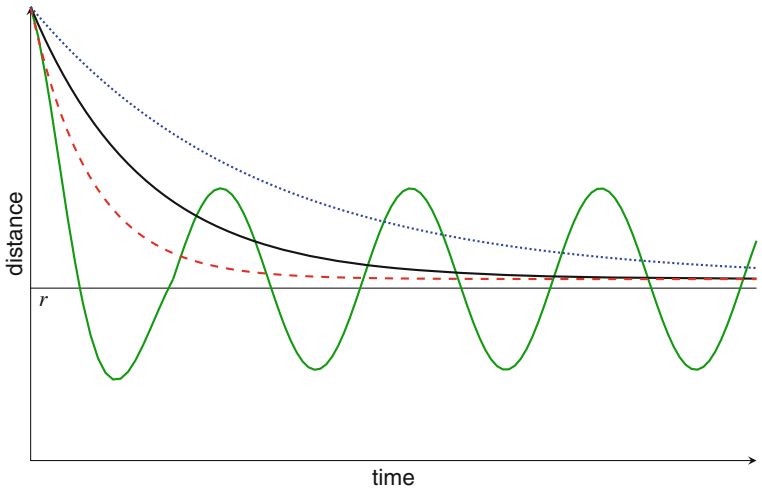



**Fig. 6.4** The effect of the gain on the P controller: lower gain (*dotted blue line*), higher gain (*dashed red line*), excessive gain (oscillating *green line*)

approach the reference distance faster, while lower gain (dotted blue line) causes the robot to approach the reference distance slower. However, if the gain is too high, the P controller functions like an on-off controller with an oscillating response (green line). We say that the controller is *unstable*.

There are situations where the P controller cannot reach the reference distance even in a ideal system. Suppose that the object itself is moving at constant speed away from the robot. The P controller will set maximum motor power to cause the robot to move rapidly towards the object. Eventually, however, as the robot approaches the object, the measured distance will become small and the P controller will set the power so low that the speed of the robot is lower than the speed of the object. The result is that the robot will never reach the reference distance. If the robot could actually reach the reference distance, the error would be zero and therefore the speed of the robot would also be zero. The object, however, is still moving away from the robot, so somewhat later the robot will start moving again and the cycle repeats. This start-and-stop motion is not the intended goal of maintaining the reference distance.

**Example** We use the same data as in the previous example except that the object moves at 20 cm/s. Table 6.2 shows the errors and power settings for three distances. Initially, the robot is going faster then the object so it will catch up. At 125 cm from the object, however, the robot is moving at the same speed as the object. It maintains this fixed distance and will not approach the reference distance of 100 cm. If somehow the robot gets closer to the object, say, 110 cm, the power is reduced to 8 causing the robot to back away from the object.

**Table 6.2** Proportional controller for a moving object and a gain of −0.8

| Distance | Error | Power |
|---|---|---|
| 150 | −50 | 40 |
| 125 | −25 | 20 |
| 110 | −10 | 8 |

In general, the robot will stabilize at a fixed distance from the reference distance. You can reduce this error by increasing the gain, but the reference distance will never be reached and the only result is that the controller becomes unstable.

**Activity 6.3: Proportional controller**

- Implement the proportional control algorithm to cause the robot to stop at a specified distance from an object. How accurately can you achieve the goal when the object does not move?
- What happens if the object moves? For the object you can use a second robot programmed to move at a fixed speed.
- Experiment with the gain and the period to see how they affect the performance of the algorithm.

## 6.4 Proportional-Integral (PI) Controller

A *proportional-integral controller* can achieve the reference distance even in the presence of friction or a moving object by taking into account the accumulated error over time. Whereas the P controller only takes into account the current error:

$$u(t) = k_p e(t)\,,$$

the PI controller adds the integral of the error from the time when the algorithm starts to run until the present time:

$$u(t) = k_p e(t) + k_i \int_0^t e(\tau)\, d\tau\,.$$

Separate gain factors are used for the proportional and integral terms to allow flexibility in the design of the controller.

When implementing a PI controller, a discrete approximation to the continuous integral is performed (Algorithm 6.4).

**Algorithm 6.4: Proportional-integral controller**

```
integer reference ← ···                    // Reference distance
integer measured                           // Measured distance
integer error                              // Error
integer error-sum ← 0                      // Cumulative error
float gain-p ← ···                         // Proportional gain
float gain-i ← ···                         // Integral gain
integer power                              // Motor power
1: error ← reference − measured            // Distances
2: error-sum ← error-sum + error           // Integral term
3: power ← gain-p * error + gain-i * error-sum // Control value
4: left-motor-power ← power
5: right-motor-power ← power
```

In the presence of friction or a moving object, the error will be integrated and cause a higher motor power to be set; this will cause the robot to converge to the reference distance. A problem with a PI controller is that the integration of the error starts from the initial state when robot is far from the object. As the robot approaches the reference distance, the integral term of the controller will have already a large value; to decrease this value the robot must move past the reference distance so that there are errors of opposite sign. This can generate oscillations (Fig. 6.5).

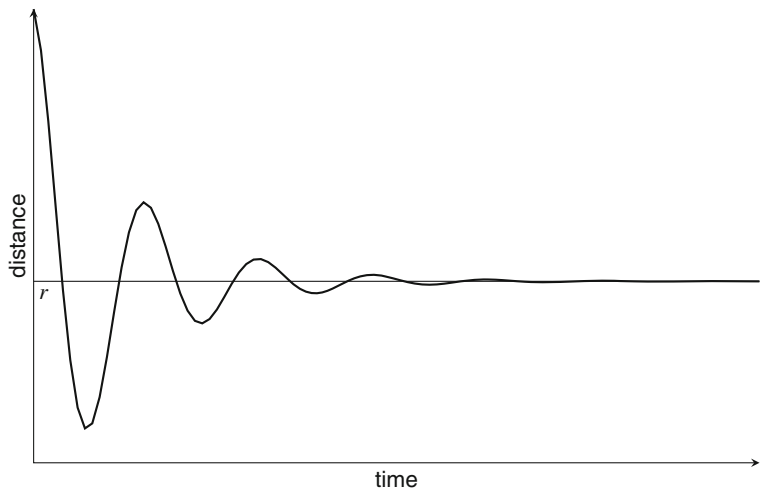



**Fig. 6.5** Behavior of the PI controller

**Activity 6.4: PI controller**

- Implement a PI controller that causes the robot to stop at a specified distance from an object.
- Compare the behavior of the PI controller with a P controller for the same task by monitoring the variables of the control algorithms over time.
- What happens if you manually prevent the robot from moving for a short time and then let it go? This demonstrates a concept called *integrator windup*. Explore the concept through online sources and find a method to fix the problem.

## 6.5 Proportional-Integral-Derivative (PID) Controller

When you throw or kick a ball to another player who is moving, you do not throw it to his current position. By the time the ball reaches him, he will have moved to a new position. Instead, you estimate where the new position will be and aim the ball there. Similarly, a robot whose task is to push a parcel onto a moving trolley must time its push to the estimated future position of the trolley when the parcel reaches it. The control algorithm of this robot cannot be an on-off, P or PI controller, because they only take into account the current value of the error (and for the PI controller the previous values).

To estimate the future error, the rate of change of the error can be taken into account. If the rate of change of the error is small, the robot can push the parcel just before the trolley approaches it, while if the rate of change of the error is large, the parcel should be pushed much earlier.

Mathematically, rate of change is expressed as a derivative. A *proportional-integral-derivative (PID)* controller adds an additional term to P and I terms:

$$u(t) = k_p e(t) + k_i \int_{\tau=0}^{t} e(\tau)\, d\tau + k_d \frac{de(t)}{dt}\,. \tag{6.1}$$

In the implementation of a PID controller, the differential is approximated by the difference between the previous error and current error (Algorithm 6.5).

**Algorithm 6.5: Proportional-integral-differential controller**

```
integer reference ← ···                    // Reference distance
integer measured                           // Measured distance
integer error                              // Error
integer error-sum ← 0                      // Cumulative error
integer previous-error ← 0                 // Previous error
integer error-diff                         // Error difference
float gain-p ← ···                         // Proportional gain
float gain-i ← ···                         // Integral gain
float gain-d ← ···                         // Derivative gain
integer power                              // Motor power
```

```
1: error ← reference − measured            // Distances
2: error-sum ← error-sum + error           // Integral term
3: error-diff ← error − previous-error     // Differential term
4: previous-error ← error                  // Save current error
5: power ← gain-p * error +                // Control value
       gain-i * error-sum + gain-d * error-diff
6: left-motor-power ← power
7: right-motor-power ← power
```

The behavior of the PID controller is shown in Fig. 6.6. The robot smoothly and rapidly converges to the reference distance.

The gains of a PID controller must be carefully balanced. If the gains for the P and I terms are too high, oscillations can occur. If the gain for the D term is too high, the controller will react to short bursts of noise.

**Activity 6.5: PID controller**

- Implement a PID controller for the task of a robot approaching an object.
- Experiment with different gains until the robot smoothly approaches the reference distance.
- Repeat the experiments with a moving object.

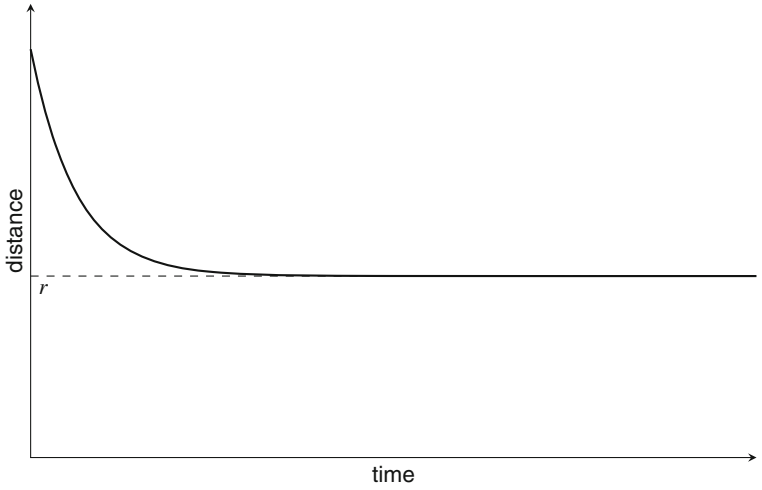



**Fig. 6.6** Behavior of the PID controller

## 6.6 Summary

A good control algorithm should converge rapidly to the desired result while avoiding abrupt motion. It must be computationally efficient, but not require constant tuning. The control algorithm has to be adapted to the specific requirements of the system and the task, and to function correctly in different environmental conditions. We have described four algorithms, from the impractical on-off algorithm through algorithms that combine proportional, integral and derivative terms. The proportional term ensures that large errors cause rapid convergence to the reference, the integral term ensures that the reference can actually be attained, while the derivative term makes the algorithm more responsive.

## 6.7 Further Reading

A modern textbook on control algorithms is [1].

## Reference

1. Åström, K.J., Murray, R.M.: Feedback Systems: An Introduction for Scientists and Engineers. Princeton University Press (2008). The draft of a second edition is available online at http://www.cds.caltech.edu/~murray/amwiki/index.php/Second_Edition

# Chapter 7
# Local Navigation: Obstacle Avoidance

A mobile robot must *navigate* from one point to another in its environment. This can be a simple task, for example, if a robot can follow an unobstructed line on the floor of a warehouse (Sect. 3.4), but the task becomes more difficult in unknown and complex environments like a rover exploring the surface of Mars or a submersible exploring an undersea mountain range. Even a self-driving car which travels along a road needs to cope with other cars, obstacles on the road, pedestrian crosswalks, road construction, and so on.

The navigation of a self-driving car can be divided into two tasks: There is the high-level task of finding a path between a starting position and a goal position. Before the development of modern computer systems, finding a path required you to study a map or to ask directions. Now there are smartphone applications which take a starting position and a goal position and compute paths between the two positions. If the application receives real-time data on traffic conditions, it can suggest which path will get you to your goal in the shortest time. The path can be computed offline, or, if you have a GPS system that can determine your current position, the path can be found in real-time and updated to take changing conditions into account.

A self-driving car must also perform the lower-level task of adapting its behavior to the environment: stopping for a pedestrian in a crosswalk, turning at intersections, avoiding obstacles in the road, and so on. While high-level path finding can be done once before the trip (or every few minutes), the low-level task of obstacle avoidance must be performed frequently, because the car never knows when a pedestrian will jump into the road or when the car it is following will suddenly brake.

Section 7.1 looks at the low-level task of obstacle avoidance. Section 7.2 shows how a robot can recognize markings while following a line so that it knows when it has reached its goal. Sections 7.3–7.5 demonstrate a higher-level behavior: finding a path without a map of the environment. This is done by analogy with a colony of ants locating a food source and communicating its location to all members of the colony.



M. Ben-Ari and F. Mondada, *Elements of Robotics*,
https://doi.org/10.1007/978-3-319-62533-1_7

## 7.1 Obstacle Avoidance

The algorithms presented so far have focused on detecting objects and moving towards them. When a robot moves toward a goal it is likely to encounter additional objects called *obstacles* that block the path and prevent the robot from reaching its goal. We assume that the robot is able to detect if there is an unobstructed path to the goal, for example, by detecting a light on the goal. This section describes three algorithms for obstacle avoidance, where the obstacles are walls that block the robot's movement:

- A straightforward wall following algorithm, which unfortunately will not work if there are multiple obstacles in the environment.
- An algorithm that can avoid multiple obstacles, but it must know the general direction of the goal (perhaps from its GPS system). Unfortunately, some obstacles can cause the robot to become trapped in a loop.
- The Pledge algorithm is a small modification of the second one that overcomes this erroneous behavior.

The algorithms will use the abstract conditional expressions wall-ahead and wall-right, which are true if there is a wall close to the front or to the right of the robot. The first algorithm will also use the conditional expression corner-right which is true if the robot is moving around an obstacle and senses a corner to its right. There are several ways of implementing these expressions which we pursue in Activity 7.1.

**Activity 7.1: Conditional expressions for wall following**

- Implement the conditional expression wall-ahead using a horizontal proximity or a touch sensor.
- Implement the conditional expression wall-right. This is easy to do with a sensor mounted on the right of the robot, or with a rotating distance sensor. If you have only a forward-facing proximity sensor, you can have the robot turn slightly to the right, detect the wall, if any, and then turn back again.
- Implement the conditional expression corner-right. This can be implemented as an extension of wall-right. When the value of wall-right changes from true to false, make a short right turn and check if wall-right becomes true again.

### *7.1.1 Wall Following*

Figure 7.1 shows a robot performing wall following by maintaining its position so that the wall is to its right (Algorithm 7.1). If a wall is detected ahead, the robot turns left so that the wall is to its right. If a wall is detected to the right, the robot continues

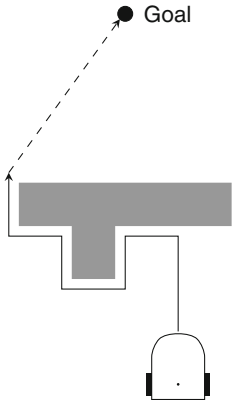



**Fig. 7.1** Wall following

moving along the wall. If a corner is detected, the robot turns right to continue moving around the obstacle. At the same time, the robot continually searches for the goal (black dot). When it detects the goal the robot moves directly towards it.

**Algorithm 7.1: Simple wall following**

```
1: while not-at-goal
2:     if goal-detected
3:         move towards goal
4:     else if wall-ahead
5:         turn left
6:     else if corner-right
7:         turn right
8:     else if wall-right
9:         move forward
10:    else
11:        move forward
```

Unfortunately, Algorithm 7.1 does not always work correctly. Figure 7.2 shows a configuration with *two* obstacles between the robot and the goal. The robot will never detect the goal so it will move around the first obstacle indefinitely.

**Activity 7.2: Simple wall following**

- Implement Algorithm 7.1 and verify that it demonstrates the behaviors shown in Figs. 7.1, 7.2.

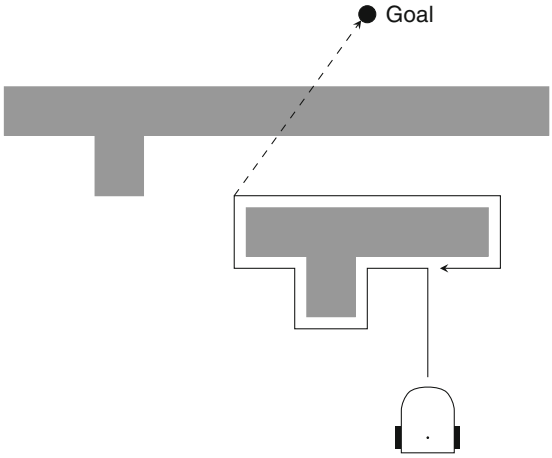



**Fig. 7.2** Simple wall following doesn't always enable the robot to reach the goal

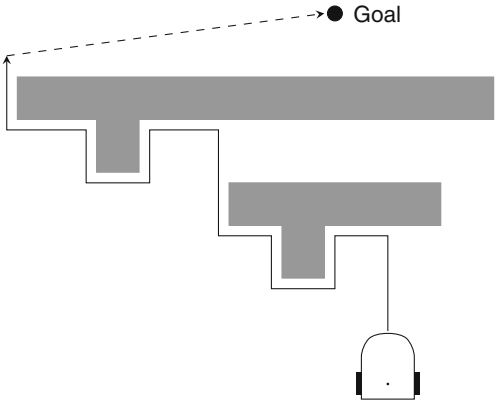



**Fig. 7.3** Wall following with direction

### *7.1.2 Wall Following with Direction*

The problem with Algorithm 7.1 is that it is a local algorithm that only looks at its immediate environment and does not take account of the fact that the higher-level navigation algorithm knows roughly the direction the robot should take to reach the goal. Figure 7.3 shows the behavior of a robot that "knows" that the goal is somewhere to its north so the robot moves at a heading of 0° relative to north. The wall following algorithm is only used if the robot cannot move north.

Algorithm 7.2 is similar to the previous algorithm except for its preference to move north if possible. It uses a variable heading to remember its current heading as

it moves around the obstacle. When heading is again north (a multiple of 360°), the robot moves forward instead of looking for a corner.

**Algorithm 7.2: Wall following**

```
    integer heading ← 0°
 1: while not-at-goal
 2:     if goal-detected
 3:         move towards goal
 4:     else if wall-ahead
 5:         turn left
 6:         heading ← heading + 90°
 7:     else if corner-right
 8:         if heading = multiple of 360°
 9:             move forward
10:         else
11:             turn right
12:             heading ← heading − 90°
13:     else if wall-right
14:         move forward
15:     else
16:         move forward
```

Unfortunately, the algorithm can fail when faced with a *G*-shaped obstacle (Fig. 7.4). After making four left turns, its heading is 360° (also north, a multiple of 360°) and it continues to move forward, encountering and following the wall again and again.

**Activity 7.3: Wall following with direction**

- Implement the wall following algorithm with direction and verify that it demonstrates the behavior shown in Fig. 7.4.

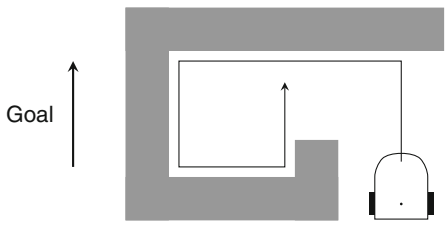



**Fig. 7.4** Why wall following with direction doesn't always work

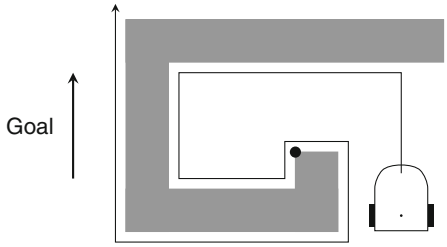



**Fig. 7.5** Pledge algorithm for wall following

- Run the simple wall following algorithm (Algorithm 7.1) with a *G*-shaped obstacle. What happens? Does this affect our claim that this algorithm is not suitable for obstacle avoidance?

### *7.1.3 The Pledge Algorithm*

The Pledge algorithm modifies line 8 of the wall following algorithm to:

if heading $= 0°$

The robot moves forward only when its cumulative heading is equal to 0° and not when it is moving north—a heading that is a multiple of 360°. The robot now avoids the "G"-shaped obstacle (Fig. 7.5): when its encounters the corner (black dot), it is moving north, but its heading is 360° after four left turns. Although 360° is a multiple of 360°, it is not equal to 0°. Therefore, the robot continues to follow the wall until four right turns subtract 360 so that the total heading is 0°.

**Activity 7.4: Pledge algorithm**

- Implement the Pledge algorithm and verify that it demonstrates the behavior shown in Fig. 7.5.

## 7.2 Following a Line with a Code

Let us return to the task of finding a path to a goal. If the path is marked by a line on the ground, line following algorithms (Sect. 3.4) can guide a robot within the

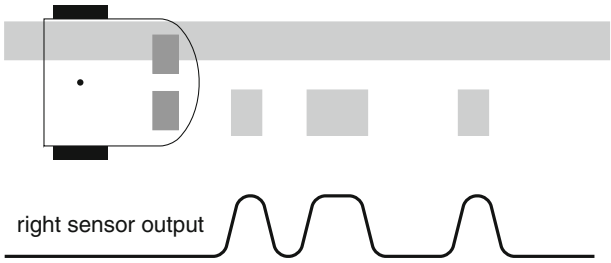



**Fig. 7.6** A robot following a line using its left sensor and reading a code with its right sensor

environment, but line following is not navigation. To navigate from one position to another we also need a localization algorithm so that the robot knows when it has reached its goals. We do not need a continuous localization algorithm like the ones in Chap. 8, we only need to know positions on the line that facilitate fulfilling the task. This is similar to navigating when driving: you only need to know about interchanges, intersections, major landmarks, and so on, in order to know where you are. Between such positions you can just follow the road.

Navigation without continuous localization can be implemented by reading a code placed on the floor next to the line. Figure 7.6 shows a robot with two ground sensors: the left one senses the line and the right one senses the code. Below the robot is a graph of the signal returned by the right sensor.

**Activity 7.5: Line following while reading a code**

- Implement line following with code reading as shown in Fig. 7.6.
- Write a program that causes a robot to follow a path.
- Place marks that encode values next to the path. The robot should display these values (using light, sound or a screen) when it moves over the codes.

**Activity 7.6: Circular line following while reading a code**

- Implement a clock using two robots, one for the minutes and one for the hours (Fig. 7.7).
- An alternate implementation would be to have the two robots move at different speeds so that one completes a revolution in one hour and the other completes a revolution in one day. Discuss the difference between the implementations.

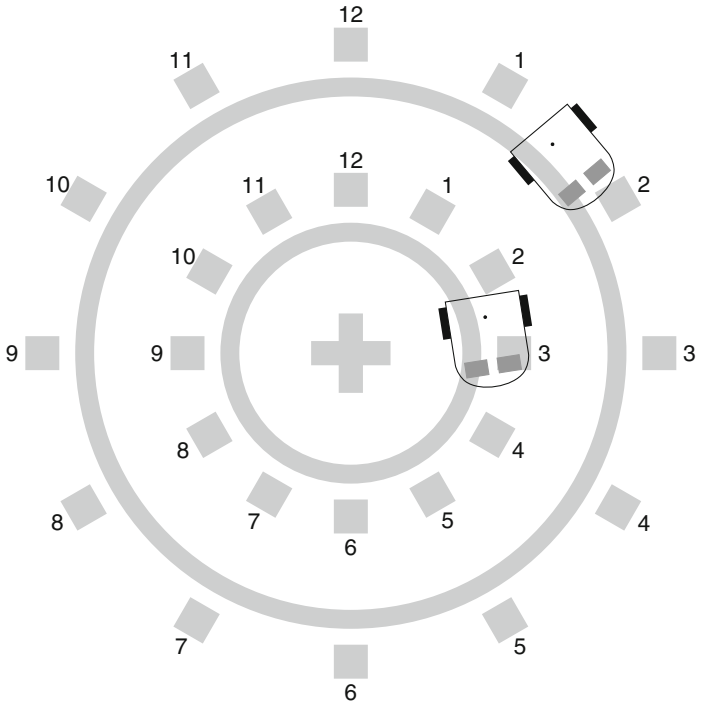



**Fig. 7.7** A robotic clock: one robot indicates the hour and the other indicates the minute

## 7.3 Ants Searching for a Food Source

Let us now return to the high-level algorithm of finding a path. If there is a line and a mechanism for localization like a code, the approach of the previous section can be used. However, even if a line does not exist, a robot may be able to *create* its own line. The interesting aspect of this method is that the robot does not need to know its location in the environment, for example, using a GPS; instead, it uses landmarks in the environment itself for navigation. The algorithm will be presented within the real-world context of ants searching for food:

> There exists a nest of ants. The ants search randomly for a source of food. When an ant finds food it returns directly to the nest by using landmarks and its memory of the path it took from the nest. During the return journey to the nest with the food, the ant deposits chemicals called *pheromones*. As more and more ants find the food source and return to the nest, the trail accumulates more pheromones than the other areas that the ants visit. Eventually, the amount of pheromones on the trail will be so strong that the ants can follow a direct path from the nest to the food source.

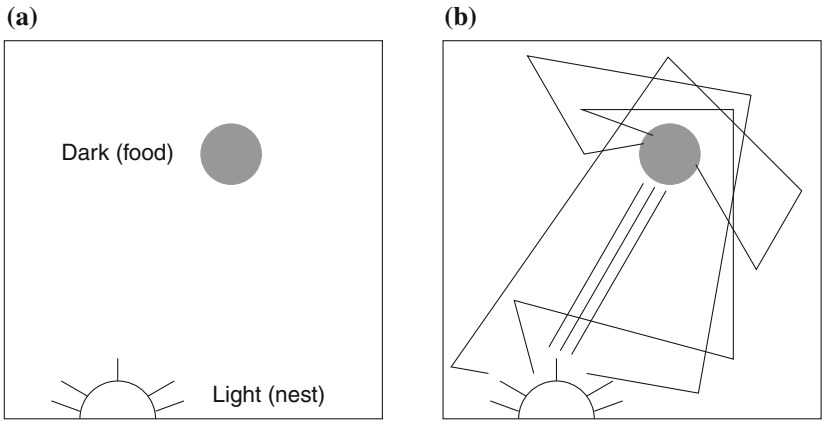



**Fig. 7.8** **a** The ants' nest and the food source. **b** Pheromones create a trail

Figure 7.8a shows the ants' nest in the lower left corner represented as a light that enables the ants to easily find their way back to the nest. The dark spot is the food source. Figure 7.8b shows three random trails that eventually discover the food source; then the ants return directly to the nest, leaving three straight lines of pheromones. This concentration can be subsequently used to find the food source directly.

The ant-like behavior can be implemented by a robot. Assume that there is a fixed area within which the robot can move. As in Fig. 7.8a there is a food source and a nest. The food source will be represented by a dark spot that can be easily detected by a ground sensor on the robot. The proximity sensors of the robot are used to detect the walls of the area. Activity 7.7 suggests two methods of representing the nest that depend on what additional sensors your robot has.

**Activity 7.7: Locating the nest**

- Implement a program that causes the robot to move to the nest no matter where it is placed in the area.
- Accelerometers: Mount the area on a slope such that one corner, the nest, is at the lowest point of the area.
- Light sensor: The nest is represented by a light source that can be detected by the light sensor regardless of the position and heading of the robot. If the light sensor is fixed and can detect light only from a certain direction, the robot will have to rotate to locate the light source.

Simulate the pheromones by covering the area with a sheet of white paper and attaching a black marker to the robot so that it draws a line wherever it moves. A

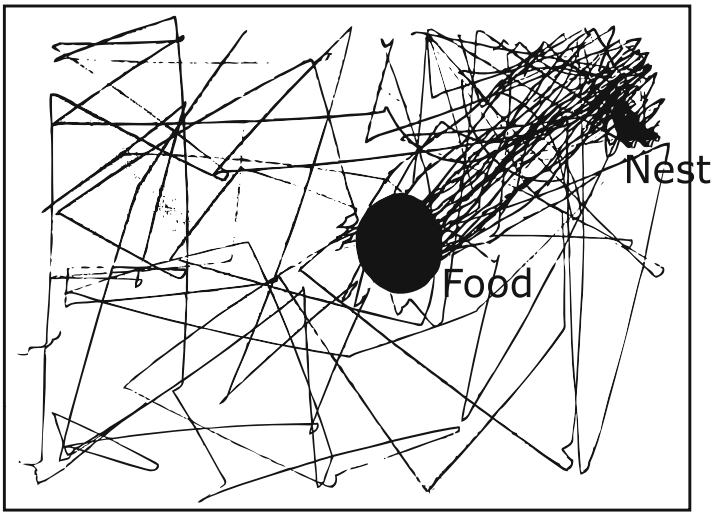



**Fig. 7.9** A robot simulating pheromones of ants

ground sensor detects the marks in the area. Figure 7.9 shows the lines resulting from the behavior of a robot running the algorithm. Activity 7.8 asks you to explore the ability of the robot to sense areas which have a high density of lines.

**Activity 7.8: Sensing areas of high density**

- In Sect. 3.4.3 we noted that sensors don't sense a single geometrical point but rather have an aperture that reads a relatively large area, perhaps even as much as a square centimeter (Fig. 3.6). Experiment with your ground sensor to see how the readings returned by the sensor depend on the width of the line. Can you come to any conclusion about the optimal width of the marker? If it is too thin the trail won't be detected and if it is too thick the markings of the random movement might be taken to be the trail.
- Represent the food source as a relatively large totally black spot and make sure that it gives a minimal reading of the ground sensor.
- Figure 7.9 shows that the trail between the food source and the nest has a high density. Experiment with various numbers of lines and define an effective threshold between the trail and areas of random motion outside the trail. See if you can get the robot to make darker lines by varying its motion or by moving back and forth along the trail.

## 7.4 A Probabilistic Model of the Ants' Behavior

A *model* is an abstraction of a system that shows how parameters impact phenomena. Models are used, for example, to study traffic patterns in order to predict the effect of new roads or traffic lights. To understand how the path from the nest to the food is generated, this section presents a simplified model the behavior of the ants.

The fundamental characteristic of the ants' behavior is that they do not have a map of their environment, so they must move randomly in order to search for the food source. Therefore, a model of their behavior must be probabilistic. Let us assume that the environment is a rectangular area that is a grid of cells. Figure 7.10 shows an area split into $6 \times 8 = 48$ cells.

**Coordinates in a grid of cells**

Throughout the book, the coordinates of a cell in a grid are given as (*row*, *column*). Rows are numbered from top to bottom and columns from left to right like matrices in mathematics, however, the numbering starts from 0, as in the array data type in computer science.

Without any information on how ants choose their movements, we assume that they can move in any direction with the same probability, so the probability $p$ of an ant being in any cell is 1 divided by the number of cells, here $p = 1/48 = 0.021$.

The probability that the ant is in the cell with the food source is $p$, the same as for any other cell. According to our specification of the ant's behavior, once it enters this cell and identifies the cell as the food source, it returns directly to the nest. In Fig. 7.10 the food source is in cell (3, 4), so an ant visiting that cell must return to the nest at cell (0, 7), passing through cells (2, 5) and (1, 6). What is the probability that the ant is in any of these three cells? There are two possibilities: either the ant is in the

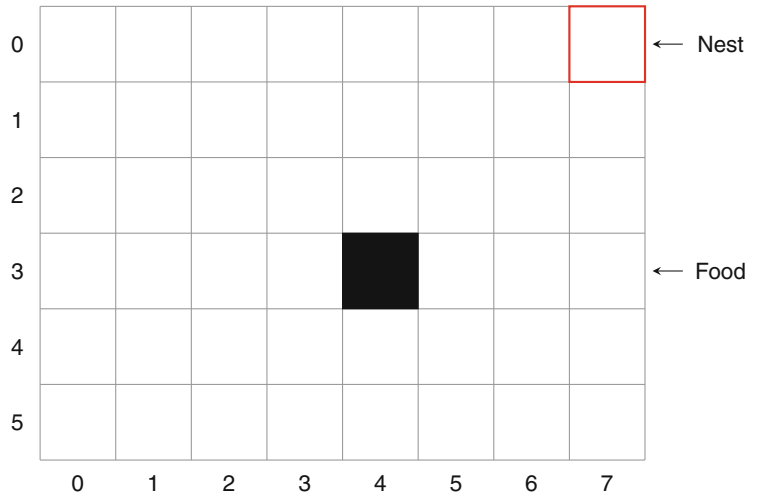



**Fig. 7.10** Representation of the environment as a grid of cells

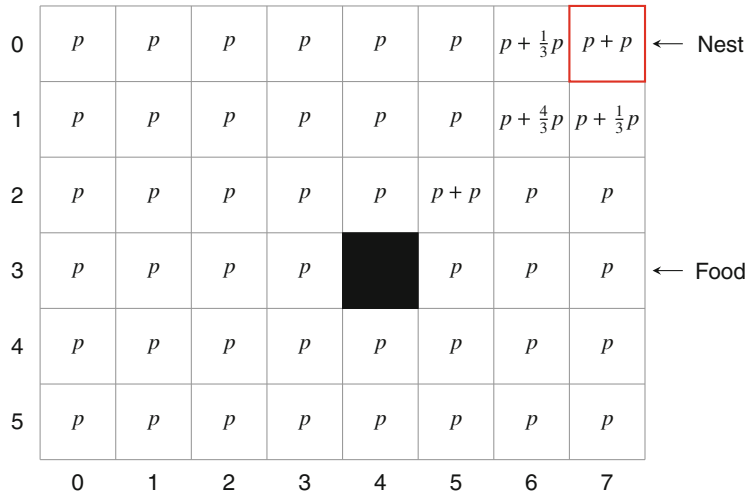



**Fig. 7.11** Probabilities for the location of the ant

cell because it randomly moved there with probability $p$, or the ant is there because it moved to the food source randomly with probability $p$ and then *with probability* 1 is moved towards the nest. Therefore, the total probability of being in any of those cells is $p + p \times 1 = p + p = 2p$.[1] If our robot is drawing lines as it moves, the cells on the diagonal should be twice as dark as the other cells.

Once the ant has reached the nest, it will move randomly again, that is, it will select a random neighbor to move to. In general a cell has eight neighbors (above and below, left and right, four on the diagonals), so the probability is $p/8$ that it will be in any one of these neighbors. The nest, however, is in the corner with only three neighbors, so the probability is $p/3$ that it will move to any one of them. Figure 7.11 shows the probability of the location of the ant after finding the food source, returning to the nest and making one additional random move. When implemented by a robot with a marker, the cells with higher probability will become darker (Fig. 7.12).

What can we conclude from this model?

- Although the ants move randomly, their behavior of returning to the nest after finding the food source causes the probability of being on the diagonal to be higher than anywhere else in the environment.
- Since the ants drop pheromones (black marks) at every cell they visit, it follows that the marks on the diagonal path between the food source and the nest will be darker than the marks on other cells. Eventually, the markings on this path will be sufficiently dark so that the robot can follow it to the food source without performing a random exploration.

[1] After the probabilities are updated they must be normalized as explained in Appendix B.2. For another example of normalization, see Sect. 8.4.

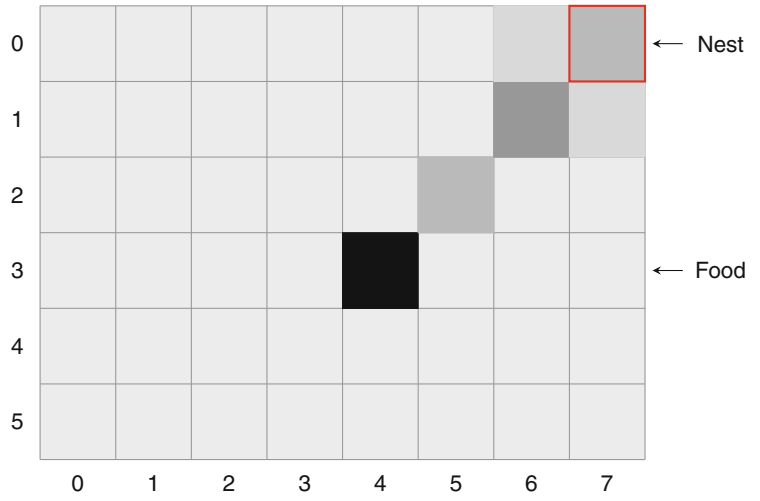



**Fig. 7.12** Probabilities for the location of a robot with a marker

- Since the robot visits the nest often, the cells in the immediate vicinity of the nest will have a probability somewhere between the uniform probability and the high probability of the trail. Therefore, it is important to emphasize the trail using methods such as those explored in Activity 7.8.

## 7.5 A Finite State Machine for the Path Finding Algorithm

An FSM for path finding by the ants is shown in Fig. 7.13. To save space the labels of the transitions use abbreviations which are explained in Table 7.1. Here is a detailed description of the behavior specified by this FSM in each state:

**search:** In this state the robot randomly searches for dark areas. It is the initial state and the transition true ⇝ fwd specifies that initially (and unconditionally) the robot is moving forwards and a timer is set to a random period. When the timer expires (timeout), the robot makes random turn, moves forwards and resets the timer. This random motion will continue until the robot encounters the wall of the area or a gray marking on the surface of the area. If it encounters a wall it makes a random turn *away* from the wall; we assume that the sensor faces directly ahead so the random turn must be in some direction to the side or the rear of the robot. Once the robot has detected a gray marking, it makes the transition to the follow state.

**follow:** The two self-transitions above and to the right of this state are transitions that implement line following (Sect. 3.4). There are three other transitions: Should a timeout occur without detecting gray, the robot is no longer following a line and must return to the search state. If the robot encounters a wall, we want it to turn away,

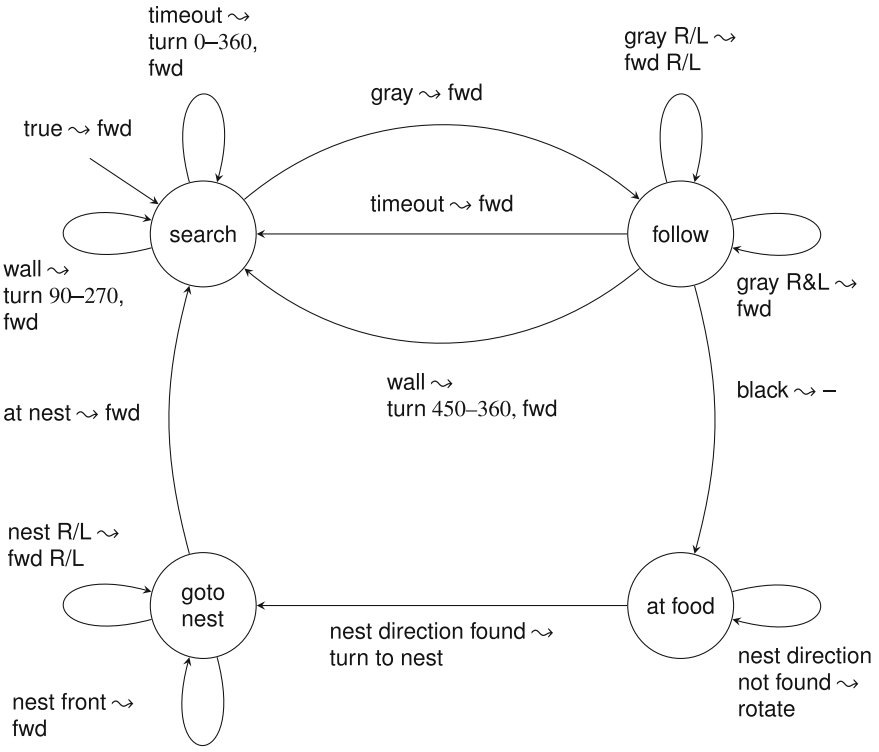



**Fig. 7.13** State machine for drawing a path between a food source and a nest see Table 7.1 for explanations of the abbreviations

**Table 7.1** Abbreviations in the state machine

| Abbreviation | Explanation |
|---|---|
| fwd | Set motor forwards |
| fwd R/L | Set motor forwards and to the right/left |
| | **fwd and fwd R/L also set the timer to a random period** |
| Wall | Wall detected |
| Timeout | Timer period expired |
| Gray R/L/R&L | Gray detected by right/left/both sensors |
| Nest front/R/L | Nest detected in front/right/left |
| Black | Black detected |
| Nest direction | Direction from food to nest found or not found |
| Turn $\theta_1 - \theta_2$ | Turn randomly in the range $\theta_1 - \theta_2$ |
| Rotate | The robot (or its sensor) rotates |

but first we ask it to make a full 360° turn to check if there is a gray marking in its vicinity. Therefore, the transition includes the action turn 450–360. Since the nest is next to a wall, this condition is also true when the robot returns to the nest. If the robot senses a high-density marking (black), it concludes that it has reached the food source and takes the transition to the state at food.

**at food:** Finally, the robot has discovered the food source. It must now return to the nest. We specified that the nest can be detected (Activity 7.7), but the robot's sensor does not necessarily face the direction of the nest. Therefore, the robot (or its sensor) must rotate until it finds the direction to the nest. When it does so, it turns towards the nest and takes the transition to the state goto nest.

**goto nest:** This state is similar to the follow state in that the robot moves forward towards the nest, turning right or left as needed to move in the direction of the nest. When it reaches the nest it returns to the search state.

Look again at Fig. 7.9 which shows an actual experiment with a robot running this algorithm. We see that there is a high density of lines between the nest and food source, but there is also a relatively high density of lines in the vicinity of the nest, not necessarily in the direction of the food source. This can cause to robot to return to random searching instead of going directly to the food source.

## 7.6 Summary

The obstacle avoidance algorithms use wall following algorithms that have been known since ancient times in the context of navigating a maze. When used for obstacle avoidance, various anomalies can cause the algorithms to fail, in particular, the $G$-shaped obstacle can trap a wall following algorithm. The Pledge algorithm overcomes this difficulty.

A colony of ants can determine a path between their nest and a food source without knowing their location and without a map by reinforcing random behavior that has a positive outcome.

## 7.7 Further Reading

There is a large literature on mazes that can be found by following the references in the Wikipedia article for *Maze*. The Pledge algorithm was discovered by 12-year-old John Pledge; our presentation is based on [1, Chap. 4]. A project based on ants following pheromones is described in [2].

## References

1. Abelson, H., diSessa, A.: Turtle Geometry: The Computer as a Medium for Exploring Mathematics. MIT Press, Cambridge (1986)
2. Mayet, R., Roberz, J., Schmickl, T., Crailsheim, K.: Antbots: A feasible visual emulation of pheromone trails for swarm robots. In: Dorigo, M., Birattari, M., Di Caro, G.A., Doursat, R., Engelbrecht, A.P., Floreano, D., Gambardella, L.M., Groß, R., Şahin, E., Sayama, H., Stützle, T. (eds.) Proceedings of Swarm Intelligence: 7th International Conference, ANTS 2010, Brussels, Belgium, 8–10 Sep. 2010, pp. 84–94. Springer Berlin Heidelberg (2010)



# Chapter 8
# Localization

Navigation by odometry (Sect. 5.4) is prone to errors and can only give an estimate of the real pose of the robot. Moreover, the further the robot moves, the larger the error in the estimation of the pose, especially in the heading. Odometry in a robot can be compared to walking with closed eyes, counting steps until we reach our destination. As with odometry, the further we walk, the more uncertain we are about our location. Even when counting steps, we should open our eyes from time to time to reduce the uncertainty of our location. For a robot, what does it mean to "count steps" and "open our eyes from time to time"? It means that for moving short distances odometry is good enough, but when moving longer distances the robot must determine its position relative to an external reference called a landmark. This process is called *localization*.

Section 8.1 starts with an activity that you can use to familiarize yourself with the relation between odometry and localization. Section 8.2 presents classical trigonometric techniques used by surveyors to determine the position of a location on earth by measuring angles and distances to positions whose location is known. Section 8.3 briefly surveys global positioning systems (GPS) which are now widely used for localization. There are environments where GPS is not effective: in buildings and when very precise positions are needed. Robots in these environments use probabilistic localization techniques that are described in Sects. 8.4–8.5.

## 8.1 Landmarks

Landmarks, such as lines on the ground or doors in a corridor can be detected and identified by the robot and used for localization. The following activity—which uses neither a computer nor a robot—will help you understand the importance of identifying landmarks to correct errors in odometry.



M. Ben-Ari and F. Mondada, *Elements of Robotics*,
https://doi.org/10.1007/978-3-319-62533-1_8

**Activity 8.1: Play the landmark game**

- Choose a path in your home that requires you to pass several doors, for example, from the bed in your room through a hallway to a sofa in the living room and then back.
- Close your eyes and walk along the path applying the following rules:
  - You get 30 points at the beginning of the activity.
  - From time to time you can open your eyes for one second; this action costs 1 point.
  - If you touch a wall it costs 10 points.
- How many points do you have when you complete the path?
- Is it better to open the eyes frequently or to walk the path without ever opening your eyes?

## 8.2 Determining Position from Objects Whose Position Is Known

In this section we describe two methods which a robot can use to determine its position by measuring angles and distances to an object whose position is known. The first method assumes that the robot can measure the distance to the object and its *azimuth*, the angle of the object relative to north. The second method measures the angles to the object from two different positions. Both methods use trigonometry to compute the coordinates of the robot relative to the object. If the absolute coordinates $(x_0, y_0)$ of the object are known, the absolute coordinates of the robot can then be easily computed.

### *8.2.1 Determining Position from an Angle and a Distance*

Figure 8.1 shows the geometry of a robot relative to an object. In the diagram the object is denoted by the large dot placed at the origin $(x_0, y_0)$ of a coordinate system. The azimuth of the robot $\theta$ is the angle between north and the forward direction of the robot; it can be measured by a compass. A laser scanner is used to measure the distance $s$ to the object and the angle $\phi$ between the forward direction of the robot and the object. The relative coordinates $\Delta x$ and $\Delta y$ can be determined by simple trigonometry:

$$\Delta x = s \sin(\theta - \phi), \quad \Delta y = s \cos(\theta - \phi) .$$

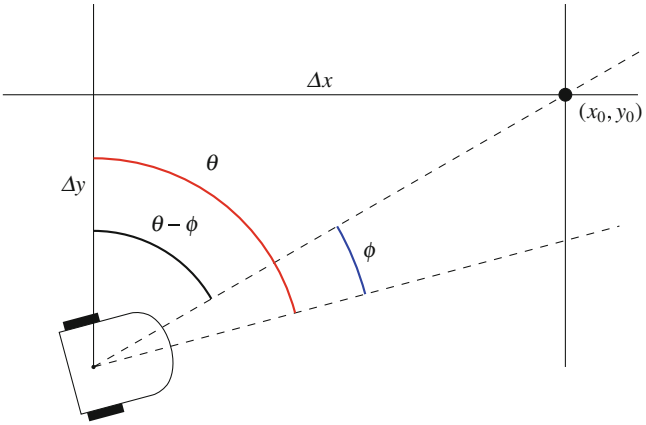



**Fig. 8.1** Determining position from an angle and a distance

From the known absolute coordinates of the object $(x_0, y_0)$, the coordinates of the robot can be determined.

**Activity 8.2: Determining position from an angle and a distance**

- Implement the algorithm.
- To measure the azimuth place the robot lined up with a edge of the table and call it north. Measure the angle to the object using several horizontal sensors or using one sensor and rotating either the sensor or the robot.
- The distance and the angle are measured from the position where the sensor is mounted, which is not necessarily the center of the robot. You may have to make corrections for this.

### *8.2.2 Determining Position by Triangulation*

*Triangulation* is used for determining coordinates when it is difficult or impossible to measure distances. It was widely used in surveying before lasers were available, because it was impossible to measure long distances accurately. The principle of triangulation is that from two angles of a triangle and the length of the included side, you can compute the lengths of the other sides. Once these are known, the relative position of a distant object can be computed.

Figure 8.2 shows the robot measuring the angles $\alpha$ and $\beta$ to the object from two positions separated by a distance $c$. If the two positions are close, the distance can be measured using a tape measure. Alternatively, the distance can be measured by

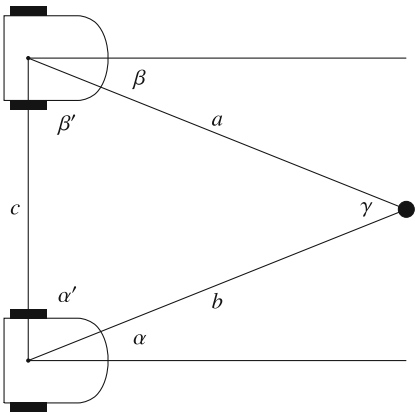



**Fig. 8.2** Triangulation

odometry as the robot moves from one position to another, though this may be less accurate. In surveying, if the coordinates of the two positions are known, the distance between them can be computed and used to determine the coordinates of the object.

The lengths $a$ and $b$ are computed using the *law of sines*:

$$\frac{a}{\sin\alpha'} = \frac{b}{\sin\beta'} = \frac{c}{\sin\gamma}\,,$$

where $\alpha' = 90° - \alpha,\ \beta' = 90° - \beta$ are the interior angles of the triangle. To use the law, we need $c$, which has been measured, and $\gamma$, which is:

$$\gamma = 180° - \alpha' - \beta' = 180° - (90° - \alpha) - (90° - \beta) = \alpha + \beta\,.$$

From the law of sines:

$$b = \frac{c\sin\beta'}{\sin\gamma} = \frac{c\sin(90° - \beta)}{\sin(\alpha + \beta)} = \frac{c\cos\beta}{\sin(\alpha + \beta)}\,.$$

A similar computation gives $a$.

**Activity 8.3: Determining position by triangulation**

- Implement triangulation.
- Initially, perform a measurement of the angle at one position and then pick the robot up and move it to a new position for the second measurement, carefully measuring the distance $c$.
- Alternatively, cause the robot move by itself from the first position to the second, calculating $c$ by odometry.

## 8.3 Global Positioning System

In recent years, the determination of location has been made easier and more accurate with the introduction of the *Global Positioning System (GPS)*.[1] GPS navigation is based upon orbiting satellites. Each satellite knows its precise position in space and its local time. The position is sent to the satellite by ground stations and the time is measured by a highly accurate atomic clock on the satellite.

A GPS receiver must be able to receive data from four satellites. For this reason, a large number of satellites (24–32) is needed so that there is always a line-of-sight between any location and at least four satellites. From the time signals sent by a satellite, the distances from the satellites to the receiver can be computed by multiplying the times of travel by the speed of light. These distances and the known locations of the satellites enable the computation of the three-dimensional position of the receiver: latitude, longitude and elevation.

The advantage of GPS navigation is that it is accurate and available anywhere with no additional equipment beyond an electronic component that is so small and inexpensive that it is found in every smartphone. There are two problems with GPS navigation:

- The position error is roughly 10 m. While this is sufficient to navigate your car to choose the correct the road at an intersection, it is not sufficient to perform tasks that need higher accuracy, for example, parking your car.
- GPS signals are not strong enough for indoor navigation and are subject to interference in dense urban environments.

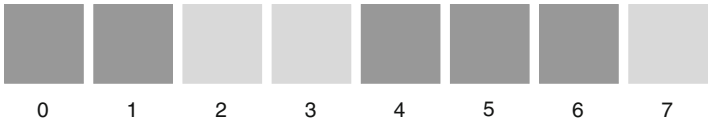
**Relativity and GPS**

You have certainly heard of Albert Einstein's theory of relativity and probably considered it to be an esoteric theory of interest only to physicists. However, Einstein's theories are used in the GPS computations! According to the special theory of relativity, the clocks on the satellites run *slower* than they do on the Earth (by 7.2 microseconds per day), because the satellites are moving fast relative to the Earth. According to the general theory of relativity, the clocks run *faster* (by 45.9 microseconds per day), because the force of the Earth's gravity is smaller at the distant satellite than it is for us on the surface. The two effects do not cancel out and a correction factor is used when broadcasting the time signals.

## 8.4 Probabilistic Localization

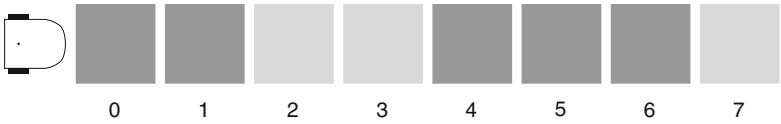
Consider a robot that is navigating within a known environment for which it has a *map*. The following map shows a wall with five doors (dark gray) and three areas where there is no door (light gray):

[1] The generic term is *global navigation satellite system (GNSS)* since GPS refers to the specific system operated by the United States. Similar systems are operated by the European Union (Galileo), Russia (GLONASS) and China (BeiDou), but GPS is frequently used to refer to all such systems and we do so here.

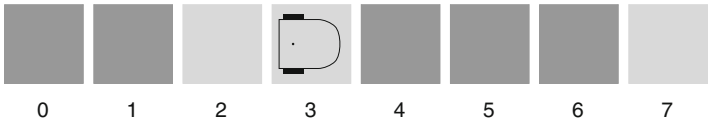



For clarity the doors and walls are drawn as if they are on the floor and the robot moves over them, measuring intensity with ground sensors.

The task of the robot is to enter a specific door, say the one at position 4. But how can the robot know where it is? By odometry, the robot can determine its current position given a known starting position. For example, if the robot is at the left end of the wall:




it knows that it has to move five times the width of each door, while if the robot is at the following position:



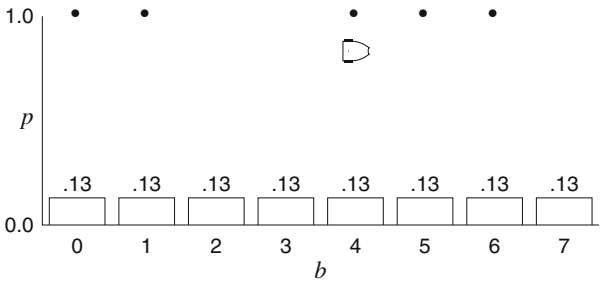

the required door is the next one to the right. Because of errors in odometry, it is quite likely that as time goes by the robot will get lost. In this section, we implement a one-dimensional version of a probabilistic *Markov localization algorithm* that takes into account uncertainty in the sensors and in the robot's motion, and returns the most probable locations of the robot.

Appendix B.1 contains a short tutorial on conditional probability and Bayes rule, including an example of the details of the calculations of uncertainty.

### *8.4.1 Sensing Increases Certainty*

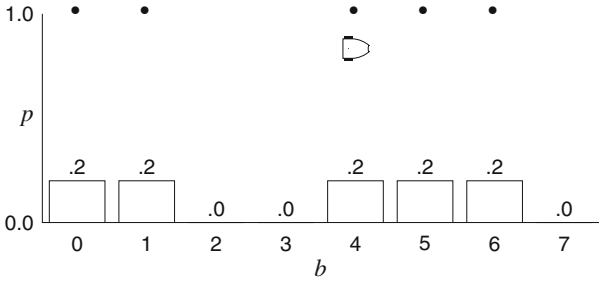
Consider a robot in the above environment of walls and doors that has no information as to its location. The robot assigns a probability to the eight positions where it might be located. Initially, it has no idea where it is, so each position will be assigned the probability $b[i] = 1.0/8 = 0.125 \approx 0.13$, where $b$ is called the *belief array*[2]:

[2] All probabilities will be rounded to two decimal digits for display.

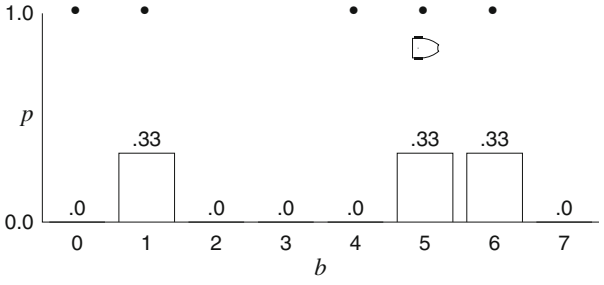



In the plots the dots denote the positions of the doors and a small icon indicates the actual position of the robot which is facing right.

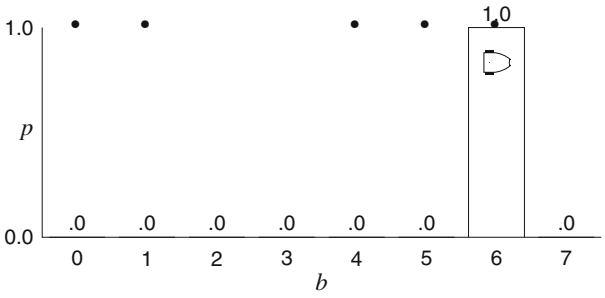
Suppose now that the robot's sensors detect a dark gray area. Its uncertainty is reduced, because it knows that it must be in front of one of the five doors. The belief array shows 0.2 for each of the doors and 0.0 for each of the walls:



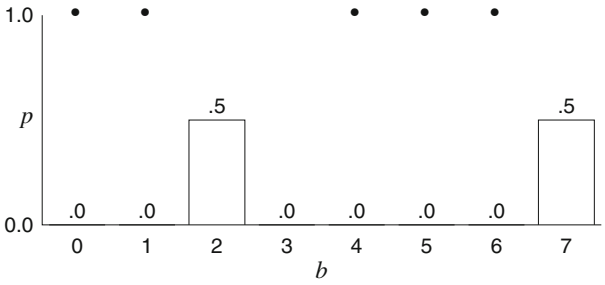

Next the robot moves forwards and again senses a dark gray area. There are now only three possibilities: it was at position 0 and moved to 1, it was at 4 and moved to 5, or it was at 5 and moved to 6. If the robot's initial position were 1 or 6, after moving right it would no longer detect a dark gray area, so it could not have been there. The probability is now 0.33 for each of the three positions 1, 4, 5:



After the next step of the robot, if it again detects a door, it is without doubt at position 6:




If the robot did not detect a door, it is either at position 2 or position 7:




The robot maintains a belief array and integrates new data when it detects the presence or absence of a door. As time goes on, the uncertainty decreases: the robot knows with greater certainty where it is actually located. In this example, eventually the robot knows its position in front of door 6 with complete certainty or it has reduced its uncertainty to one of the two positions 2, 7.

### *8.4.2 Uncertainty in Sensing*

The values returned by the robot's sensors reflect the intensity of the light reflected by the gray colors of the doors and walls. If the difference in the color of a dark gray door and a light gray wall is not very great, the robot may occasionally detect a dark gray door as a light gray wall, or conversely. This can occur due to changes in the ambient lighting or to errors in the sensors themselves. It follows that the robot cannot distinguish between the two with complete certainty.

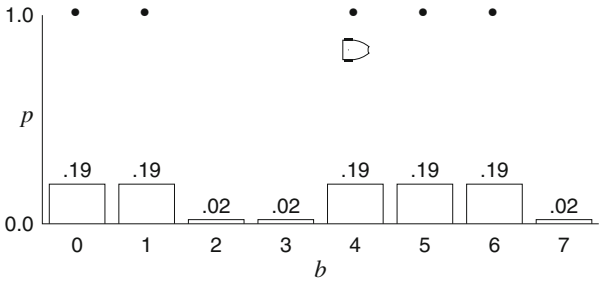
We model this aspect of the world by assigning probabilities to the detection. If the robot senses dark gray, we specify that the probability is 0.9 that it has correctly detected a door and 0.1 that it has mistakenly detected a wall where there was in fact a door. Conversely, if it senses light gray, the probability is 0.9 that it has correctly detected a wall and 0.1 that it has mistakenly detected a door where there was a wall.

**Table 8.1** Localization with uncertainty in sensing, sensor = after multiplying by the sensor uncertainty, norm = after normalization, right = after moving right one position

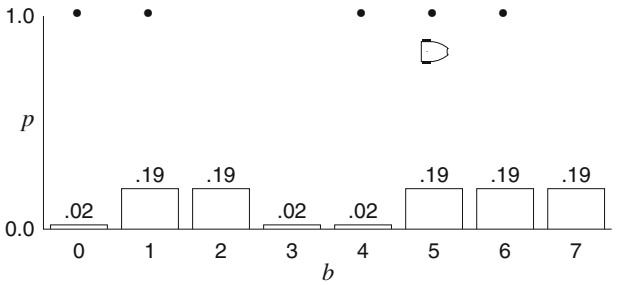
| Position door? | 0 • | 1 • | 2 | 3 | 4 • | 5 • | 6 • | 7 |
|---|---|---|---|---|---|---|---|---|
| Initial | 0.13 | 0.13 | 0.13 | 0.13 | 0.13 | 0.13 | 0.13 | 0.13 |
| Sensor | 0.11 | 0.11 | 0.01 | 0.01 | 0.11 | 0.11 | 0.11 | 0.01 |
| Norm | 0.19 | 0.19 | 0.02 | 0.02 | 0.19 | 0.19 | 0.19 | 0.02 |
| Right | 0.02 | 0.19 | 0.19 | 0.02 | 0.02 | 0.19 | 0.19 | 0.19 |
| Sensor | 0.02 | 0.17 | 0.02 | 0.00 | 0.02 | 0.17 | 0.17 | 0.02 |
| Norm | 0.03 | 0.29 | 0.03 | 0.00 | 0.03 | 0.29 | 0.29 | 0.03 |
| Right | 0.03 | 0.03 | 0.29 | 0.03 | 0.00 | 0.03 | 0.29 | 0.29 |
| Sensor | 0.03 | 0.03 | 0.03 | 0.00 | 0.00 | 0.03 | 0.26 | 0.03 |
| Norm | 0.07 | 0.07 | 0.07 | 0.01 | 0.01 | 0.07 | 0.63 | 0.07 |

We continue to display the computations in graphs but you may find it easier to follow them in Table 8.1. Each row represents the belief array of the robot following the action written in the first column.

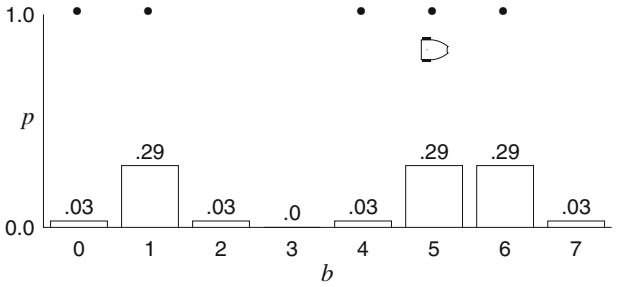
Initially, after sensing dark gray at a position where there is a door, we only know with probability $0.125 \times 0.9 = 0.1125$ that a door has been correctly detected; however, there is still a $0.125 \times 0.1 = 0.0125$ probability that it has mistakenly detected a wall. After normalizing (Appendix B.2), the belief array is:



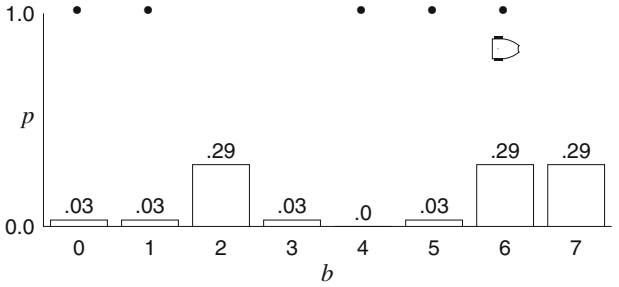

What happens when the robot moves one position to the right? Its belief array must also move one position to the right. For example, the probability 0.19 that the robot was at position 1 now becomes the probability that it is at position 2. Similarly, the probability 0.02 that the robot was at position 3 now becomes the probability that it is at position 4. The probability is now 0 that the robot is at position 0 and the probability $b_7$ becomes $b_8$, so the indices become 1–8 instead of 0–7. To simplify the computations and the diagrams in the example, the indices 0–7 are retained and the value of $b_8$ is stored in $b_0$ as if the map were cyclic. The belief array after the robot moves right is:



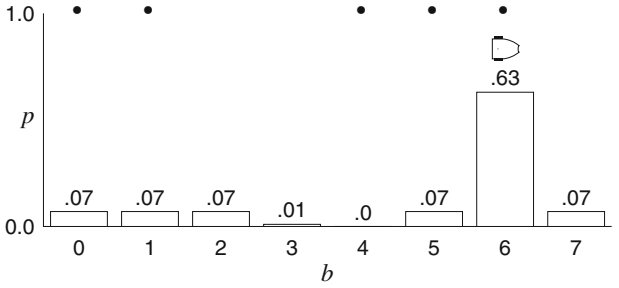

If the robot again senses dark gray, the probability of being at positions 1, 5 or 6 should increase. Computing the probabilities and normalizing gives:




Now the robot moves right again:




and senses a third dark gray area. The belief array becomes:




Not surprisingly, the robot is almost certainly at position 6.

**Activity 8.4: Localization with uncertainty in the sensors**

- Implement probabilistic localization with uncertainty in the sensor.
- How does the behavior of the algorithm change when the uncertainty is changed?
- Run the algorithm for different starting positions of the robot.

## 8.5 Uncertainty in Motion

As well as uncertainty in the sensors, robots are subject to uncertainty in their motion. We can ask the robot to move one position to the right, but it might move two positions, or it might move very little and remain in its current position. Let us modify the algorithm to take this uncertainty into account.

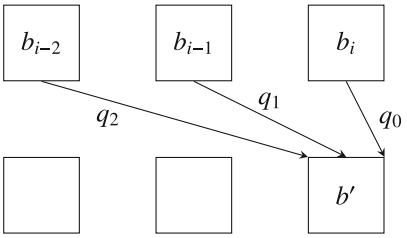
Let $b$ be the belief array. The belief array is updated using the formula:

$$b'_i = p_i \, b_i \, ,$$

where $b'_i$ is the new value of $b_i$ and $p_i$ is the probability of detecting a door (in the example, $p_i$ is 0.9 for $i = 0, 1, 4, 5, 6$ and $p_i$ is 0.1 for $i = 2, 3, 7$). If the motion is certain the robot moves one position to the right, but with uncertain motion, the following computation takes into account the probabilities $q_j$ that the robot actually moves $j = 0, 1, 2$ positions:

$$b'_i = p_i \, (b_{i-2} \, q_2 + b_{i-1} \, q_1 + b_i \, q_0) \, ,$$

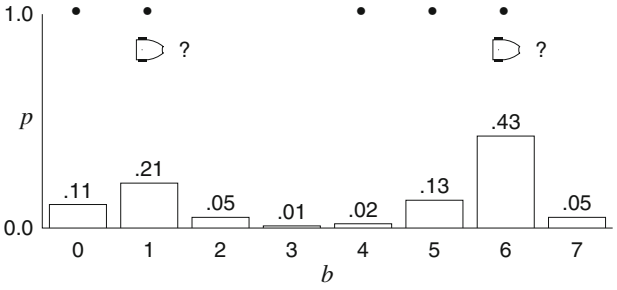
as shown in the following diagram:




It is highly likely that the robot will move correctly, so reasonable values are $q_1 = 0.8$ and $q_0 = q_2 = 0.1$. With these values for the uncertainty of the motion and the previous values for $p_i$, the calculation of the belief array after three moves is shown in Table 8.2 and its final value is shown in the following diagram:

**Table 8.2** Localization with uncertainty in sensing and motion, sensor = after multiplying by the sensor uncertainty, norm = after normalization, right = after moving right one position

| Position door? | 0 • | 1 • | 2 | 3 | 4 • | 5 • | 6 • | 7 |
|---|---|---|---|---|---|---|---|---|
| Initial | 0.13 | 0.13 | 0.13 | 0.13 | 0.13 | 0.13 | 0.13 | 0.13 |
| Sensor | 0.11 | 0.11 | 0.01 | 0.01 | 0.11 | 0.11 | 0.11 | 0.01 |
| Norm | 0.19 | 0.19 | 0.02 | 0.02 | 0.19 | 0.19 | 0.19 | 0.02 |
| Right | 0.05 | 0.19 | 0.17 | 0.04 | 0.04 | 0.17 | 0.19 | 0.17 |
| Sensor | 0.05 | 0.17 | 0.02 | 0.00 | 0.03 | 0.15 | 0.17 | 0.02 |
| Norm | 0.08 | 0.27 | 0.03 | 0.01 | 0.06 | 0.25 | 0.28 | 0.03 |
| Right | 0.06 | 0.12 | 0.23 | 0.05 | 0.01 | 0.07 | 0.23 | 0.25 |
| Sensor | 0.05 | 0.10 | 0.02 | 0.01 | 0.01 | 0.06 | 0.21 | 0.02 |
| Norm | 0.11 | 0.21 | 0.05 | 0.01 | 0.02 | 0.13 | 0.43 | 0.05 |




The robot is likely at position 6, but we are less certain because the probability is only 0.43 instead of 0.63. There is a non-negligible probability of 0.21 that the robot is at position 1.

**Activity 8.5: Localization with uncertainty in the motion**

- Implement probabilistic localization with uncertainty in the computation of the motor power.
- How does the behavior of the algorithm change when the uncertainty is changed?
- Run the algorithm for different starting positions of the robot.

## 8.6 Summary

Odometry provides an estimation of the position of a robot. A robot can use surveying techniques to compute its position relative to an object of known position. GPS gives excellent data on location, however, it may not be accurate enough and interference with reception from the satellites limits its use in indoor environments. If there are multiple known objects that the robot can sense and if it has a map of its environment, it can use probabilistic localization to estimate its position with high probability, although the probability will be reduced if there is a lot of uncertainty in the sensors or in the motion of the robot.

## 8.7 Further Reading

Probabilistic methods in robotics are treated in depth in [1]. There is a wealth of information on GPS at http://www.gps.gov. An implementation of probabilistic localization using the Thymio educational robot is described in [2].

## References

1. Thrun, S., Burgard, W., Fox, D.: Probabilistic Robotics. MIT Press, Cambridge (2005)
2. Wang, S., Colas, F., Liu, M., Mondada, F., Magnenat, S.: Localization of inexpensive robots with low-bandwidth sensors. In: Distributed Autonomous Robotic Systems (DARS). IEEE (2016)



# Chapter 9
# Mapping

We have seen that a robot can use its capability to detect obstacles to localize itself, based on information about the position of the obstacles or other information on the environment. This information is normally provided by a map. It is relatively easy to construct a map of industrial environments such as factories, since the machines are anchored in fixed locations. Maps are less relevant for a robotic vacuum cleaner, because the manufacturer cannot prepare maps of the apartments of every customer. Furthermore, the robots would be too difficult to use if customers had to construct maps of their apartments and change them whenever a piece of furniture is moved. It goes without saying that it is impossible to construct in advance maps of inaccessible locations like the ocean floor.

The solution is to have the robot build its own map of the environment. To build a map requires localization so that the robot knows where it is, but localization needs a map, which needs …. To overcome this chicken-and-egg problem, robots use *simultaneous localization and mapping (SLAM)* algorithms. To perform SLAM robots use information known to be valid even in unexplored parts of the environment, refining the information during the exploration.

This is what humans did to create geographic maps. Observations of the sun and stars were used for localization and maps were created as exploration proceeded. Initially, tools for localization were poor: it is relatively easy to measure latitude using a sextant to observe the height of the sun at noon, but accurate measurements of longitude were impossible until accurate clocks called chronometers were developed in the late eighteenth century. As localization improved so did maps, including not only land and seacoasts, but also terrain features like lakes, forests and mountains, as well as artificial structures like buildings and roads.

Sections 9.1 and 9.2 introduce methods of representing maps in a computer. Section 9.3 describes how a robot can create a map using the frontier algorithm. Section 9.4 explains how partial knowledge of the environment helps in constructing



M. Ben-Ari and F. Mondada, *Elements of Robotics*,
https://doi.org/10.1007/978-3-319-62533-1_9

a map. A SLAM algorithm is the subject of the final three sections. The algorithm is first presented in Sect. 9.5 using a relatively simple example. Activities for SLAM are collected in Sects. 9.6 and 9.7 explains the formal algorithm.

## 9.1 Discrete and Continuous Maps

We are used to graphical maps that are printed on paper, or, more commonly these days, displayed on computers and smartphones. A robot, however, needs a non-visual representation of a map that it can store in its memory. There are two techniques for storing maps: discrete maps (also called *grid maps*) and continuous maps.

Figure 9.1a shows an $8 \times 8$ grid map with a triangular object. The location of the object is stored as a list of the coordinates of each grid cell covered by the object. The object in the figure consists of the cells at:

$$(5, 3),\ (5, 4),\ (5, 5)\ ,\ (4, 5)\ ,\ (5, 6),\ (4, 6),\ (3, 6)\ .$$

Figure 9.1b shows a *continuous map* of the same object. Instead of storing the positions of the object, the coordinates of boundary positions are stored:

$$A = (6, 3),\ B = (3, 7),\ C = (6, 7)\ .$$

Discrete maps are not very accurate: it is hard to recognize the object in Fig. 9.1a as a triangle. To improve accuracy, a finer grid must be used: $16 \times 16$ or even $256 \times 256$. Of course, as the number of grid points increases, so must the size of the memory in the robot. In addition, a more powerful computer must be used to process the grid

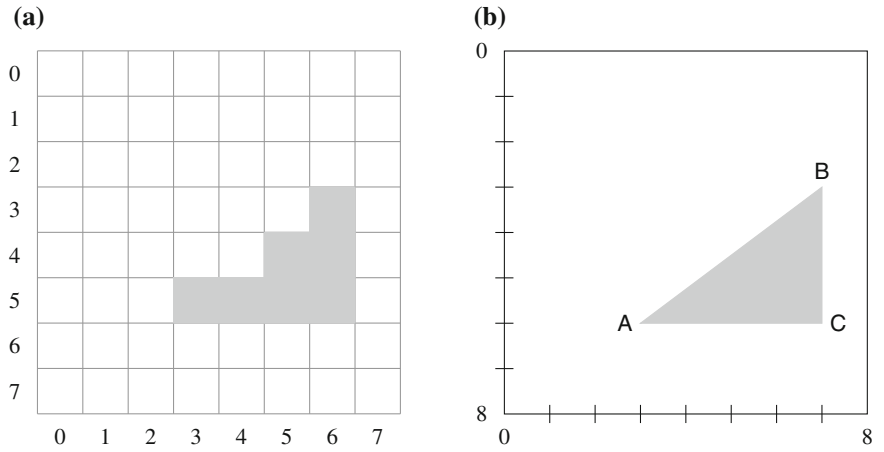



**Fig. 9.1** **a** A discrete map of the occupied cells of an object, **b** A continuous map of the same object

cells. Mobile robots have constraints on weight, cost, battery capacity, and so on, so very fine grids may not be practical.

If the objects in the environment are few and have a simple shape, a continuous map is more efficient, in addition to being more accurate. In Fig. 9.1b, three pairs of numbers represent the triangle with far greater accuracy than the seven pairs of the discrete map. Furthermore, it is easy to compute if a point is in the object or not using analytic geometry. However, if there are many objects or if they have very complex shapes, continuous maps are no longer efficient either in memory or in the amount of computation needed. The object in Fig. 9.1b is bounded by straight lines, but if the boundary were described by high-order curves, the computation would become difficult. Consider a map with 32 objects of size one, none of which touch each other. The discrete map would have 32 coordinates, while the continue map would need to store the coordinates of the four corners of each object.

In mobile robotics, discrete maps are commonly used to represent maps of environments, as we did in Chap. 8.

## 9.2 The Content of the Cells of a Grid Map

A geographic map uses conventional notations to describe the environment. Colors are used: green for forests, blue for lakes, red for highways. Symbols are used: dots of various sizes to denote villages and towns, and lines for roads, where the thickness and color of a line is used to indicate the quality of the road. Robots use grid maps where each cell stores a number and we need to decide what a number encodes.

The simplest encoding is to allocate one bit to each cell. A value of 1 indicates that an object exists in that cell and a value of 0 indicates that the cell is empty. In Fig. 9.1a, gray represents the value 1 and white represents the value 0.

However, sensors are not accurate and it may be difficult to be certain if a cell is occupied by an object or not. Therefore, it makes sense to assign a probability to each cell indicating how certain we are that the object is in that cell. Figure 9.2 is a copy of Fig. 9.1a with the probabilities listed for each cell. Cells without a number are assumed to have a probability of 0.

It can be seen that cells with a probability of at least 0.7 are the ones considered in Fig. 9.1a to be cells occupied by the object. Of course, we are free to choose some other threshold, for example 0.5, in which case more cells are considered to be occupied. In this example, we know that the object is triangular, so we can see that a threshold of 0.5 makes the object bigger than it actually is, while the higher threshold 0.7 gives a better approximation.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | | | | | | | | |
| 1 | | | | | | | | |
| 2 | | | | | | .4 | .6 | .3 |
| 3 | | | .1 | .2 | .4 | .6 | .9 | .5 |
| 4 | | | .3 | .6 | .5 | .7 | .8 | .6 |
| 5 | | | .4 | .8 | 1.0 | 1.0 | .9 | .5 |
| 6 | | | .2 | .3 | .2 | .3 | .3 | .1 |
| 7 | | | | | | | | |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

**Fig. 9.2** A probabilistic grid map

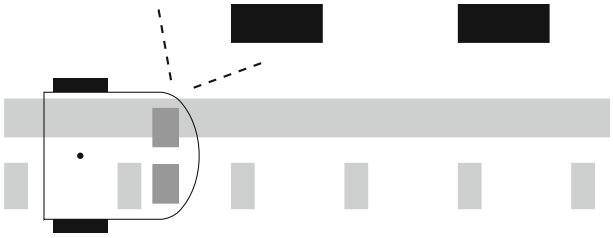


**Fig. 9.3** The *black rectangles* are the obstacles to be measured. The *gray line* guides the robot and the *gray marks* are used for localization

**Activity 9.1: Probabilistic map of obstacles**

- Place your robot on a line in front of a set of obstacles that the robot can detect with a lateral sensor (Fig. 9.3). If you implemented localization as described in Activity 8.4, you can use this information to establish where the robot is. Otherwise, draw regular lines on the ground that can be read by the robot for localization. Build a probabilistic map of the obstacles.
- How do the probabilities change when obstacles are added or removed?

## 9.3 Creating a Map by Exploration: The Frontier Algorithm

Consider a robotic vacuum cleaner newly let loose in your apartment. Obviously, it does not come pre-programmed with a map of your apartment. Instead, it must explore the environment to gather information that will be used to construct its own map. There are several ways of exploring the environment, the simplest of which is random exploration. The exploration will be much more efficient if the robot has a partial map that it can use to guide its exploration.

### *9.3.1 Grid Maps with Occupancy Probabilities*

The map in Fig. 9.4 is a grid map where each cell is labeled with its *obstacle probability*, which is the probability that there is an obstacle in the cell. The obstacle can be a wall, a table or anything that does not allow the robot to pass through this cell. The question marks represent cells that have not yet been explored. In the absence of any knowledge about the contents of a cell, we can assume that the probability that there is an obstacle there is 0.5, since it could just as easily be occupied or not. A question mark is used instead of the value 0.5 to clarify the unexplored status of the cells.

The center of the map is free from obstacles and the occupancy probabilities of these cells, called *open cells*, are low (0.1 or 0.2). There are three known obstacles, at the top right, the top left and the bottom center. The obstacles are characterized by high occupancy probabilities (0.9 or 1.0) and are denoted by gray cells. A *frontier cell* is an open cell that is adjacent (left, right, up, down) to one or more unknown

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? |
| ? | ? | ? | ? | ? | 1 | ? | ? | ? | ? | ? | 0.9 | 1 | 0.9 | ? | ? |
| ? | ? | ? | ? | ? | 1 | 0.1 | 0.1 | ? | ? | ? | 1 | 0.2 | 1 | ? | ? |
| ? | ? | ? | ? | ? | 1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.2 | 1 | ? | ? |
| ? | ? | ? | ? | ? | 0.9 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.2 | 1 | ? | ? |
| ? | ? | ? | ? | ? | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | ? | ? | ? |
| ? | ? | ? | ? | ? | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | ? | ? | ? |
| ? | ? | ? | ? | ? | ? | ? | 0.2 | 0.1 | 0.1 | 0.2 | 0.2 | 0.1 | ? | ? | ? |
| ? | ? | ? | ? | ? | ? | ? | 1 | 1 | 0.9 | 1 | ? | ? | ? | ? | ? |
| ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? |
| ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? |

**Fig. 9.4** Grid map of an environment with occupancy probabilities

cells. The set of frontier cells is called the *frontier*. The red lines of small squares in Fig. 9.4 represent the boundary between the frontier and the unknown cells in the environment. The unknown cells adjacent to the frontier are the interesting ones that should be explored in order to expand the current map.

### 9.3.2 The Frontier Algorithm

The *frontier algorithm* is used to expand the map by exploring the frontier. The robot moves to the closest frontier cell, senses if there are obstacles in unknown adjacent cells and updates the map accordingly.

The grid map in Fig. 9.5 is the same as the map in Fig. 9.4 with the addition of the robot which occupies a cell colored blue. The frontier cell closest to the robot is the cell two steps above its initial location. The arrow shows that the robot has moved to that cell. The robot uses its local sensors to determine if there are obstacles in adjacent unknown cells. (The sensors can detect obstacles in all eight adjacent cells, including the ones on the diagonal.) Suppose that the cell to its upper left certainly contains an obstacle (probability 1.0), while the cells directly above and to the right almost certainly do not contain an obstacle (probability 0.1). Figure 9.6 shows the map updated with this new information and the new position of the frontier.

Figure 9.7 shows the result of the next iteration of the algorithm. The robot has moved up one cell to the closest frontier cell, detected obstacles in the two adjacent unknown cells and updated the map. The upper right obstacle is completely known and there is no frontier cell in the vicinity of the current position of the robot.

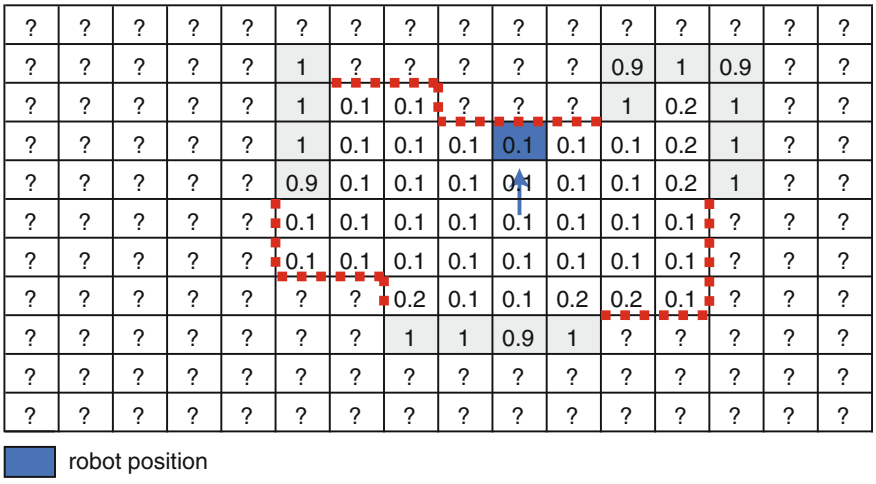



**Fig. 9.5** The robot moves to the frontier

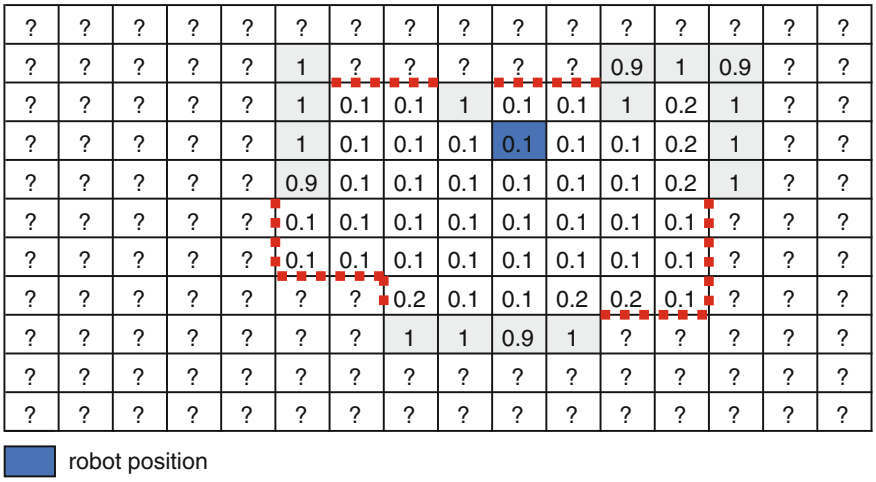



**Fig. 9.6** The robot updates unknown cells adjacent to the frontier

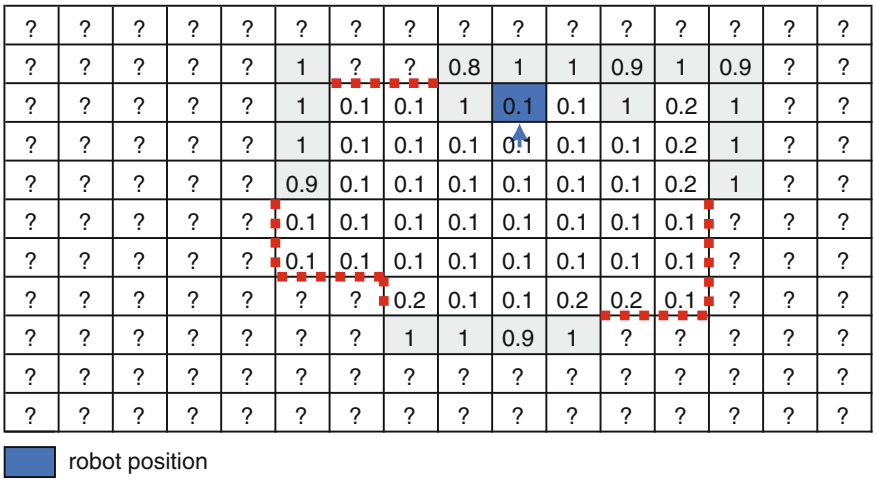



**Fig. 9.7** Second iteration of the frontier algorithm

Figure 9.8 shows the next iteration of the algorithm. The robot is blocked by the upper right obstacle and has to avoid it as it moves to the nearest frontier cell.

Figure 9.9 shows the completed map constructed by the robot after it has explored the entire frontier as shown by the path with the blue arrows.

Algorithm 9.1 formalizes the frontier algorithm. For simplicity, the algorithm recomputes the frontier at each step. A more sophisticated algorithm would examine the cells in a neighborhood of the robot's position and add or remove the cells whose status as frontier cells has changed.

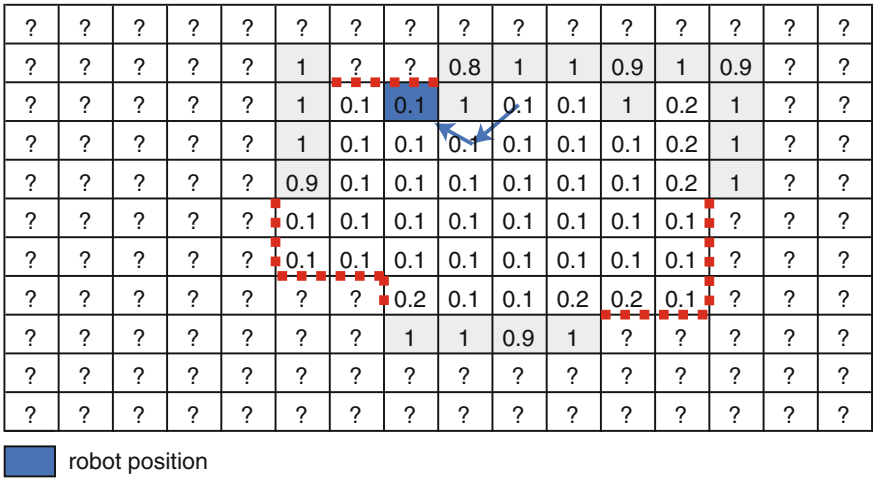



**Fig. 9.8** The robot avoids an obstacle while moving to the next frontier

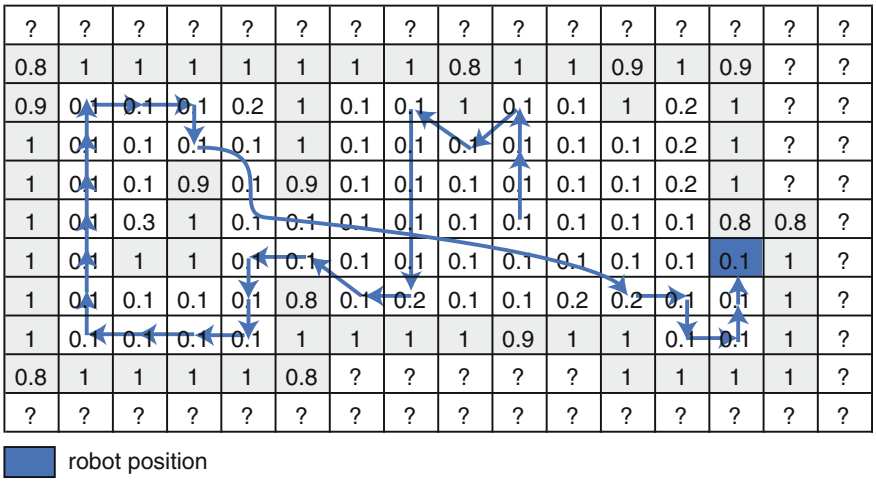



**Fig. 9.9** The map constructed by the frontier algorithm and the path explored by the robot

The example to which we applied the frontier algorithm is a relatively simple environment consisting of two rooms connected by a door (at the sixth column from the left in Fig. 9.9), but otherwise closed to the outside environment. However, the frontier algorithm works in more complex environments.

The frontier algorithm can be run in parallel by multiple robots. Each robot will explore that portion of the frontier that is closest to its position. The robots share their partial maps so that consistency of the maps is maintained. Since each robot explores a different area of the environment, the construction of the map will be much more efficient.

**Algorithm 9.1: Frontier algorithm**

```
        float array grid                    // Grid map
        cell list frontier                  // List of frontier cells
        cell robot                          // Cell with robot
        cell closest                        // Closest cell to robot
        cell c                              // Index over cells
        float low                           // Low occupancy probability
 1: loop
 2:     frontier ← empty
 3:     for all known cells c in the grid
 4:         if grid(c) < low and
 5:             exists unknown neighbor of c
 6:                 append c to frontier
 7:     exit if frontier empty

 8:     closest ← cell in frontier nearest robot
 9:     robot ← closest
10:     for all unknown neighbors c of closest
11:             sense if c is occupied
12:             mark grid(c) with occupancy probability
```

### *9.3.3 Priority in the Frontier Algorithm*

The frontier algorithm can use criteria other than distance to choose which frontier to explore. Consider the exploration of the grid map shown in Fig. 9.10. The robot is at cell (3, 3) marked with the blue circle. There are six known obstacle cells and five known open cells, of which the three cells (1, 3), (2, 2), (3, 2), marked with red squares, are the frontier cells. (Here the diagonal neighbors are not considered as adjacent.)

In Algorithm 9.1, the robot uses distance to a frontier cell as the criterion for deciding where to move. In Fig. 9.10 the cell to the robot's left at (3, 2) is the closest cell since it is only one step away, while the other two frontier cells are two steps away. We can consider a different criterion by taking into account the number of unknown cells adjacent to a frontier cell. Starting with a frontier cell with more

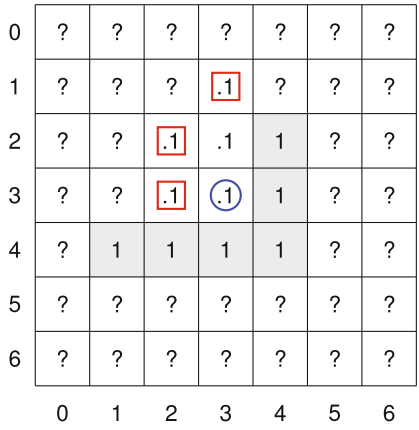



**Fig. 9.10** Exploration of a labyrinth

unknown cells might make the algorithm more efficient. We define the priority of a frontier cell as:

$$p_{cell} = \frac{a_{cell}}{d_{cell}},$$

where $a_{cell}$ is the number of adjacent unknown cells and $d_{cell}$ is the distance from the robot. The priorities of the three frontier cells are:

$$p_{(3,2)} = 1/1 = 1, \quad p_{(2,2)} = 2/2 = 1, \quad p_{(1,3)} = 3/2 = 1.5\,.$$

The priority of cell (1, 3) is the highest and the exploration starts there.

**Activity 9.2: Frontier algorithm**

- Implement the frontier algorithm. You will need to include an algorithm to move accurately from one cell to another and an algorithm to move around obstacles.
- Run the program on the grid map in Fig. 9.9. Do you get the same path? If not, why not?
- Run the program on the grid map in Fig. 9.10.
- Modify Algorithm 9.1 to use the priority described in Sect. 9.3.3. Is the path different from the one generated by the original program?
- Try to implement the frontier algorithm on your robot. What is the most difficult aspect of the implementation?

## 9.4 Mapping Using Knowledge of the Environment

Now that we know how to explore an environment, let us consider how to build a map during the exploration. In Chap. 8 we saw that a robot can localize itself with the help of external landmarks and their representation in a map. Without such external landmarks, the robot can only rely on odometry or inertial measurement, which are subject to errors that increase with time (Fig. 5.6). How is it possible to make a map when localization is subject to large errors?

Even with bad odometry, the robot can construct a better map if it has some information on the structure of the environment. Suppose that the robot tries to construct the plan of a room by following its walls. Differences in the real speeds of the left and right wheels will lead the robot to conclude that the walls are not straight (Fig. 9.11a), but if the robot knows *in advance* that the walls are straight and perpendicular to each other, the robot can construct the map shown in Fig. 9.11b. When it encounters a sharp turn, it understands that the turn is a 90° corner where two walls meet, so its mapping of the angles will be correct. There will also be an error when measuring the lengths of the walls and this can lead to the gap shown in the figure between the first and last walls. The figure shows a small gap which would not be important, but if the robot is mapping a large area, the problem of *closing a loop* in a map is hard to solve because the robot has only a local view of the environment.

Consider a robotic lawnmower given the task of mowing a lawn by moving back and forth; it has to close the loop by returning to its charging station (Fig. 9.12). It is not possible to implement this behavior using odometry alone, since small errors in velocity and heading lead to large errors in the position of the robot. It is highly unlikely that through odometry alone the robot will mow the entire surface of the lawn and return to its charging station. Landmarks such as signaling cables in the ground need to be used to close the loop.

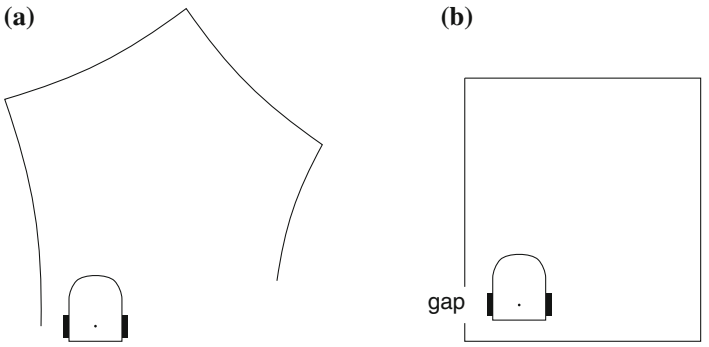



**Fig. 9.11** **a** Perceived motion of a robot based on odometry, **b** Odometry together with knowledge of the geometry of the walls

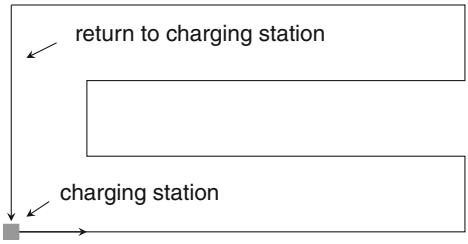



**Fig. 9.12** A robotic lawnmower mowing an area and returning to its charging station

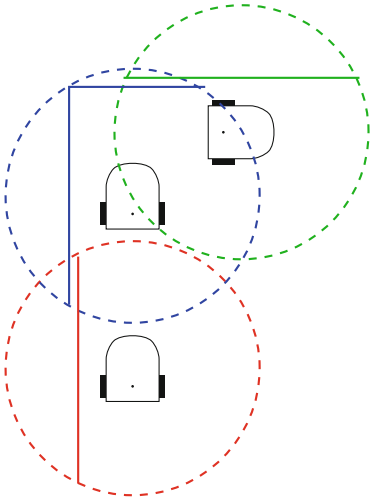


**Fig. 9.13** Long range sensor measurements can detect overlap

Map construction can be significantly improved by using sensor data that can give information on regular features in the environment, in particular, at a long range. The regular features can be lines on the ground, a global orientation, or the detection of features that *overlap* with other measurements. Suppose that we have a distance sensor that can measure distances over a large area (Fig. 9.13). The measurement over a large area enables the robot to identify features such as walls and corners from a measurement taken at a single location. Large area measurements facilitate identifying overlaps between the local maps that are constructed at each location as the robot moves through the environment. By comparing local maps, the localization can be corrected and map accurately updated. This is the topic of the SLAM algorithm described in the next section.

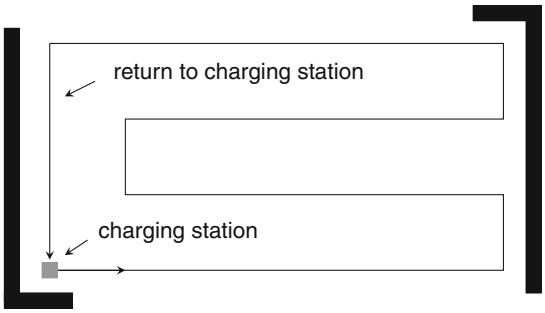



**Fig. 9.14** A robotic lawnmower with landmarks

**Activity 9.3: Robotic lawnmower**

- Write a program that causes a robotic lawnmower to move along a path using odometry alone (Fig. 9.12). Run the program several times. Does the robot return to the charging station? If not, how large are the errors? Are the errors consistent or do they change from one run to the next?
- Place landmarks of black tape around the lawn (Fig. 9.14). Program your robot to recognize the landmarks and correct its localization. How large do the landmarks have to be so that the robot detects them reliably?

## 9.5 A Numerical Example for a SLAM Algorithm

The SLAM algorithm is quite complicated so we first compute a numerical example and later give the formal presentation.

Figure 9.15a shows a robot in a room heading towards the top of the diagram. The robot is near a corner of the room and there is a projection from the wall to the robot's left, perhaps a supporting pillar of the building. Figure 9.15b is the corresponding map. The large dot shows the position of the robot and the associated arrow shows its heading. The thick dotted line represents the real wall. The white cells represent locations that are known to be free, the gray cells represent obstacles, and the cells with question marks are still unexplored. A cell is considered to be part of the obstacle if a majority of the area of the cell is behind the wall. For example, the two horizontal segments of the projection from the wall are near the boundary of the cells they pass through, but these cells are considered part of the obstacle because almost all their area is behind the wall.

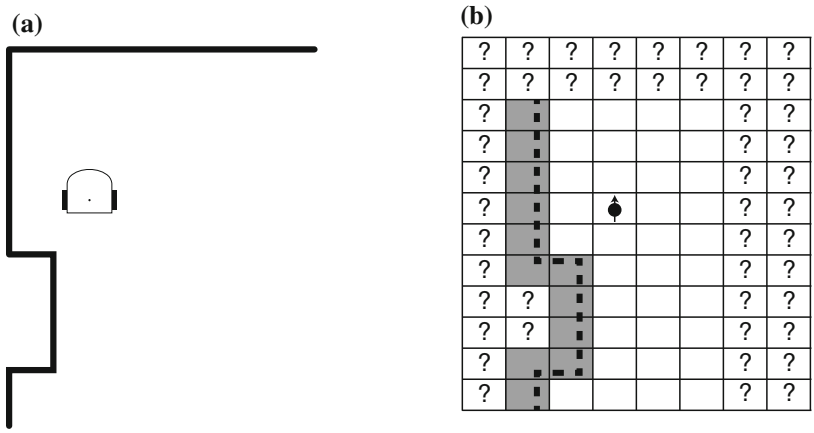



**Fig. 9.15** **a** A robot near the wall of a room, **b** The corresponding map

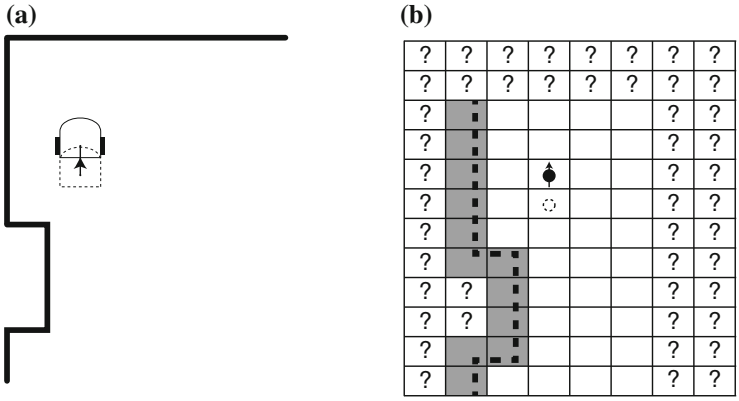



**Fig. 9.16** **a** The intended motion of the robot, **b** The map for the intended motion

For the purpose of presenting the details of the SLAM algorithm, the map is highly simplified. First, the cells are much too large, roughly the same size as the robot itself. In practice, the cells would be much smaller than the robot. Second, we specify that each (explored) cell is either free (white) or it contains an obstacle (gray); real SLAM algorithms use a probabilistic representation (Sect. 9.2).

Suppose that the robot in Fig. 9.15a intends to move forwards to the new position shown in Fig. 9.16a. Figure 9.16b shows the map corresponding to the position after the intended move, where the robot has moved the height of one cell up from its initial position. Unfortunately, the right wheel moves over an area of low friction and although the robot ends up in the correct position, its heading is too far to the right. The actual position of the robot is shown in Fig. 9.17a and b is the corresponding map.

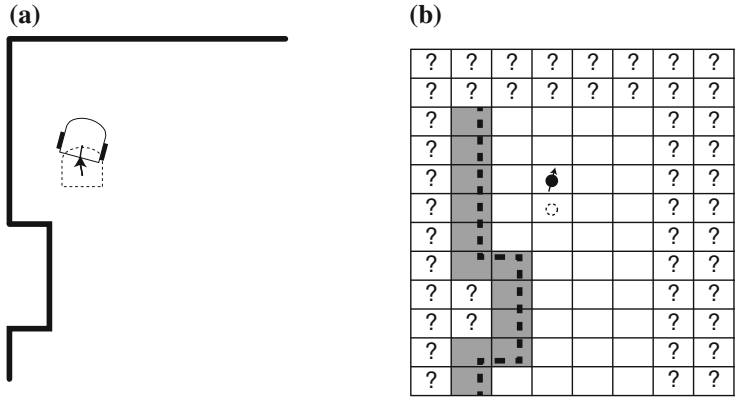



**Fig. 9.17** **a** The actual motion of the robot, **b** The map for the actual motion

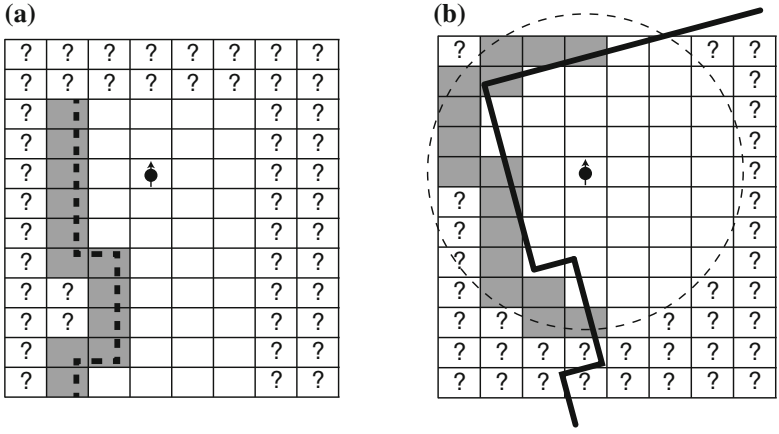



**Fig. 9.18** **a** The intended perception of the robot, **b** The actual perception of the robot

Figure 9.18a (which is the same as Fig. 9.16b) shows the *intended* perception of the robot because the robot has moved one cell upwards relative to the map in Fig. 9.15b. From this position it can detect the obstacle to its left and investigate the unknown cells in front of it.

However, because of the error in odometry, the *actual* perception of the robot is different. Figure 9.18b shows the actual position of the wall as seen by the robot overlaid on top of the cells, where cells are colored gray if the majority of their area is known to be behind the wall. (Examine several cells to verify that this is true.) We assume that the robot can sense walls at a distance of up to five times the size of a cell as shown by the dashed circle and we also assume that the robot knows that any wall is one cell thick.

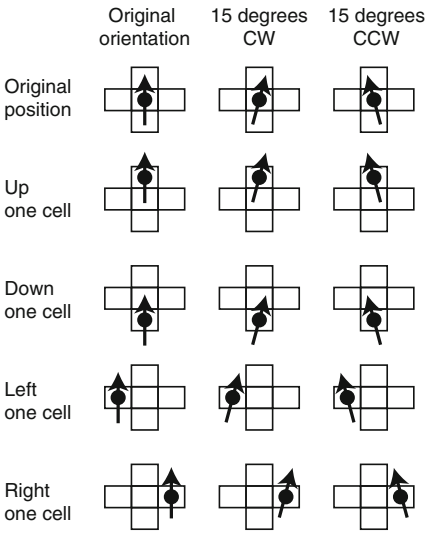



**Fig. 9.19** Possible poses of the robot

There is a clear mismatch between the current map and the sensor data which should correspond to the known part of the map. Obviously, the robot is not where it is expected to be based on odometry. How can this mismatch be corrected? We assume that odometry does give a reasonable estimation of the pose (position and heading) of the robot. For each relatively small possible error in the pose, we compute what the perception of the current map would be and compare it with the actual perception computed from the sensor data. The pose that gives the best match is chosen as the actual pose of the robot and the current map updated accordingly.

In the example, assume that the robot is either in the expected cell or in one of its four neighbors (left, right, up, down) and that the heading of the robot is either correct or turned slightly to the right (15° clockwise (CW)) or slightly to the left (15° counterclockwise (CCW)). The $5 \times 3 = 15$ possible poses are shown in Figs. 9.19 and 9.20 shows the perception of the map computed from the current map for each pose. (To save space, only a $8 \times 5$ fragment of the $12 \times 8$ map is displayed.)

The next step is to choose the map that gives the best fit with the sensor measurements. First transform the $8 \times 5$ maps into $8 \times 5$ matrices, assigning $-1$ to empty cells, $+1$ to obstacle cells and 0 to other cells. The left matrix in Fig. 9.21 is associated with the current map and the center matrix in the figure is associated with the perception map corresponding to the pose where the robot is in the correct cell but the heading is 15° CW (the middle element of the top row of Fig. 9.20).

To compare the maps, multiply elements of corresponding cells. Let $m(i, j)$ be the $(i, j)$'th cell of the current map and $p(i, j)$ be the $(i, j)$'th cell of the perception map obtained from sensor values. $S(i, j)$, the *similarity* of $(i, j)$'th cell, is:

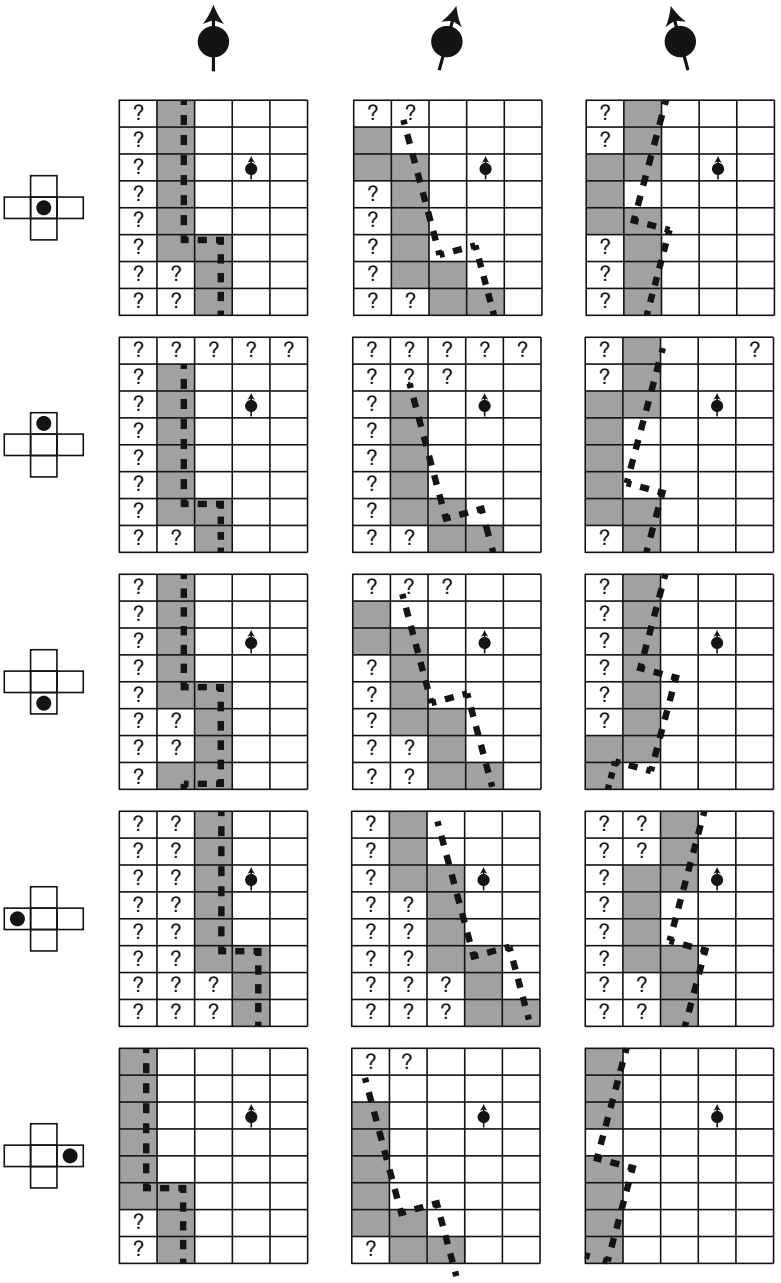


**Fig. 9.20** Estimations of perception of the robot for different poses

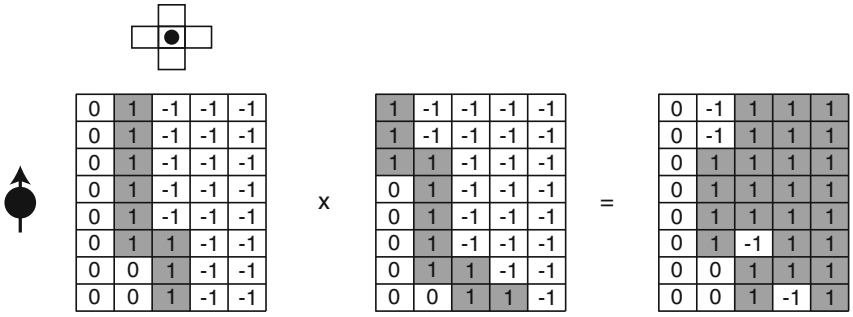



**Fig. 9.21** Computation of the matching between two maps

**Table 9.1** Similarity $\mathcal{S}$ of the sensor-based map with the current map

| | Intended orientation | 15° CW | 15° CCW |
|---|---|---|---|
| Intended position | 22 | **32** | 20 |
| Up one cell | 23 | 25 | 16 |
| Down one cell | 19 | 28 | 21 |
| Left one cell | 6 | 7 | 18 |
| Right one cell | 22 | 18 | 18 |

$$S(i, j) = m(i, j)\, p(i, j)\,,$$

which can also be expressed as:

$$\begin{aligned} S(i, j) &= 1 \quad \text{if } m(i, j) \neq 0,\ p(i, j) \neq 0,\ m(i, j) = p(i, j) \\ S(i, j) &= -1 \quad \text{if } m(i, j) \neq 0,\ p(i, j) \neq 0,\ m(i, j) \neq p(i, j) \\ S(i, j) &= 0 \quad \text{if } m(i, j) = 0 \text{ or } p(i, j) = 0\,. \end{aligned}$$

The right matrix in Fig. 9.21 shows the result of this computation for the two matrices to its left. There are a lot of 1's meaning that the matrices are similar and thus we can conclude that the perception maps are similar. For a quantitative result, compute the sum of the similarities to obtain a single value for any pair $m$, $p$:

$$\mathcal{S} = \sum_{i=1}^{8} \sum_{j=1}^{5} S(i, j)\,.$$

Table 9.1 gives the values of the similarity $\mathcal{S}$ for all the perception maps in Fig. 9.20 compared with the current map. As expected, the highest similarity is obtained for the map corresponding to the pose with the correct position and with the heading rotated by 15° CW.

Once we have this result, we correct the pose of the robot and use data from the perception map to update the current map stored in the robot's memory (Fig. 9.22).

## 9.6 Activities for Demonstrating the SLAM Algorithm

The following two activities demonstrate aspects of the SLAM algorithm. Activity 9.4 follows the algorithm and is intended for implementation in software. Activity 9.5 demonstrates a key element of the algorithm that can be implemented on an educational robot.

The activities are based on the configuration shown in Fig. 9.23. The robot is located at the origin of the coordinate system with pose $((x, y), \theta) = ((0, 0), 0°)$.[1] Given the uncertainty of the odometry, the robot might actually be located at any of the coordinates $(-1, 0)$, $(1, 0)$, $(0, -1)$, $(0, 1)$ and its orientation might be any of $-15°$, $0$, $-15°$ (as shown by the dashed arrows), giving 15 possible poses. The three gray dots at coordinates $(2, 2)$, $(2, 0)$, $(2, -2)$ represent known obstacles on the current map. (To save space the dots are displayed at coordinates $(2, 1)$, $(2, 0)$, $(2, -1)$.) The obstacles can be sensed by multiple horizontal proximity sensors, but for the purposes of the activities we specify that there are three sensors.

In the SLAM algorithm the *perception* of an obstacle is the value returned by a distance sensor. To avoid having to define model for the sensors, the activities will define a perception as the *distance* and *heading* from the sensor to the obstacle.

**Activity 9.4: Localize the robot from the computed perceptions**

- For each of the 15 poses compute the set of perceptions of each obstacle. For example, if the robot is at the pose $((0.0, 1.0), -15.0°)$, the set of perceptions of the three obstacles is:

$$[(2.2, 41.6°),\ (2.2, -11.6°),\ (3.6, -41.3°)]\,.$$

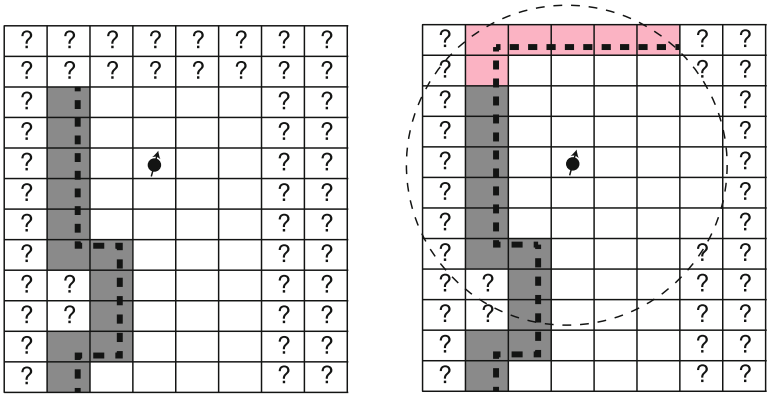


**Fig. 9.22** Map before and after the update using data from the perception map

[1] It is convenient to take the heading of the robot as 0°.

- Given a set of measured perceptions, compute their similarities to the perceptions of the 15 poses of the robot. Choose the pose with the best similarity as the actual pose of the robot. For example, for the set of measured perceptions:

$$[(2.0, 32.0°),\ (2.6, -20.0°),\ (3.0, -30.0°)]\,,$$

  and the similarity computed as the sum of the absolute differences of the elements of the perceptions, the pose with the best similarity is $((0.0, 1.0), -15.0°)$.
- Experiment with different similarity functions.
- We have computed that the robot's pose is approximately $((0.0, 1.0), -15.0°)$. Suppose that a new obstacle is placed at coordinate $(3, 0)$. Compute the perception $(d, \theta)$ of the object from this pose and then compute the coordinate $(x, y)$ of the new obstacle. The new obstacle can be added to the map with this coordinate.
- Is the computed coordinate significantly different from the coordinate $(3, 0)$ that would be obtained if the robot were at its intended pose $((0, 0), 0°)$?

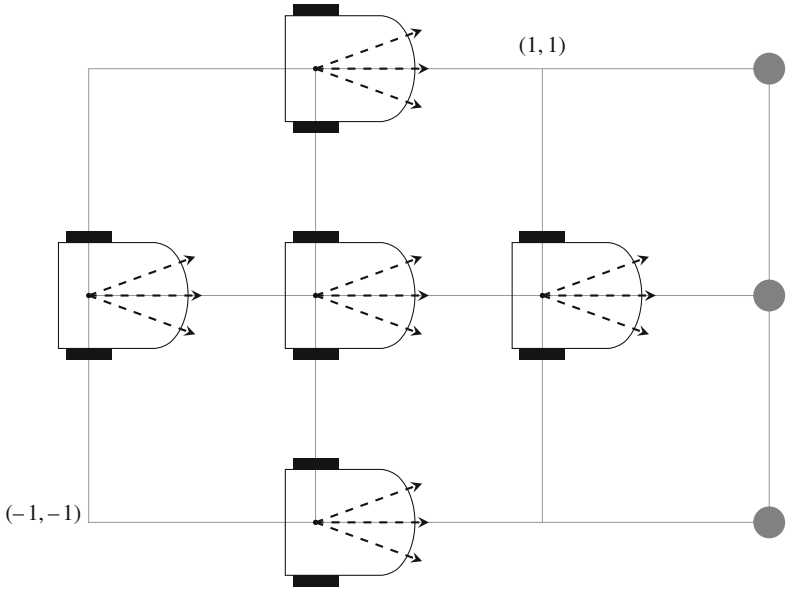



**Fig. 9.23** Configuration for the SLAM algorithm

**Activity 9.5: Localize the robot from the measured perceptions**

- Place three objects as shown in Fig. 9.23.
- Write a program that stores the set of values returned by the three horizontal proximity sensors. Place your robot successively at each of the 15 poses and record the sets of values. You now have a database of perceptions: a set of three sensor readings for each pose.
- Place the robot at one of the poses and store the set of values returned by the sensors. Compute the similarity of this set to each of the sets in the database. Display the pose associated with the best similarity.
- Experiment with placing the robot at various poses and several times at each pose. How precise is the determination of the pose?
- Experiment with different similarity functions.

## 9.7 The Formalization of the SLAM Algorithm

Algorithm 9.2 is a SLAM algorithm that finds the position whose perception map is closest to the perception map obtained from the sensor data. The robot is localized to this position and the map updated to what is perceived at this position.

The algorithm is divided into three phases. In the first phase (lines 2–4), the robot moves a short distance and its new position is computed by odometry. The perception map at this location is obtained by analyzing the sensor data.

Assuming that the odometry error is relatively small, we can define a set of test positions where the robot might be. In the second phase (lines 5–8), the expected map at each of these positions is computed and compared with the current map. The best match is saved.

In the third phase (lines 9–11), the position with the best match becomes the new position and the current map is updated accordingly.

In practice, the algorithm is somewhat more complicated because it has to take into account that the perception map obtained from the sensors is limited by the range of the sensors. The overlap will be partial both because the sensor range does not cover the entire current map and because the sensors can detect obstacles and free areas outside the current map. Therefore, the size of the perceived map **p** will be much smaller than the expected map **e** and the function compare(**p**, **e**) will only compare the areas that overlap. Furthermore, when updating the current map, areas not previously in the map will be added. In Fig. 9.22 there are cells in the current map that are outside the five-cell radius of the sensor and will not be updated. The light red cells were unknown in the current map as indicated by the question marks, but in the perception map they are now known to be part of the obstacle. This information is used to update the current map to obtain a new current map.

**Algorithm 9.2: SLAM**

```
                matrix m ← partial map          // Current map
                matrix p                        // Perception map
                matrix e                        // Expected map
                coordinate c ← initial position // Current position
                coordinate n                    // New position
                coordinate array T              // Set of test positions
                coordinate t                    // Test position
                coordinate b ← none             // Best position
 1: loop
 2:     move a short distance
 3:     n ← odometry(c)                         // New position based on odometry
 4:     p ← analyze sensor data

 5:     for every t in T                        // T is the positions around n
 6:         e ← expected(m, t)                  // Expected map at test position
 7:         if compare(p,e) better than b
 8:             b ← t                           // Best test position so far

 9:     n ← b                                   // Replace new position by best position
10:     m ← update(m,p,n)                       // Update map based on new position
11:     c ← n                                   // Current position is new position
```

## 9.8 Summary

Accurate robotic motion in an uncertain environment requires that the robot have a map of the environment. The map must be maintained in the robot's computer; it can be either a grid map of cells or a graph representation of a continuous map. In an uncertain environment, a map will typically not be available to the robot before it begins its tasks. The frontier algorithm is used by a robot to construct a map as it explores its surroundings. More accurate maps can be constructed if the robot has some knowledge of its environment, for example, that the environment is the inside of a building consisting of rectangular rooms and corridors. Simultaneous localization and mapping (SLAM) algorithms use an iterative process to construct a map while also correcting for errors in localization.

## 9.9 Further Reading

Two textbooks on path and motion planning are [4, 5]. See also [6, Chap. 6].

The frontier algorithm was proposed by Yamauchi [8] who showed that a robot could use the algorithm to successfully explore an office with obstacles.

Algorithms for SLAM use probability, in particular Bayes rule. Probabilistic methods in robotics are the subject of the textbook [7].

A two-part tutorial on SLAM by Durrant-Whyte and Bailey can be found in [1, 2]. A tutorial on graph-based SLAM is [3].

Sebastian Thrun's online course *Artificial Intelligence for Robotics* is helpful: https://classroom.udacity.com/courses/cs373.

## References

1. Bailey, T., Durrant-Whyte, H.: Simultaneous localization and mapping: part ii. IEEE Robot. Autom. Mag. **13**(3), 108–117 (2006)
2. Durrant-Whyte, H., Bailey, T.: Simultaneous localization and mapping: part i. IEEE Robot. Autom. Mag. **13**(2), 99–110 (2006)
3. Grisetti, G., Kümmerle, R., Stachniss, C., Burgard, W.: A tutorial on graph-based slam. IEEE Intell. Transp. Syst. Mag. **2**(4), 31–43 (2010)
4. Latombe, J.C.: Robot Motion Planning. Springer, Berlin (1991)
5. LaValle, S.M.: Planning Algorithms. Cambridge University Press, Cambridge (2006)
6. Siegwart, R., Nourbakhsh, I.R., Scaramuzza, D.: Introduction to Autonomous Mobile Robots, 2nd edn. MIT Press, Cambridge (2011)
7. Thrun, S., Burgard, W., Fox, D.: Probabilistic Robotics. MIT Press, Cambridge (2005)
8. Yamauchi, B.: A frontier-based approach for autonomous exploration. In: IEEE International Symposium on Computational Intelligence in Robotics and Automation, pp. 146–151 (1997)



# Chapter 10
# Mapping-Based Navigation

Now that we have a map, whether supplied by the user or discovered by the robot, we can discuss *path planning*, a higher-level algorithm. Consider a robot used in a hospital for transporting medications and other supplies from storage areas to the doctors and nurses. Given one of these tasks, what is the best way of going from point A to point B? There may be multiple ways of moving through the corridors to get to the goal, but there may also be short paths that the robot is not allowed to take, for example, paths that go through corridors near the operating rooms.

We present three algorithms for planning the shortest path from a starting position `S` to a goal position `G`, assuming that we have a map of the area that indicates the positions of obstacles in the area. Edsgar W. Dijkstra, one of the pioneers of computer science, proposed an algorithm for the shortest path problem. Section 10.1 describes the algorithm for a grid map, while Sect. 10.2 describes the algorithm for a continuous map. The A* algorithm, an improvement on Dijkstra's algorithm based upon heuristic methods, is presented in Sect. 10.3. Finally, Sect. 10.4 discusses how to combine a high-level path planning algorithm with a low-level obstacle avoidance algorithm.

## 10.1 Dijkstra's Algorithm for a Grid Map

Dijkstra described his algorithm for a discrete graph of nodes and edges. Here we describe it for a grid map of cells (Fig. 10.1a). Cell `S` is the starting cell of the robot and its task is to move to the goal cell `G`. Cells that contain obstacles are shown in black. The robot can sense and move to a *neighbor* of the cell `c` it occupies. For simplicity, we specify that the neighbors of `c` are the four cells next to it horizontally and vertically, but not diagonally. Figure 10.1b shows a shortest path from `S` to `G`:

$$(4, 0) \rightarrow (4, 1) \rightarrow (3, 1) \rightarrow (2, 1) \rightarrow (2, 2) \rightarrow (2, 3) \rightarrow (3, 3) \rightarrow (3, 4) \rightarrow (3, 5) \rightarrow (4, 5)\,.$$



M. Ben-Ari and F. Mondada, *Elements of Robotics*,
https://doi.org/10.1007/978-3-319-62533-1_10

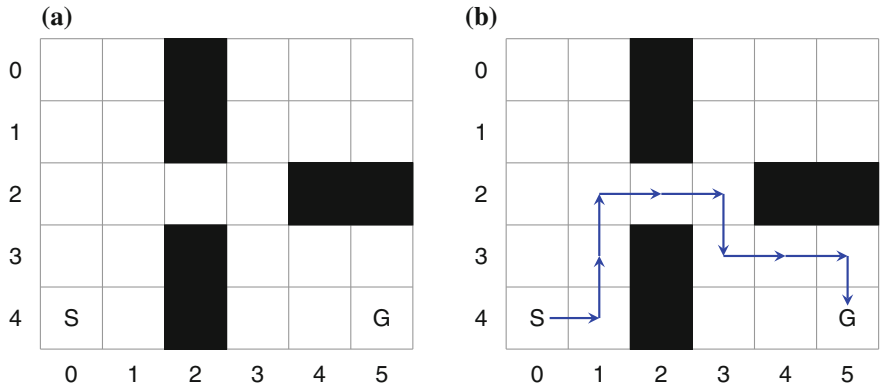



**Fig. 10.1** **a** Grid map for Dijkstra's algorithm. **b** The shortest path found by Dijkstra's algorithm

Two versions of the algorithm are presented: The first is for grids where the cost of moving from one cell to one of its neighbors is constant. In the second version, each cell can have a different cost associated with moving to it, so the shortest path geometrically is not necessarily the shortest path when the costs are taken into account.

### *10.1.1 Dijkstra's Algorithm on a Grid Map with Constant Cost*

Algorithm 10.1 is Dijkstra's algorithm for a grid map. The algorithm is demonstrated on the $5 \times 6$ cell grid map in Fig. 10.2a. There are three obstacles represented by the black cells.

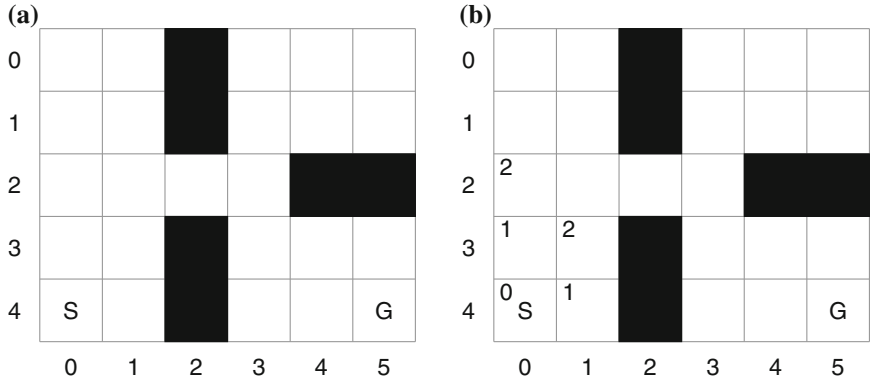



**Fig. 10.2** **a** Grid map for Dijkstra's algorithm. **b** The first two iterations of Dijkstra's algorithm

**Algorithm 10.1: Dijkstra's algorithm on a grid map**

```
integer n ← 0                          // Distance from start
cell array grid ← all unmarked         // Grid map
cell list path ← empty                 // Shortest path
cell current                           // Current cell in path
cell c                                 // Index over cells
cell S ← ···                           // Source cell
cell G ← ···                           // Goal cell
```

```
1: mark S with n
2: while G is unmarked
3:     n ← n + 1
4:     for each unmarked cell c in grid
5:             next to a marked cell
6:         mark c with n
7: current ← G
8: append current to path
9: while S not in path
10:    append lowest marked neighbor c
11:        of current to path
12:    current ← c
```

The algorithm incrementally marks each cell c with the number of steps needed to reach c from the start cell S. In the figures, the step count is shown as a number in the upper left hand corner of a cell. Initially, mark cell S with 0 since no steps are needed to reach S from S. Now, mark every neighbor of S with 1 since they are one step away from S; then mark every neighbor of a cell marked 1 with 2. Figure 10.2b shows the grid map after these two iterations of the algorithm.

The algorithm continues iteratively: if a cell is marked $n$, its unmarked neighbors are marked with $n+1$. When G is finally marked, we know that the shortest distance from S to G is $n$. Figure 10.3a shows the grid map after five iterations and Fig. 10.3b shows the final grid map after nine iterations when the goal cell has been reached.

It is now easy to find a shortest path by working backwards from the goal cell G. In Fig. 10.3b a shortest path consists of the cells that are colored gray. Starting with the goal cell at coordinate $(4, 5)$, the previous cell must be either $(4, 4)$ or $(3, 5)$ since they are eight steps away from the start. (This shows that there is more than one shortest path.) We arbitrarily choose $(3, 5)$. From each selected cell marked $n$, we choose a cell marked $n - 1$ until the cell S marked 0 is selected. The list of the selected cells is:

$$(4, 5),\ (3, 5),\ (3, 4),\ (3, 3),\ (2, 3),\ (2, 2),\ (2, 1),\ (3, 1),\ (4, 1),\ (4, 0)\,.$$

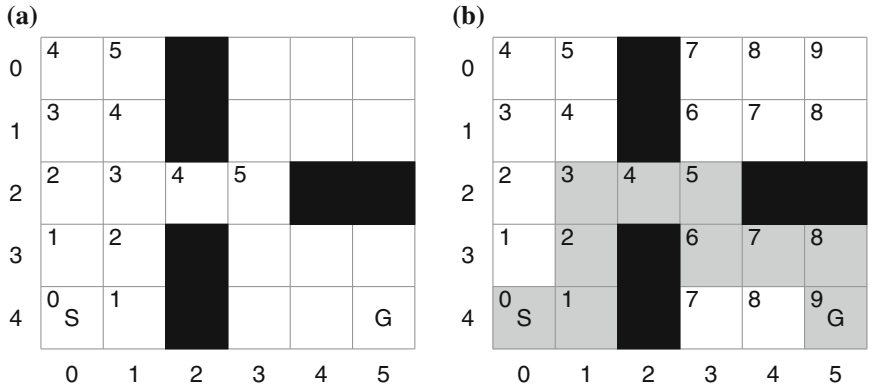



**Fig. 10.3** **a** After five iterations of Dijkstra's algorithm. **b** The final grid map with the shortest path marked

By reversing the list, the shortest path from S to G is obtained. Check that this is the same path we found intuitively (Fig. 10.1b).

**Example** Figure 10.4 shows how Dijkstra's algorithm works on a more complicated example. The grid map has $16 \times 16$ cells and the goal cell G is enclosed within an obstacle and difficult to reach. The upper left diagram shows the grid map after three iterations and the upper right diagram shows the map after 19 iterations. The algorithm now proceeds by exploring cells around both sides of the obstacle. Finally, in the lower left diagram, G is found after 25 iterations, and the shortest path is indicated in gray in the lower right diagram. We see that the algorithm is not very efficient for this map: although the shortest path is only 25 steps, the algorithm has explored $256 - 25 = 231$ cells!

### *10.1.2 Dijkstra's Algorithm with Variable Costs*

Algorithm 10.1 and the example in Fig. 10.4 assume that the cost of taking a step from one cell to the next is constant: line three of the algorithm adds 1 to the cost for each neighbor. Dijkstra's algorithm can be modified to take into account a variable cost of each step. Suppose that an area in the environment is covered with sand and that it is more difficult for the robot to move through this area. In the algorithm, instead of adding 1 to the cost for each neighboring cell, we can add $k$ to each neighboring sandy cell to reflect the additional cost.

The grid on the left of Fig. 10.5 has some cells marked with a diagonal line to indicate that they are covered with sand and that the cost of moving through them is 4 and not 1. The shortest path, marked in gray, has 17 steps and also costs 17 since it goes around the sand.

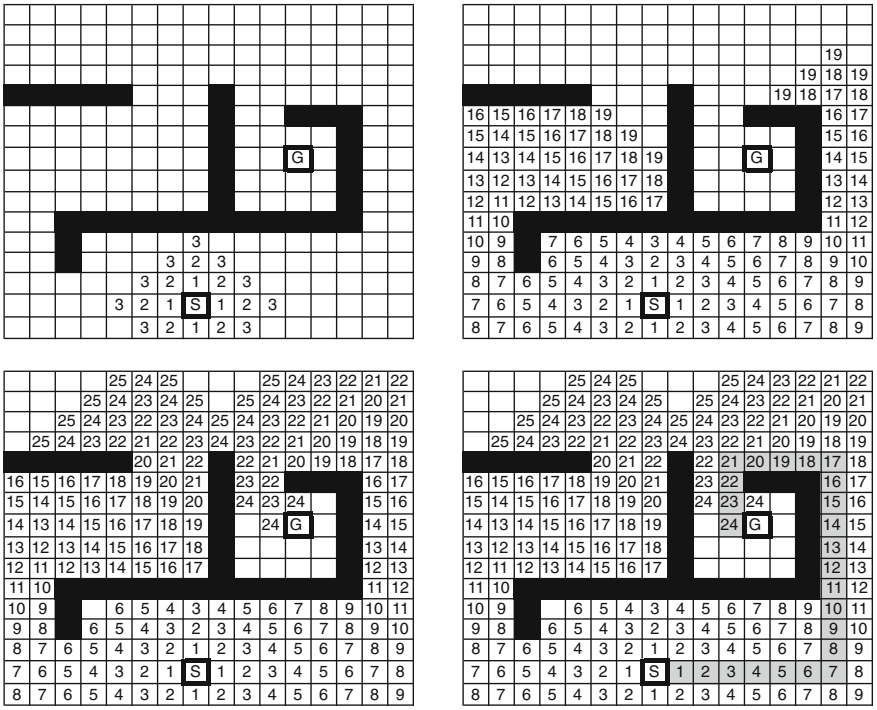



**Fig. 10.4** Dijkstra's algorithm for path planning on a grid map. Four stages in the execution of the algorithm are shown starting in the *upper left* diagram

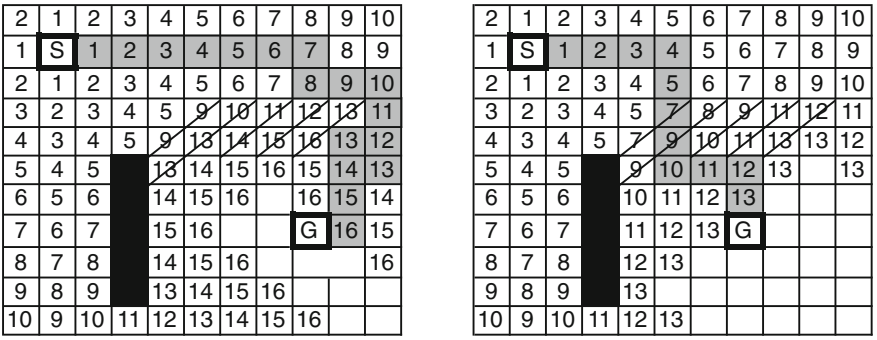



**Fig. 10.5** Dijkstra's algorithm with a variable cost per cell (*left* diagram, cost = 4, *right* diagram, cost = 2)

The shortest path depends on the cost assigned to each cell. The right diagram shows the shortest path if the cost of moving through a cell with sand is only 2. The path is 12 steps long although its cost is 14 to take into account moving two steps through the sand.

**Activity 10.1: Dijkstra's algorithm on a grid map**

- Construct a grid map and apply Dijkstra's algorithm.
- Modify the map to include cells with a variable cost and apply the algorithm.
- Implement Dijkstra's algorithm.
  - Create a grid on the floor.
  - Write a program that causes the robot to move from a known start cell to a known goal cell. Since the robot must store its current location, use this to create a map of the cells it has moved through.
  - Place some obstacles in the grid and enter them in the map of the robot.

## 10.2 Dijkstra's Algorithm for a Continuous Map

In a continuous map the area is an ordinary two-dimensional geometric plane. One approach to using Dijkstra's algorithm in a continuous map is to transform the map into a discrete graph by drawing vertical lines from the upper and lower edges of the environment to each corner of an obstacle. This divides the area into a finite number of segments, each of which can be represented as a node in a graph. The left diagram of Fig. 10.6 shows seven vertical lines that divide the map into ten segments which are shown in the graph in Fig. 10.7. The edges of the graph are defined by the adjacency relation of the segments. There is a directed edge from segment A to segment B if A and B share a common border. For example, there are edges from node 2 to nodes 1 and 3 since the corresponding segments share an edge with segment 2.

What is the shortest path between vertex 2 representing the segment containing the starting point and vertex 10 representing the segment containing the goal? The result of applying Dijkstra's algorithm is S $\rightarrow$ 2 $\rightarrow$ 3 $\rightarrow$ 6 $\rightarrow$ 9 $\rightarrow$ 10 $\rightarrow$ G. Although this is the shortest path in terms of the number of edges of the graph, it is not the shortest path in the environment. The reason is that we assigned constant cost to each edge, although the segments of the map have various size.

Since each vertex represents a large segment of the environment, we need to know how moving from one vertex to another translates into moving from one segment to another. The right diagram in Fig. 10.6 shows one possibility: each segment is associated with its geometric center, indicated by the intersection of the dotted diagonal lines in the figure. The path in the environment associated with the path in the graph goes from the center of one segment to the center of the next segment, except that

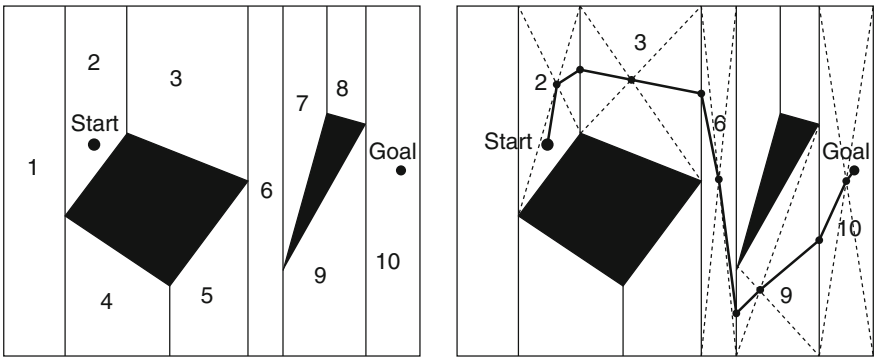



**Fig. 10.6** Segmenting a continuous map by *vertical lines* and the path through the segments

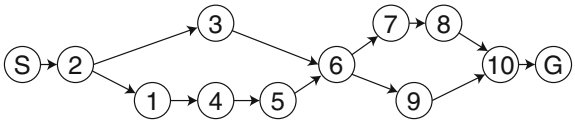



**Fig. 10.7** The graph constructed from the segmented continuous map

the start and goal positions are at their geometric locations. Although this method is reasonable without further knowledge of the environment, it does not give the optimal path which should stay close to the borders of the obstacles.

Figure 10.8 shows another approach to path planning in a continuous map. It uses a *visibility graph*, where each vertex of the graph represents a corner of an obstacle, and there are vertices for the start and goal positions. There is an edge from vertex $v_1$ to vertex $v_2$ if the corresponding corners are visible. For example, there is an edge $C \rightarrow E$ because corner $E$ of the right obstacle is visible from corner $C$ of the left obstacle. Figure 10.9 shows the graph formed by these nodes and edges. It represents all candidates for the shortest path between the start and goal locations.

It is easy to see that the paths in the graph represent paths in the environment, since the robot can simply move from corner to corner. These paths are the shortest paths because no path, say from $A$ to $B$, can be shorter than the straight line from $A$ to $B$. Dijkstra's algorithm gives the shortest path as $S \rightarrow D \rightarrow F \rightarrow H \rightarrow G$. In this case, the shortest path in terms of the number of edges is also the geometrically shortest path.

Although this is the shortest path, a real robot cannot follow this path because it has a finite size so its center cannot follow the border of an obstacle. The robot must maintain a minimum distance from each obstacle, which can be implemented by expanding the size of the obstacles by the size of the robot (right diagram of Fig. 10.9). The resulting path is optimal and can be traversed by the robot.

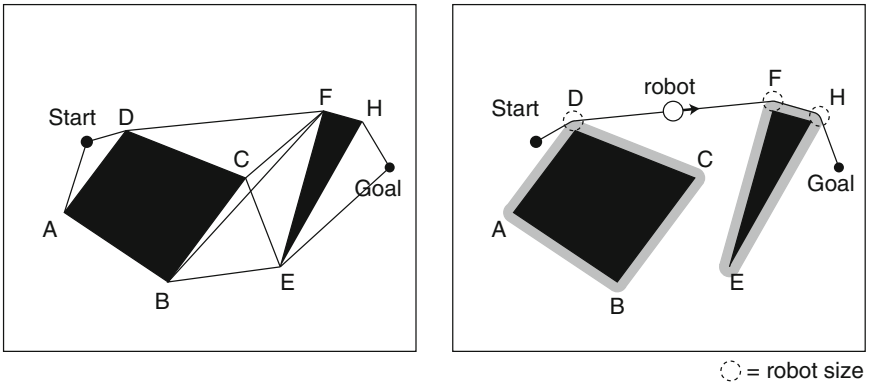



**Fig. 10.8** A continuous map with lines from corner to corner and the path through the corners

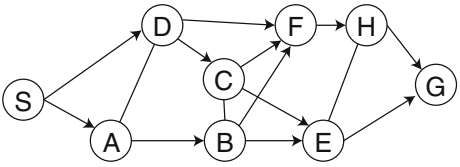



**Fig. 10.9** The graph constructed from the segmented continuous map

**Activity 10.2: Dijkstra's algorithm for continuous maps**

- Draw a larger version of the map in Fig. 10.8. Measure the length of each segment. Now apply Dijkstra's algorithm to determine the shortest path.
- Create your own continuous map, extract the visibility graph and apply Dijkstra's algorithm.

## 10.3 Path Planning with the A* Algorithm

Dijkstra's algorithm searches for the goal cell in all directions; this can be efficient in a complex environment, but not so when the path is simple, for example, a straight line to the goal cell. Look at the top right diagram in Fig. 10.4: near the upper right corner of the center obstacle there is a cell at distance 19 from the start cell. After two more steps to the left, there will be a cell marked 21 which has a path to the goal cell that is not blocked by an obstacle. Clearly, there is no reason to continue to explore the region at the left of the grid, but Dijkstra's algorithm continues to do so. It would be more efficient if the algorithm somehow knew that it was close to the goal cell.

The A* *algorithm* (pronounced "A star") is similar to the Dijkstra's algorithm, but is often more efficient because it uses extra information to guide the search. The A* algorithm considers not only the number of steps from the start cell, but also a *heuristic function* that gives an indication of the preferred direction to search. Previously, we used a cost function $g(x, y)$ that gives the actual number of steps from the start cell. Dijkstra's algorithm expanded the search starting with the cells marked with the highest values of $g(x, y)$. In the A* algorithm the cost function $f(x, y)$ is computed by adding the values of a heuristic function $h(x, y)$:

$$f(x, y) = g(x, y) + h(x, y) .$$

We demonstrate the A* algorithm by using as the heuristic function the number of steps from the goal cell G to cell $(x, y)$ *without taking the obstacles into account*. This function can be precomputed and remains available throughout the execution of the algorithm. For the grid map in Fig. 10.2a, the heuristic function is shown in Fig. 10.10a.

In the diagrams, we will keep track of the values of the three functions $f, g, h$ by displaying them in different corners of each cell $\boxed{\begin{matrix} g & f \\ & h \end{matrix}}$. Figure 10.10b shows the grid map after two steps of the A* algorithm. Cells (3, 1) and (3, 0) receive the same cost $f$: one is closer to S (by the number of steps counted) and the other is closer to G (by the heuristics), but both have the same cost of 7.

The algorithm needs to maintain a data structure of the *open cells*, the cells that have not yet been expanded. We use the notation $(r, c, v)$, where $r$ and $c$ are the row and column of the cell and $v$ is the $f$ value of the cell. Each time an open cell is expanded, it is removed form the list and the new cells are added. The list is *ordered* so that cells with the lowest values appear first; this makes it easy to decide which cell to expand next. The first three lists corresponding to Fig. 10.10b are:

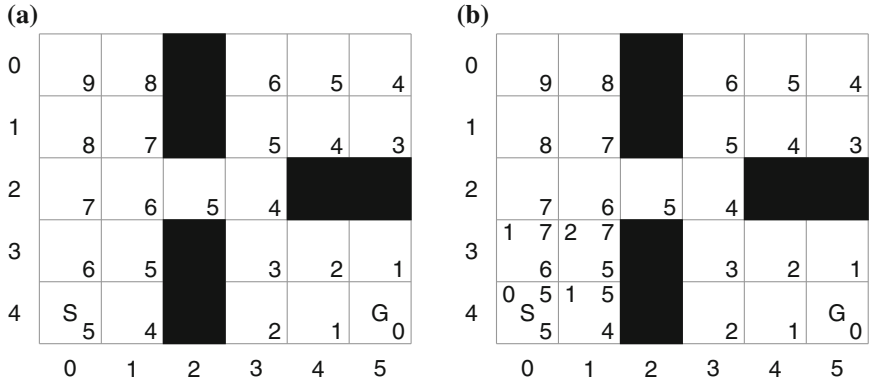



**Fig. 10.10** **a** Heuristic function. **b** The first two iterations of the A* algorithm

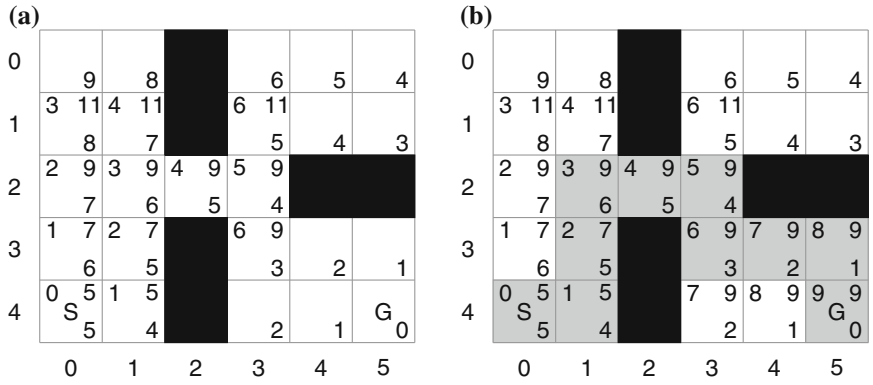



**Fig. 10.11** **a** The A* algorithm after 6 steps. **b** The A* algorithm reaches the goal cell and finds a shortest path

$$(4, 0, 5)$$
$$(4, 1, 5),\ (3, 0, 7)$$
$$(3, 0, 7),\ (3, 1, 7)\,.$$

Figure 10.11a shows the grid map after six steps. This can be seen by looking at the $g$ values in the upper left corner of each cell. The current list of open cells is:

$$(3, 3, 9),\ (1, 0, 11),\ (1, 1, 11),\ (1, 3, 11)\,.$$

The A* algorithm chooses to expand cell $(3, 3, 9)$ with the lowest $f$. The other cells in the list have an $f$ value of 11 and are ignored at least for now. Continuing (Fig. 10.11b), the goal cell is reached with $f$ value 9 and a shortest path in gray is displayed. The last list before reaching the goal is:

$$(3, 5, 9),\ (4, 4, 9),\ (1, 0, 11),\ (1, 1, 11),\ (1, 3, 11)\,.$$

It doesn't matter which of the nodes with value 9 is chosen: in either case, the algorithm reaches the goal cell $(4, 5, 9)$.

All the cells in the upper right of the grid are not explored because cell $(1, 3)$ has $f$ value 11 and that will never be the smallest value. While Dijkstra's algorithm explored all 24 non-obstacle cells, the A* algorithm explored only 17 cells.

### A More Complex Example of the A* Algorithm

Let apply the A* algorithm to the grid map in Fig. 10.5. Recall that this map has sand on some of its cells, so the $g$ function will give higher values for the cost of moving to these cells. The upper left diagram of Fig. 10.12 shows the $g$ function as computed by Dijkstra's algorithm, while the upper right diagram shows the heuristic function $h$, the number of steps from the goal in the absence of obstacles and the sand. The rest of the figure shows four stages of the algorithm leading to the shortest path to the goal.

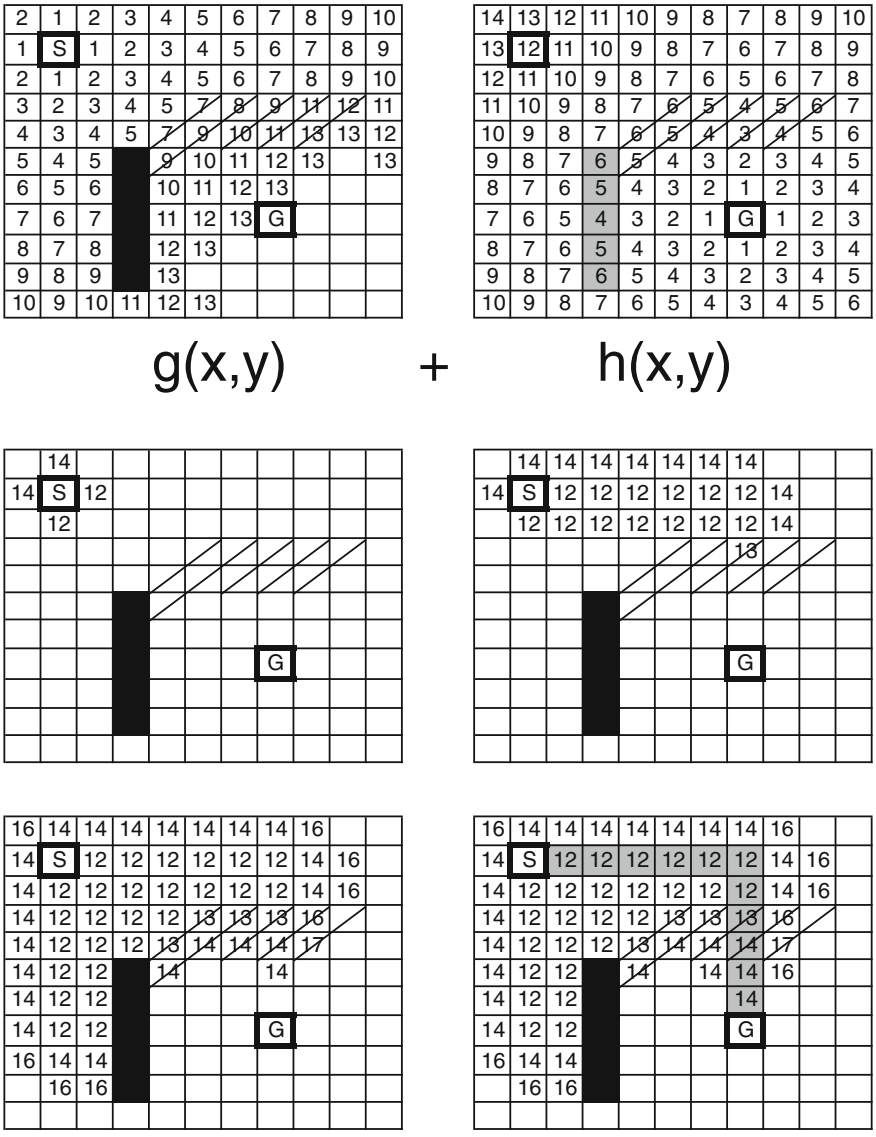



**Fig. 10.12** The A* algorithm. *Upper left* the number of steps to the goal. *Upper right* the heuristic function. The *middle* and *bottom* diagrams show four steps of the algorithm

Already from the middle left diagram, we see that it is not necessary to search towards the top left, because the $f$ values of the cells above and to the left of S are higher than the values to the right of and below S. In the middle right diagram, the first sand cell has a value of 13 so the algorithm continues to expand the cells with the lower cost of 12 to the left. In the bottom left diagram, we see that the search does not continue to the lower left of the map because the cost of 16 is higher than the cost of 14 once the search leaves the sand. From that point, the goal cell G is found very quickly. As in Dijkstra's algorithm, the shortest path is found by tracing back through cells with lower $g$ values until the start cell is reached.

Comparing Figs. 10.4 and 10.12 we see that the A* algorithm needed to visit only 71% of the cells visited by Dijkstra's algorithm. Although the A* algorithm must perform additional work to compute the heuristic function, the reduced number of cells visited makes the algorithm more efficient. Furthermore, this heuristic function depends only on the area searched and not on the obstacles; even if the set of obstacles is changed, the heuristic function does not need to be recomputed.

**Activity 10.3: A* algorithm**

- Apply the A* algorithm and Dijkstra's algorithm on a small map without obstacles: place the start cell in the center of the map and the goal in an arbitrary cell. Compare the results of the two algorithms. Explain your results. Does the result depend on the position of the goal cell?
- Define other heuristic functions and compare the results of the A* algorithms on the examples in this chapter.

## 10.4 Path Following and Obstacle Avoidance

This chapter and the previous ones discussed two different but related tasks: high-level path planning and low-level obstacle avoidance. How can the two be integrated? The simplest approach is to prioritize the low-level algorithm (Fig. 10.13). Obviously, it is more important to avoid hitting a pedestrian or to drive around a pothole than it is to take the shortest route to the airport. The robot is normally in its drive state, but if an obstacle is detected, it makes a transition to the avoid obstacle state. Only when the obstacle has been passed does it return to the state plan path so that the path can be recomputed.

The strategy for integrating the two algorithms depends on the environment. Repairing a road might take several weeks so it makes sense to add the obstacle to the map. The path planning algorithm will take the obstacle into account and the resulting path is likely to be better than one that is changed at the last minute by an obstacle avoidance algorithm. At the other extreme, if there are a lot of moving

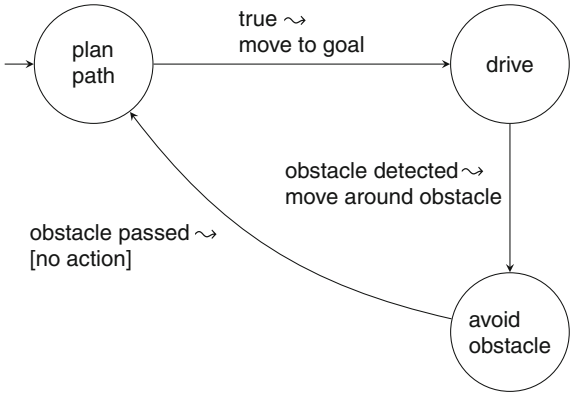



**Fig. 10.13** Integrating path planning and obstacle avoidance

obstacles such as pedestrians crossing a street, the obstacle avoidance behavior could be simply to stop moving and wait until the obstacles move away. Then the original plan can simply be resuming without detours.

**Activity 10.4: Combining path planning and obstacle avoidance**

- Modify your implementation of the line-following algorithm so that the robot behaves correctly even if an obstacle is placed on the line. Try several of the approaches listed in this section.
- Modify your implementation of the line-following algorithm so that the robot behaves correctly even if additional robots are moving randomly in the area of the line. Ensure that the robots do not bump into each other.

## 10.5 Summary

Path planning is a high-level behavior of a mobile robot: finding the shortest path from a start location to a goal location in the environment. Path planning is based upon a map showing obstacles. Dijkstra's algorithm expands the shortest path to any cell encountered so far. The A* algorithm reduces the number of cells visited by using a heuristic function that indicates the direction to the goal cell.

Path planning is based on a graph such as a grid map, but it can also be done on a continuous map by creating a graph of obstacles from the map. The algorithms can take into account variables costs for visiting each cell.

Low-level obstacle avoidance must be integrated into high-level path planning.

## 10.6 Further Reading

Dijkstra's algorithm is presented in all textbooks on data structures and algorithms, for example, [1, Sect. 24.3]. Search algorithms such as the A* algorithm are a central topic of artificial intelligence [2, Sect. 3.5].

## References

1. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press, Cambridge (2009)
2. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach, 3rd edn. Pearson, Boston (2009)



# Chapter 11
# Fuzzy Logic Control

The control algorithms in Chap. 6 used exact mathematical computations to determine the signals used to control the behavior of a robot. An alternate approach is to use *fuzzy logic* control algorithms based upon *rules*. A cruise control system might have rules of the form:

- If the *car in front is far away* or the *car in back is near*, set the *speed to fast*.
- If the *car in front is near*, set the *speed to slow*.

The logic is "fuzzy" because the rules are expressed in terms of *linguistic variables* like *speed* whose values do not have precise mathematical definitions, but only imprecise linguistic specifications like *fast* and *slow*.

A fuzzy logic controller consists of three phases that are run sequentially:

**Fuzzify** The values of the sensors are converted into values of the linguistic variables, such as far, closing, near, called *premises*. Each premise specifies a *certainty* which is the probability of our belief that the variable is true.

**Apply rules** A set of *rules* expresses the control algorithm. Given a set of premises, a *consequent* is inferred. Consequents are also linguistic variables such as very fast, fast, cruise, slow, stop.

**Defuzzify** The consequents are combined in order to produce a *crisp* output, which is a numerical value that controls some aspect of the robot such as the power applied to the motors.

The following sections present the three phases of fuzzy control for the task of a robot approaching an object and stopping when it is very close to the object.

## 11.1 Fuzzify

When approaching an object, the value read by the horizontal proximity sensor increases from 0 to 100. The value returned by the sensor is fuzzified by converting it to a value of a linguistic variable. Figure 11.1 shows three graphs for converting



M. Ben-Ari and F. Mondada, *Elements of Robotics*,
https://doi.org/10.1007/978-3-319-62533-1_11

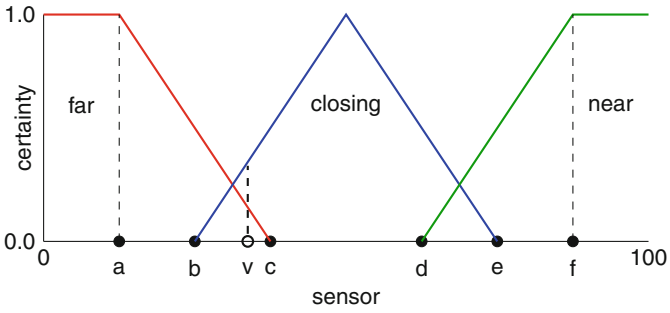



**Fig. 11.1** Fuzzify the value of the horizontal proximity sensor

the sensor values into certainties of the linguistic variables far, closing and near. The $x$-axis is the value returned by the sensor and the $y$-axis gives the premise for each variable, the certainty that the linguistic variable is true.

The labeled points on the $x$-axis refer to thresholds: (a) far_low, (b) closing_low, (c) far_high, (d) near_low, (e) closing_high, (f) near_high. If the value of the sensor is below far_low, then we are completely certain that the object is far away and the certainty is 1. If the value is between closing_low and far_high then we are somewhat certain that the object is far away, but also somewhat certain that it is closing. The fuzziness results from the overlapping ranges: when the value is between point (b) and point (c), we can't say with complete certainty if the object is far away or closing. For the sensor value v of about 33 the certainty of far is about 0.15 and the certainty of closing is about 0.25.

## 11.2 Apply Rules

The three premises, the certainties of far, closing and near, are used to compute five consequents using the following rules:

1. If far then very fast
2. If far and closing then fast
3. If closing then cruise
4. If closing and near then slow
5. If near then stop

The certainties of the consequents resulting from rules 1, 3, 5 are the same as the certainties of the corresponding premises. When there are two premises, as in rules 2 and 4, the certainties of the consequents are computed from the minimum of the certainties of the premises. Since *both* of the premises must apply, we can't be *more certain* of the consequent than we are of the smaller of the premises. For the value v in Fig. 11.1, rule 2 applies and the certainty of the consequent is $\min(0.15, 0.25) = 0.15$.

Another way of combining premises is to take their joint probability:

$$p(A \cap B) = P(A) \cdot P(B) .$$

For value v, the certainty of the consequent is $0.15 \times 0.25 = 0.0375$, much less than the certainty obtained from the minimum function.

## 11.3 Defuzzify

The next step is to combine the consequents, taking into account their certainties. Figure 11.2 shows the output motor powers for each of the five consequents. For example, if we are completely certain that the output is cruise, the center graph in the figure shows that the motor power should be set to 50, but if we were less certain, the motor power should be less or more.

Suppose that the certainty of the consequent of cruise is computed to be 0.4. Then the center triangle in Fig. 11.2 is no longer relevant because the certainty can never be more than 0.4, which is displayed as a trapezoid in Fig. 11.3.

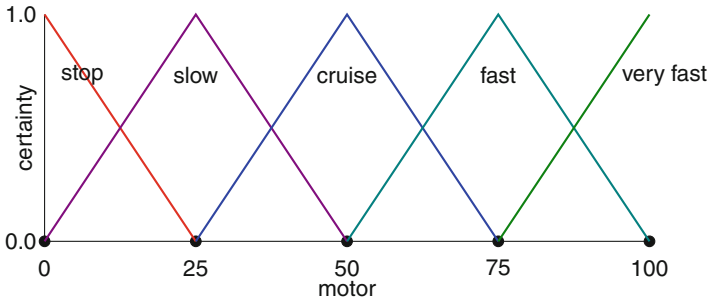



**Fig. 11.2** Defuzzify to obtain the crisp motor setting

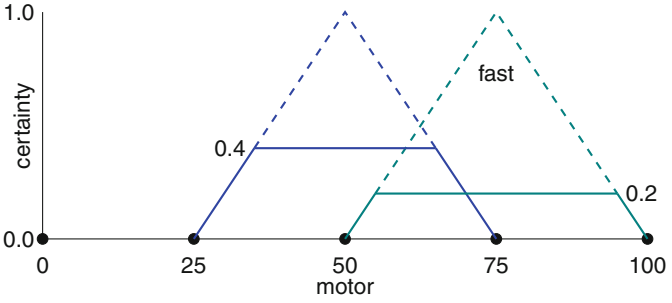



**Fig. 11.3** Areas defined by the certainties of the consequents

Let $w$ and $h$ be the width and height of a triangle. Then the area of a trapezoid bounded by the line at height $h'$ is given by the formula[1]:

$$wh'\left(1-\frac{h'}{2h}\right).$$

It is possible that more than one consequent has positive values. Figure 11.3 shows the trapezoids for the consequent cruise with a certainty of 0.4 and the consequent fast with a certainty of 0.2. For $w = 50$, $h = 1$, $h'_c = 0.4$ (cruise), $h'_f = 0.2$ (fast), the areas of the trapezoids $a_c$ (cruise) and $a_f$ (fast) are:

$$a_c = 50 \times 0.4\left(1-\tfrac{0.4}{2}\right) = 16$$
$$a_f = 50 \times 0.2\left(1-\tfrac{0.2}{2}\right) = 9\,.$$

To obtain a crisp value, the *center of gravity* is computed. This is the sum of the areas of the trapezoids weighted by the value at the center of the base of each trapezoid divided by the sum of the areas:

$$\frac{16 \times 50 + 9 \times 75}{16+9} = 59\,.$$

The value is closer to the value associated with cruise than it is to the value associated with fast. This is not surprising since the certainty of cruise is greater than the certainty of fast.

**Activity 11.1: Fuzzy logic**

- Implement the fuzzy logic controller for a robot approaching an object.
- Define appropriate thresholds for the proximity sensor and appropriate values for defuzzifying to obtain a crisp motor speed.
- Compare the results with the proportional control algorithm you implemented in Activity 6.3.

## 11.4 Summary

Fuzzy logic control is an alternative to the classical mathematical control algorithms described in Chap. 6. The advantage of fuzzy logic control is that it does not demand precise mathematical specifications of the robot's behavior which may be difficult to define. We gave an example of fuzzy definitions of speed; other examples would

[1] The derivation of the formula is given in Appendix B.6.

be color (when does a shade of red become orange?) and temperature (when does a warm room become hot?). The disadvantage is that the behavior of fuzzy logic control is not as transparent as that of classical control algorithms.

## 11.5 Further Reading

The original work on fuzzy logic was done by Lotfi Zadeh [3]. A textbook on the application of fuzzy logic to control is [2]. Fuzzy logic is also used in image processing [1, Sect. 3.8].

## References

1. Gonzalez, R.C., Woods, R.E.: Digital Image Processing, 3rd edn. Pearson, Boston (2008)
2. Passino, K.M., Yurkovich, S.: Fuzzy Control. Addison-Wesley, Reading (1998)
3. Zadeh, L.A.: Fuzzy sets. Inf. Control **8**(3), 338–353 (1965)



# Chapter 12
# Image Processing

The distance sensor on your self-driving car detects an object 100 m in front of your car. Are you following the car in front of you at a safe distance or has a pedestrian jumped into the road? The robotics algorithms presented so far have been based upon the measurement of physical properties like distance, angles and reflectance. More complex tasks require that a robot obtain detailed information on its surroundings, especially when the robot is intended to function autonomously in an unfamiliar environment.

For us, the obvious way to make sense of our environment is to use vision. We take vision for granted and don't realize how complex our visual system—our eyes and brain—really is. In fact, about 30% of the brain is used for vision. We can instantly distinguish between a moving car and a pedestrian crossing the road and react quickly.

For almost two hundred years it has been possible to automatically record images using a camera, but the interpretation of images remained a task for humans. With the advent of computers, it became possible to automatically process and interpret images. Digital images are familiar: weather maps from satellites, medical images (X-rays, CT and MRI scans, ultrasound images), and the photos we take with our smartphones. The field of *digital image processing* is one of the most intensely studied fields of computer science and engineering, but image processing systems have not yet reached the capability of the human visual system.

In this chapter, we present a taste of algorithms for digital image processing and describe how they are used in robotics systems. Sections 12.1 and 12.2 provide an overview of imaging systems and digital image processing. Sections 12.3–12.6 describe algorithms for image processing: enhancement by digital filters and histogram manipulation, segmentation (edge detection), and feature recognition (detection of corners and blobs, identification of multiple features).

For reasons of cost and computing power, few educational robots use cameras, so to study image processing algorithms you can implement the algorithms on a personal computer using images captured with a digital camera. Nevertheless, we propose some activities that demonstrate image processing algorithms on an educational



M. Ben-Ari and F. Mondada, *Elements of Robotics*,
https://doi.org/10.1007/978-3-319-62533-1_12

robot. The robot moves over a one-dimensional image and samples are read by a ground sensor. This results in a one-dimensional array of pixels that can be processed using simplified versions of the algorithms that we present.

## 12.1 Obtaining Images

In this section we give an overview of design considerations for imaging systems.

***Optics***

The optical system of a camera consists of a lens that focuses light on a sensor. The wider the lens, the more light that can be collected, which is important for systems that need to work in dark environments. The longer the focal length (which is related to the distance between the lens and the sensor), the greater the magnification. That is why professional photographers carry heavy cameras with long lenses. Manufacturers of smartphones are faced with a dilemma: we want our phones to be thin and elegant, but that limits the focal length of the camera. For most robotics applications, magnification is not worth the size and weight required to achieve a long focal length.

***Resolution***

Once upon a time, images were captured on film by a chemical reaction caused by light hitting a sheet of plastic covered with an emulsion of tiny silver particles. In principle, each particle could react independently so the resolution was extremely high. In digital images, light is captured by semiconductor devices such as *charge-coupled devices (CCD)*. A digital camera contains a chip with a fixed number of elements in a rectangular array. Each element measures the light intensity independently and these measurements are called *pixels*. The more pixels captured by a chip of a given area, the higher the resolution. Currently, even inexpensive cameras in smartphones can capture millions of pixels in a single image.

The problem with high resolution images is the large amount of memory needed to store them. Consider a high-resolution computer screen with $1920 \times 1080$ pixels and assume that each pixel uses 8 bits to store intensity in the range 0–255. A single image requires about 2 megabytes (MB) of memory. An embedded computer could analyze a single such image, but a mobile robot may need to store several images per second.

Even more important than the amount of memory required is the computing power required to analyze the images. Image processing algorithms require the computer to perform a computation on each individual pixel. This is not a problem for an astronomer analyzing images sent to earth from a space telescope, but it is a problem for a self-driving car which needs to make decisions in a fraction of a second.

***Color***

Our visual system has the capability of distinguishing a range of wavelengths called *visible light*. We discern different wavelengths as different colors. Light of longer

wavelengths is called *red*, while light of shorter wavelengths is called *violet*. The human eye can distinguish millions of different colors although we name only a few: red, orange, yellow, green, cyan, blue, violet, etc. Color is one of the primary tools that we use to identify objects.

Sensors are able to measure light of wavelengths outside the range we call visual light: *infrared* light of longer wavelengths and *ultraviolet* light of shorter wavelengths. Infrared images are important in robotics because hot objects such as people and cars can be detected as bright infrared light.

The problem with color is that it triples the requirements for storing and processing images. All colors can be formed by taking varying amounts of the three *primary colors*: red, green and blue (RGB). Therefore, a color image requires three bytes for each pixel. A single color image of resolution $1920 \times 1080$ requires over 6 MB of memory to store and the image processing takes at least three times as long.

## 12.2 An Overview of Digital Image Processing

The optical system of a robot captures images as rectangular arrays of pixels, but the tasks of a robot are expressed in terms of objects of the environment: enter a room through a door, pick up an item off a shelf, stop if a pedestrian walks in front of the car. How can we go from pixels to objects?

The first stage is *image enhancement*. Images contain noise that results from the optics and electronics. Furthermore, the lighting in the environment can cause an image to be too dark or washed out; the image may be accidentally rotated; the image may be out of focus. All these problems are independent of the content. It doesn't matter if an image that is out of focus shows a cat or a child. Image enhancement algorithms typically work by modifying the values assigned to individual pixels without regard to their meaning.[1]

Image enhancement is difficult because there is no formal definition of what it means to enhance an image. A blurred blob might be dirt on a camera's lens or an unknown galaxy. Section 12.3 presents two approaches to image enhancement: filtering removes noise by replacing a pixel with an average of its neighboring pixels and histogram manipulation modifies the brightness and contrast of an image.

Objects are distinguished by lines, curves and areas. A door consists of three straight edges of a rectangle with one short side missing. A traffic light consists of three bright disks one above another. Before a door or traffic light can be identified, image processing algorithms must determine which pixels represent lines, edges, etc. This process is called *segmentation* or *feature extraction* because the algorithms have to determine which pixels are part of a segment of an image.

[1] We limit ourselves to *spatial* processing algorithms that work on the pixels themselves. There is another approach called *frequency* processing algorithms, but that requires mathematical techniques beyond the scope of this book.

Segmentation would be easy if edges, lines and curves were uniform, but this is not what occurs in real images. An edge may be slanted at an arbitrary angle and some of its pixels may obscured by shadows or even missing. We are familiar with *captchas* where letters are intentionally distorted to make automatic recognition very difficult whereas humans can easily identify distorted letters. Enhancement algorithms can make segmentation easier, for example, by filling in missing pixels, but they may also introduce artificial segments. Section 12.4 demonstrates one segmentation technique: a filter that detects edges in an image.

The final phase of image processing is to recognize objects. In Sect. 12.5, we present two algorithms for detecting corners: by locating the intersection of two edges and by counting neighbors with similar intensities. Section 12.6 describes how to recognize *blobs*, which are areas whose pixels have similar intensities but which are not bounded by regular features such as lines and curves. Finally, Activity 12.6 demonstrates the recognition of an object that is defined by more than one feature, such as a door defined by two edges that are at an arbitrary distance from each other.

## 12.3 Image Enhancement

Figure 12.1a shows an image of a rectangle whose intensity is uniform horizontally and shaded dark to light from top to bottom. The representation of the image as a $6 \times 10$ rectangular array of pixels is shown in Fig. 12.2a, where each pixel is represented by a light intensity level in the range 0–100. Now look at Fig. 12.1b: the image is no longer *smooth* in the sense that there are three points whose intensity is not similar to the intensities of its neighbors. Compare the pixel array in Fig. 12.2b with the one in Fig. 12.2a: the intensities of the pixels at locations (2, 3), (3, 6), (4, 4) are different. This is probably the result of noise and not an actual feature of the object being photographed.

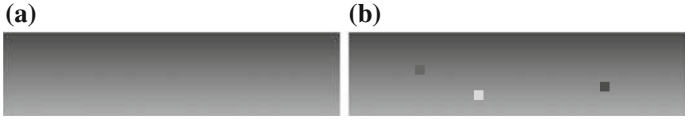



**Fig. 12.1** **a** Image without noise. **b** Image with noise

**(a)**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| 1 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 |
| 2 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 |
| 3 | 40 | 40 | 40 | 40 | 40 | 40 | 40 | 40 | 40 | 40 |
| 4 | 50 | 50 | 50 | 50 | 50 | 50 | 50 | 50 | 50 | 50 |
| 5 | 60 | 60 | 60 | 60 | 60 | 60 | 60 | 60 | 60 | 60 |

**(b)**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| 1 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 | 20 |
| 2 | 30 | 30 | 30 | **20** | 30 | 30 | 30 | 30 | 30 | 30 |
| 3 | 40 | 40 | 40 | 40 | 40 | 40 | **10** | 40 | 40 | 40 |
| 4 | 50 | 50 | 50 | 50 | **90** | 50 | 50 | 50 | 50 | 50 |
| 5 | 60 | 60 | 60 | 60 | 60 | 60 | 60 | 60 | 60 | 60 |

**Fig. 12.2** **a** Pixel array without noise. **b** Pixel array with noise

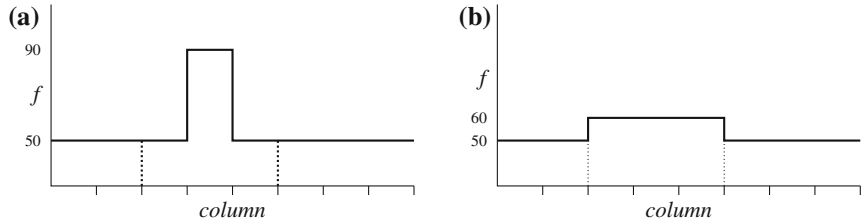



**Fig. 12.3** **a** Intensity plot before averaging. **b** Intensity plot after averaging

It doesn't really matter where the noise comes from: from the object itself, dust on the camera lens, non-uniformity in the sensor or noise in the electronics. It is impossible to get rid of the noise entirely, because we can never be sure whether a pixel is noise or an actual feature of the object, but we do want to enhance the image so that the noise is no longer noticeable.

### *12.3.1 Spatial Filters*

Consider row 4 in the pixel array in Fig. 12.2b:

$$50,\ 50,\ 50,\ 50,\ 90,\ 50,\ 50,\ 50,\ 50,\ 50\,.$$

Figure 12.3a is a plot of the light intensity $f$ for the pixels in that row. It is clear that one of the pixels has an unlikely value because its value is so different from its neighbors. A program can make each pixel more like its neighbors by replacing the intensity of the pixel with the average of its intensity and the intensities of its neighbors. For most of the pixels in the row, this doesn't change their values: $(50 + 50 + 50)/3 = 50$, but the noise pixel and its two neighbors receive new values: $(50 + 90 + 50)/3 \approx 60$ (Fig. 12.3b). Averaging has caused two pixels to receive "wrong" values, but overall the image will be visually enhanced because the intensity of the noise will be reduced.

Taking the average of a sequence of pixels is the discrete version of integrating a continuous intensity function. Integration smooths out local variation of the function. The dotted lines in Fig. 12.3a, b indicate a three-pixel sequence and it can be seen that the areas they bound are about the same.

The averaging operation is performed by applying a *spatial filter* at each pixel of the image.[2] For the two-dimensional array of pixels, the filter is represented by a $3 \times 3$ array, where each element of the array specifies the factor by which the pixel and its neighbors are multiplied. Each pixel has four or eight neighbors, depending on whether we include the diagonal neighbors. Here, we include the diagonal pixels in the filters.

[2]The mathematical term for applying a function $g$ at every point of a function $f$ is called (discrete) *convolution*. For continuous functions integration is used in place of the addition of averaging.

**(a)**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| 1 | 20 | 20 | 18 | 18 | 18 | 20 | 20 | 20 | 20 | 20 |
| 2 | 30 | 30 | 28 | 28 | 28 | 26 | 26 | 26 | 30 | 30 |
| 3 | 40 | 40 | 38 | 43 | 43 | 41 | 36 | 36 | 40 | 40 |
| 4 | 50 | 50 | 50 | 54 | 54 | 51 | 46 | 46 | 50 | 50 |
| 5 | 60 | 60 | 60 | 60 | 60 | 60 | 60 | 60 | 60 | 60 |

**(b)**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| 1 | 20 | 20 | 19 | 19 | 19 | 20 | 20 | 20 | 20 | 20 |
| 2 | 30 | 30 | 29 | 25 | 29 | 28 | 28 | 28 | 30 | 30 |
| 3 | 40 | 40 | 39 | 41 | 41 | 40 | 25 | 38 | 40 | 40 |
| 4 | 50 | 50 | 50 | 52 | 70 | 50 | 48 | 48 | 50 | 50 |
| 5 | 60 | 60 | 60 | 60 | 60 | 60 | 60 | 60 | 60 | 60 |

**Fig. 12.4** **a** Smoothing with the box filter. **b** Smoothing with a weighted filter

The *box filter* is:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

The results of the multiplications are added and the sum is divided by 9 to scale the result back to an intensity value.

The application of the filter to each pixel $(r, c)$ can be written explicitly as:

$$\begin{aligned} g(r,c) = ( & \\ & f(r-1,c-1) + f(r-1,c) + f(r-1,c+1) + \\ & f(r,c-1) + f(r,c) + f(r,c+1) + \\ & f(r+1,c-1) + f(r+1,c) + f(r+1,c+1) \\ & ) / 9 . \end{aligned}$$

The result of applying the box filter to the noisy image in Fig. 12.2b is shown in Fig. 12.4a.[3] The intensity values are no longer uniform but they are quite close to the original values, except where a noise pixels existed. The second row from the bottom shows that the noise value of 90 no longer appears; instead, all the values in the row are close together in the range 46–54.

The box filter gives equal importance to the pixel and all its neighbors, but a *weighted filter* uses different factors for different pixels. The following filter gives much more weight to the pixel itself than to its neighbors:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 8 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

It would be appropriate to use this filter if we think that a pixel almost certainly has its correct value, but we still want its neighbors to influence its value. After applying this filter, the result must be divided by 16 to scale the sum to an intensity value. Figure 12.4b shows the result of using the weighted filter. Looking again at the second

[3] The filter is not applied to the pixels in the boundary of the image to avoid exceeding the bounds of the array. Alternatively, the image can be padded with extra rows and columns.

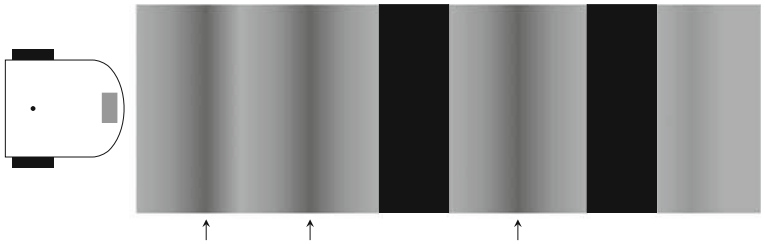


**Fig. 12.5** One-dimensional image enhancement

row from the bottom, the value of 90 has only been reduced to 70 because greater weight is given to the pixel relative to its neighbors.

**Activity 12.1: Image enhancement: smoothing**

- Print a sheet of paper with a gray-level pattern like the one shown in Fig. 12.5. The pattern has two black lines that we wish to detect but also three dark-gray areas (indicated by the arrows) that are likely to be incorrectly detected as lines.
- Program the robot so that it moves from left to right over the pattern, sampling the output of the ground sensor. Examine the output and set a threshold so that the robot detects both the black lines and the dark-gray areas. Modify the program so that in its second pass, it indicates (by light or sound) when it has detected a black line and a dark area.
- Modify the program so that it replaces every sample by the average of the intensity of the sample and its two neighbors. The robot should now detect the two black lines but not the gray areas.
- Experiment with different weights for the average.
- Experiment with different sampling rates. What happens if you sample the ground sensor at very short intervals?

### *12.3.2 Histogram Manipulation*

Figure 12.6a shows the pixels of a binary image: an image where each pixel is either black or white.[4] The image shows a $3 \times 5$ white rectangle on the black background. Figure 12.6b shows the same image with a lot of random noise added. By looking

[4]The values 10 for black and 90 for white have been used instead of the more usual 0 and 100 for clarity in printing the array.

**(a)**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| 1 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| 2 | 10 | 10 | 10 | **90** | **90** | **90** | **90** | **90** | 10 | 10 |
| 3 | 10 | 10 | 10 | **90** | **90** | **90** | **90** | **90** | 10 | 10 |
| 4 | 10 | 10 | 10 | **90** | **90** | **90** | **90** | **90** | 10 | 10 |
| 5 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |

**(b)**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 19 | 17 | 37 | 19 | 26 | 11 | 46 | 27 | 37 | 10 |
| 1 | 11 | 24 | 17 | 30 | 14 | 43 | 29 | 22 | 34 | 46 |
| 2 | 31 | 37 | 38 | 63 | 72 | 86 | 65 | 64 | 27 | 47 |
| 3 | 33 | 38 | 49 | 73 | 63 | 66 | 59 | 76 | 40 | 10 |
| 4 | 47 | 13 | 44 | 90 | 86 | 56 | 63 | 65 | 18 | 44 |
| 5 | 10 | 34 | 29 | 14 | 35 | 31 | 26 | 42 | 15 | 25 |

**Fig. 12.6** **a** Binary image without noise. **b** Binary image with noise

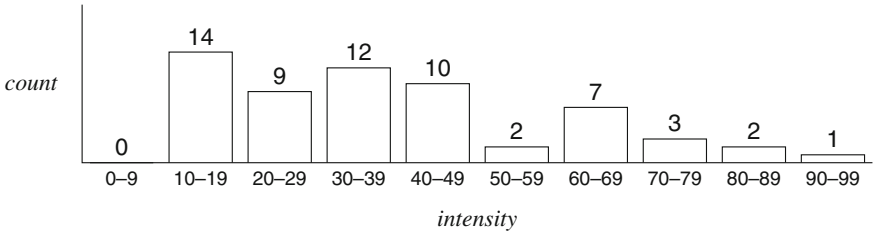



**Fig. 12.7** Histogram of the noisy image

at the image it is possible to identify the rectangle, but it is very difficult to do and smoothing the image won't help.

Let us now construct a *histogram* of the intensities (Fig. 12.7). A histogram is constructed of *bins*, where each bin stores a count of the pixels having a range of intensities. The histogram in the figure contains ten bins for intensities in the ranges 0–9, 10–19, …, 91–99. If we assume that the white rectangle is small relative to the background, it is easy to see from the histogram that there are two groups of pixels, those that are relatively dark and those that are relatively bright. A threshold of 50 or 60 should be able to distinguish between the rectangle and the background even in the presence of noise. In fact, a threshold of 50 restores the original image, while a threshold of 60 correctly restores 13 of the 15 pixels of the rectangle.

The advantage of histogram manipulation is that it is very efficient to compute even on large images. For each pixel, divide the intensity by the number of bins and increment the bin number:

```
for each pixel p
    bin_number ← intensity(p) / number_of_bins
    bins[bin_number] ← bins[bin_number] + 1
```

Compare this operation with the application of a $3 \times 3$ spatial filter which requires 9 multiplications, 8 additions and a division at each pixel. Furthermore, little memory is needed. We chose 10 bins so that Fig. 12.7 could display the entire histogram, but a full 8-bit grayscale histogram requires only 256 bins.

Choosing a threshold by examining a plot of the histogram is easy, and if you know roughly the fraction of the background covered by objects, the selection of the threshold can be done automatically.

Algorithms for histogram manipulation can perform more complex enhancement than the simple binary threshold we described here. In particular, there are algorithms for enhancing images by modifying the brightness and contrast of an image.

**Activity 12.2: Image enhancement: histogram manipulation**

- Modify the program in Activity 12.1 so that it computes the histogram of the samples.
- How does the histogram change if the number of samples is increased?
- Examine the histogram to determine a threshold that will be used to distinguish between the black lines and the background.
- Compute the sum of the contents of the bins until the sum is greater than a fraction (perhaps one-third) of the samples. Use the index of the last bin to set the threshold.

## 12.4 Edge Detection

Medical image processing systems need sophisticated image enhancement algorithms for modifying brightness and contrast, removing noise, etc. However, once the image is enhanced, interpretation of the image is performed by specialists who know which lines and shadows correspond with which organs in the body, and if the organs are normal or not. An autonomous robot does not have a human to perform interpretation: it needs to identify objects such as doors in a building, boxes in a warehouse and cars on the road. The first step is to extract features or segments such as lines, edges and areas.

Consider the $6 \times 6$ array of pixels in Fig. 12.8a. The intensity level of each row is uniform but there is a sharp discontinuity between the intensity level of rows 2 and 3. Clearly, this represents an edge between the dark area at the top and the light area at the bottom. Averaging will make the intensity change smoother and we lose the sharp change at the edge.

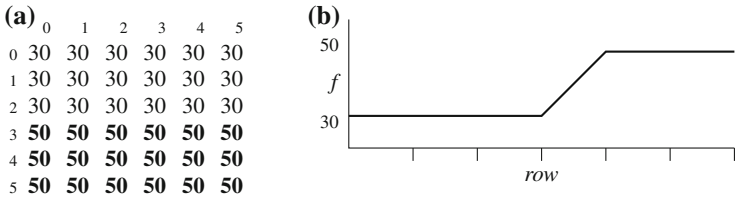



**Fig. 12.8** **a** Image with an edge. **b** Intensity of edge

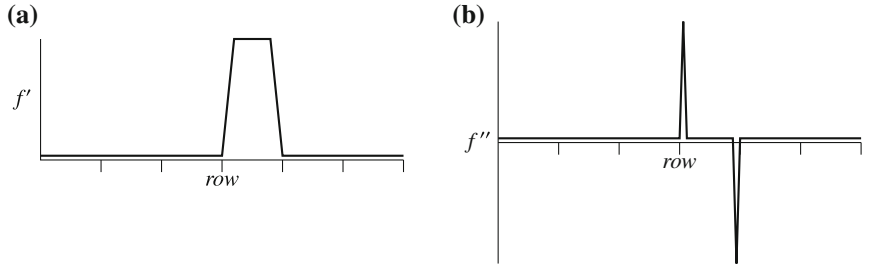



**Fig. 12.9** **a** First derivative of edge intensity. **b** Second derivative of edge intensity

Since averaging is an integrating operator that *removes* abrupt changes in intensities, it is not surprising that the differential operator can be used to *detect* abrupt changes that represent edges. Figure 12.8b plots the intensity against the row number along a single column of Fig. 12.8a, although the intensities are shown as lines instead of as discrete points. The intensity doesn't change for the first three pixels, then it rapidly increases and continues at the higher level. The first derivative $f'$ of a function $f$ is zero when $f$ is constant, positive when $f$ increases and negative when $f$ decreases. This is shown in Fig. 12.9a. An edge can be detected by searching for a rapid increase or decrease of the first derivative of the image intensity.

In practice, it is better to use the second derivative. Figure 12.9b shows a plot of $f''$, the derivative of $f'$ in Fig. 12.9a. The positive spike followed by the negative spike indicates a transition from dark to light; if the transition were from light to dark, the negative spike would precede the positive spike.

There are many digital derivative operators. A simple but effective one is the *Sobel filter*. There are two filters, one for detecting horizontal edges (on the left) and other for detecting vertical edges (on the right):

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \qquad \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}.$$

A characteristic of a derivative filter is that the sum of its elements must equal zero. The reason is that if the operator is applied to a pixel whose intensity is the same as that of all its neighbors, the result must be zero. Look again at Figs. 12.8b and 12.9a where the derivative is zero when the intensity is constant.

When the Sobel filters are applied to the pixel array in Fig. 12.8a, the result clearly detects that there is a horizontal edge (Fig. 12.10a) but no vertical edge (Fig. 12.10b).

Sobel filters are very powerful because they can not only detect an edge but also compute the angle of the edge within the image. Figure 12.11 shows an image with an edge running diagonally from the upper left to the lower right. The results of applying the two Sobel filters are shown in Fig. 12.12a, b. From the magnitudes and signs of the elements of these arrays, the angle of the edge can be computed as described in [3, Sect. 4.3.1].

**(a)**

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 80 | 80 | 80 | 80 | 0 |
| 3 | 0 | 80 | 80 | 80 | 80 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 |

**(b)**

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 |

**Fig. 12.10** **a** Sobel horizontal edge. **b** Sobel vertical edge

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 30 | 30 | 30 | 30 | 30 | 30 |
| 1 | **50** | 30 | 30 | 30 | 30 | 30 |
| 2 | **50** | 50 | 30 | 30 | 30 | 30 |
| 3 | **50** | **50** | **50** | 30 | 30 | 30 |
| 4 | **50** | **50** | **50** | **50** | 30 | 30 |
| 5 | **50** | **50** | **50** | **50** | **50** | 30 |

**Fig. 12.11** Diagonal edge

**(a)**

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | **60** | 20 | 0 | 0 | 0 |
| 2 | 0 | **60** | **60** | 20 | 0 | 0 |
| 3 | 0 | 20 | **60** | **60** | 20 | 0 |
| 4 | 0 | 0 | 20 | **60** | **60** | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 |

**(b)**

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | **−60** | −20 | 0 | 0 | 0 |
| 2 | 0 | **−60** | **−60** | −20 | 0 | 0 |
| 3 | 0 | −20 | **−60** | **−60** | −20 | 0 |
| 4 | 0 | 0 | −20 | **−60** | **−60** | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 |

**Fig. 12.12** **a** Sobel horizontal filter on a diagonal edge. **b** Sobel vertical filter on a diagonal edge

**Activity 12.3: Detecting an edge**

- Print out a pattern with a sharp edge (Fig. 12.13).
- Adapt the program from Activity 12.1 to cause the robot to sample and store the ground sensor as the robot moves over the pattern from left to right. Apply a derivative filter to the samples.
- During a second pass over the pattern, the robot indicates when the value of the derivative is not close to zero.
- What happens if the robot moves over the pattern from right to left?
- When applying the filter, the results must be stored in a separate array, not in the array used to store the pixels. Why?

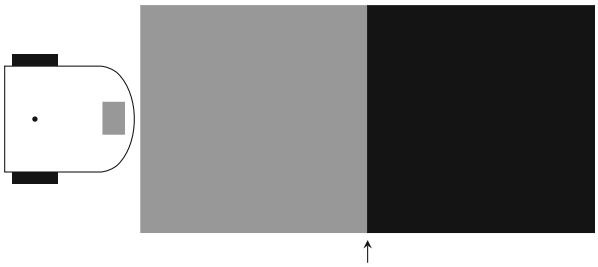


**Fig. 12.13** Edge detection activity

## 12.5 Corner Detection

The black rectangle within the gray background in Fig. 12.14a is more than just a set of edges. The vertical edges form two corners with the horizontal edge. Here we describe two algorithms for identifying corners in an image. For simplicity, we assume that the corners are aligned with the rectangular image.

We know how to detect edges in an image. A corner is defined by the intersection of a vertical edge and a horizontal edge. Figure 12.14b is the $6 \times 10$ pixel array for the image in Fig. 12.14a. If we apply the Sobel edge detectors to this pixel array, we obtain two vertical edges (Fig. 12.15a) and one horizontal edge (Fig. 12.15b).

The intersection is defined for pixels for which the sum of the absolute values in the two Sobel edge arrays is above a threshold. With a threshold of 30, the edges intersect in the pixels (2, 3) and (2, 7) which are the corners.

**(a)**

**(b)**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 |
| 1 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 |
| 2 | 30 | 30 | 30 | **50** | **50** | **50** | **50** | **50** | 30 | 30 |
| 3 | 30 | 30 | 30 | **50** | **50** | **50** | **50** | **50** | 30 | 30 |
| 4 | 30 | 30 | 30 | **50** | **50** | **50** | **50** | **50** | 30 | 30 |
| 5 | 30 | 30 | 30 | **50** | **50** | **50** | **50** | **50** | 30 | 30 |

**Fig. 12.14** **a** Image of a corner. **b** Pixel array of a corner

**(a)**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | **20** | **20** | 0 | 0 | 0 | **−20** | **−20** | 0 |
| 2 | 0 | 0 | **60** | **60** | 0 | 0 | 0 | **−60** | **−60** | 0 |
| 3 | 0 | 0 | **80** | **80** | 0 | 0 | 0 | **−80** | **−80** | 0 |
| 4 | 0 | 0 | **80** | **80** | 0 | 0 | 0 | **−80** | **−80** | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**(b)**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | **20** | **60** | **80** | **80** | **80** | **60** | **20** | 0 |
| 2 | 0 | 0 | **20** | **60** | **80** | **80** | **80** | **60** | **20** | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Fig. 12.15** **a** Vertical edges. **b** Horizontal edge

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 8 | 7 | 6 | 5 | 5 | 5 | 6 | 7 | 0 |
| 2 | 0 | 8 | 6 | **3** | 5 | 5 | 5 | **3** | 6 | 0 |
| 3 | 0 | 8 | 5 | 5 | 8 | 8 | 8 | 5 | 5 | 0 |
| 4 | 0 | 8 | 5 | 5 | 8 | 8 | 8 | 5 | 5 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Fig. 12.16** Similar neighbors

A uniform area, an edge and a corner can be distinguished by analyzing the neighbors of a pixel. In a uniform area, all the neighbors of the pixel have approximately the same intensity. At an edge, the intensities of the neighbors of the pixel are very different in one direction but similar in the other direction. At a corner, the intensities of the neighbors of the pixel show little similarity. To detect a corner, count the number of similar neighbors for each pixel, find the minimum value and identify as corners those pixels with that minimum value. Figure 12.16 shows the counts for the pixels in Fig. 12.14b. As expected, the corner pixels $(2, 3)$ and $(2, 7)$ have the minimum number of similar neighbors.

**Activity 12.4: Detecting a corner**

- Implement corner detection by intersecting edges using a robot with two ground proximity sensors. The robot moves from the bottom of the image in Fig. 12.14a to the top. If placed over the black rectangle it does not detect a corner, while if it is placed so that one sensor is over the black rectangle and the other over the gray background it does detect the corner.
- Implement corner detection by similar neighbors. Repeatedly, check the current samples from the left and right sensors and the previous samples from the left and right sensors. If only one of the four samples is black, a corner is detected.

## 12.6 Recognizing Blobs

Figure 12.17a shows a *blob*: a roughly circular area of 12 pixels with high intensity on a background of low intensity. The blob does not have a well-defined boundary like a rectangle. The next to the last row also shows two artifacts of high intensity that are not part of the blob, although they may represent distinct blobs. Figure 12.17b shows the pixels after adding random noise. The task is to identify the blob in the presence of noise, without depending on a predefined intensity threshold and ignoring the artifacts. The independence of the identification from the overall intensity is

**(a)**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 |
| 1 | 30 | 30 | 30 | 30 | **80** | **80** | 30 | 30 | 30 | 30 |
| 2 | 30 | 30 | 30 | **80** | **80** | **80** | **80** | 30 | 30 | 30 |
| 3 | 30 | 30 | 30 | **80** | **80** | **80** | **80** | 30 | 30 | 30 |
| 4 | **80** | 30 | 30 | 30 | **80** | **80** | 30 | 30 | 30 | **80** |
| 5 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 |

**(b)**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 46 | 42 | 40 | 50 | 46 | 44 | 40 | 33 | 30 | 34 |
| 1 | 32 | 46 | 46 | 46 | **67** | **73** | 39 | 47 | 39 | 30 |
| 2 | 33 | 40 | 40 | **73** | **68** | **63** | **73** | 44 | 42 | 31 |
| 3 | 35 | 41 | 50 | **67** | **60** | **71** | **60** | 37 | 30 | 49 |
| 4 | **68** | 46 | 32 | 44 | **61** | **77** | 48 | 42 | 45 | **62** |
| 5 | 39 | 37 | 38 | 34 | 33 | 40 | 35 | 37 | 34 | 32 |

**Fig. 12.17** **a** Blob. **b** Blob with noise

**(a)**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 67 | 73 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 73 | 68 | 63 | 73 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 67 | 60 | 71 | 60 | 0 | 0 | 0 |
| 4 | 68 | 0 | 0 | 0 | 61 | 77 | 0 | 0 | 0 | 62 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**(b)**

**Fig. 12.18** **a** Blob after threshold. **b** Blob to detect

important so that the robot can fulfill its task regardless of the lighting conditions in the environment.

To ignore noise without predefining a threshold, we use a threshold which is defined in terms of the average intensity of the image. To separate the blobs from one another, we first find a pixel whose intensity is above the threshold and then *grow* the blob by adding neighboring pixels whose intensity is above the threshold. For the noisy image in Fig. 12.17b, the average intensity is 54. Since the blob presumably occupies a relatively small part of the background, it might be a good idea to take a threshold somewhat higher than the average, say 60.

Figure 12.18a shows the image after assigning 0 to all pixels below the threshold. The blob has been detected but so have the two artifacts. Algorithm 12.1 is an algorithm for isolating a single blob. First, search for some pixel that is non-zero; starting from the top left, this will be pixel $p_1 = (1, 4)$ with intensity 67. Now grow the blob that adding all neighbors of $p_1$ whose intensities are non-zero; these are $p_2 = (1, 5)$, $p_3 = (2, 3)$, $p_4 = (2, 4)$, $p_5 = (2, 5)$. Continue adding non-zero neighbors of each $p_i$ to the blob until no more pixels are added. The result will be the 12-pixel blob without the artifacts at $(4, 0)$, $(4, 9)$.

The algorithm works because the first non-zero pixel found belonged to the blob. If there were an isolated non-zero pixel at $(1, 1)$, this artifact would have been detected as a blob. If there is an estimate for the minimum size of a blob, the algorithm should be followed by a check that the blob is at least this size.

**Algorithm 12.1: Detecting a blob**

```
integer threshold
pixel p
set not-explored ← empty-set
set blob ← empty-set
1: set the threshold to the average intensity
2: set pixels below threshold to zero
3: find a non-zero pixel and add to not-explored
4: while not-explored not empty
5:     p ← some element of not-explored
6:     add p to blob
7:     remove p from not-explored
8:     add non-zero neighbors of p to not-explored
```

Check that Algorithm 12.1 is not sensitive to the intensity level by subtracting the constant value 20 from all elements of the noisy image (Fig. 12.17b) and rerunning the algorithm. It should still identify the same pixels as belonging to the blob.

**Activity 12.5: Detecting a blob**

- Write a program that causes the robot to sample the ground sensor as it moves from left to right over the pattern in Fig. 12.18b.
- Compute the average intensity and set the threshold to the average.
- On a second pass over the pattern, after detecting the first sample from the black rectangle that is below the threshold, the robot provides an indication (by light or sound) as long as it moves over the rectangle.
- The robot should consider the second black rectangle as an artifact and ignore it.

**Activity 12.6: Recognizing a door**

- In Fig. 12.19a, the gray rectangle represents an open door in a dark wall represented by the black rectangles. Figure 12.19b represents a dark wall between two gray open doors. If you run the program from Activity 12.3, you will see that two edges are detected for both patterns. Modify the program so that the robot can distinguish between the two patterns.

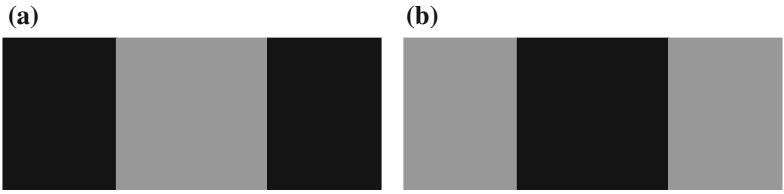



**Fig. 12.19** **a** Recognize the door. **b** This is not a door

## 12.7 Summary

In human beings and most animals vision is the most important sensor and a large portion of the brain is devoted to interpreting visual signals. Robots can use vision to perform advanced tasks within an environment that is constantly changing. The technology of digital cameras is highly advanced and the cameras can transfer high-resolution pixel arrays to the robot's computer. Algorithms for digital image processing enhance and interpret these images.

Enhancement algorithms remove noise, improve contrast and perform other operations that do not depend on what objects appear in an image. They use spatial filters that modify the intensity of each pixel based on the intensities of its neighbors. Histogram modification uses the global distribution of intensities in an image to modify individual pixels.

Following image enhancement, algorithms identify the objects in the image. They start by detecting simple geometric properties like edges and corners, and then proceed to identify the objects that appear in the image.

## 12.8 Further Reading

Gonzalez and Woods [1] is a comprehensive textbook on digital image processing that includes the mathematical fundamentals of the topic. Russ [2] is a reference work on image processing. Szeliski [4] is a book on computer vision which goes beyond image processing and focuses on constructing 3D models of images. For applications of image processing in robotics, see [3, Chap. 4].

## References

1. Gonzalez, R.C., Woods, R.E.: Digital Image Processing, 3rd edn. Pearson, Boston (2008)
2. Russ, J.C.: The Image Processing Handbook, 6th edn. CRC Press, Boca Raton (2011)
3. Siegwart, R., Nourbakhsh, I.R., Scaramuzza, D.: Introduction to Autonomous Mobile Robots, 2nd edn. MIT Press, Cambridge (2011)
4. Szeliski, R.: Computer Vision: Algorithms and Applications. Springer, Berlin (2011)

# Chapter 13
# Neural Networks

Chapter 3 described reactive behaviors inspired by the work of Valentino Braitenberg. The control of simple Braitenberg vehicles is very similar to the control of a living organism by its biological *neural network*. This term refers to the nervous system of a living organism, including its brain and the nerves that transmit signals through the body. Computerized models of neural networks are an active topic of research in artificial intelligence. *Artificial neural networks (ANNs)* enable complex behavior to be implemented using a large number of relatively simple abstract components that are modeled on neurons, the components of biological neural networks. This chapter presents the use of ANNs to control the behavior of robots.

Following a brief overview of the biological nervous system in Sect. 13.1, Sect. 13.2 defines the ANN model and Sect. 13.3 shows how it can be used to implement the behavior of a Braitenberg vehicle. Section 13.4 presents different network topologies. The most important characteristic of ANNs is their capability for learning which enables them to adapt their behavior. Section 13.5 presents an overview of learning in ANNs using the Hebbian rule.

## 13.1 The Biological Neural System

The nervous system of living organisms consists of cells called *neurons* that process and transmit information within the body. Each neuron performs a simple operation, but the combination of these operations leads to complex behavior. Most neurons are concentrated in the brain, but others form the nerves that transmit signals to and from the brain. In *vertebrates* like ourselves, many neurons are concentrated in the spinal cord which efficiently transmit signals throughout the body. There are an immense number of neurons in a living being: the human brain has about 100 billion neurons while even a mouse brain has about 71 million neurons [2].



M. Ben-Ari and F. Mondada, *Elements of Robotics*,
https://doi.org/10.1007/978-3-319-62533-1_13

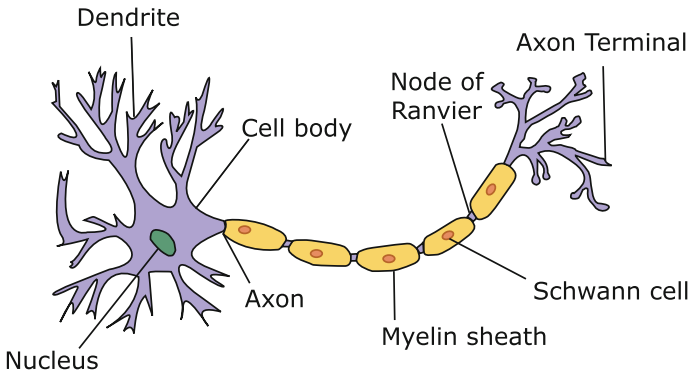



**Fig. 13.1** Structure of a neuron. *Source* https://commons.wikimedia.org/wiki/File:Neuron.svg by Dhp1080 [CC BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0) or GFDL https://en.wikipedia.org/wiki/en:GNU_Free_Documentation_License)], via Wikimedia Commons

Figure 13.1 shows the structure of a neuron. It has a main body with a *nucleus* and a long fiber called an *axon* that allows one neuron to connect with another. The body of a neuron has projections called *dendrites*. Axons from other neurons connect to the dendrites through *synapses*. Neurons function through biochemical processes that are well understood, but we can abstract these processes into *pulses* that travel from one neuron to another. Input pulses are received through the synapses into the dendrites and from them to the body of the neuron, which processes the pulses and in turn transmits an output pulse through the axon. The processing in the body of a neuron can be abstracted as a function from the input pulses to an output pulse, and the synapses regulate the transmission of the signals. Synapses are adaptive and are the primary element that makes memory and learning possible.

## 13.2 The Artificial Neural Network Model

An artificial neuron is a mathematical model of a biological neuron (Fig. 13.2a, b; see Table 13.1 for a list of the symbols appearing in ANN diagrams). The body of the neuron is a node that performs two functions: it computes the sum of the weighted input signals and it applies an output function to the sum. The input signals are multiplied by weights before the sum and output functions are applied; this models the synapse. The output function is usually nonlinear; examples are: (1) converting the neuron's output to a set of discrete values (turn a light on or off); (2) limiting the range of the output values (the motor power can be between $-100$ and 100; (3) normalizing the range of output values (the volume of a sound is between 0 (mute) and 1 (maximum).

Artificial neurons are analog models, that is, the inputs, outputs, weights and functions can be floating point numbers. Here we start with an unrealistic activity

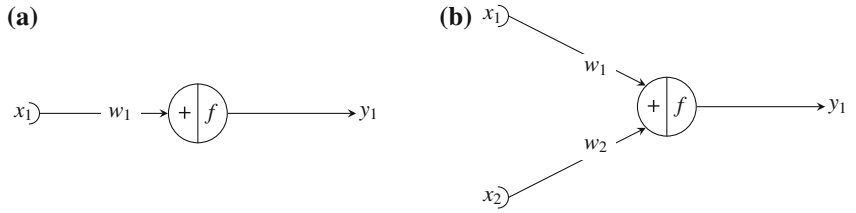



**Fig. 13.2** **a** ANN: one neuron with one input. **b** ANN: one neuron with two inputs

**Table 13.1** Symbols used in ANN diagrams

| Symbol | Meaning |
|---|---|
| $f$ | Neuron output function |
| $+$ | Sum of the inputs |
| $x_i$ | Inputs |
| $y_i$ | Outputs |
| $w_i$ | Weights for the inputs |
| 1 | Constant input of value 1 |

that demonstrates how artificial neurons work within the familiar context of digital logic gates.

Figure 13.3a shows an artificial neuron with two inputs, $x_1$ and 1, and one output $y$. The meaning of the input 1 is that the input is not connected to an external sensor, but instead returns a constant value of 1. The input value of $x_1$ is assumed to be 0 or 1. The function $f$ is:

$$f(x) = 0 \quad \text{if } x < 0$$
$$f(x) = 1 \quad \text{if } x \geq 0\,.$$

Show that with the given weights the neuron implements the logic gate for not.

**Activity 13.1: Artificial neurons for logic gates**

- The artificial neuron in Fig. 13.3b has an additional input $x_2$. Assign weights $w_0$, $w_1$, $w_2$ so that $y$ is 1 only if the values of $x_1$ or $x_2$ (or both) are 1. This implements the logic gate for or.
- Assign weights $w_0$, $w_1$, $w_2$ so that $y$ is 1 only if the values of $x_1$ and $x_2$ are both 1. This implements the logic gate for and.
- Implement the artificial neurons for logic gates on your robot. Use two sensors, one for $x_1$ and one for $x_2$. Use the output $y$ (mapped by $f$, if necessary) so that an output of 0 gives one behavior and an output of 1 another behavior, such as turning a light on or off, or starting and stopping the robot.

The following activity explores analog processing in an artificial neuron.

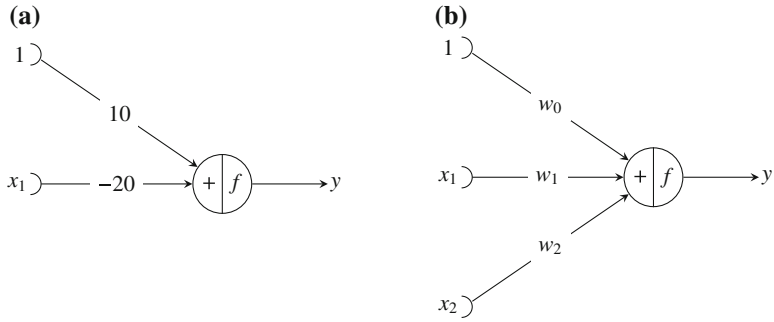



**Fig. 13.3** **a** Artificial neuron for the not gate. **b** Artificial neuron for the and and or gates

**Activity 13.2: Analog artificial neurons**

- Implement the artificial neuron shown in Fig. 13.2a so that it demonstrates the following behavior. The input to the neuron will be the reading of a proximity sensor at the front of the robot. The output will be one or both of the following: (1) the intensity of a light on the robot or the volume of the sound from a speaker on the robot; (2) the motor power applied to both the left and right motors so that the robot retreats from an object detected by the sensor.
- The output value will be proportional to the input value: the closer the object, the greater the intensity (or volume); the closer the object, the faster the robot retreats from the object.
- Modify the implementation so that there are two inputs from two proximity sensors (Fig. 13.2b). Give different values to the two weights $w_1$, $w_2$ and show that the sensor connected to the input with the larger weight has more effect on the output.

## 13.3 Implementing a Braintenberg Vehicle with an ANN

Figure 13.4 shows a robot inspired by a Braitenberg vehicle whose behavior is implemented using a simple neural network. We describe the ANN in detail and then give several activities that ask you to design and implement the algorithm.

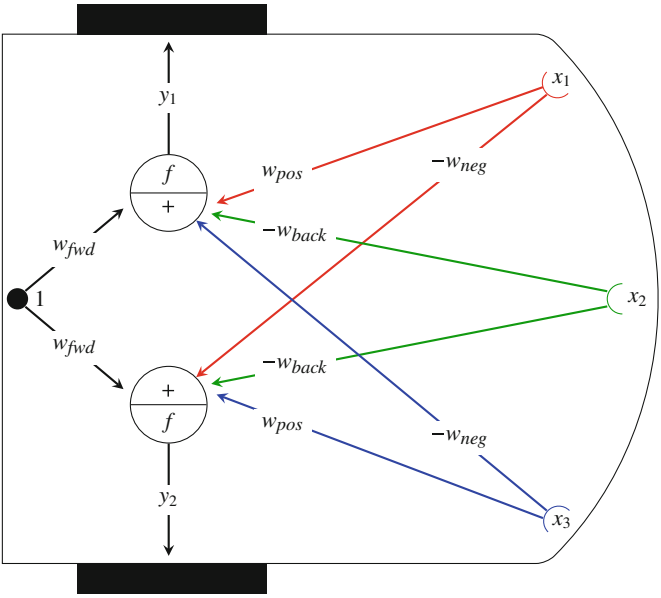



**Fig. 13.4** Neural network for obstacle avoidance

**Specification (obstacle avoidance)**:
The robot has three forward-facing sensors.

- The robot moves forwards unless it detects an obstacle.
- If the obstacle is detected by the center sensor, the robot moves slowly backwards.
- If the obstacle is detected by the left sensor, the robot turns right.
- If the obstacle is detected by the right sensor, the robot turns left.

Figure 13.4 shows the two neurons whose outputs control the power sent to the motors of the wheels of the robot. Table 13.2 lists the symbols used in the figure.

Each neuron has four inputs. The function $f$ must be nonlinear in order to limit the maximum forward and backward speeds. The large dot at the back of the robot

**Table 13.2** Symbols in Fig. 13.4 in addition to those in Table 13.1

| Symbol | Meaning |
|---|---|
| $w_{fwd}$ | Weight for forward movement |
| $w_{back}$ | Weight for backward movement |
| $w_{pos}$ | Weight for positive wheel rotation |
| $w_{neg}$ | Weight for negative wheel rotation |

denotes a constant input of 1 that is weighted by $w_{fwd}$. This ensures that in the absence of signals from the sensors, the robot will move forwards. When implementing the ANN you need to find a weight so that the output motor powers are reasonable in the absence input from the sensors. The weight should also ensure that the constant input is similar to inputs from the sensors.

The $x_1, x_2, x_3$ values come from the sensors that return zero when there is no object and an increasing positive value when approaching an object. The center sensor is connected to both neurons with a negative weight $-w_{back}$ so that if an obstacle is detected the robot will move backwards. This weight should be set to a value that causes the robot to move backwards slowly.

The left and right sensors are connected to the neurons with a positive weight for the neuron controlling the near wheel and a negative weight for the neuron controlling the far wheel. This ensures that robot turns away from the obstacle.

The following activity asks you to think about the relative values of the weights.

**Activity 13.3: ANN for obstacle avoidance: design**

- What relation must hold between $w_{fwd}$ and $w_{back}$?
- What relation must hold between $w_{fwd}$ and $w_{pos}$ and between $w_{fwd}$ and $w_{neg}$?
- What relation must hold between $w_{back}$ and $w_{pos}$ and between $w_{back}$ and $w_{neg}$?
- What relation must hold between $w_{pos}$ and $w_{neg}$?
- What happens if the obstacle is detected by both the left and center sensors?

In the following activities, you will have to experiment with the weights and the functions to achieve the desired behavior. Your program should use a data structure like an array so that it is easy to change the values of the weights.

**Activity 13.4: ANN for obstacle avoidance: implementation**

- Write a program for obstacle avoidance using the ANN in Fig. 13.4.

**Activity 13.5: ANN for obstacle attraction**

- Write a program to implement obstacle attraction using an ANN:
  - The robot moves forwards.
  - If the center sensor detects that the robot is *very close* to the obstacle, it stops.
  - If an obstacle is detected by the left sensor, the robot turns left.
  - If an obstacle is detected by the right sensor, the robot turns right.

## 13.4 Artificial Neural Networks: Topologies

The example in the previous section is based on an artificial neural network composed of a single layer of two neurons, each with several inputs and a single output. This is a very simple topology for an artificial neural network; many other topologies can implement more complex algorithms (Fig. 13.5). Currently, ANNs with thousands or even millions of neurons arranged in many layers are used to implement *deep learning*. In this section we present an overview of some ANN topologies.

### *13.4.1 Multilayer Topology*

Figure 13.6a shows an ANN with several layers of neurons. The additional layers can implement more complex computations than a single layer. For example, with a single layer it is not possible to have the robot move forward when only one sensor detects an obstacle and move backwards when several sensors detect an obstacle. The reason

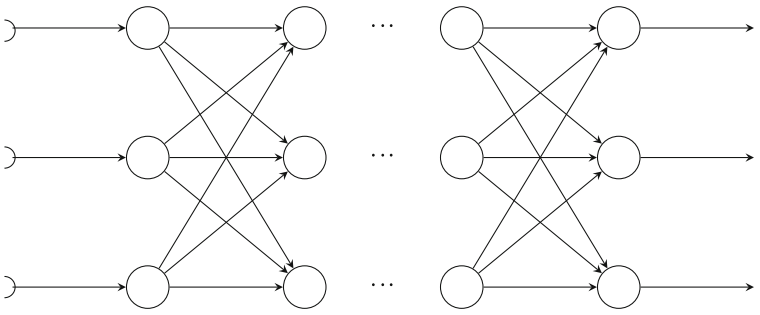


**Fig. 13.5** Neural network for deep learning

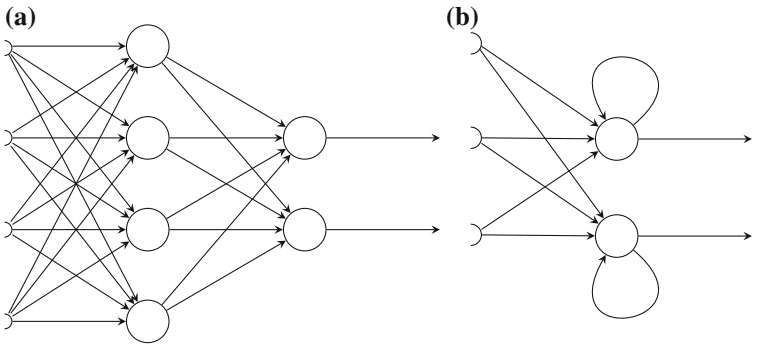



**Fig. 13.6** **a** Multilayer ANN. **b** ANN with memory

is that the function in a single layer linking the sensors and the motors is monotonic, that is, it can cause the motor to go faster when the sensor input increases or slower when the sensor input increases, but not both. The layer of neurons connected to the output is called the *output layer* while the internal layers are called the *hidden layers*.

**Activity 13.6: Multilayer ANNs**

- The goal of this activity is to understand how multilayer ANNs can perform computations that a single-layer ANN cannot. For the activity, assume that the inputs $x_i$ are in the range $-2.0$ to 2.0, the weights $w_i$ are in the range $-1.0$ to 1.0, and the functions $f$ limit the output values to the range $-1.0$ to 1.0.
- For the ANN consisting of a single neuron (Fig. 13.2a) with $w_1 = -0.5$, compute $y_1$ for inputs in increments of 0.2: $x_1 = -2.0, -1.8, \ldots, 0.0, \ldots, 1.8, 2.0$. Plot the results in a graph.
- Repeat the computation for several values of $w_1$. What can you say about the relationship between the output and the input?
- Consider the two-layer ANN shown in Fig. 13.7 with weights:

$$w_{11} = 1,\ w_{12} = 0.5,\ w_{21} = 1,\ w_{22} = -1\,.$$

Compute the values and draw graphs of the outputs of the neurons of the hidden layer (the left neurons) and the output layer (the right neuron). Can you obtain the same output from an ANN with only one layer?

**Activity 13.7: Multilayer ANN for obstacle avoidance**

- Design an ANN that implements the following behavior of a robot: There are two front sensors. When an object is detected in front of one of the sensors, the robot turns to avoid the object, but when an object is detected by both sensors, the robot moves backwards.

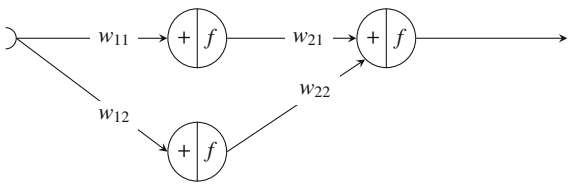



**Fig. 13.7** Two-layer ANN

### 13.4.2 Memory

An artificial neural network can have *recurrent connections* from the output of a neuron to the input of a neuron in the same layer (including itself). Recurrent connections can be used to implement memory. Consider the Braitenberg vehicle for obstacle avoidance (Fig. 13.4). It only turns when obstacles are detected by the sensors. When they are no longer detected, the robot does not continue to turn. By adding recurrent connections we can introduce a memory effect that causes the robot to continue turning. Suppose that each of the sensors causes an input of 0.75 to the neurons and that causes the output to be saturated to 1.0 by the non-linear output function. If the sensors no longer detect the obstacle, the inputs become 0, but the recurrent connection adds an input of 1.0 so the output remains 1.0.

**Activity 13.8: ANN with memory**

- Consider the network in Fig. 13.6b with an output function that saturates to 0 and 1. The inputs and most weights are also between 0 and 1. What happens if the weight of the recurrent connections in the figure is higher than 1? What happens if it is between 0 and 1?
- Modify the implementation of the network in Fig. 13.4 to add recurrent connections on the two output neurons. What is their effect in the obstacle-avoidance behavior of the robot?

### 13.4.3 Spatial Filter

A camera is a sensing device constructed from a large number of adjacent sensors (one for each pixel). The values of the sensors can be the inputs to an ANN with a large number of neurons in the first layer (Fig. 13.8). Nearby pixels will be input to adjacent neurons. The network can be used to extract local features such as differences of intensity between adjacent pixels in an image, and this local property can be used for tasks like identifying edges in the image. The number of layers may be one or more. This topology of neurons is called a *spatial filter* because it can be used as a filter before a layer that implements an algorithm for obstacle avoidance.

**Example** The ANN in Fig. 13.8 can be used to distinguish between narrow and wide objects. For example, both a leg of a chair and a wall are detected as objects, but the former is an obstacle that can be avoided by a sequence of turns whereas a wall cannot be avoided so the robot must turn around or follow the wall.

Suppose that the leg of the chair is detected by the middle sensor with a value of 60, but since the leg is narrow the other sensors return the value 0. The output values of the ANN (from top to bottom) are:

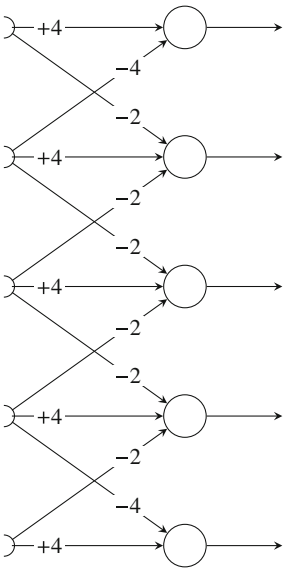



**Fig. 13.8** ANN for spatial filtering

$$
\begin{aligned}
(0 \times 4) + (0 \times -4) &= 0 \\
(0 \times -2) + (0 \times 4) + (60 \times -2) &= -120 \\
(0 \times -2) + (60 \times 4) + (0 \times -2) &= +400 \\
(60 \times -2) + (0 \times 4) + (0 \times -2) &= -120 \\
(0 \times 4) + (0 \times -4) &= 0\,.
\end{aligned}
$$

When the robot approaches a wall all the sensors will return more or less the same values, say, 45, 50, 40, 55, 50. The output values of the ANN are:

$$
\begin{aligned}
(45 \times 4) + (50 \times -4) &= -20 \\
(45 \times -2) + (50 \times 4) + (40 \times -2) &= +30 \\
(50 \times -2) + (40 \times 4) + (55 \times -2) &= -50 \\
(40 \times -2) + (55 \times 4) + (50 \times -2) &= +40 \\
(55 \times -4) + (50 \times 4) &= -20\,.
\end{aligned}
$$

Even though 48, the average value returned by the sensors detecting the wall, is about the same as the value 60 returned when detecting the leg of the chair, the outputs of the ANNs are clearly distinguishable. The first set of values has a high peak value surrounded by neighbors with large negative values, while second set is a relatively flat set of values in the range $-50$ to 40. The layer of neurons can confidently identify whether the object is narrow or wide.

**Activity 13.9: ANN for spatial filtering**

- Implement the ANN for spatial filtering in Fig. 13.8.
- The inputs to the ANN are the readings of five proximity sensors facing forwards. If only one sensor detects an object, the robot turns to face the object. If the center sensor is the one that detects the object, the robot moves forwards.
- Implement three behaviors of the robot when it detects a wall defined as all five sensors detecting an object:
  - The robot stops.
  - The robot moves forwards.
  - The robot moves backwards.

  Remember that there are no if-statements in an artificial neural network; you can only add additional neurons or change the weights associated with the inputs of the neurons. Look again at Activity 13.7 which used two levels of neurons to implement a similar behavior.
- The implementation will involve adding two additional neurons whose inputs are the outputs of the first layer. The output of the first neuron will set the power setting of the left motor and the output of the second neuron will set the power setting of the right motor.
- What happens if an object is detected by two adjacent sensors?

## 13.5 Learning

Setting the weights manually is difficult even for very small networks such as those presented in the previous sections. In biological organisms synapses have a plasticity that enables *learning*. The power of artificial neural networks comes from their ability to learn, unlike ordinary algorithms that have to be specified to the last detail. There are many techniques for learning in ANNs; we describe one of the simpler techniques in this section and show how it can be used in the neural network for obstacle avoidance.

### *13.5.1 Categories of Learning Algorithms*

There are three main categories of learning algorithms:

- **Supervised learning** is applicable when we know what output is expected for a set of inputs. The error between the desired and the actual outputs is used to correct the weights in order to reduce the error. Why is it necessary to train a network if we already know how it should behave? One reason is that the network is required to provide outputs in situations for which it was not trained. If the weights are adjusted so that the network behaves correctly on known inputs, it is reasonable to assume that its behavior will be more or less correct on other inputs. A second reason for training a network is to simplify the learning process: rather than directly relate the outputs $\{y_i\}$ to specific values of the inputs $\{x_i\}$, it is easier to place the network in several different situations and to tell it which outputs are expected in each situation.
- In **reinforcement learning** we do not specify the exact output value in each situation; instead, we simply tell the network if the output it computes is good or not. Reinforcement is appropriate when we can easily distinguish correct behavior from incorrect behavior, but we don't really care what the exact output is for each situation. In the next section, we present reinforcement learning for obstacle avoidance by a robot; for this task it is sufficient that the robot avoid the obstacle and we don't care what motor settings are output by the network as long as the behavior is correct.
- **Unsupervised learning** is learning without external feedback, where the network adapts to a large number of inputs. Unsupervised learning is not appropriate for achieving specified goals; instead, it is used in classification problems where the network is presented with raw data and attempts to find trends within the data. This approach to learning is the topic of Chap. 14.

### *13.5.2 The Hebbian Rule for Learning in ANNs*

The *Hebbian rule* is a simple learning technique for ANNs. It is a form of reinforcement learning that modifies the weights of the connections between the neurons. When the network is doing something good we reinforce this good answer: if the output value of two connected neurons is similar, we increase the weight of the connection linking them, while if they are different, we decrease the weight. If the robot is doing something wrong, we can either decrease the weights of similar connected neurons or do nothing.

The change in the weight of the connection between neuron $k$ and neuron $j$ is described by the equation:

$$\Delta w_{kj} = \alpha\, y_k\, x_j\,,$$

where $w_{kj}$ is the weight linking the neurons $k$ and $j$, $\Delta w_{kj}$ is the change of $w_{kj}$, $y_k$ is the output of neuron $k$ and $x_j$ the input of neuron $j$, and $\alpha$ is a constant that defines the speed of learning.

The Hebbian rule is applicable under two conditions:

- The robot is exploring its environment, encountering various situations, each with its own inputs for which the network computes a set of outputs.
- The robot receives information on which behaviors are good and which are not.

The evaluation of the quality of the robot's behavior can come from a human observer manually giving feedback; alternatively, an automatic system can be used to evaluate the behavior. For example, in order to teach the robot to avoid obstacles, an external camera can be used to observe the robot and to evaluate its behavior. The behavior is classified as bad when the robot is approaching an obstacle and as good when the robot is moving away from all the obstacles. It is important to understand that what is evaluated is not the *state* of the robot (close to or far from an obstacle), but rather the *behavior* of the robot (approaching or avoiding an obstacle). The reason is that the connections of the neural network generate behavior based on the state as measured by the sensors.

**Learning to avoid an obstacle**

Suppose that we want to teach a robot to avoid an obstacle. One way would be to let the robot move randomly in the environment, and then touch one key when it successfully avoids the obstacle and another when it crashes into the obstacle. The problem with this approach is that it will probably take a very long time for the robot to exhibit behavior that can be definitely characterized as positive (avoiding the obstacle) or negative (crashing it into the obstacle).

Alternatively, we can present the robot with several known situations and the required behavior: (1) detecting obstacle on the left and turning right is good; (2) detecting an obstacle on the right and turning left is good; (3) detecting an obstacle in front and moving backwards is good; (4) detecting an obstacle in front and moving forwards is bad.

This looks like supervised learning but it is not, because the feedback to the robot is used only to reinforce the weights linked to good behavior. Supervised learning would consist in quantifying the error between the desired and actual outputs (to the motors in this case), and using this error to adjust the weights to compute exact outputs. Feedback in reinforcement learning is binary: the behavior is good or not.

**The algorithm for obstacle avoidance**

Let us now demonstrate the Hebbian rule for learning on the problem of obstacle avoidance. Figure 13.9 is similar to Fig. 13.4 except that proximity sensors have been added to the rear of the robot. We have also changed the notation for the weights to make them more appropriate for expressing the Hebbian rule; in particular, the negative signs have been absorbed into the weights.

The obstacle-avoidance algorithm is implemented using several concurrent processes and will be displayed as a set of three algorithms. Algorithm 13.1 implements an ANN which reads the inputs from the sensors and computes the outputs to the motors. The numbers of inputs and outputs are taken from Fig. 13.9. Algorithm 13.2 receives evaluations of the robot's behavior from a human. Algorithm 13.3 performs the computations of the Hebbian rule for learning.

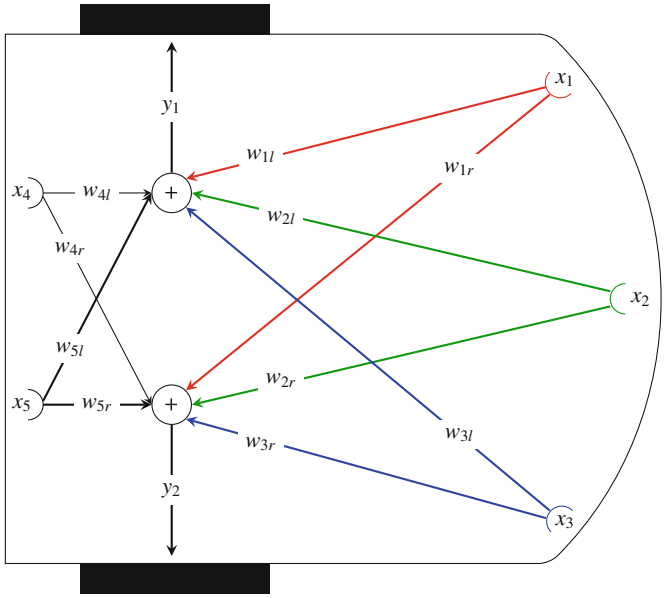



**Fig. 13.9** Neural network for demonstrating Hebbian learning

In Algorithm 13.1 a timer is set to a period such as 100 ms. The timer is decremented by the operating system (not shown) and when it expires, the outputs $y_1$ and $y_2$ are computed by Eq. 13.3 below. These outputs are then used to set the power of the left and right motors, and, finally, the timer is reset.

**Algorithm 13.1: ANN for obstacle avoidance**

```
integer period ← ···     // Timer period (ms)
integer timer ← period
float array[5]   x
float array[2]   y
float array[2,5] W
1: when timer expires
2:     x ← sensor values
3:     y ← W x
4:     left-motor-power ← y[1]
5:     right-motor-power ← y[2]
6:     timer ← period
```

There are five sensors that are read into the five input variables:

$$
\begin{aligned}
x_1 &\leftarrow \text{front left sensor} \\
x_2 &\leftarrow \text{front center sensor} \\
x_3 &\leftarrow \text{front right sensor} \\
x_4 &\leftarrow \text{rear left sensor} \\
x_5 &\leftarrow \text{rear right sensor}
\end{aligned}
$$

We assume that the values of the sensors are between 0 (obstacle not detected) and 100 (obstacle very close), and that the values of the motor powers are between $-100$ (full backwards power) and 100 (full forwards power). If the computation results in saturation, the values are truncated to the end points of the range, that is, a value less than $-100$ becomes $-100$ and a value greater than 100 becomes 100.[1] Recall that a robot with differential drive turns right by setting $y_1$ (the power of the left motor) to 100 and $y_2$ (the power of the right motor) to $-100$, and similarly for a left turn.

To simplify the presentation of Algorithm 13.1 we used vector notation where the inputs are given as a single column vector:

$$
\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} .
$$

Referring again to Fig. 13.9, the computation of the outputs is given by:

$$
y_1 \leftarrow w_{1l}x_1 + w_{2l}x_2 + w_{3l}x_3 + w_{4l}x_4 + w_{5l}x_5 \tag{13.1}
$$

$$
y_2 \leftarrow w_{1r}x_1 + w_{2r}x_2 + w_{3r}x_3 + w_{4r}x_4 + w_{5r}x_5 \, . \tag{13.2}
$$

Expressed in vector notation this is:

$$
\mathbf{y} = \begin{bmatrix} y_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} w_{1l} & w_{2l} & w_{3l} & w_{4l} & w_{5l} \\ w_{1r} & w_{2r} & w_{3r} & w_{4r} & w_{5r} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \mathbf{W}\,\mathbf{x} \, . \tag{13.3}
$$

To cause the algorithm to learn, feedback is used to modify the weights $\mathbf{W}$ (Algorithms 13.2, 13.3). Let us assume that there are four buttons on the robot or on a remote control, one for each direction forwards, backwards, left and right. Whenever we note that the robot is in a situation that requires a certain behavior, we touch the corresponding button. For example, if the left sensor detects an obstacle,

[1] Algorithm 13.1 declared all the variables as float because the weights are floating point numbers. If the sensor inputs and motor outputs are integers, type conversion will be necessary.

the robot should turn right. To implement this, there is a process for each button. These processes are shown together in Algorithm 13.2, where the forwards slashes / separate corresponding events and actions.

**Algorithm 13.2: Feedback on the robot's behavior**

```
when button {forward / backward / left / right} touched
    y1 ← {100 / −100 / −100 / 100}
    y2 ← {100 / −100 / 100 / −100}
```

The next phase of the algorithm is to update the connection weights according to the Hebbian rule (Algorithm 13.3).

**Algorithm 13.3: Applying the Hebbian rule**

```
x ← sensor values
for j in {1, 2, 3, 4, 5}
    wjl ← wjl + α y1 xj
    wjr ← wjr + α y2 xj
```

**Example** Assume that initially the weights are all zero. By Eqs. 13.1 and 13.2 the outputs are zero and the robot will not move.

Suppose now that an obstacle is placed in front of the left sensor so that $x_1 = 100$ while $x_2 = x_3 = x_4 = x_5 = 0$. Without feedback nothing will happen since the weights are still zero. If we touch the right button (informing the robot that the correct behavior is to turn right), the outputs are set to $y_1 = 100$, $y_2 = -100$ to turn right, which in turn leads to the following changes in the weights (assuming a learning factor $\alpha = 0.0001$):

$$w_{1l} \leftarrow 0 + (0.0001 \times 100 \times 100) = 10$$
$$w_{1r} \leftarrow 0 + (0.0001 \times -100 \times 100) = -10\,.$$

The next time that an obstacle is detected by the left sensor, the outputs will be non-zero:

$$y_1 \leftarrow (10 \times 100) + 0 + 0 + 0 + 0 = 1000$$
$$y_2 \leftarrow (-10 \times 100) + 0 + 0 + 0 + 0 = -1000\,.$$

After truncating to 100 and $-100$ these outputs will cause the robot to turn right.

The learning factor $\alpha$ determines the magnitude of the effect of $y_k\, x_j$ on the values of $w_{kj}$. Higher values of the factor cause larger effects and hence faster learning. Although one could think that a faster learning is always better, if the learning is too fast it can cause unwanted changes such as forgetting previous good situations or strongly emphasizing mistakes. The learning factor must be adjusted to achieve optimal learning.

**Activity 13.10: Hebbian learning for obstacle avoidance**

- Implement Algorithms 13.1–13.3 and teach your robot to avoid obstacles.
- Modify the program so that it *learns* to move forwards when it does not detect an obstacle.

## 13.6 Summary

Autonomous robots must function in environments characterized by a high degree of uncertainty. For that reason it is difficult to specify precise algorithms for robot behavior. Artificial neural networks can implement the required behavior in an uncertain environment by learning, that is, by modifying and improving the algorithm as the robot encounters additional situations. The structure of ANNs makes learning technically simple: An ANN is composed of a large number of small, simple components called neurons and learning is achieved by modifying the weights assigned to the connections between the neurons.

Learning can be supervised, reinforcement or unsupervised. Reinforcement learning is appropriate for learning robotic behavior because it requires the designer to specify only if an observed behavior is good or bad without quantifying the behavior. The Hebbian rule modifies the weights connecting neurons by multiplying the output of one neuron by the input of the neuron it is connected to. The result is multiplied by a learning factor that determines the size of the change in the weight and thus the rate of learning.

## 13.7 Further Reading

Haykin [1] and Rojas [4] are comprehensive textbooks on neural networks. David Kriesel wrote an online tutorial [3] that can be freely downloaded.

## References

1. Haykin, S.O.: Neural Networks and Learning Machines, 3rd edn. Pearson, Boston (2008)
2. Herculano-Houzel, S.: The human brain in numbers: a linearly scaled-up primate brain. Front. Hum. Neurosci. **3**, 31 (2009)
3. Kriesel, D.: A Brief Introduction to Neural Networks. http://www.dkriesel.com/en/science/neural_networks (2007)
4. Rojas, R.: Neural Networks: A Systematic Introduction. Springer, Berlin (1996)

# Chapter 14
# Machine Learning

Consider a robot that recognizes and grasps yellow objects (Fig. 14.1). It can use a color camera to identify yellow objects, but the objects will appear different in different environments, such as in sunlight, in a dark room, or in a showroom. Furthermore, it is hard to precisely define what "yellow" means: what is the boundary between yellow and lemon-yellow or between yellow and orange? Rather than write detailed instructions for the robot, we would prefer that the robot learn color recognition as it is performing the task, so that it could adapt to the environment where the task takes place. Specifically, we want to design a *classification algorithm* that can be *trained* to perform the task without supplying all the details in advance.

Classifications algorithms are a central topic in *machine learning*, a field of computer science and statistics that develops computations to recognize patterns and to predict outcomes without explicit programming. These algorithms extract *rules* from the raw data acquired by the system during a training period. The rules are subsequently used to classify a new object and then to take the appropriate action according to the class of the object. For the color-recognition task, we train the robot by presenting it with objects of different colors and telling the robot which objects are yellow and which are not. The machine learning algorithm generates a rule for color classification. When presented with new objects, it uses the rule to decide which objects are yellow and which are not.

The previous chapter presented artificial neural networks which perform a form of machine learning based upon *reinforcement*. In this chapter, we discuss statistical techniques based upon *supervised learning*: during the training period we tell the robot the precise answers, such as whether an object is yellow or not. Section 14.1 introduces the statistical techniques by developing an algorithm for distinguishing between objects of two colors. We present a technique for machine learning called *linear discriminant analysis (LDA)*. Sections 14.2 and 14.3 present LDA in the same context of distinguishing between two colors. LDA is based on the assumption that



M. Ben-Ari and F. Mondada, *Elements of Robotics*,
https://doi.org/10.1007/978-3-319-62533-1_14

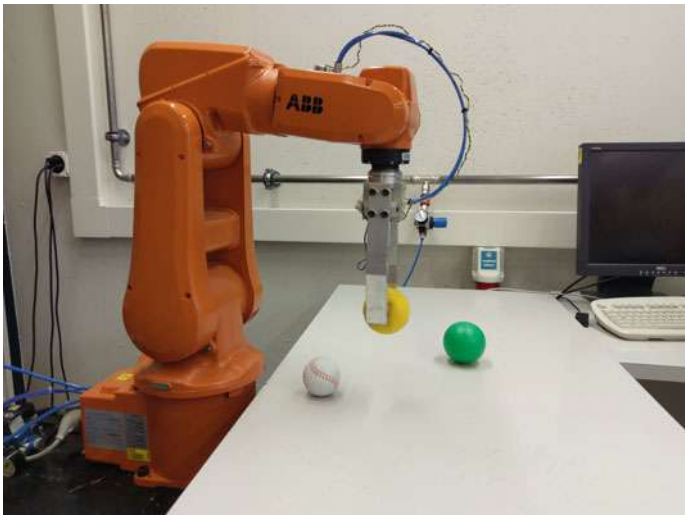



**Fig. 14.1** Robotic arm sorting colored balls

the data has specific statistical properties; if these do not hold, *perceptrons* can be used for classification as described in Sect. 14.4.

This chapter assumes that you are familiar with the concepts of *mean*, *variance* and *covariance*. Tutorials on these concepts appear in Appendices B.3 and B.4.

## 14.1 Distinguishing Between Two Colors

We start with the problem of distinguishing yellow balls from non-yellow balls. To simplify the task, we modify the problem to one of distinguishing dark gray areas from light gray areas that are printed on paper taped to the floor (Fig. 14.2). The robot uses two ground sensors that sample the reflected light as the robot moves over the two areas.

Figure 14.3 shows a plot of the values returned by sampling the two sensors.[1] The robot takes about 70 s to move from left to right, sampling the reflected light once per second. It is easy to see that the data shows significant variation, which is probably due to noise in the sensors and uneven printing. Even more problematic is the variability in the results returned by the two sensors. How can the robot learn to distinguish between the shades of gray given the variability in the samples and the sensors? We want to automatically create a rule to distinguish between the two shades of gray.

[1] The data used in this chapter are real data taken from an experiment with the Thymio robot.

**Fig. 14.2** Distinguishing two shades of *gray*

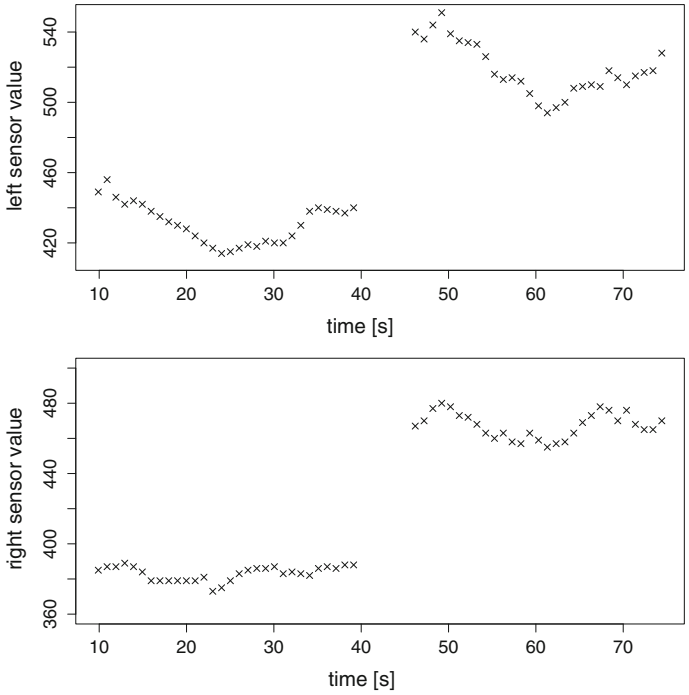



**Fig. 14.3** Plots of reflected light versus time for the left sensor (*top*) and the right sensor (*bottom*)

### *14.1.1 A Discriminant Based on the Means*

By examining the plots in Fig. 14.3, it is easy to see which samples are from the dark gray area and which are from the light gray area. For the left sensor, the values of the light gray area are in the range 500–550, while the values of the dark gray area are in the range 410–460. For the right sensor, the ranges are 460–480 and 380–400. For the left sensor, a threshold of 480 would clearly distinguish between light and dark gray, while for the right sensor a threshold of 440 would clearly distinguish between light and dark gray. But how can these optimal values be chosen automatically and how can we reconcile the thresholds of the two sensors?

Let us first focus on the left sensor. We are looking for a *discriminant*, a value that distinguishes samples from the two colors. Consider the values $max_{dark}$, the

maximum value returned by sampling dark gray, and $min_{light}$, the minimum value returned by sampling light gray. Under the reasonable assumption that $max_{dark} < min_{light}$, any value $x$ such that $max_{dark} < x < min_{light}$ can distinguish between the two shades of gray. The midpoint between the two values would seem to offer the most robust discriminant.

From Fig. 14.3 we see that $max_{dark} \approx 460$ occurs at about 10 s and $min_{light} \approx 500$ occurs at about 60 s, so we choose their average 480 as the discriminant. While this is correct for this particular data set, in general it is not a good idea to use the maximum and minimum values because they could be *outliers*: extreme values resulting from unusual circumstances, such as a hole in the paper which would incorrectly return a very high value in the dark gray area.

A better solution is to use *all* the data and the simplest function of all the data is the *mean* of the values. Let $\mu_{dark}$ denote the mean of the dark gray samples and $\mu_{light}$ the mean of the light gray samples. A good discriminant $\Delta$ is the midpoint between the two means:

$$\Delta = \frac{\mu_{dark} + \mu_{light}}{2} .$$

For the data in Fig. 14.3 the means for the left sensor and the discriminant are[2]:

$$\mu_{dark}^{left} = 431, \quad \mu_{light}^{left} = 519, \quad \Delta^{left} = \frac{431 + 519}{2} = 475 .$$

A similar computation gives the discriminant for the right sensor:

$$\Delta^{right} = 425 .$$

In order to obtain optimal recognition, we want an algorithm that is able to automatically decide which of the two discriminants is better. This is a preliminary stepping stone to the method described in Sect. 14.2, where a discriminant is computed by combining data from both sensors.

Intuitively, the greater the difference between the means of the light and dark areas:

$$\left| \mu_{dark}^{left} - \mu_{light}^{left} \right| , \quad \left| \mu_{dark}^{right} - \mu_{light}^{right} \right| ,$$

the easier it will be to place a discriminant between the two classes. The difference between the means of the left sensor (88) is a bit greater than the difference between the means of the right sensor (84). This leads us to choose the discriminant (475) computed from the means of the left sensor. However, from the plot in Fig. 14.4 it appears that this might not be the best choice because of the large variation in the samples of the left sensor.

[2] In this chapter, values will be rounded to the nearest integer.

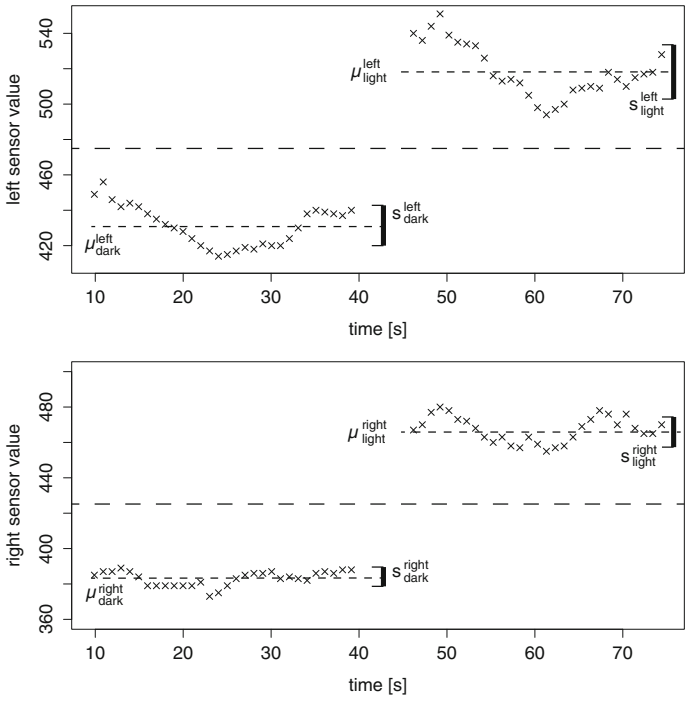



**Fig. 14.4** Figure 14.3 with means (*short dashes*), variances (*brackets*), discriminants (*long dashes*)

### *14.1.2 A Discriminant Based on the Means and Variances*

A better discriminant can be obtained if we consider not only the difference of the means but also the spread of the sample values around the mean. This is called the *variance* of the samples. The variance $s^2$ of a set of values $\{x_1, x_2, \ldots, x_{n-1}, x_n\}$ is[3]:

$$s^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \mu)^2 ,$$

where $\mu$ is the mean of the values in the set.

The variance computes the average of the distances of each sample from the mean of the samples. The distances are squared because a sample can be greater than or less than the mean, but we want a positive distance that shows how far the sample is from the mean.

The brackets in Fig. 14.4 show the four variances for the sets of samples of the light and dark areas for the left and right sensors. The difference between the left

[3] Appendix B.3 explains why $n - 1$ is used instead of $n$.

means is somewhat greater than the difference between the right means:

$$\left| \mu_{dark}^{left} - \mu_{light}^{left} \right| > \left| \mu_{dark}^{right} - \mu_{light}^{right} \right| ,$$

but the variances of the right sensor are much smaller than the corresponding variances of the left sensor:

$$\left(s_{dark}^{right}\right)^2 \ll \left(s_{dark}^{left}\right)^2 , \quad \left(s_{light}^{right}\right)^2 \ll \left(s_{light}^{left}\right)^2 .$$

The use of the variances enables better classification of the two sets, since a sensor with less variance is more stable and this facilitates classification.

A good discriminant can be obtained by combining information from the means and the variances. The quality of a discriminant $J_k$, for $k = left, right$, is given by:

$$J_k = \frac{\left(\mu_{dark}^k - \mu_{light}^k\right)^2}{\left(s_{dark}^k\right)^2 + \left(s_{light}^k\right)^2} . \tag{14.1}$$

To maximize $J$, the numerator—the distance between the means—should be large, and the denominator—the variances of the samples—should be small.

Table 14.1 displays the computations for the data set from Fig. 14.4. The quality criterion $J$ for the right sensor is much larger than the one for the left sensor. It follows that the midpoint between the means of the right sensor:

$$\Delta^{right} = \frac{383 + 467}{2} = 425$$

is a better discriminant than the midpoint of the means of the left sensor that would be chosen by considering only the differences of the means $|\mu_{dark} - \mu_{light}|$, which is slightly larger for the left sensor than for the right sensor.

**Table 14.1** The difference of the means and the quality criteria $J$

| | Left | | Right | |
|---|---|---|---|---|
| | Dark | Light | Dark | Light |
| $\mu$ | 431 | 519 | 383 | 467 |
| $s^2$ | 11 | 15 | 4 | 7 |
| $\lvert\mu_{dark} - \mu_{light}\rvert$ | | 88 | | 84 |
| $J$ | | 22 | | 104 |

### 14.1.3 Algorithm for Learning to Distinguish Colors

These computations are done by the robot itself, so the choice of the better discriminant and the better sensor is automatic. The details of the computation are given Algorithms 14.1 and 14.2.[4]

There are two classes $C_1, C_2$ and two sensors. During the learning phase, the robot samples areas of the two gray levels independently and then computes the criterion of quality $J$. The sampling and computation are done for each sensor, either one after another or simultaneously. After the learning phase, the robot uses midpoint of the means with the best $J$ value for recognition of gray levels.

**Algorithm 14.1: Distinguishing classes (learning phase)**

| | |
|---|---|
| float $\mathbf{X_1}, \mathbf{X_2}$ | // Sets of samples |
| float $\mu_1, \mu_2$ | // Means of $C_1, C_2$ |
| float $s_1, s_2$ | // Variances of $C_1, C_2$ |
| float $\mu[2]$ | // Means of $\mu_1, \mu_2$ |
| float $J[2]$ | // Criteria of quality |
| integer $k$ | // Index of max($J[1], J[2]$) |

1: for sensor i=1, 2
2: Collect a set of samples $\mathbf{X_1}$ from $C_1$
3: Collect a set of samples $\mathbf{X_2}$ from $C_2$
4: Compute means $\mu_1$ of $\mathbf{X_1}$ and $\mu_2$ of $\mathbf{X_2}$
5: Compute variances $s_1$ of $\mathbf{X_1}$ and $s_2$ of $\mathbf{X_2}$
6: Compute the mean $\mu[i] = \dfrac{\mu_1 + \mu_2}{2}$
7: Compute the criterion $J[i]$ from Eq. 14.1
8: $k \leftarrow$ index of max($J[1], J[2]$)
9: Output $\mu[k]$

**Algorithm 14.2: Distinguishing classes (recognition phase)**

| | |
|---|---|
| float $\mu \leftarrow$ input $\mu[k]$ from the learning phase | |
| float x | |

1: loop
2: $x \leftarrow$ get new sample
3: if $x < \mu$
4: assign $x$ to class $C_1$
5: else
6: assign $x$ to class $C_2$

[4] Boldface variables represent vectors or matrices.

**Activity 14.1: Robotic chameleon**

- Construct an environment as shown in Fig. 14.2. Print two pieces of paper with different uniform gray levels and tape them to the floor.
- Write a program that causes the robot to move at a constant speed over the area of one color and sample the reflected light periodically. Repeat for the other color.
- Plot the data, compute the means and the discriminant.
- Implement a program that classifies the measurements of the sensor. When the robot classifies a measurement it displays which color is recognized like a chameleon (or gives other feedback if changing color cannot be done).
- Apply the same method with a second sensor and compare the separability of the classes using the criterion $J$.
- Repeat the exercise with two very close levels of gray. What do you observe?

## 14.2 Linear Discriminant Analysis

In the previous section we classified samples of two levels of gray based on the measurements of one sensor out of two; the sensor was chosen automatically based on a quality criterion. This approach is simple but not optimal. Instead of choosing a discriminant based on one sensor, we can achieve better recognition by combining samples from both sensors. One method is called *linear discriminant analysis (LDA)* and is based upon pioneering work in 1936 by the statistician Ronald A. Fisher.

### *14.2.1 Motivation*

To understand the advantages of combining samples from two sensors, suppose that we need to classify objects of two colors: *electric violet (ev)* and *cadmium red (cr)*. Electric violet is composed largely of blue with a bit of red, while cadmium red is composed largely of red with a bit of blue. Two sensors are used: one measures the level of red and the other measures the level of blue. For a set of samples, we can compute their means $\mu_j^k$ and variances $(s_j^k)^2$, for $j = ev, cr$, $k = blue, red$.

The left plot in Fig. 14.5 shows samples of the electric violet objects contained within a dashed ellipse at the upper left and samples of the cadmium red objects contained within a dashed ellipse at the lower right. The centers of the ellipses are the means because the samples are symmetrically distributed with respect to the ellipses. The $y$-coordinate of $\mu_{ev}$ is the mean of the samples of the electric violet objects by the blue sensor and its $x$-coordinate is the mean of the samples of these

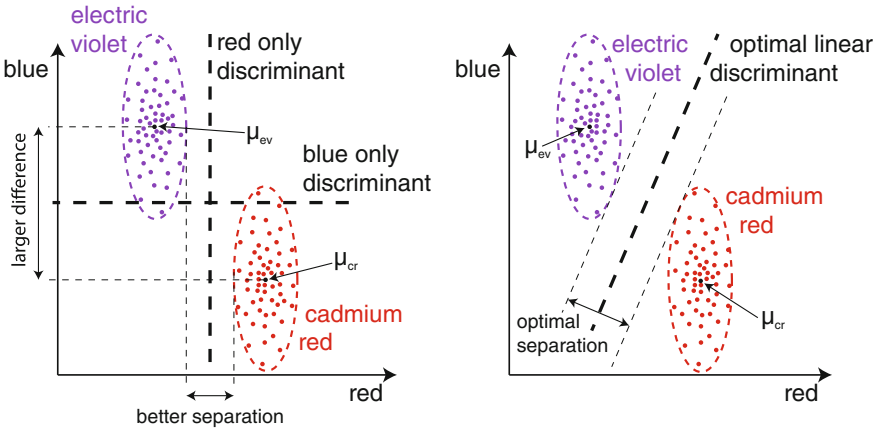



**Fig. 14.5** An optimal linear discriminant for distinguishing two colors

objects by the red sensor. The means of the sensors when sampling the cadmium red objects are similarly displayed. It is easy to see that the electric violet objects have more blue (they are higher up the $y$-axis), while the cadmium red objects have more red (they are further out on the $x$-axis).

From the diagram we see that there is a larger difference between the means for the blue sensor than between the means for the red sensor. At first glance, it appears that using the blue sensor only would give a better discriminant. However, this is not true: the dashed lines show that the red-only discriminant completely distinguishes between electric violet and cadmium red, while the blue-only discriminant falsely classifies some electric violet samples as cadmium red (some samples are below the line) and falsely classifies some cadmium red samples as electric violet (some samples are above the line).

The reason for this unintuitive result is that the blue sensor returns values that are widely spread out (have a large variance), while the red sensor returns values that are narrowly spread out (have a small variance), and we saw in Sect. 14.1 that classification is better if the variance is small. The right plot in Fig. 14.5 shows that by constructing a discriminant from both sensors it is possible to better separate the electric violet objects from the cadmium red objects. The discriminant is still linear (a straight line) but its slope is no longer parallel to one of the axes. This line is computed using the variances as well as the means. The method is called linear discriminant analysis because the discriminant is linear.

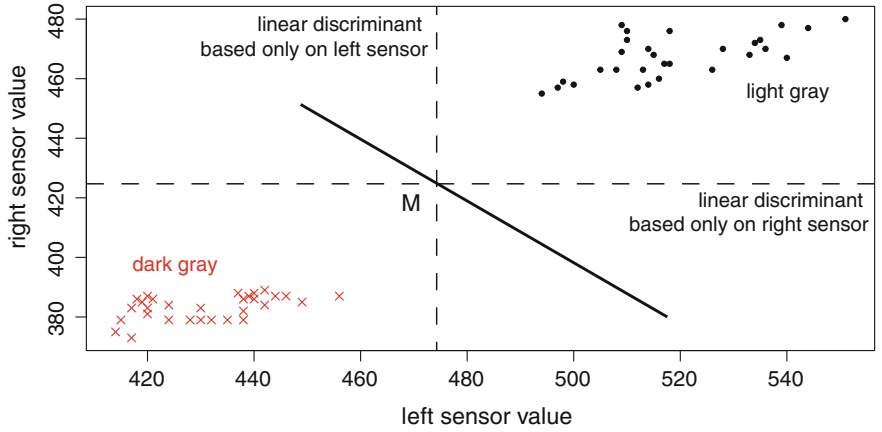



**Fig. 14.6** $x$–$y$ plot of *gray* levels with single-sensor discriminants and an optimal discriminant

### *14.2.2 The Linear Discriminant*

Figure 14.5 is an $x$–$y$ plot of data sampled from two sensors. Each value is plotted at the point whose $x$-coordinate is the value returned by the red sensor and whose $y$-coordinate is the value returned by the blue sensor. Similarly, Fig. 14.6 is an $x$–$y$ plot of the data from Figs. 14.3 and 14.4 that were collected as the robot moved over two gray areas. In those graphs, the sensor values were plotted as function of time, but time has no role in classification except to the link samples that were measured at the same time by the two sensors.

In Fig. 14.6, classification based only on the left sensor corresponds to the vertical dashed line, while classification based only on the right sensor corresponds to the horizontal dashed line. Both the horizontal and vertical separation lines are not optimal. Suppose that classification based on the left sensor (the vertical line) is used and consider a sample for which the left sensor returns 470 and the right sensor returns 460. The sample will be classified as dark gray even though the classification as light gray is better. Intuitively, it is clear that the solid diagonal line in the graph is a far more accurate discriminant than either of the two discriminants based on a single sensor.

How can such a linear discriminant be defined mathematically so that it can be discovered automatically?[5] The general equation for a line in the plane is $y = mx+a$, where $m$ is the slope and $a$ is the intersect of the line and the $y$-axis when $x = 0$. Another equation for a line is:

$$w_1 x_1 + w_2 x_2 = c\,, \tag{14.2}$$

[5]The following presentation is abstract and will be easier to understand if read together with the numerical example in Sect. 14.2.5.

where $x_1$ is the horizontal axis for the left sensor values, $x_2$ is the vertical axis for the right sensor values. $c$ is a constant and $w_1$, $w_2$ are coefficients of the variables.

It is convenient to represent the coefficients as a column vector:

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}.$$

In this representation the vector $\mathbf{w}$ is normal to the discriminant line and therefore defines its slope, while the constant $c$ enables the discriminant to be any line of that slope, that is, any one of the infinite number of parallel lines with a given slope. Once the slope of $\mathbf{w}$ is determined, $c$ is obtained by entering a given point ($x_1$,$x_2$) into Eq. 14.2.

Linear discriminant analysis automatically defines the vector $\mathbf{w}$ and constant $c$ that generates an optimal discriminant line between the data sets of the two classes. The first step is to choose a point on the discriminant line. Through that point there are an infinite number of lines and we have to choose the line whose slope gives the optimal discriminant. Finally, the value $c$ can be computed from the slope and the chosen point. The following subsections describe each of these steps in detail.

### 14.2.3 Choosing a Point for the Linear Discriminant

How can we choose a point? LDA is based upon the *assumption* that the values of both classes have the same distribution. Informally, when looking at an $x$–$y$ plot, both sets of points should have similar size and shape. Although the distributions will almost certainly not be exactly the same (say a Gaussian distribution) because they result from measurements in the real world, since both sensors are subject to the same types of variability (sensor noise, uneven floor surface) it is likely that they will be similar.

If the two distributions are similar, the means of the samples of each sensor will be more-or-less in the same place with respect to the distributions. The average of the means for each sensor will be equidistant from corresponding points in the data sets. The discriminant is chosen to be some line that passes through the point $M$ (Fig. 14.6) whose coordinates are:

$$\left( \frac{\mu_{light}^{left} + \mu_{dark}^{left}}{2}, \frac{\mu_{light}^{right} + \mu_{dark}^{right}}{2} \right).$$

### 14.2.4 Choosing a Slope for the Linear Discriminant

Once we have chosen the point $M$ on the discriminant line, the next step is to choose the slope of the line. From Fig. 14.6, we see that there are infinitely many lines

through the point $M$ that would distinguish between the two sets of samples. Which is the best line based on the statistical properties of our data?

In Sect. 14.1 we looked for a way to decide between two discriminants, where each discriminant was a line parallel to the $y$-axis at the midpoint between the means of the values returned by a sensor (Fig. 14.4). The decision was based on the quality criterion $J_k$ (Eq. 14.1, repeated here for convenience):

$$J_k = \frac{\left(\mu^k_{dark} - \mu^k_{light}\right)^2}{\left(s^k_{dark}\right)^2 + \left(s^k_{light}\right)^2}, \tag{14.3}$$

where $k = left, right$. The discriminant with the larger value of $J_k$ was chosen. To maximize $J_k$, the numerator—the distance between the means—should be large, and the denominator—the variances of the samples—should be small.

Now, we no longer want to compute a quality criterion based on the values returned by each sensor separately, but instead to compute a criterion from all the values returned by both sensors. Figure 14.7 shows an $x$–$y$ plot of the values of two sensors, where the values of the two classes are represented by red squares and blue circles. Clearly, there is significant overlap along the $x$ or $y$ axes separately and it is impossible to find lines parallel to the axes that can distinguish between the groups. However, if we project the groups of measurements onto the line defined by a vector:

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix},$$

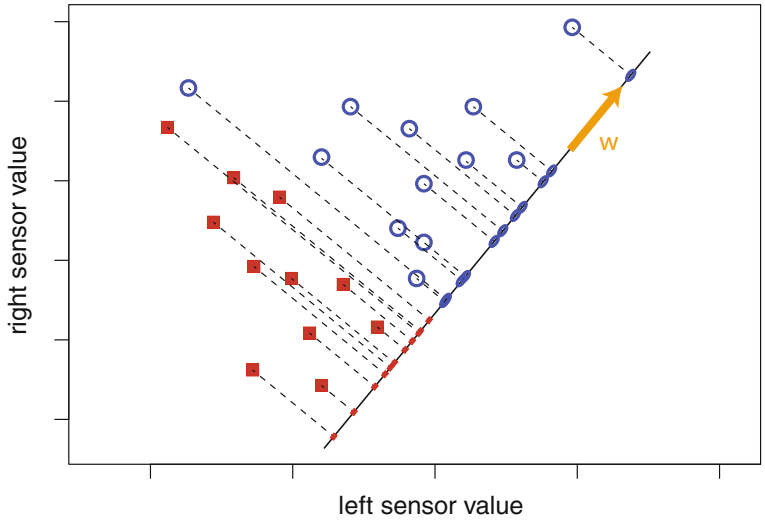



**Fig. 14.7** Projection of the samples of two classes onto a line defined by **w**

the two groups can be distinguished by the separator line defined by Eq. 14.2:

$$w_1 x_1 + w_2 x_2 = c\,,$$

where $c$ is defined by a point on the projection line situated between the red and blue points. In analogy with Eq. 14.3, we need to define a quality criterion $J(\mathbf{w})$, such that larger values of $J(\mathbf{w})$ give better discriminant lines. Then, we need to find the value of $\mathbf{w}$ that maximizes $J(\mathbf{w})$ by differentiating this function, setting the derivative to zero and solving for $\mathbf{w}$.

The definition of $J(\mathbf{w})$ is based on the means and variances of the two classes but is too complex for this book. We give without proof that the value of $\mathbf{w}$ that maximizes $J(\mathbf{w})$ is:

$$\mathbf{w} = \mathbf{S}^{-1}\left(\boldsymbol{\mu}_{light} - \boldsymbol{\mu}_{dark}\right), \tag{14.4}$$

where:

$$\boldsymbol{\mu}_{light} = \begin{bmatrix} \mu_{light}^{left} \\ \mu_{light}^{right} \end{bmatrix}, \quad \boldsymbol{\mu}_{dark} = \begin{bmatrix} \mu_{dark}^{left} \\ \mu_{dark}^{right} \end{bmatrix}$$

are the mean vectors of the two classes and $\mathbf{S}^{-1}$ is the inverse of the average of the covariance matrices of the two classes[6]:

$$\mathbf{S} = \frac{1}{2}\left(\begin{bmatrix} s^2\left(\mathbf{x}_{light}^{left}\right) & cov\left(\mathbf{x}_{light}^{left}, \mathbf{x}_{light}^{right}\right) \\ cov\left(\mathbf{x}_{light}^{right}, \mathbf{x}_{light}^{left}\right) & s^2\left(\mathbf{x}_{light}^{right}\right) \end{bmatrix} + \begin{bmatrix} s^2\left(\mathbf{x}_{dark}^{left}\right) & cov\left(\mathbf{x}_{dark}^{left}, \mathbf{x}_{dark}^{right}\right) \\ cov\left(\mathbf{x}_{dark}^{right}, \mathbf{x}_{dark}^{left}\right) & s^2\left(\mathbf{x}_{dark}^{right}\right) \end{bmatrix}\right).$$

Compared with the single-sensor $J_k$, the means are two-dimensional vectors because we have one mean for each of the sensors. The sum of variances becomes the covariance matrix, which takes into account both the individual variances of the two sensors and the covariances that express how the two sensors are related.

When the values of $M$ and $\mathbf{w}$ have been computed, all that remains is to compute the constant $c$ to fully define the discriminant line. This completes the learning phase of the LDA algorithm. In the recognition phase, the robot uses the line defined by $\mathbf{w}$ and $c$ for classifying new samples.

The computation is formalized in Algorithms 14.3 and 14.4, where we want to distinguish between two classes $C_1$, $C_2$ using two sensors. Compare this algorithm with Algorithm 14.1: the two sets of samples $\mathbf{X_1}$, $\mathbf{X_2}$ and the means $\boldsymbol{\mu_1}$, $\boldsymbol{\mu_2}$ and variances $\mathbf{s_1}$, $\mathbf{s_2}$ are vectors with two elements, one element for each of the sensors.

[6] See Appendix B.4 for the definition of covariance and Sect. 14.2.5 for a numerical example that will clarify the computations.

| **Algorithm 14.3: Linear discriminant analysis (learning phase)** | |
|---|---|
| float array[$n_1$,2] $\mathbf{X_1}$ | // Sets of samples of first area |
| float array[$n_2$,2] $\mathbf{X_2}$ | // Sets of samples of second area |
| float array[2] $\boldsymbol{\mu_1}, \boldsymbol{\mu_2}$ | // Means of $C_1, C_2$ |
| float array[2] $\boldsymbol{\mu}$ | // Mean of the means |
| float array[2] $\mathbf{s_1}, \mathbf{s_2}$ | // Variances of $C_1, C_2$ |
| float $cov_1, cov_2$ | // Covariances of $C_1, C_2$ |
| float array[2] $\mathbf{S_{inv}}$ | // Inverse of average |
| float $c$ | // Constant of the line |
| 1: Collect a set of samples $\mathbf{X_1}$ from $C_1$ | |
| 2: Collect a set of samples $\mathbf{X_2}$ from $C_2$ | |
| 3: Compute means $\boldsymbol{\mu_1}$ of $\mathbf{X_1}$ and $\boldsymbol{\mu_2}$ of $\mathbf{X_2}$ | |
| 4: $\boldsymbol{\mu} \leftarrow (\boldsymbol{\mu_1} + \boldsymbol{\mu_2})/2$ | |
| 5: Compute variances $\mathbf{s_1}$ of $\mathbf{X_1}$ and $\mathbf{s_2}$ of $\mathbf{X_2}$ | |
| 6: Compute covariances $cov_1, cov_2$ of $\mathbf{X_1}$ and $\mathbf{X_2}$ | |
| 7: Compute $\mathbf{S_{inv}}$ of covariance matrix | |
| 8: Compute $\mathbf{w}$ from Eq. 14.4 | |
| 9: Compute point $M$ from $\boldsymbol{\mu}$ | |
| 10: Compute $c$ from $M$ and $\mathbf{w}$ | |
| 11: Output $\mathbf{w}$ and $c$ | |

| **Algorithm 14.4: Linear discriminant analysis (recognition phase)** |
|---|
| float $\mathbf{w} \leftarrow$ input from the learning phase |
| float $c \leftarrow$ input from the learning phase |
| float $\mathbf{x}$ |
| 1: loop |
| 2: $\mathbf{x} \leftarrow$ new sample |
| 3: if $\mathbf{x} \cdot \mathbf{w} < c$ |
| 4: assign $\mathbf{x}$ to class $C_1$ |
| 5: else |
| 6: assign $\mathbf{x}$ to class $C_2$ |

### *14.2.5 Computation of a Linear Discriminant: Numerical Example*

Figure 14.8 is a plot of the samples from two ground sensors measured as a robot moves over two very similar gray areas, one that is 83.6% black and the other 85% black. The levels are so close that the human eye cannot distinguish between them. Can the linear discriminant computed by Algorithm 14.3 do better?

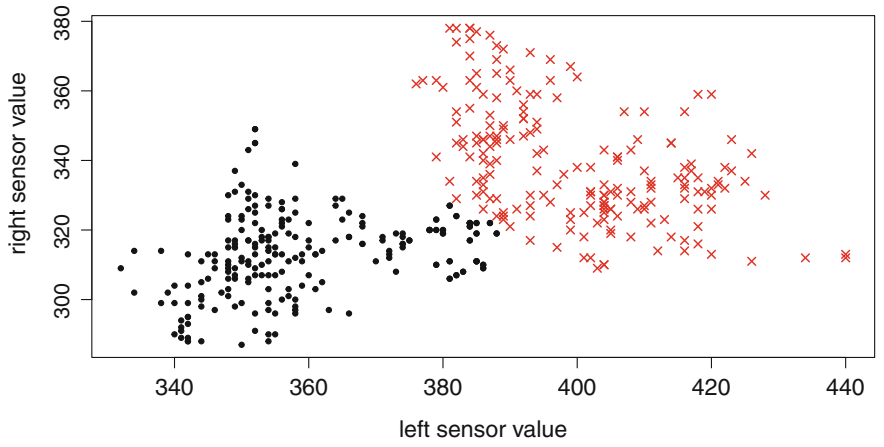



**Fig. 14.8** Similar *gray* areas (*black dots* from the darker area, *red x'es* from the lighter area)

Class $C_1$ is the light gray area and class $C_2$ is the dark gray area. The elements of the sets of vectors $\mathbf{X_1}[n_1]$, $\mathbf{X_2}[n_2]$ are vectors:

$$\mathbf{x} = \begin{bmatrix} x^{left} \\ x^{right} \end{bmatrix}$$

of samples measured by the left and right sensors, where $n_1 = 192$ and $n_2 = 205$.

First we compute the means of the data of Fig. 14.8:

$$\boldsymbol{\mu_1} = \frac{1}{192}\left(\begin{bmatrix} 389 \\ 324 \end{bmatrix} + \begin{bmatrix} 390 \\ 323 \end{bmatrix} + \ldots + \begin{bmatrix} 389 \\ 373 \end{bmatrix}\right) \approx \begin{bmatrix} 400 \\ 339 \end{bmatrix}$$

$$\boldsymbol{\mu_2} = \frac{1}{205}\left(\begin{bmatrix} 358 \\ 297 \end{bmatrix} + \begin{bmatrix} 358 \\ 296 \end{bmatrix} + \ldots + \begin{bmatrix} 352 \\ 327 \end{bmatrix}\right) \approx \begin{bmatrix} 357 \\ 312 \end{bmatrix}.$$

More samples were taken from the second area (205) than from the first area (192), but that is not important for computing the means. As expected, the means $\boldsymbol{\mu_1}$ of the samples from the light gray area are slightly higher (more reflected light) than the means $\boldsymbol{\mu_2}$ of the samples from the dark gray areas. However, the left sensor consistently measures higher values than the right sensor. Figure 14.9 shows the data from Fig. 14.8 with thin dashed lines indicating the means. There are two lines for each area: the horizontal line for the right sensor and the vertical line for the left sensor.

The covariance matrix is composed from the variances of the two individual sensors and their covariance. For $i = 1, 2$:

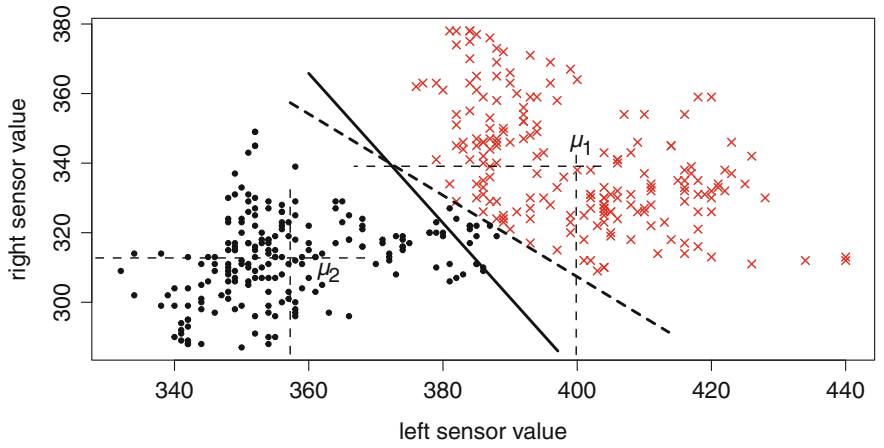



**Fig. 14.9** Means for each class (*thin dashed lines*), LDA discriminant line (*solid line*), discriminant line that fully distinguishes the classes (*thick dashed line*)

$$\mathbf{S}_i = \begin{bmatrix} s^2\left(X_i^{left}\right) & cov\left(X_i^{left}, X_i^{right}\right) \\ cov\left(X_i^{right}, X_i^{left}\right) & s^2\left(X_i^{right}\right) \end{bmatrix}.$$

For the data from Fig. 14.8, the variances of the samples of the light gray area are:

$$s^2(X_1^{left}) = \frac{1}{191}\left((389-400)^2 + (390-400)^2 + \cdots + (389-400)^2\right) \approx 187$$

$$s^2(X_1^{right}) = \frac{1}{191}\left((324-339)^2 + (323-339)^2 + \ldots + (373-339)^2\right) \approx 286\,.$$

Equation B.4 from Appendix B.4 is used to compute the covariance. Since covariance is symmetric, the value need be computed only once.

$$cov(X_1^{left}, X_1^{right}) = \frac{1}{191}\left((389-400)(324-339) + \ldots + (389-400)(373-339)\right)$$
$$\approx -118\,.$$

Putting the results together and performing a similar computation for $X_2$ gives the covariance matrices:

$$\mathbf{S}_1 = \begin{bmatrix} 187 & -118 \\ -118 & 286 \end{bmatrix} \qquad \mathbf{S}_2 = \begin{bmatrix} 161 & 44 \\ 44 & 147 \end{bmatrix}.$$

The next step is to compute the mean of the covariance matrices:

$$\mu_{\mathbf{S}} = \frac{1}{2}\left(\begin{bmatrix} 187 & -118 \\ -118 & 286 \end{bmatrix} + \begin{bmatrix} 161 & 44 \\ 44 & 147 \end{bmatrix}\right) = \begin{bmatrix} 174 & -38 \\ -37 & 216 \end{bmatrix},$$

and to find its inverse[7]:

$$\mathbf{S}^{-1} = \begin{bmatrix} 0.006 & 0.001 \\ 0.001 & 0.005 \end{bmatrix}.$$

We can now use Eq. 14.4 to compute $\mathbf{w}$:

$$\mathbf{w} = \begin{bmatrix} 0.006 & 0.001 \\ 0.001 & 0.005 \end{bmatrix} \cdot \left(\begin{bmatrix} 400 \\ 339 \end{bmatrix} - \begin{bmatrix} 357 \\ 313 \end{bmatrix}\right) = \begin{bmatrix} 0.28 \\ 0.17 \end{bmatrix}.$$

The vector $\mathbf{w}$ gives the direction of the projection line which is perpendicular to discriminant line. We now use Eq. 14.2, repeated here:

$$w_1 x_1 + w_2 x_2 = c$$

to compute the constant $c$, assuming we know the coordinate $(x_1, x_2)$ of some point. But we specified that the midpoint between the means must be on the discriminant line. Its coordinates are:

$$\boldsymbol{\mu} = \frac{1}{2}(\boldsymbol{\mu_1} + \boldsymbol{\mu_2}) = \frac{1}{2}\left(\begin{bmatrix} 400 \\ 339 \end{bmatrix} + \begin{bmatrix} 357 \\ 313 \end{bmatrix}\right) \approx \begin{bmatrix} 379 \\ 326 \end{bmatrix}.$$

Therefore:

$$c = 0.28 \cdot 379 + 0.17 \cdot 326 \approx 162\,,$$

and the equation of the discriminant line is:

$$0.28x_1 + 0.17x_2 = 162\,.$$

The discriminant line is shown as a solid line in Fig. 14.9. Given a new sample $(a, b)$, compare the value of $0.28a + 0.17b$ to 162: If it is larger, the sample is classified as belonging to class $C_1$ and if it is is smaller, the sample is classified as belong to class $C_2$.

[7] See Appendix B.5.

### 14.2.6 Comparing the Quality of the Discriminants

If we compare the linear discriminant found above with the two simple discriminants based upon the means of a single sensor, we see a clear improvement. Because of the overlap between the classes in a single direction, the simple discriminant for the right sensor correctly classifies only 84.1% of the samples, while the simple discriminant for the left sensor is somewhat better, classifying 93.7% of samples correctly. The linear discriminant found using LDA is better, correctly classifying 97.5% of the samples.

It might be surprising that there are discriminant lines that can correctly classify *all* of the samples! One such discriminant is shown by the thick dashed line in Fig. 14.9. Why didn't LDA find this discriminant? LDA assumes both classes have a similar distribution (spread of values) around the mean and the LDA discriminant is optimal under this assumption. For our data, some points in the second class are far from the mean and thus the distributions of the two classes are slightly different. It is hard to say if these samples are outliers, perhaps caused by problem when printing the gray areas on paper. In that case, it is certainly possible that subsequent sampling of the two areas would result in distributions that are similar to each other, leading to the correct classification by the LDA discriminant.

### 14.2.7 Activities for LDA

Activities for LDA are collected in this section.

**Activity 14.2: Robotic chameleon with LDA**

- Construct an environment as shown in Fig. 14.2 but with two gray levels very similar to each other.
- Write a program that causes the robot to move at a constant speed over the area of one color and sample the reflected light periodically. Repeat for the other color.
- Plot the data.
- Compute the averages, the covariance matrices and the discriminant.
- Implement a program that classifies measurements of the sensor. When the robot classifies a measurement it displays which color is recognized (or gives other feedback if changing color cannot be done).

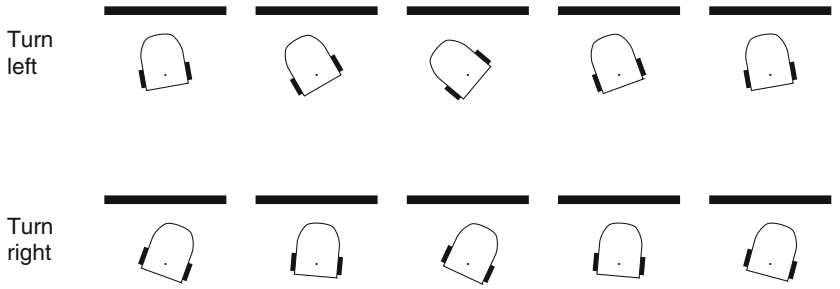



**Fig. 14.10** Training for obstacle avoidance

**Activity 14.3: Obstacle avoidance with two sensors**

- Figure 14.10 shows a robot approaching a wall. The upper part of the diagram shows various situations where the robot detects the wall with its right sensors; therefore, it should turn left to move around the wall. Similarly, in the lower part of the diagram the robot should turn right.
- Write a program that stores the sensor values from both the right and left sensors when a button is pressed. The program also stores the identity of which button was pressed; this represents the class we are looking for when doing obstacle avoidance.
- Train the robot: Place the robot next to a wall and run the program. Touch the left button if the robot should turn left or the right button if the robot should turn right. Repeat many times.
- Plot the samples from the two sensors on an $x$–$y$ graph and group them by class: turn right or left to avoid of the wall. You should obtain a graph similar to the one in Fig. 14.11.
- Draw a discriminant line separating the two classes.
- How successful is your discriminant line? What percentage of the samples can it successfully classify?
- Compute the optimal discriminant using LDA. How successful is it? Do the assumptions of LDA hold?

**Activity 14.4: Following an object**

- Write a program that causes the robot to follow an object. The robot moves forward if it detects the object in front; it moves backwards if it is too close to the object. The robot turns right if the object is to its right and the robot turns left if the object is to its left.

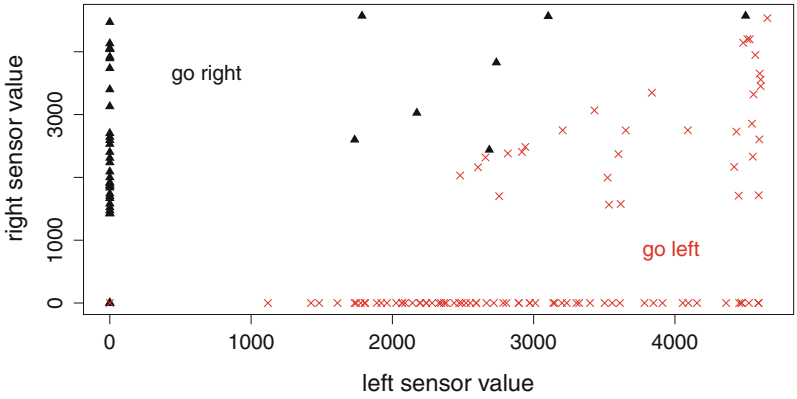



**Fig. 14.11** Obstacle avoidance data from the class "go left" (*red x'es*) and the class "go right" (*black triangles*)

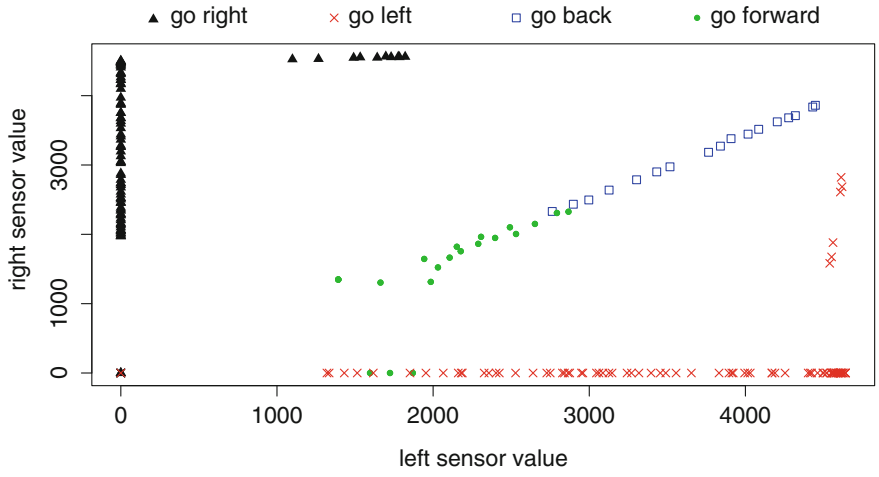



**Fig. 14.12** Data acquired in a learning phase for following an object

- Use two sensors so we can visualize the data on an $x$–$y$ plot.
- Acquire and plot the data as in Activity 14.3. The plot should be similar to the one shown in Fig. 14.12.
- Explain the classifications in Fig. 14.12. What is the problem with classifying a sample as going forwards or going backwards? Why do the samples for going forwards and backwards have different values for the left and right sensors?
- Suggest an algorithm for classifying the four situations. Could you use a combination of linear separators?

## 14.3 Generalization of the Linear Discriminant

In this section we point out some ways in which LDA can be extended and improved.

First, we can have more sensors. The mathematics becomes more complex because with $n$ sensors, the vectors will have $n$ elements and the covariance matrix will have $n \times n$ elements, requiring more computing power and more memory. Instead of a discriminant line, the discriminant will be an $n - 1$ dimension hyperplane. Classification with multiple sensors is used with electroencephalography (EEG) signals from the brain in order to control a robot by thought alone.

Activity 14.4 demonstrated another generalization: classification into more than two classes. Discriminants are used to classify between each pair of classes. Suppose you have three classes $C_1$, $C_2$, and $C_3$, and discriminants $\Delta_{12}$, $\Delta_{13}$, $\Delta_{23}$. If a new sample is classified in class $C_2$ by $\Delta_{12}$, in class $C_1$ by $\Delta_{12}$, and in class $C_2$ by $\Delta_{23}$, the final classification will be into class $C_2$ because more discriminants assign the sample to that class.

A third generalization is to use a higher order curve instead of a straight line, for example, a quadratic function. A higher order discriminant can separate classes whose data sets are not simple clusters of samples.

## 14.4 Perceptrons

LDA can distinguish between classes only under the assumption that the samples have similar distributions in the classes. In this section, we present another approach to classification using *perceptrons* which are related to neural networks (Chap. 13). There we showed how learning rules can generate specified behaviors linking sensors and motors; here we show how they can be used to classify data into classes.

### *14.4.1 Detecting a Slope*

Consider a robot exploring difficult terrain. It is important that the robot identify steep slopes so it won't fall over, but it is difficult to specify in advance all dangerous situations since these depend on characteristics such as the geometry of the ground and its properties (wet/dry, sand/mud). Instead, we wish to train the robot to adapt its behavior in different environments.

To simplify the problem, assume that the robot can move just forwards and backwards, and that it has accelerometers on two axes *relative to the body of the robot*: one measures acceleration forwards and backwards, and the other measures acceleration upwards and downwards. A robot that is stationary on a level surface will measure zero acceleration forwards and backwards, and an downwards acceleration of 9.8 m/sec$^2$ due to gravity. Gravitational acceleration is relatively strong compared

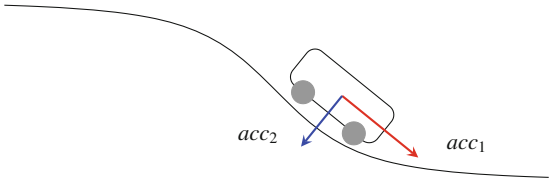



**Fig. 14.13** A robot with accelerometers moving in difficult terrain

with the acceleration of a slow-moving robot, so the relative values of the accelerometer along the two axes will give a good indication of its attitude.

Figure 14.13 shows a robot moving forwards on a slope. The values returned by both accelerometers are similar, $acc_1 \approx acc_2$, so we can infer that the robot is on a slope. If $acc_2 \gg acc_1$, we would infer that the robot is on level ground because the up/down accelerometer measures the full force of gravity while the front/back accelerometer measures only the very small acceleration of the moving robot. The task is to distinguish between a safe position of the robot and one in which the slope starts to become too steep so that the robot is in danger of falling.

Figure 14.14 shows data acquired during a training session as the robot moves down a slope. When the operator of the training session determines that the robot is stable (class $C_1$), she initiates a measurement indicated by a red $\times$; when she determines that the robot is in a dangerous situation (class $C_2$), she initiates a measurement indicated by a black triangle. The challenge is to find a way of distinguishing samples of the two classes.

The dashed lines in the figure show the means for the two data sets. It is clear that they do not help us classify the data because of the large overlap between the two sets of samples. Furthermore, LDA is not appropriate because there is no similarity in

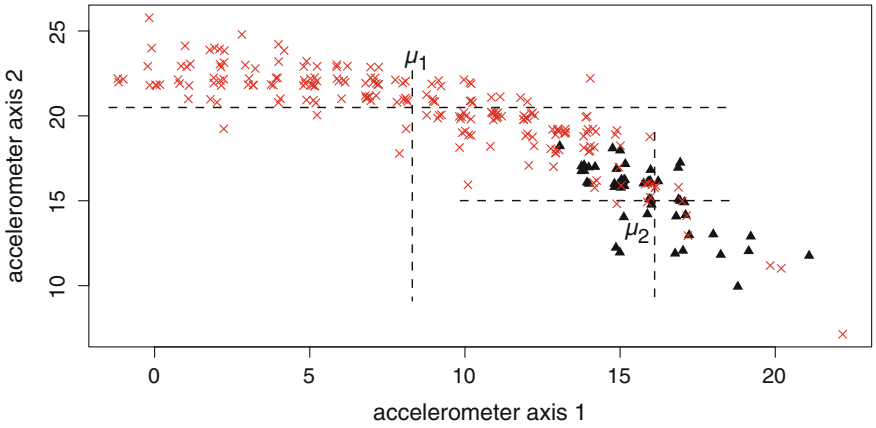



**Fig. 14.14** Detecting a dangerous slope using data from the accelerometers

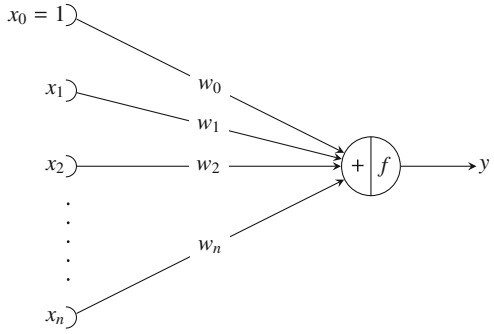



**Fig. 14.15** A perceptron

the distributions: samples when the robot is stable appear in many parts of the plot, while samples from dangerous situations are concentrated in a small area around their mean.

### *14.4.2 Classification with Perceptrons*

A *perceptron* is an artificial neuron with a specific structure (Fig. 14.15). It has a summation unit with inputs $\{x_1, \ldots, x_n\}$ and each input $x_i$ is multiplied by a factor $w_i$ before summation. An additional input $x_0$ has the constant value 1 for setting a bias independent of the inputs. The output of the perceptron is obtained by applying a function $f$ to the result of the addition.

When used as a classifier, the inputs to the perceptron are the values returned by the sensors for a sample to be classified and the output will be one of two values that indicate the class to which the sample is assigned. Usually, the output function $f$ is just the sign of the weighted sum:

$$y = sign\left(\sum_{i=0}^{n} w_i\, x_i\right) = \pm 1\,, \tag{14.5}$$

where one class corresponds to $+1$ and the other to $-1$.

The data are normalized so that all inputs are in the same range, usually $-1 \leq x_i \leq +1$. The data in Fig. 14.14 can be normalized by dividing each value by 30.

Given a set of input values $\{x_0 = 1, x_1, \ldots, x_n\}$ of a sample, the object of a training session is to find a set of weights $\{w_0, w_1, \ldots, w_n\}$ so that the output will be the value $\pm 1$ that assigns the sample to the correct class.

If the sample is close to the border between two classes, the weighted sum will be close to zero. Therefore, a perceptron is also a linear classifier: to distinguish between outputs of $\pm 1$, the discriminant dividing the two classes is defined by the

set of weights giving an output of zero:

$$\sum_{i=0}^{n} w_i \, x_i = 0 \,,$$

or

$$w_0 + w_1 x_1 + \cdots + w_n x_n = 0 \,. \tag{14.6}$$

The presentation of LDA for a two-dimensional problem ($n = 2$) led to Eq. 14.2, which is the same as Eq. 14.6 when $c = -w_0$. The difference between the two approaches is in the way the weights are obtained: in LDA statistics are used while for perceptrons they result from an iterative learning process.

### *14.4.3 Learning by a Perceptron*

The iterative search for values of the weights $\{w_0, w_1, \ldots, w_n\}$ starts by setting them to a small value such as 0.1. During the learning phase, a set of samples is presented to the perceptron, together with the expected output (the class) for each element of the set. The set of samples must be constructed randomly and include elements from all the classes; furthermore, the elements from a single class must also be chosen randomly. This is to prevent the learning algorithm from generating a discriminant that is optimal in one specific situation, and to ensure that the process converges rapidly to an overall optimal discriminant, rather than spending too much time optimizing for specific cases.

The adjustment of the weights is computed as follows:

$$w_i(t+1) = w_i(t) + \eta \, x_i \, y \,, \quad 0 \leq i \leq n \,. \tag{14.7}$$

This is essentially the Hebbian rule for ANNs (Sect. 13.5.2). $w_i(t)$ and $w_i(t+1)$ are the $i$'th weights before and after the correction, $\eta$ defines the learning rate, $x_i$ is the normalized input, and $y$ is the desired output. Since the sign function is applied to the sum of the weighted inputs, $y$ is 1 or $-1$, except on the rare occasions where the sum is exactly zero.

Equation 14.7 corrects the weights by adding or subtracting a value that is proportional to the input, where the coefficient of proportionality is the learning rate. A small value for the learning rate means that the corrections to the weights will be in small increments, while a high learning rate will cause the corrections to the weights to be in larger increments. Once learning is completed, the weights are used to classify subsequent samples.

Algorithms 14.5 and 14.6 are a formal description of classification by perceptrons. The constant $N$ is the size of the set of samples for the learning phase, while $n$ is the number of sensor values returned for each sample.

**Algorithm 14.5: Classification by a perceptron (learning phase)**

```
float array[N,n] X                          // Set of samples
float array[n + 1] w ← [0.1, 0.1, ...]      // Weights
float array[n] x                            // Random sample
integer c                                   // Class of the random sample
integer y                                   // Output of the perceptron
1: loop until learning terminated
2:     x ← random element of X
3:     c ← class to which x belongs
4:     y ← output according to Eq. 14.5
5:     if y does not correspond to class c
6:         adjust w_i according to Eq. 14.7
7: Output w
```

**Algorithm 14.6: Classification by a perceptron (recognition phase)**

```
float w ← weights from the learning phase
float x
integer y
1: loop
2:     x ← new sample
3:     y ← output of perceptron for x, w
4:     if y = 1
5:         assign x to class C_1
6:     else if y = −1
7:         assign x to class C_2
```

When should the learning phase be terminated? One could specify an arbitrary value, for example: terminate the learning phase when 98% of the samples are classified correctly. However, it may not be possible to achieve this level. A better method is to terminate the learning phase when the magnitudes of the corrections to the weights become small.

### 14.4.4 Numerical Example

We return to the robot that is learning to avoid dangerous slopes and apply the learning algorithm to the data in Fig. 14.14. The perceptron has three inputs: $x_0$ which always set to 1, $x_1$ for the data from the front/back accelerometer, and $x_2$ for the data from the up/down accelerometer. The data is normalized by dividing each sample by 30 so that values will be between 0 and 1. We specify that an output of 1 corresponds to class $C_1$ (stable) and an output of $-1$ corresponds to class $C_2$ (dangerous).

Select a random sample from the input data, for example, a sample in class $C_1$ whose sensor values are $x_1 = 14$ and $x_2 = 18$. The normalized input is $x_1 = 14/30 = 0.47$ and $x_2 = 18/30 = 0.6$. The output of the perceptron with initial weights 0.1 is:

$$\begin{aligned} y &= sign(w_0 \times 1 + w_1 x_1 + w_2 x_2) \\ &= sign(0.1 \times 1 + 0.1 \times 0.47 + 0.1 \times 0.6) \\ &= sign(0.207) \\ &= 1\,. \end{aligned}$$

This output is correct so the weights need not be corrected. Now choose a random sample in class $C_2$ whose sensor values are $x_1 = 17$ and $x_2 = 15$. The normalized input is $x_1 = 17/30 = 0.57$ and $x_2 = 15/30 = 0.5$. The output of the perceptron is:

$$\begin{aligned} y &= sign(w_0 \times 1 + w_1 x_1 + w_2 x_2) \\ &= sign(0.1 \times 1 + 0.1 \times 0.57 + 0.1 \times 0.5) \\ &= sign(0.207) \\ &= 1\,. \end{aligned}$$

This output is not correct: the sample is from class $C_2$ which corresponds to $-1$. The weights are now adjusted using Eq. 14.7 with a learning rate $\eta = 0.1$:

$$\begin{aligned} w_0(t+1) &= w_0(t) + \eta\, x_0 y &&= 0.1 + 0.1 \times 1 \times -1 &&= 0 \\ w_1(t+1) &= w_1(t) + \eta\, x_1 y &&= 0.1 + 0.1 \times 0.57 \times -1 &&= 0.043 \\ w_2(t+1) &= w_2(t) + \eta\, x_2 y &&= 0.1 + 0.1 \times 0.5 \times -1 &&= 0.05\,. \end{aligned}$$

These will be the new weights for the next iteration. If we continue for 2000 iterations, the weights evolve as shown in Fig. 14.16. At the end of the learning process, the weights are:

$$w_0 = -0.1, \quad w_1 = -0.39, \quad w_2 = 0.53\,.$$

These weights can now be used by the recognition phase of the classification Algorithm 14.6. The discriminant line built by the perceptron (Eq. 14.6) is:

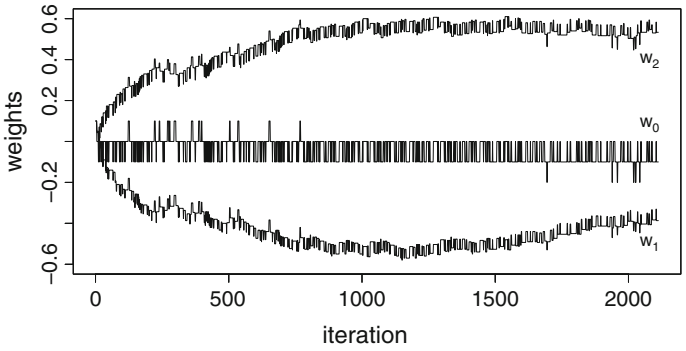



**Fig. 14.16** Evolution of the weights for learning by a perceptron

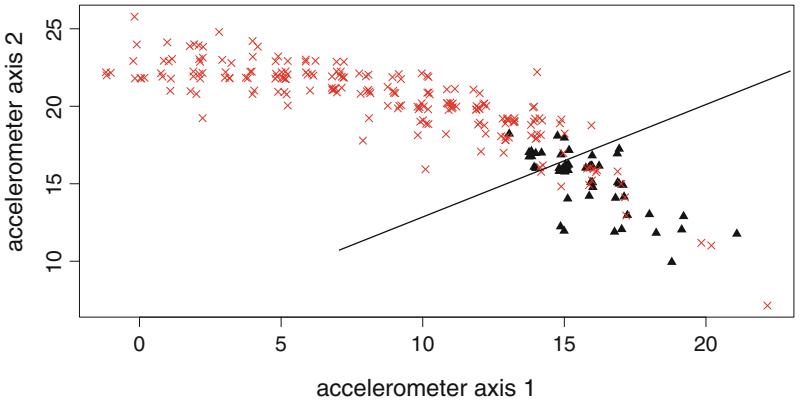



**Fig. 14.17** Discriminant line computed from the perceptron weights

$$-0.1 - 0.39x_1 + 0.53x_2 = 0\,.$$

The coordinates of this line are the normalized values, but they can be transformed back into the raw values obtained from the accelerometers. The line is shown in Fig. 14.17 and considering the large overlap of the classes, it does a reasonably good job of distinguishing between them.

### *14.4.5 Tuning the Parameters of the Perceptron*

The performance of a perceptron is determined by the number of iterations and the learning rate. Figure 14.16 shows that there is a strong variation in the weights at the beginning, but the weights stabilize as the number of iterations increases. Thus it is relatively simple to monitor the weights and terminate the computation when the weights stabilize.

This evolution of the weights depends strongly on the learning rate. Increasing the learning rate speeds the variation at the beginning, but strong corrections are not beneficial when the weights begin to stabilize. From Fig. 14.16, it is clear that even at the end of the run, there are significant variations in the weights which oscillate around the optimal value. This suggests that we reduce the learning rate to reduce the oscillations, but doing so will slow down the convergence to the optimal weights at the beginning of the learning phase.

The solution is to use a variable learning rate that is not constant. It should start out large to encourage rapid convergence to the optimal values, and then become smaller to reduce oscillations. For example, we can start with the learning rate of 0.1 and then decrease it continually using the equation:

$$\eta\,(t+1) = \eta\,(t) \times 0.997\,.$$

Figure 14.18 shows the evolution of the weights when this variable learning rate is used. The exponential decrease of $\eta$ is also plotted in the figure. A comparison of Figs. 14.16 and 14.18 clearly shows the superiority of the variable learning rate and this improvement is obtained with very little additional computation.

**Activity 14.5: Learning by a perceptron**

- Take a set of measurements of the accelerometers on your robot on various slopes and plot the data. For each sample you will have to decide if the robot is in danger of falling off the slope.
- Classify the data using a perceptron. What discriminant line do you find?
- Use a perceptron to classify the gray areas using the data of Activity 14.2. What discriminant do you find? Compare the discriminant found by the perceptron to the discriminant found by LDA.

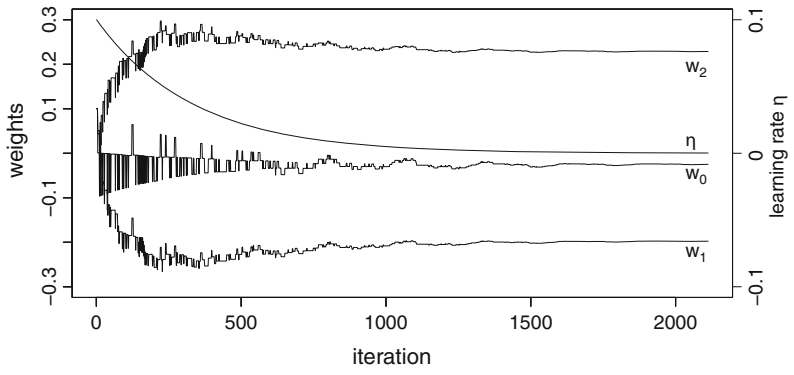



**Fig. 14.18** Evolution of the weights for learning by a perceptron with a variable learning rate

## 14.5 Summary

Samples of two classes can be distinguished using their means alone or using both the means and the variances. Linear discriminant analysis is a method for classification that is based on computing the covariances between the samples of the classes. LDA performs well only when the distributions of the samples of the classes are similar. When this assumption does not hold, perceptrons can be used. For optimum performance, the learning rate of a perceptron must be adjusted, if possible dynamically during the learning phase.

## 14.6 Further Reading

A detailed mathematical presentation of linear discriminant analysis can be found in Izenman [2, Chap. 8]. Textbooks on machine learning techniques are [1, 3].

## References

1. Harrington, P.: Machine Learning in Action, vol. 5. Manning, Greenwich (2012)
2. Izenman, A.J.: Modern Multivariate Statistical Techniques. Springer, Berlin (2008)
3. Kubat, M.: An Introduction to Machine Learning. Springer, Berlin (2015)

# Chapter 15
# Swarm Robotics

Factories use multiple robots to achieve goals such as painting and welding a car (Fig. 1.3). The use of multiple robots shortens manufacturing time by performing different tasks simultaneously such as welding parts on both sides of a car. These tasks are typically designed to be independent with no close collaboration between the robots. Later, robots, especially mobile robots, were designed to collaborate with each other *directly* to perform multiple actions simultaneously in different places.

Here are some examples of tasks that require multiple robots to collaborate:

- Manipulating large structural elements in buildings, as well as in environments that are hard for humans to access such as in space or underwater.
- Performing a task through the collaboration of different types of robots: in a large-scale disaster, a flying drone can locate areas where victims are likely to be found, while a tracked robot on the ground searches these areas, focusing on places that may be hidden from aerial observation by trees or rubble.
- Performing simultaneous measurements in different locations: measuring sound disturbances in different parts of a building, or monitoring pollution after an industrial accident.

What is common to these situations is that multiple robots participate in performing a task and they need to coordinate with each other because they are acting on the same physical object even though they are not next to each other in the environment.

Collaboration among robots can also be used to speed up the execution of a task by having several robots perform the task in parallel. Consider measuring pollution over a large area: a single robot can roam the entire area (like a robotic vacuum cleaner in an apartment), but the task will be accomplished much faster if multiple robots divide up the areas to be covered among themselves.



M. Ben-Ari and F. Mondada, *Elements of Robotics*,
https://doi.org/10.1007/978-3-319-62533-1_15

## 15.1 Approaches to Implementing Robot Collaboration

There are two main approaches to the design of systems composed of multiple robots. The first is a *centralized system*, where a central component (one of the robots or an external computer) coordinates all the robots and their tasks. The advantage of a centralized system is that it is relatively simple to implement. The main disadvantage is that it is difficult to expand because adding more robots adds processing load to the central station where all the intelligence is concentrated. In a centralized system, the robots themselves can be "dumb," but most robots have significant computing power that is not well utilized in this architecture. Another serious disadvantage of a centralized system is that the central component is a single point of failure. If it stops working the entire systems fails. In critical environments it is unacceptable to employ a system that is not robust under the failure of a single component.

If we look to the animal world, we see that many activities are *distributed*, that is, independent individuals work together to achieve common goals of the entire population. Ants optimize their path to food sources not by depending on one ant to dispatch others to search and then to process the information returned, but by a distributed effort of the entire ant colony. Individual ants mark the ground with pheromones that are sensed by the other ants. If a few ants are eaten by a predator, the rest of the colony survives, as does the knowledge embodied in the locations of the pheromones. The efficiency and robustness of this approach was demonstrated in the algorithms in Chap. 7.

*Swarm robotics* is a distributed approach to robotics that tries to coordinate behavior by copying mechanisms inspired by the behavior of social animals. These mechanisms, often local and simple, allow a group to achieve global performance that could not be achieved by an individual on its own. Distributed systems have the following advantages:

- They are robust. Losing one out of ten robots only reduces the performance of the system by about ten percent instead of causing failure of the entire system.
- They are flexible and scalable. The number of robots can be adapted to the task. If there are ten robots in the system but five are sufficient to perform the task, the other five can be assigned to other tasks, while if ten robots cannot perform the task efficiently, another ten can easily be added.

These advantages come at the cost of the effort required to design and implement coordination among the robots. In swarm robotics, as well as in nature, there are relatively simple coordination mechanisms that make distributed systems feasible.

This chapter presents two approaches to coordination in swarm robotics:

- Information-based coordination (Sect. 15.2), where the interaction is in the form of communications between robots. This can be either directly by explicitly passing electronic messages or indirectly by placing messages in the environment.
- Physical coordination (Sect. 15.3), where individual robots interact at the mechanical level, either directly by exerting forces on each other or indirectly by manipulating a common object.

## 15.2 Coordination by Local Exchange of Information

Communications can be either global or local. Suppose that you receive a call from your friends and they inform you: "We see an ice-cream store on the left." This *global* information is useless unless they give you their current location. However, if you are walking side-by-side with them and one says: "I see an ice-cream store on the left," from this *local* information you immediately know the approximate location of the store and can easily locate it visually.

Inspired by nature, swarm robotics uses local communications within a distributed architecture. A few zebras on the outer edges of a herd look for predators and signal the others by sound or movement. Herding is a successful survival strategy for animals because local communications enables large numbers of animals to flee immediately upon detection of a predator by a small numbers of alert watchers.

### *15.2.1 Direct Communications*

*Direct* local exchange of information is achieved when a friend talks to you. Animals don't talk but they do use sound as well as movement and physical contact to achieve direct local exchange of information. Robots implement direct local communications electronically (such as local WiFi or Bluetooth), or by transmitting and receiving light or sound. Alternatively, they can use a camera to detect changes in another robot such as turning on a light.

Local communications can be either *directional* or *non-directional*. Radio communications such as Bluetooth is local (just a few meters) and generally the receiving robot does not attempt to determine the direction to the transmitting robot. Local directional communications can be implemented using a light source as the transmitter and a narrow-aperture detector or a camera as the receiver.

### *15.2.2 Indirect Communications*

*Indirect* local communications refers to communications through a medium that can store a transmitted message for later access. The most familiar example is mail, either email or regular mail, where the transmitter composes the message and sends it, but the message remains at the server or the post office until it is delivered to the receiver who, in turn, may not access it immediately. Indirect communications in animals is called *stigmergy*; animals leave messages by depositing chemical substances that other animals can sense. We mentioned the use of pheromones by ants; another example is the use of urine by a dog to mark its territory.

Chemical messages are hard to implement in robots, but robots can leave optical markings on the ground as we did in the algorithm simulating a colony of ants

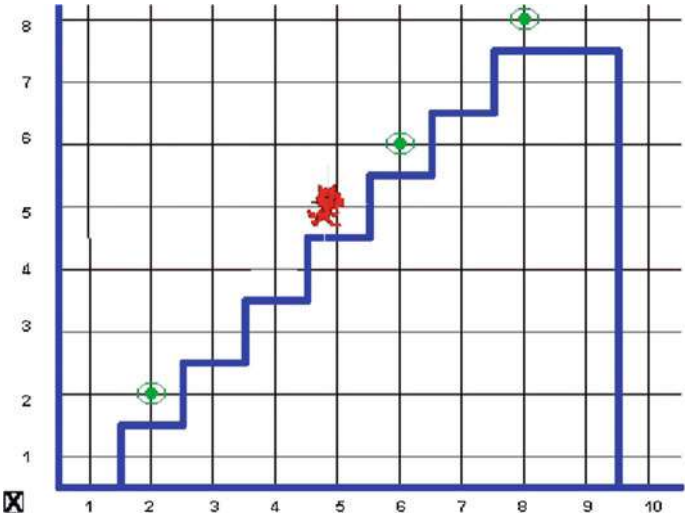



**Fig. 15.1** Karel the Robot implemented in Scratch; the *green dots* are the beepers

searching for a food source (Sect. 7.3). There only one robot was used, but the algorithms could be easily implemented with multiple robots because it is the marking that is important, not the identity of the robot that created the marking.

Indirect communications can also be implemented by placing or manipulating objects in the environment. Robotic vacuum cleaners use *beacons* that can be placed at the entrance of rooms that the robot should avoid. It is possible to conceive of beacons that record when a room has already been cleaned and this information is (indirectly) communicated to other vacuum cleaners.

*Karel the Robot* is an environment used to teach programming. Commands move a virtual robot around a grid on a computer screen and the robot can deposit and sense "beepers" place on squares of the grid (Fig. 15.1).[1]

Indirect communications when combined with manipulation can generate interesting patterns. Figure 15.2a shows an environment filled with small objects and five mobile robots equipped with grippers. The robots follow a simple set of rules:

- If the robot finds an isolated object it picks the object up;
- If the robot finds an isolated object but already holds one, it puts the new object down next to the one that was found;
- The robot avoids walls and groups of several objects.

It seems that these rules will cause the robots to eventually place all the objects in groups of two, but this does not happen as shown in Fig. 15.2b. The reason is that a group of objects *seen from the side* can look like an isolated object, so an additional

[1]The image is taken from the first author's implementation of Karel the Robot in Scratch (https://scratch.mit.edu/studios/520857).

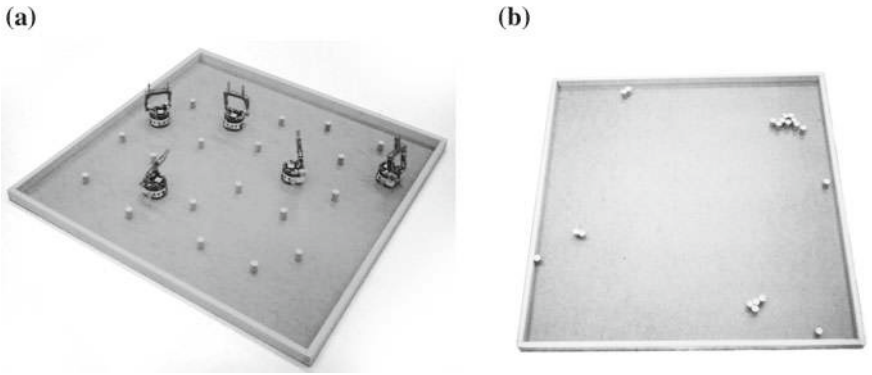



**Fig. 15.2** **a** Gripper robots in an environment filled with small objects. **b** The objects have been collected into groups

object is placed in the group. The result is that large groups of objects are assembled purely by indirect communication.

### *15.2.3 The BeeClust Algorithm*

The BeeClust algorithm is a swarm algorithm inspired by the behavior of bees. It uses a distributed architecture and local communications to generate a global result. The BeeClust algorithm is based on the way that very young bees cluster around locations of optimal temperature in the darkness of their nest. They measure local temperatures and detect collisions with other bees. The algorithm can be used by a swarm of robots to locate pollution; instead of measuring temperature, each robot measures some physical quantity that indicates levels of pollution. In time, the robots will come together in groups at locations of high levels of pollution.

Figure 15.3 shows a state machine implementing the algorithm. The robot moves randomly until it hits another robot, at which point it measures the temperature at the location of the collision. It waits at this location for a period of time proportional to the temperature that it found and then returns to moving randomly. When the robot moves it avoids obstacles such as walls. The algorithm uses what is perhaps the simplest form of communications: detecting a collision with another robot. The localized nature of the communications is essential for the correct functioning of the algorithm.

Initially, the robots will collide at random places, but those that are in locations with higher temperatures will remain there for longer periods of time which in turn causes additional collisions. In the long term, most of the robots will form a cluster in the area with the highest temperatures. They collide with each other frequently since being in a crowd increases the number of collisions. Of course this mechanism

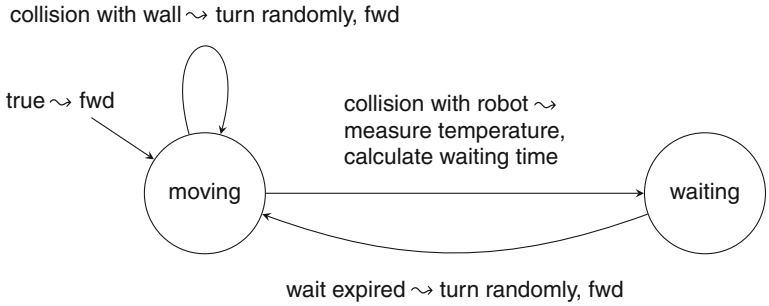



**Fig. 15.3** BeeClust algorithm

can only work with a large number of robots that generate many collisions leading to numerous measurements and clustering.

### *15.2.4 The ASSISIbf Implementation of BeeClust*

Within the ASSISIbf project, researchers at the University of Graz used Thymio robots to implement the BeeClust algorithm as part of research into the behavior of bees in a nest. The algorithm simulates a group of young bees in a cold circular arena, where two virtual sources of heat are placed on the right and on the left of the arena. Initially, only one of these sources of heat is active. The heat sources are simulated by two clusters of three robots at the edges of the arena (Fig. 15.4a).

The three heat-emitting robots on the left side transmit their temperature. Bee-robots in their vicinity detect this signal and stop for a period of time proportional to the temperature. During their stay in this area they also transmit a temperature signal. Furthermore, if the heat-emitting robots detect nearby robots they increase their heat and transmit the new temperature. Figure 15.4 shows the evolution of the behavior of the robots: (a) the initial state when the robots start moving and only the left temperature source is on; (b) the robots start clustering around the left source; (c) the highest level of clustering occurs. The bee-robots can leave the cluster, explore the environment and return to the cluster as shown in (d). This is caused by the random nature of the algorithm and is essential to prevent the system from being trapped in local minima. Figure 15.4e shows the moment when the right source is also switched on. After about 21 min the robots form two smaller clusters (f).

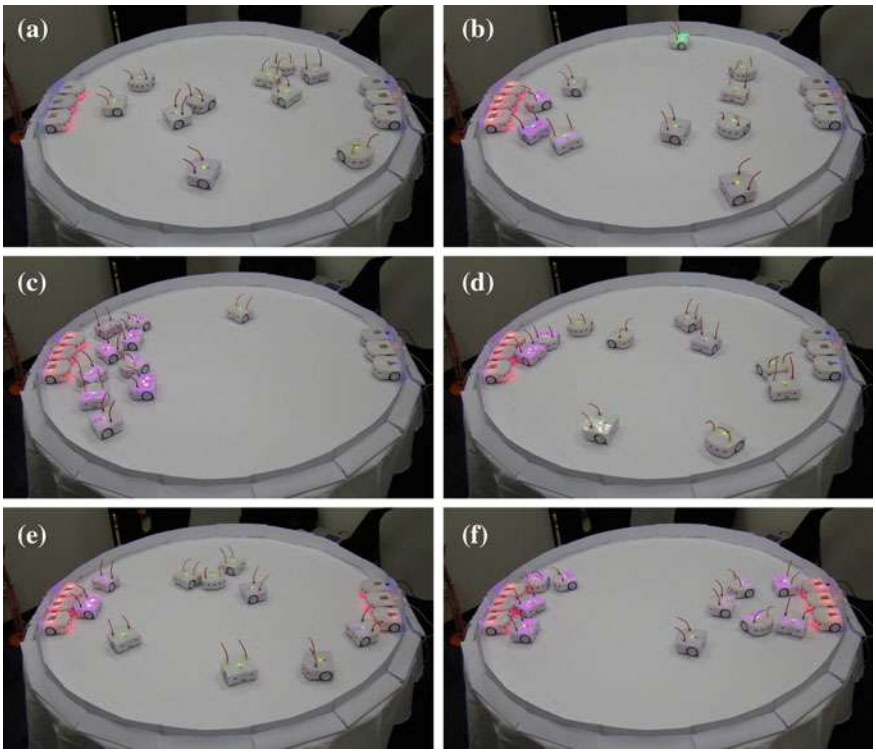



**Fig. 15.4** BeeClust implementation. The photographs were taken at the following times (min:sec) from the beginning of the experiment: **a** 1:40, **b** 2:30, **c** 9:40, **d** 12:40, **e** 13:20, **f** 21:10

**Activity 15.1: BeeClust algorithm**

- Implement the BeeClust algorithm to cause a group of robots to cluster at the brightest location in a room.
- Use three sensors: a light sensor to measure the ambient light, a sensor to detect other robots and a sensor to detect the limits of the arena.
- Solution 1: Use proximity sensors to detect other robots and ground sensors to detect a line defining the limits of the arena.
- Solution 2: Define the limits of the arena with a wall and use robot-to-robot communications to distinguish between robots and the wall. To avoid confusion do not use the same sensor to detect the other robots and the wall.

## 15.3 Swarm Robotics Based on Physical Interactions

In Sect. 7.2 we looked at a typical example of efficient swarm behavior: a colony of ants finding a path from their nest to a source of food. That example used indirect communications in the form of pheromones deposited on the ground. In this section we look at another form of swarm behavior that is mediated by physical interactions. We start with ants collaborating on the task of pulling a stick from the ground and a robotic version of this task. This is followed by a discussion of how forces exerted by several robots can be combined, demonstrated by a simple but clever algorithm called occlusion-based collective pushing.

### *15.3.1 Collaborating on a Physical Task*

Figure 15.5a shows two ants extracting a stick from the ground for use in building a nest. The stick is embedded so deeply that one ant cannot extract it by itself. We want to design a robotic system to accomplish this task (Fig. 15.5b). Each robot searches randomly until it finds a stick. It then pulls on the stick as hard as possible. If it successfully extracts the stick, it takes it back to a nest; otherwise, since it has only partially extracted the stick, it waits until it feels that another robot is pulling harder and releases the stick. If no robot comes to its help for a period of time, the robot releases its hold and returns to a random search. This ensures that if there are more sticks than robots, the system won't deadlock with each robot trying to extract one stick and waiting indefinitely.

The finite state machine for this algorithm is shown in Fig. 15.6. Although this behavior is simple and local, when applied by two robots it results in the extraction of the stick from the ground. The robot on the right in Fig. 15.5b pulls part of the stick as far as possible out of the ground using the maximum movement of its arm. When it detects that another robot has found the same stick and starts pulling, the first robot releases the stick to allow the second robot to extract it. By combining the

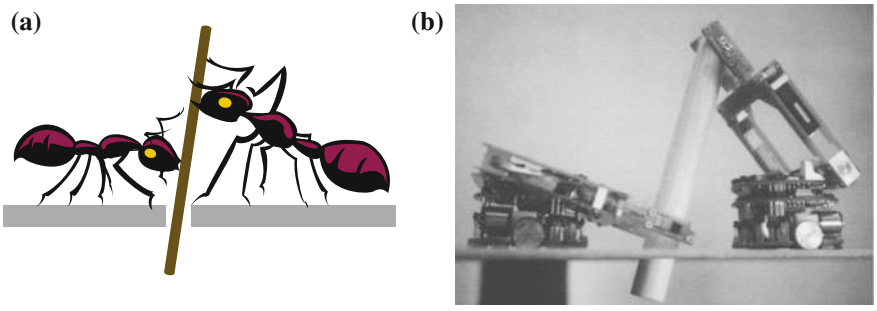



**Fig. 15.5** **a** Ants pulling a stick from the ground. **b** Robots pulling a stick from the ground

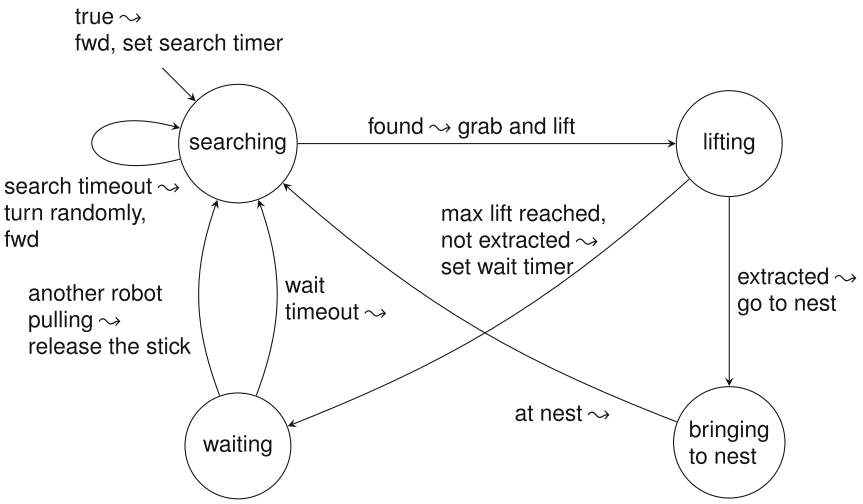



**Fig. 15.6** Algorithm for distributed stick pulling

physical capability of two robots with a simple specification of behavior, we achieve a result that neither of the robots could achieve alone.

### *15.3.2 Combining the Forces of Multiple Robots*

Figure 15.7 shows a differential-drive robot moving backwards (from left to right). It exerts a force $F_r$ that can be used to pull an object. Figure 15.8 shows two robots connected together so that they exert a force $F_{total}$ when moving from left to right.

What is the relationship between $F_r$ and $F_{total}$? There are three possibilities:

- $F_{total} < 2F_r$: The connected robots lose efficiency because the force they exert is less than that exerted by the two robots pulling separately.
- $F_{total} = 2F_r$: The connected robots achieve the same efficiency as two separate robots.
- $F_{total} > 2F_r$: The connected robots are more efficient that two separate ones.

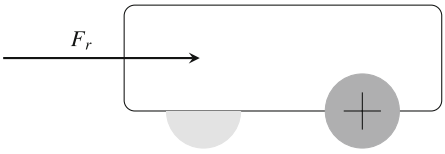



**Fig. 15.7** One robot pulling with a given force $F_r$

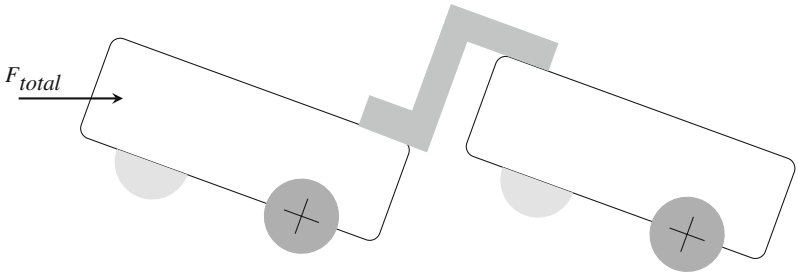



**Fig. 15.8** Two connected robots pulling with a given force $F_{total}$

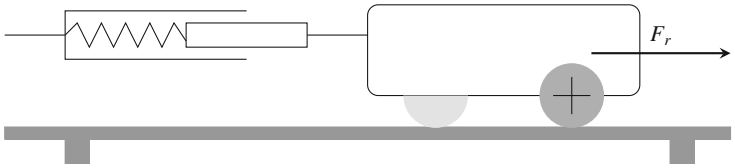



**Fig. 15.9** Measuring the force with a dynamometer

Robots can achieve $F_{total} > 2F_r$, where the total force is greater than the sum of the forces exerted by the individual robots, because in some mechanical configurations the connected robots are more stable since their center of gravity better placed.

**Activity 15.2: Pulling force by several robots.**

- Connect two robots and check whether they exert a force that is less than, the same or greater than twice that exerted by a single robot. You can connect the robots with a rigid connection as in Fig. 15.8 or with a flexible connection using string.
- Measure $F_r$, the force exerted by a single robot, and then measure $F_{total}$ the force exerted by the connected robots.
- A *dynamometer* (Fig. 15.9) is the best instrument for measuring forces. If you do not have one available, you can use a cable, a pulley and weights (or a scale) as shown in Fig. 15.10.
- Change the orientation of the robots: pulling forwards instead of backwards. Does this change the result?
- Experiment on different surfaces (hard ground, carpet, paper) and determine the effect of the surface on the resulting force.

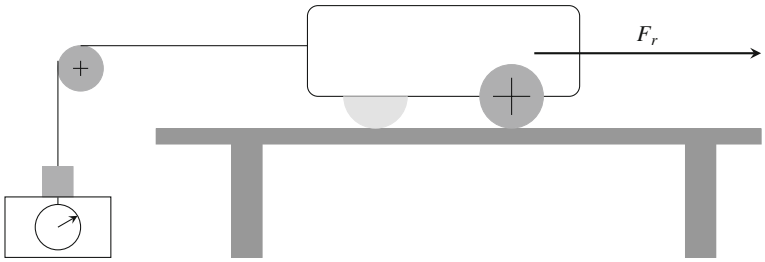



**Fig. 15.10** Measuring the force with a weight and a scale

### *15.3.3 Occlusion-Based Collective Pushing*

Let us consider another example of combining the force of multiple robots. Instead of a physical link as in Fig. 15.8, we use many simple robots to push an object, imitating a group of ants. Again, the advantages of a distributed system are flexibility—engaging just the resources needed for the task—and robustness—if one robot fails, the object might move a bit slower but complete failure of the task does not occur. These advantages come at the cost of additional resources and the complexity of implementing coordination among the robots.

Figure 15.11 shows a group of small robots pushing a large object represented by the circle towards a goal. One approach would be to determine the direction to the

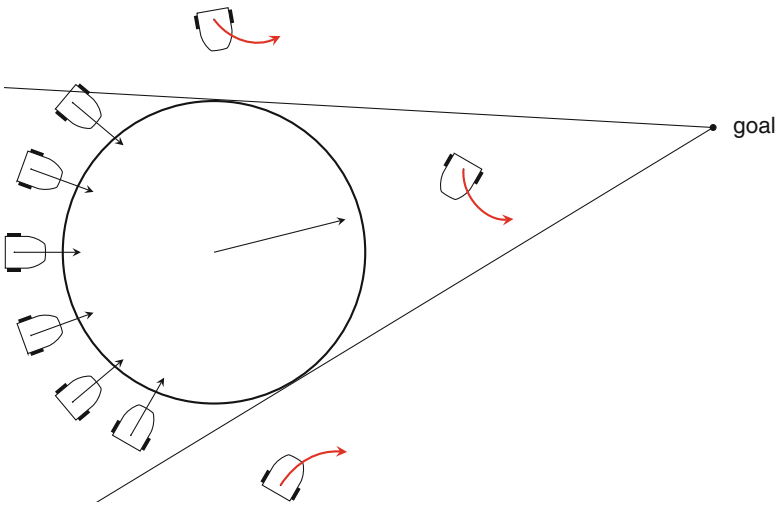



**Fig. 15.11** Occlusion based coordination: straight *black arrows* for the robots pushing the object and curved *red arrows* for the robots searching for an occluded position

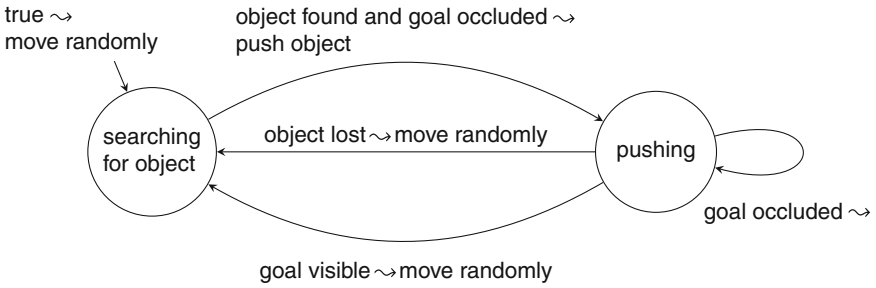



**Fig. 15.12** Algorithm for occlusion-based coordination

goal, to share this information among all the robots and to have each robot compute the direction it should push. This is not as simple as it may appear since a simple robot might not be able to determine its absolute position and heading.

A swarm robotics approach is called *occlusion-based pushing*, which assumes that the robots have only local knowledge, not global knowledge of what the other robots are doing. The robots are able to determine whether they can detect the goal or not. For example, a bright light could be mounted on the goal and the robots equipped with a light sensor.

Figure 15.12 shows the finite state machine for this algorithm. The robots search for the object; when they find it they place themselves perpendicular to the surface and push (straight black arrows). This could be implemented using a touch sensor or a proximity sensor. The robots continue to push as long as they *do not* detect the goal. If they do detect the goal, they stop pushing, move away (curved red arrows) and commence a new search for an occluded position where they resume pushing. The result is that the vector sum of the forces exerted by the robots is in a direction that moves the object toward the goal. The occlusion algorithm leads to the task being performed without a central control unit and even without inter-robot communications.

**Activity 15.3: Total force**

- Consider the configuration shown in Fig. 15.13. Robot 1 is pushing an object at a 45° angle with force $f_1$, robot 2 is pushing horizontally with force $f_2$ and robot 3 is pushing vertically with force $f_3$. Show that the $f_{total}$, the total force on the object, has magnitude:

$$\sqrt{\left(f_2 + \frac{f_1}{\sqrt{2}}\right)^2 + \left(f_3 - \frac{f_1}{\sqrt{2}}\right)^2}.$$

  in the direction:

$$\alpha = \tan^{-1} \frac{2f_3 - \sqrt{2}f_1}{2f_2 + \sqrt{2}f_1}.$$

- Compute the magnitude and direction of $f_{total}$ for various values of the individual forces. If $f_1 = f_2 = f_3 = 1$, the magnitude is $\sqrt{3}$ and the direction is 9.7°. If $f_1 = f_2 = 1$, what must $f_3$ be so that $\alpha = 45°$?
- Use three robots to implement this configuration and check that the object moves in the computed direction.

**Activity 15.4: Occlusion-based pushing**

- Place three robots around an object and a place a goal some distance away from the robot. Implement a mechanism for the robots to determine the direction to the goal, for example, attach a light to the goal or place the goal at the lowest point of an inclined surface.
- Implement a mechanism that enables the robots to distinguish between the object and the boundary of the surface on which they move. For example, use a black tape on the surface for its boundary and detect the object using proximity sensors.
- Implement a mechanism so that robots do not push each other. One method would be to use a color sensor and attach colored tape to the robots.
- Implement the occlusion-based pushing algorithm.
- Discuss whether occlusion-based pushing could be used in three dimensions, for example, underwater robots pushing an object.

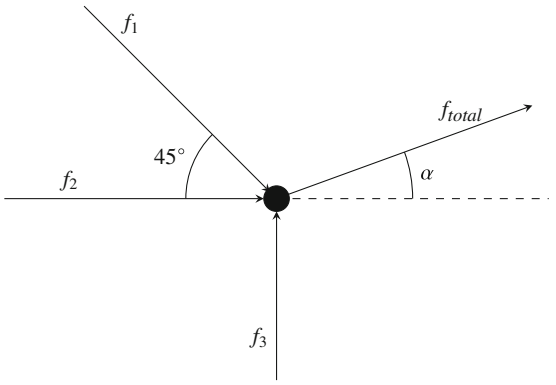



**Fig. 15.13** The total force from three robots

## 15.4 Summary

Swarm robotics uses multiple robots in a distributed architecture to perform a task. With a distributed architecture, the system is robust to the failure of individual robots and flexible in its ability to add or remove robots as the scale of the task changes. A distributed architecture can perform tasks that individual robots cannot, as we saw in the examples of robots combining forces. Finally, multiple robots can act simultaneously at different locations that are far from each other. These advantages come at a price: the increased cost of the multiple robots, the complexity of the coordination mechanisms that must be implemented, and in some cases the loss of performance due to the overlap between the action of different robots.

## 15.5 Further Reading

An overview of collective robotics is given in [5] and the collection [8] focuses on swarm robotics. For the specific projects presented in this chapters see:

- Karel the Robot [6].
- BeeClust [1, 7].
- ASSISIbf [9] and http://assisi-project.eu.
- Physical interaction (pulling a stick) [4].
- Occlusion-based transport [2, 3].

## References

1. Bodi, M., Thenius, R., Szopek, M., Schmickl, T., Crailsheim, K.: Interaction of robot swarms using the honeybee-inspired control algorithm beeclust. Math. Comput. Model. Dyn. Syst. **18**(1), 87–100 (2012)
2. Chen, J., Gauci, M., Groß, R.: A strategy for transporting tall objects with a swarm of miniature mobile robots. In: IEEE International Conference on Robotics and Automation, pp. 863–869 (2013)
3. Chen, J., Gauci, M., Li, W., Kolling, A., Groß, R.: Occlusion-based cooperative transport with a swarm of miniature mobile robots. IEEE Trans. Robot. **31**(2), 307–321 (2015)
4. Ijspeert, A.J., Martinoli, A., Billard, A., Gambardella, L.M.: Collaboration through the exploitation of local interactions in autonomous collective robotics: the stick pulling experiment. Auton. Robots **11**(2), 149–171 (2001)
5. Kernbach, S.: Handbook of Collective Robotics: Fundamentals and Challenges. CRC Press, Boca Raton (2013)
6. Pattis, R.E., Roberts, J., Stehlik, M.: Karel the Robot: A Gentle Introduction to the Art of Programming. Wiley, New York (1995)
7. Şahin, E., Spears, W.M. (eds.): Swarm robotics: from sources of inspiration to domains of application. Swarm Robotics: SAB 2004 International Workshop, pp. 10–20. Springer, Berlin (2005)

8. Schmickl, T., Thenius, R., Moeslinger, C., Radspieler, G., Kernbach, S., Szymanski, M., Crailsheim, K.: Get in touch: cooperative decision making based on robot-to-robot collisions. Auton. Agents Multi-Agent Syst. **18**(1), 133–155 (2009)
9. Schmickl, T., Bogdan, S., Correia, L., Kernbach, S., Mondada, F., Bodi, M., Gribovskiy, A., Hahshold, S., Miklic, D., Szopek, M., et al.: Assisi: mixing animals with robots in a hybrid society. In: Conference on Biomimetic and Biohybrid Systems, pp. 441–443. Springer, Berlin (2013)



# Chapter 16
# Kinematics of a Robotic Manipulator

Our presentation has focused on mobile robots. Most educational robots are mobile robots and you may have encountered commercial mobile robots such as robotic vacuum cleaners. You probably have not encountered *robotic manipulators*, but you have seen pictures of factories that assemble electronic circuits or weld frames of cars (Fig. 1.3). The most important difference between mobile and fixed robots is the environment in which they work. A mobile robot moves within an environment that has obstacles and uneven ground, so the environment is not fully known in advance. A robotic vacuum cleaner does not ask you to give it a map of your apartment with the location of each piece of furniture, nor do you have to reprogram it whenever you move a sofa. Instead, the robot autonomously senses the layout of the apartment: the rooms and the position of the furniture. While maps and odometry are helpful in moving a robot to an approximate position, sensors must be used to precisely locate the robot within its environment.

A robotic manipulator in a factory is fixed to a stable concrete floor and its construction is robust: repeatedly issuing the same commands will move the manipulator to precisely the same position. In this chapter we present algorithms for the *kinematics* of manipulators: how the commands to a manipulator and the robot's motion are related. The presentation will be in terms of an arm with two links in a plane whose joints can rotate.

There are two complementary tasks in kinematics:

- *Forward kinematics* (Sect. 16.1): Given a sequence of commands, what is the final position of the robotic arm?
- *Inverse kinematics* (Sect. 16.2): Given a desired position of the robotic arm, what sequence of commands will bring it to that position?

Forward kinematics is relatively easy to compute because the calculation of the change in position that results from moving each joint involves simple trigonometry. If there is more than one link, the final position is calculated by performing the calculations for one joint after another. Inverse kinematics is very difficult, because



M. Ben-Ari and F. Mondada, *Elements of Robotics*,
https://doi.org/10.1007/978-3-319-62533-1_16

you start with one desired position and have to look for a sequence of commands to reach that position. A problem in inverse kinematics may have one solution, multiple solutions or even no solution at all.

Kinematic computations are performed in terms of coordinate frames. A frame is attached to each joint of the manipulator and motion is described as transformations from one frame to another by rotations and translations. Transformation of coordinate frames in two-dimensions is presented in Sects. 16.3 and 16.4. Most robots manipulators are three-dimensional. The mathematical treatment of 3D motion is beyond the scope of this book, but we hope to entice you to study this subject by presenting a taste of 3D rotations in Sects. 16.5 and 16.6.

## 16.1 Forward Kinematics

We develop the kinematics of a two-dimensional robotic arm with two links, two joints and an *end effector* such as a gripper, a welder or a paint sprayer (Fig. 16.1). The first joint can rotate but it is mounted on a base that is fixed to a table or the floor. Link $l_1$ connects this joint to a second joint that can move and rotate; a second link $l_2$ connects this joint to the fixed end effector.

A two-dimensional coordinate system is assigned with the first joint at $(0, 0)$. The lengths of the two links are $l_1$ and $l_2$. Rotate the first joint by $\alpha$ to move the end of the first link with the second joint to $(x', y')$. Now rotate the second joint by $\beta$. What

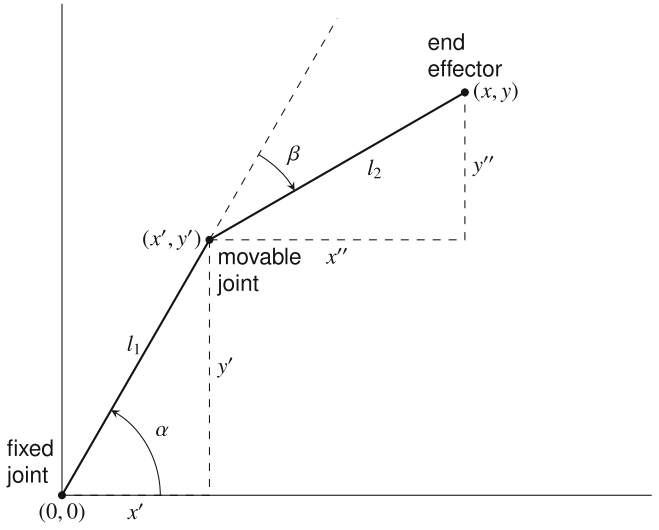



**Fig. 16.1** Forward kinematics of a two-link arm

are the coordinates $(x, y)$ of the end of the arm, in terms of the two constants $l_1, l_2$ and the two parameters $\alpha, \beta$?

Project $(x', y')$ on the $x$- and $y$-axes; by trigonometry its coordinates are:

$$x' = l_1 \cos\alpha$$
$$y' = l_1 \sin\alpha\,.$$

Now take $(x', y')$ as the origin of a new coordinate system and project $(x, y)$ on its axes to obtain $(x'', y'')$. The position of the end effector *relative to* the new coordinate system is:

$$x'' = l_2 \cos(\alpha + \beta)$$
$$y'' = l_2 \sin(\alpha + \beta)\,.$$

In Fig. 16.1, $\beta$ is negative (a clockwise rotation) so $\alpha + \beta$ is the angle between the second link and the line parallel to the $x$-axis.

Combining the results gives:

$$x = l_1 \cos\alpha + l_2 \cos(\alpha + \beta)$$
$$y = l_1 \sin\alpha + l_2 \sin(\alpha + \beta)\,.$$

**Example** Let $l_1 = l_2 = 1, \alpha = 60°, \beta = -30°$. Then:

$$x = 1 \cdot \cos 60 + 1 \cdot \cos(60 - 30) = \frac{1}{2} + \frac{\sqrt{3}}{2} = \frac{1+\sqrt{3}}{2}$$
$$y = 1 \cdot \sin 60 + 1 \cdot \sin(60 - 30) = \frac{\sqrt{3}}{2} + \frac{1}{2} = \frac{1+\sqrt{3}}{2}\,.$$

Let us check if this result makes sense. Figure 16.2 shows a triangle formed by adding a line between $(0, 0)$ and $(x, y)$. The complement of the angle $\beta$ is $180-30 = 150$ and the triangle is isosceles since both sides are 1, so the other angles of the triangle are equal and their values are $(180 - 150)/2 = 15$. The angle that the new line forms with the $x$-axis is $60 - 15 = 45$, which is consistent with $x = y$.

**Activity 16.1: Forward kinematics**

- Program your robot so that it traces the path of the arm in Fig. 16.1: turn left 60°, move forward one unit (1m or some other convenient distance), turn right 30°, move forward one unit.
- Measure the $x$- and $y$-distances of the robot from the origin and compare them to the values computed by the equations for forward kinematics.

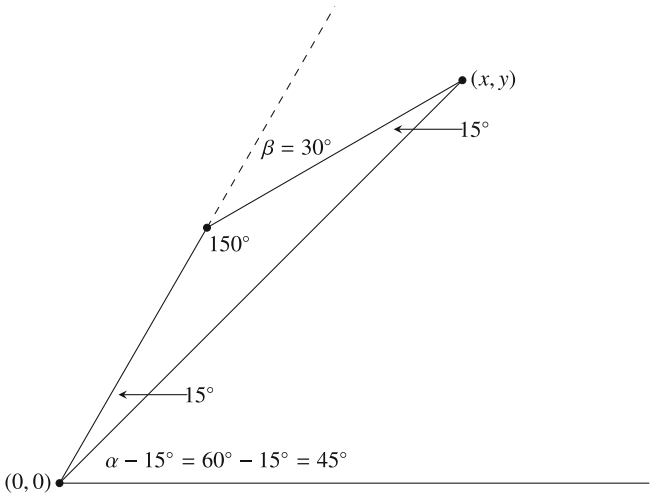



**Fig. 16.2** Computing the angles

## 16.2 Inverse Kinematics

The gray ring in Fig. 16.3 shows the *workspace* of the two-link arm, the set of positions that the end effector can reach. (We assume that $l_2 < l_1$.) The workspace is circularly symmetric since we assume that there are no limitations of the rotation of the joints in a full circle between $-180°$ and $180°$. Any point like $a$ on the circumference of the outer circle is a furthest position of the arm from the origin; it is obtained when the two links are lined up so the arm length is $l_1 + l_2$. The closest positions to the origin in the workspace are points like $b$ on the circumference of the inner circle; they are obtained when the second link is bend back on the first link giving a length of $l_1 - l_2$. Another reachable position $c$ is shown; there are *two* configurations (rotations of the joints) that cause the arm to be positioned at $c$.

Under the assumption that $l_2 < l_1$, no sequence of rotations can position the end of the arm closer to the origin that $l_1 - l_2$ and no position at a distance greater than $l_1 + l_2$ from the origin is accessible. From the figure we learn that a problem in inverse kinematics—finding commands to reach a specified point—can have zero, one or many solutions.

The computation of the inverse kinematics uses the *lassw of cosines* (Fig. 16.4):

$$a^2 + b^2 - 2ab\cos\theta = c^2\,.$$

In a right triangle $\cos 90° = 0$ and the law reduces to the Pythagorean theorem.

Suppose now that we are given a point $(x, y)$ and we want values for $\alpha$, $\beta$ (if any exist) which will bring the arm to that point. Figure 16.5 similar to Fig. 16.2 except that the specific values are replaced by arbitrary angles and lengths.

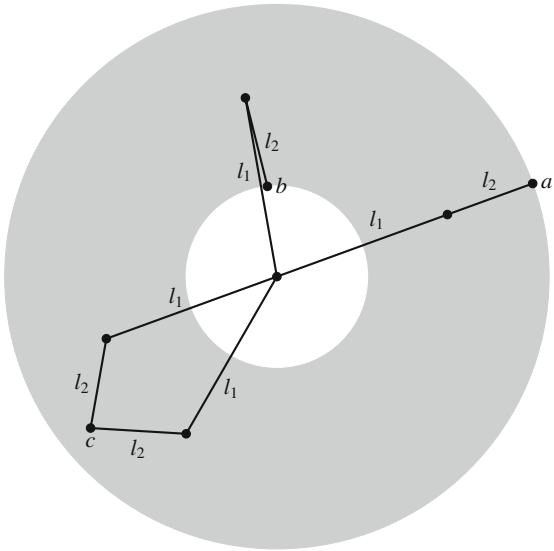



**Fig. 16.3** Workspace of a two-lever arm

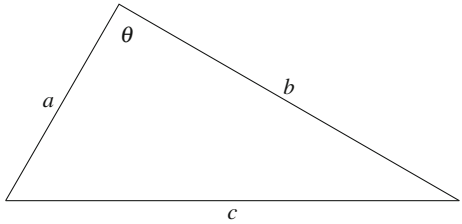



**Fig. 16.4** Law of cosines

By the Pythagorean theorem, $r = \sqrt{x^2 + y^2}$.

The law of cosines gives:

$$l_1^2 + l_2^2 - 2l_1 l_2 \cos(180^\circ - \beta) = r^2 \,,$$

which can be solved for $\beta$:

$$\cos(180^\circ - \beta) = \frac{l_1^2 + l_2^2 - r^2}{2l_1 l_2}$$

$$\beta = 180^\circ - \cos^{-1}\left(\frac{l_1^2 + l_2^2 - r^2}{2l_1 l_2}\right) .$$

To obtain $\gamma$ and then $\alpha$, use the law of cosines with $\gamma$ as the central angle:

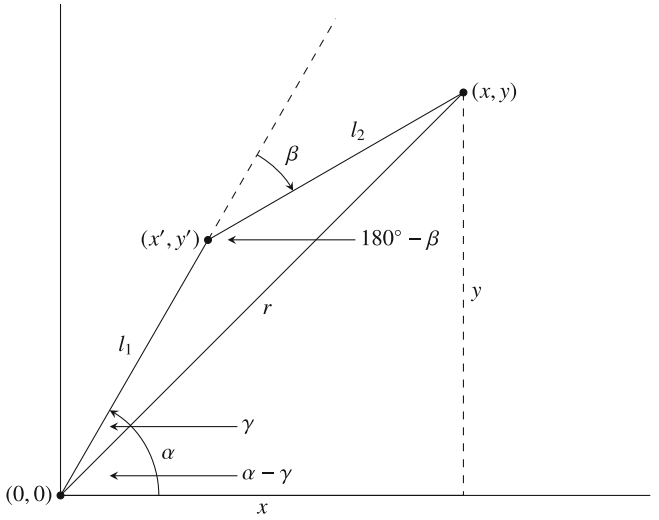



**Fig. 16.5** Inverse kinematics of a two-link arm

$$\cos\gamma = \frac{l_1^2 + r^2 - l_2^2}{2l_1 r}\,.$$

From the right triangle formed by $(x, y)$ we have:

$$\tan(\alpha - \gamma) = \frac{y}{x}$$
$$\alpha = \tan^{-1}\frac{y}{x} + \gamma\,,$$

so:

$$\alpha = \tan^{-1}\frac{y}{x} + \cos^{-1}\left(\frac{l_1^2 + r^2 - l_2^2}{2l_1 r}\right).$$

**Example** Assume again that $l_1 = l_2 = 1$ and that the end effector is at the point computed from the forward kinematics:

$$(x, y) = \left(\frac{1+\sqrt{3}}{2}, \frac{1+\sqrt{3}}{2}\right).$$

First, compute $r^2$:

$$r^2 = x^2 + y^2 = \left(\frac{1+\sqrt{3}}{2}\right)^2 + \left(\frac{1+\sqrt{3}}{2}\right)^2 = 2 + \sqrt{3}\,,$$

and use it in the computation of $\beta$:

$$\begin{aligned}\beta &= 180^\circ - \cos^{-1}\left(\frac{1^2 + 1^2 - (2+\sqrt{3})}{2 \cdot 1 \cdot 1}\right) \\ &= 180^\circ - \cos^{-1}\left(-\frac{\sqrt{3}}{2}\right) \\ &= 180^\circ \pm 150^\circ \\ &= \pm 30^\circ ,\end{aligned}$$

since $330^\circ = -30^\circ \pmod{360^\circ}$. There are two solutions because there are two ways of moving the arm to $(x, y)$.

Next compute $\gamma$:

$$\gamma = \cos^{-1}\left(\frac{1^2 + r^2 - 1^2}{2 \cdot 1 \cdot r}\right) = \cos^{-1}\left(\frac{r}{2}\right) = \cos^{-1}\left(\frac{\sqrt{2+\sqrt{3}}}{2}\right) = \pm 15^\circ . \tag{16.1}$$

The inverse cosine can be obtained numerically on a calculator or algebraically as shown in Appendix B.7.

Since $x = y$, the computation of $\alpha$ is easy:

$$\alpha = \tan^{-1}\frac{y}{x} + \gamma = \tan^{-1} 1 + \gamma = 45^\circ \pm 15^\circ = 60^\circ \text{ or } 30^\circ .$$

The solution $\alpha = 60^\circ$, $\beta = -30^\circ$ corresponds to the rotation of the joints in Fig. 16.1, while the solution $\alpha = 30^\circ$, $\beta = 30^\circ$ corresponds to rotating both of the joints $30^\circ$ counterclockwise.

In this simple case, it is possible to solve the forward kinematics equation to obtain formulas for the inverse kinematics. In general this is not possible so approximate numerical solutions are used.

**Activity 16.2: Inverse kinematics**

- Use the formulas for the inverse kinematics to program your robot to move to a specified coordinate.
- Measure the $x$- and $y$-distances of the robot from the origin and compare them to the specified coordinates.
- If your robot's computer does not have the capability to compute the formulas, compute them offline and then input the commands to the robot.

## 16.3 Rotations

The motion of a robotic manipulator is described in terms of *coordinate frames*. Three frames are associated with the arm in Fig. 16.1: one frame is associated with the joint at the origin (which we assume is fixed to a table or the floor), a second frame is associated with the joint between the two links, and a third frame is associated with the end effector at the end of the second link.

In this section we describe how the rotational motion of a robotic arm can be mathematically modeled using *rotation matrices*. The links in robotic arms introduce *translations*: the second joint is offset by a linear distance of $l_1$ from the first joint, and the end effector is offset by a linear distance of $l_2$ from the second joint. The mathematical treatment of translations uses an extension of rotation matrices called *homogeneous transforms*.

Rotations can be confusing because a rotation matrix can have three interpretations that are described in the following subsections: rotating a vector, rotating a coordinate frame and transforming a vector from one coordinate frame to another.

### *16.3.1 Rotating a Vector*

Consider a vector with cartesian coordinates $(x, y)$ and polar coordinates $(r, \phi)$ (Fig. 16.6a). Now rotate the vector by an angle $\theta$ (Fig. 16.6b). Its polar coordinates are $(r, \phi + \theta)$. What are its cartesian coordinates?

Using the trigonometric identities for the sum of two angles and the conversion of $(r, \phi)$ to $(x, y)$ we have:

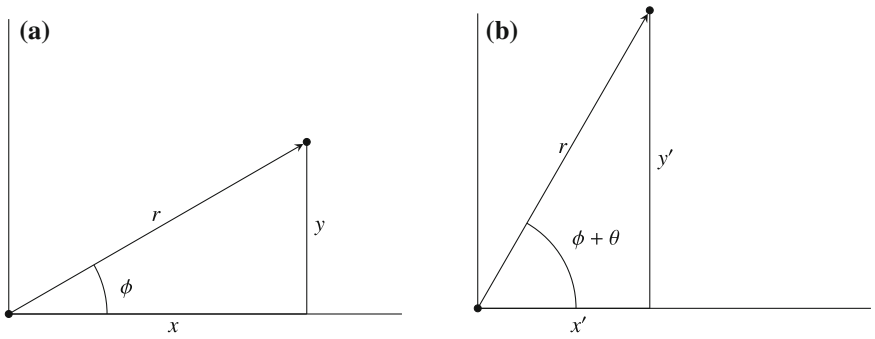



**Fig. 16.6** **a** A vector. **b** The vector rotated by $\theta$

$$\begin{aligned} x' &= r\cos(\phi+\theta) \\ &= r\cos\phi\cos\theta - r\sin\phi\sin\theta \\ &= (r\cos\phi)\cos\theta - (r\sin\phi)\sin\theta \\ &= x\cos\theta - y\sin\theta\,, \end{aligned}$$

$$\begin{aligned} y' &= r\sin(\phi+\theta) \\ &= r\sin\phi\cos\theta + r\cos\phi\sin\theta \\ &= (r\sin\phi)\cos\theta + (r\cos\phi)\sin\theta \\ &= y\cos\theta + x\sin\theta \\ &= x\sin\theta + y\cos\theta\,. \end{aligned}$$

These equations can be expressed as the multiplication of a matrix called the *rotation matrix* and a vector:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}.$$

**Example** Let $p$ be the point at the tip of a vector of length $r = 1$ that forms an angle of $\phi = 30°$ with the positive $x$-axis. The cartesian coordinates of $p$ are $\left(\frac{\sqrt{3}}{2}, \frac{1}{2}\right)$. Suppose that the vector is rotated by $\theta = 30°$. What are the new cartesian coordinates of $p$? Using matrix multiplication:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} \\ \frac{1}{2} & \frac{\sqrt{3}}{2} \end{bmatrix} \begin{bmatrix} \frac{\sqrt{3}}{2} \\ \frac{1}{2} \end{bmatrix} = \begin{bmatrix} \frac{1}{2} \\ \frac{\sqrt{3}}{2} \end{bmatrix}.$$

The result makes sense because rotating a vector whose angle with the $x$-axis is 30° by 30° should give a vector whose angle with the $x$-axis is 60°.

Suppose that the vector is rotated by an additional 30°; its new coordinates are:

$$\begin{bmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} \\ \frac{1}{2} & \frac{\sqrt{3}}{2} \end{bmatrix} \left( \begin{bmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} \\ \frac{1}{2} & \frac{\sqrt{3}}{2} \end{bmatrix} \begin{bmatrix} \frac{\sqrt{3}}{2} \\ \frac{1}{2} \end{bmatrix} \right) = \begin{bmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} \\ \frac{1}{2} & \frac{\sqrt{3}}{2} \end{bmatrix} \begin{bmatrix} \frac{1}{2} \\ \frac{\sqrt{3}}{2} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \tag{16.2}$$

This result also makes sense. Rotating a vector whose angle is 30° twice by 30° (for a total of 60°) should give 90°. The cosine of 90° is 0 and the sine of 90° is 1.

Since matrix multiplication is associative, the multiplication could also be performed as follows:

$$\left( \begin{bmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} \\ \frac{1}{2} & \frac{\sqrt{3}}{2} \end{bmatrix} \begin{bmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} \\ \frac{1}{2} & \frac{\sqrt{3}}{2} \end{bmatrix} \right) \begin{bmatrix} \frac{\sqrt{3}}{2} \\ \frac{1}{2} \end{bmatrix}. \tag{16.3}$$

**Activity 16.3: Rotation matrices**

- Demonstrate the associativity of matrix multiplication by showing that the multiplication in Eq. 16.3 gives the same result as the multiplication in Eq. 16.2.
- Compute the matrix for a rotation of $-30°$ and show that multiplying this matrix by the matrix for a rotation of $30°$ gives the matrix for a rotation of $0°$.
- Is this multiplication commutative?
- Is multiplication of two-dimensional rotational matrices commutative?

### *16.3.2 Rotating a Coordinate Frame*

Let us reinterpret Fig. 16.6a, b. Figure 16.7a shows a coordinate frame (blue) defined by two orthogonal unit vectors:

$$\mathbf{x} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

Figure 16.7b shows the *coordinate frame* rotated by $\theta$ degrees (red). The new unit vectors $\mathbf{x}'$ and $\mathbf{y}'$ can be obtained by multiplication by the rotation matrix derived above:

$$\mathbf{x}' = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix}$$

$$\mathbf{y}' = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} -\sin\theta \\ \cos\theta \end{bmatrix}.$$

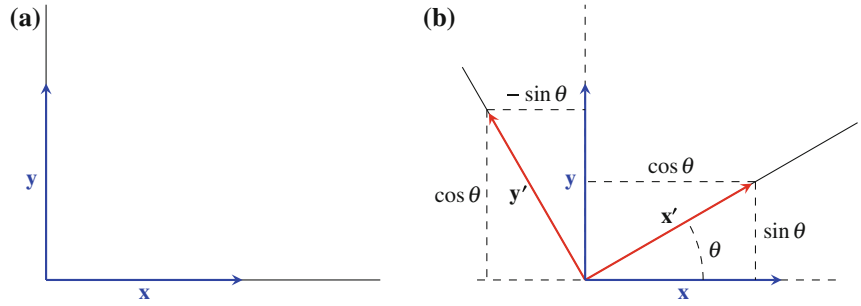



**Fig. 16.7** **a** Original coordinate frame (*blue*). **b** New coordinate frame (*red*) obtained by rotating the original coordinate frame (*blue*) by $\theta$

**Example** For the unit vectors in Fig. 16.7a and a rotation of 30°:

$$\mathbf{x}' = \begin{bmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} \\ \frac{1}{2} & \frac{\sqrt{3}}{2} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{3}}{2} \\ \frac{1}{2} \end{bmatrix}$$

$$\mathbf{y}' = \begin{bmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} \\ \frac{1}{2} & \frac{\sqrt{3}}{2} \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} -\frac{1}{2} \\ \frac{\sqrt{3}}{2} \end{bmatrix} .$$

### *16.3.3 Transforming a Vector from One Coordinate Frame to Another*

Let the origin of a coordinate frame $b$ (blue) represent the joint of an end effector such as a welder and let point $p$ be the tip of the welder (Fig. 16.8a). By convention in robotics, the coordinate frame of an entity is denoted by a "pre" superscript.[1] In the frame $b$, the point ${}^b p$ has polar coordinates $(r, \phi)$ and cartesian coordinates $({}^b x, {}^b y)$ related by the usual trigonometric formulas:

$${}^b p = ({}^b x, {}^b y) = (r \cos\phi,\ r \sin\phi) .$$

Suppose that the joint (*with its coordinate frame*) is rotated by the angle $\theta$. The coordinates of the point relative to $b$ remain the same, but the coordinate frame has moved so we ask: What are the coordinates ${}^a p = ({}^a x, {}^a y)$ of the point in the coordinate frame before it was moved? In Fig. 16.8b the original frame $b$ is shown rotated to a new position (and still shown in blue), while the coordinate frame $a$ is in the old

[1] The convention is to use uppercase letters for both the frame and the coordinates, but we use lowercase for clarity.

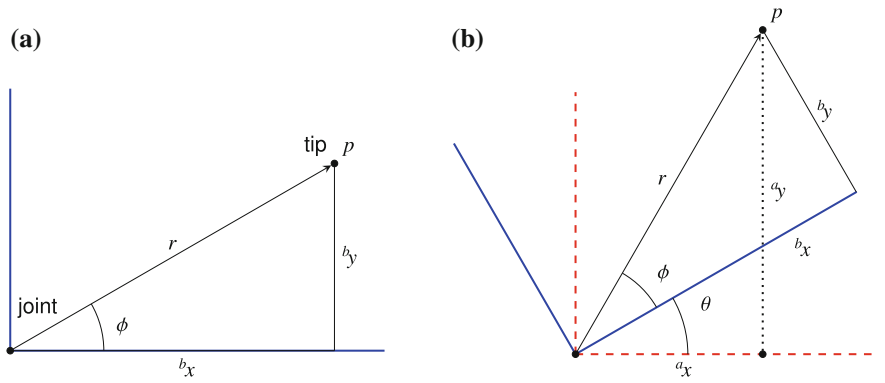



**Fig. 16.8** **a** Point $p$ at the tip of an end effector in coordinate frame $b$ (*blue*). **b** Point $p$ in coordinate frames $a$ (*red*) and $b$ (*blue*)

position of $b$ and is shown as red dashed lines. In the previous section we asked how to transform one coordinate frame into another; here, we are asking how to transform the coordinates of a point in a frame to its coordinates in another frame.

In terms of the robotic arm: we know $({}^b x, {}^b y)$, the coordinates of the tip of the end effector relative to the frame of the end effector, and we now ask for its coordinates ${}^a p = ({}^a x, {}^a y)$ relative to the fixed base. This is important because if we know ${}^a p$, we can compute the distance and angle from the tip of the welder to the parts of the car it must now weld.

We can repeat the computation used for rotating a vector:

$$\begin{aligned} {}^a x &= r\cos(\phi+\theta) \\ &= r\cos\phi\cos\theta - r\sin\phi\sin\theta \\ &= {}^b x\cos\theta - {}^b y\sin\theta\,, \end{aligned}$$

$$\begin{aligned} {}^a y &= r\sin(\phi+\theta) \\ &= r\sin\phi\cos\theta + r\cos\phi\sin\theta \\ &= {}^b x\sin\theta + {}^b y\cos\theta\,, \end{aligned}$$

to obtain the rotation matrix:

$$\begin{bmatrix} {}^a x \\ {}^a y \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} {}^b x \\ {}^b y \end{bmatrix}. \tag{16.4}$$

The matrix is called the rotation matrix *from* frame $b$ *to* frame $a$ and denoted ${}^{a}_{b}R$. Pre-multiplying the point ${}^{b}p$ in frame $b$ by the rotation matrix gives ${}^{a}p$ its coordinates in frame $a$:

$$ {}^{a}p = {}^{a}_{b}R \, {}^{b}p \, . $$

**Example** Let ${}^{b}p$ be the point in frame $b$ at the tip of a vector of length $r = 1$ that forms an angle of $\phi = 30^\circ$ with the positive $x$-axis. The coordinates of ${}^{b}p$ are $\left(\frac{\sqrt{3}}{2}, \frac{1}{2}\right)$. Suppose that the coordinate frame $b$ (together with the point $p$) is rotated by $\theta = 30^\circ$ to obtain the coordinate frame $a$. What are the coordinates of ${}^{a}p$? Using Eq. 16.4:

$$ {}^{a}p = \begin{bmatrix} {}^{a}x \\ {}^{a}y \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} \\ \frac{1}{2} & \frac{\sqrt{3}}{2} \end{bmatrix} \begin{bmatrix} \frac{\sqrt{3}}{2} \\ \frac{1}{2} \end{bmatrix} = \begin{bmatrix} \frac{1}{2} \\ \frac{\sqrt{3}}{2} \end{bmatrix} . $$

If frame $a$ is now rotated 30°, we obtain the coordinates of the point in a third frame $a1$. Pre-multiply ${}^{a}p$ by the rotation matrix for 30° to obtain ${}^{a1}p$:

$$ {}^{a1}p = \begin{bmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} \\ \frac{1}{2} & \frac{\sqrt{3}}{2} \end{bmatrix} \left( \begin{bmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} \\ \frac{1}{2} & \frac{\sqrt{3}}{2} \end{bmatrix} \begin{bmatrix} \frac{\sqrt{3}}{2} \\ \frac{1}{2} \end{bmatrix} \right) = \begin{bmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} \\ \frac{1}{2} & \frac{\sqrt{3}}{2} \end{bmatrix} \begin{bmatrix} \frac{1}{2} \\ \frac{\sqrt{3}}{2} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} . $$

The product of the two rotation matrices:

$$ \begin{bmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} \\ \frac{1}{2} & \frac{\sqrt{3}}{2} \end{bmatrix} \begin{bmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} \\ \frac{1}{2} & \frac{\sqrt{3}}{2} \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & -\frac{\sqrt{3}}{2} \\ \frac{\sqrt{3}}{2} & \frac{1}{2} \end{bmatrix} $$

results in the rotation matrix for rotating the original coordinate frame $b$ by 60°:

$$ \begin{bmatrix} \frac{1}{2} & -\frac{\sqrt{3}}{2} \\ \frac{\sqrt{3}}{2} & \frac{1}{2} \end{bmatrix} \begin{bmatrix} \frac{\sqrt{3}}{2} \\ \frac{1}{2} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} . $$

Given a sequence of rotations, pre-multiplying their rotation matrices gives the rotation matrix for the rotation equivalent to the sum of the individual rotations.

## 16.4 Rotating and Translating a Coordinate Frame

The joints on robotics manipulators are connected by links so the coordinate systems are related not just by rotations but also by translations. The point $p$ in Fig. 16.9 represents a point in the (red) coordinate frame $b$, but relative to the (blue dashed) coordinate frame $a$, frame $b$ is both rotated by the angle $\theta$ and its origin is translated

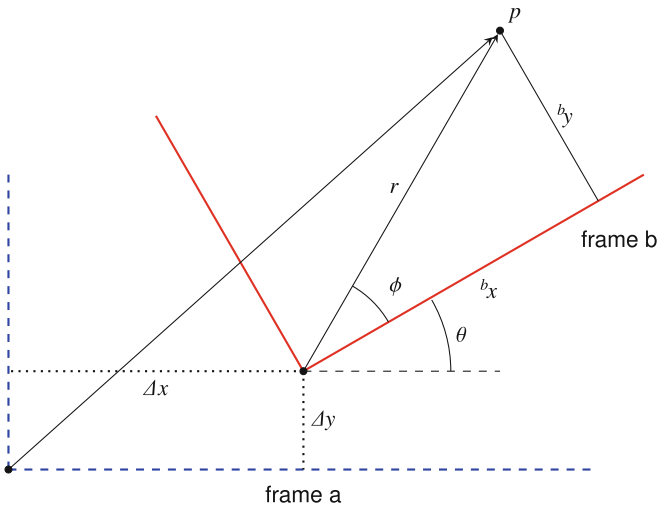



**Fig. 16.9** Frame $b$ is rotated and translated to frame $a$

by $\Delta x$ and $\Delta y$. If ${}^{b}p = ({}^{b}x, {}^{b}y)$, the coordinates of the point in frame $b$, are known, what are its coordinates ${}^{a}p = ({}^{a}x, {}^{a}y)$ in frame $a$?

To perform this computation, we define an intermediate (green) coordinate frame $a1$ that has the same origin as $b$ and the same orientation as $a$ (Fig. 16.10). What are the coordinates ${}^{a1}p = ({}^{a1}x, {}^{a1}y)$ of the point in frame $a1$? This is simply the rotation by $\theta$ that we have done before:

$$ {}^{a1}p = \begin{bmatrix} {}^{a1}x \\ {}^{a1}y \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} {}^{b}x \\ {}^{b}y \end{bmatrix} . $$

Now that we have the coordinates of the point in $a1$, it is easy to obtain the coordinates in frame $a$ by adding the offsets of the translation. In matrix form:

$$ {}^{a}p = \begin{bmatrix} {}^{a}x \\ {}^{a}y \end{bmatrix} = \begin{bmatrix} {}^{a1}x \\ {}^{a1}y \end{bmatrix} + \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} . $$

*Homogeneous transforms* are used to combine a rotation and a translation in one operator. The two-dimensional vector giving the coordinates of a point is extended with a third element that has a fixed value of 1:

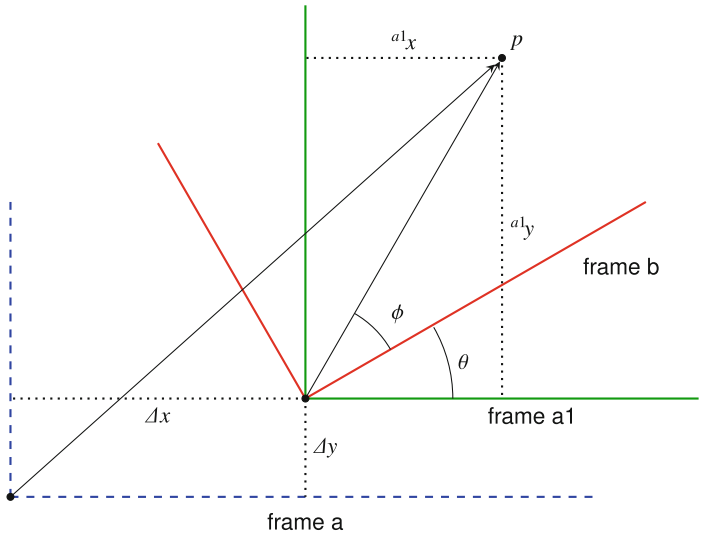



**Fig. 16.10** Frame $b$ is rotated to frame $a1$ and then translated to frame $a$

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}.$$

The rotation matrix is extended to a $3 \times 3$ matrix with a 1 in the lower right corner and zeros elsewhere. It is easy to check that multiplication of a vector in frame $b$ by the rotation matrix results in the same vector as before except for the extra 1 element:

$$\begin{bmatrix} {}^{a1}x \\ {}^{a1}y \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} {}^{b}x \\ {}^{b}y \\ 1 \end{bmatrix}.$$

The result is the coordinates of the point in the intermediate frame $a1$. To obtain the coordinates in frame $a$, we multiply by a matrix that performs the translation:

$$\begin{bmatrix} {}^{a}x \\ {}^{a}y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} {}^{a1}x \\ {}^{a1}y \\ 1 \end{bmatrix}.$$

By multiplying the two transforms, we obtain a single homogeneous transform that can perform both the rotation and that translation:

$$\begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & \Delta x \\ \sin\theta & \cos\theta & \Delta y \\ 0 & 0 & 1 \end{bmatrix}.$$

**Example** Let us extend the previous example by adding a translation of $(3, 1)$ to the rotation of $30°$. The homogeneous transform of the rotation followed by the translation is:

$$\begin{bmatrix} 1 & 0 & 3 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} & 0 \\ \frac{1}{2} & \frac{\sqrt{3}}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} & 3 \\ \frac{1}{2} & \frac{\sqrt{3}}{2} & 1 \\ 0 & 0 & 1 \end{bmatrix}.$$

The coordinates of the point in frame $a$ are:

$$\begin{bmatrix} {}^{a}x \\ {}^{a}y \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} & 3 \\ \frac{1}{2} & \frac{\sqrt{3}}{2} & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{\sqrt{3}}{2} \\ \frac{1}{2} \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{2} + 3 \\ \frac{\sqrt{3}}{2} + 1 \\ 1 \end{bmatrix}.$$

**Activity 16.4: Homogeneous transforms**

- Draw the diagram for a rotation of $-30°$ followed by a translation of $(3, -1)$.
- Compute the homogeneous transform.

## 16.5 A Taste of Three-Dimensional Rotations

The concepts of coordinate transformations and kinematics in three-dimensions are the same as in two dimensions, however, the mathematics is more complicated. Furthermore, many of us find it difficult to visualize three-dimensional motion when all we are shown are two-dimensional representations of three-dimensional objects. In this section we give a taste of three-dimensional robotics by looking at rotations in three dimensions.

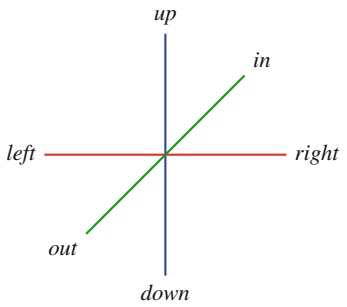



**Fig. 16.11** Three-dimensional coordinate frame

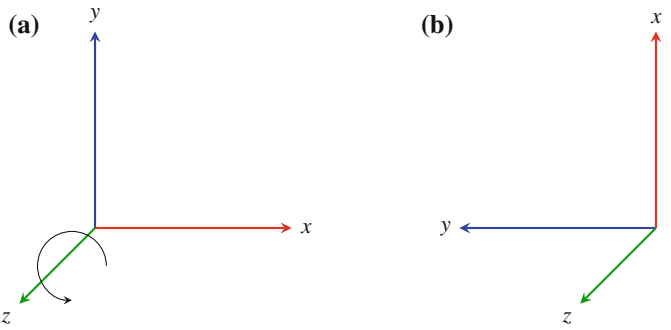



**Fig. 16.12** **a** $x$-$y$-$z$ coordinate frame. **b** $x$-$y$-$z$ coordinate frame after rotating 90° around the $z$-axis

### *16.5.1 Rotations Around the Three Axes*

A two-dimensional $x$-$y$ coordinate frame can be considered to be embedded in a three-dimensional coordinate frame by adding a $z$-axis perpendicular to the $x$- and $y$-axes. Figure 16.11 shows a two-dimensional representation of the three-dimensional frame. The $x$-axis is drawn left and right on the paper and the $y$-axis is drawn up and down. The diagonal line represents the $z$-axis which is perpendicular to the other two axes. The "standard" $x$-$y$-$z$ coordinate frame has the positive directions of its axes defined by the right-hand rule (see below). The positive directions are *right* for the $x$-axis, *up* for the $y$-axis and *out* (of the paper towards the observer) for the $z$-axis.

Rotate the coordinate frame counterclockwise around the $z$-axis, so that the $z$-axis remains unchanged (Fig. 16.12a, b). The new orientation of the frame is (*up*, *left*, *out*). Consider now a rotation of 90° around the $x$-axis (Fig. 16.13a, b). This causes the $y$-axis to "jump out" of the paper and the $z$-axis to "fall down" onto the paper, resulting in the orientation (*right*, *out*, *down*). Finally, consider a rotation of 90° around the $y$-axis (Fig. 16.14a, b). The $x$-axis "drops into" the paper and the $z$-axis "falls right" onto the paper. The new position of the frame is (*in*, *up*, *right*).

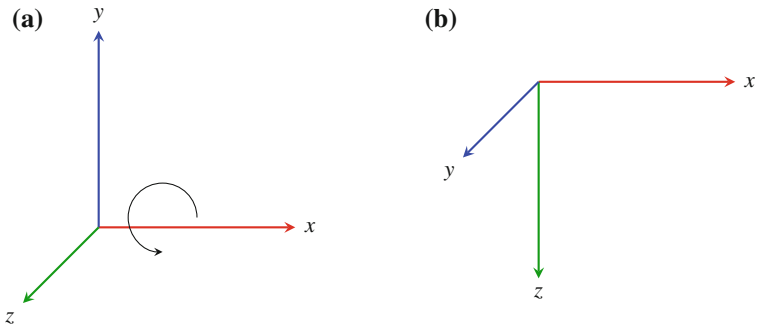



**Fig. 16.13** **a** $x$-$y$-$z$ coordinate frame. **b** $x$-$y$-$z$ coordinate frame after rotating 90° around the $x$-axis

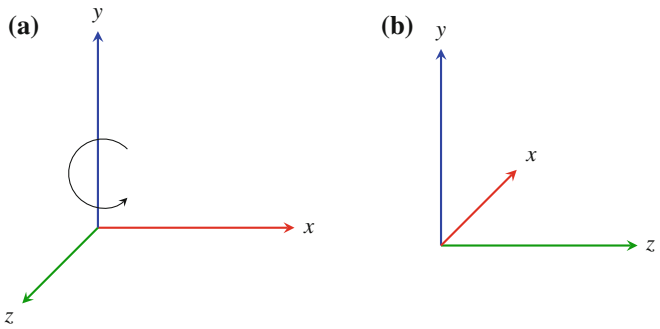



**Fig. 16.14** **a** $x$-$y$-$z$ coordinate frame. **b** $x$-$y$-$z$ coordinate frame after rotating 90° around the $y$-axis

### *16.5.2 The Right-Hand Rule*

There are two orientations for each axis, $2^3 = 8$ orientations overall. What matters is the relative orientation of one axis with respect to the other two; for example, once the $x$- and $y$-axes have been chosen to lie in the plane of the paper, the $z$-axis can have its positive direction pointing out of the paper or into the paper. The choice must be consistent. The convention in physics and mechanics is the *right-hand rule*. Curl the fingers of your right hand so that they go from the one axis to another axis. Your thumb now points in the positive direction of the third axis. For the familiar $x$- and $y$-axes on paper, curl your fingers on the path from the $x$-axis to the $y$-axis. When you do so your thumb points out of the paper and this is taken as the positive direction of the $z$-axis. Figure 16.15 shows the right-hand coordinate system displayed with each of the three axes pointing *out of the paper*. According to the right-hand rule the three rotations are:

Rotate *from* x *to* y around z,
Rotate *from* y *to* z around x,
Rotate *from* z *to* x around y.

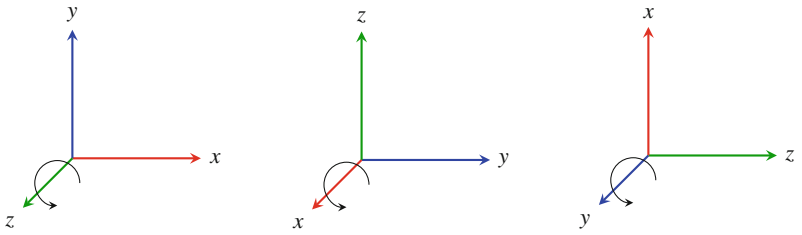



**Fig. 16.15** The right-hand rule

### *16.5.3 Matrices for Three-Dimensional Rotations*

A three-dimensional rotation matrix is a $3 \times 3$ matrix because each point $p$ in a frame has three coordinates $p_x$, $p_y$, $p_z$ that must be moved. Start with a rotation of $\psi$ around the $z$-axis, followed by a rotation of $\theta$ around the $y$ axis and finally a rotation of $\phi$ around the $x$-axis. For the first rotation around the $z$-axis, the $x$ and $y$ coordinates are rotated as in two dimensions and the $z$ coordinate remains unchanged. Therefore, the matrix is:

$$R_{z(\psi)} = \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

For the rotation by $\theta$ around the $y$-axis, the $y$ coordinate is unchanged and the $z$ and $x$ coordinates are transformed "as if" they were the $x$ and $y$ coordinates of a rotation around the $z$-axis:

$$R_{y(\theta)} = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}.$$

For the rotation by $\phi$ around the $x$-axis, the $x$ coordinate is unchanged and the $y$ and $z$ coordinates are transformed "as if" they were the $x$ and $y$ coordinates of a rotation around the $z$-axis:

$$R_{x(\phi)} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix}.$$

It may seem strange that in the matrix for the rotation around the $y$-axis the signs of the sine function have changed. To convince yourself that matrix for this rotation

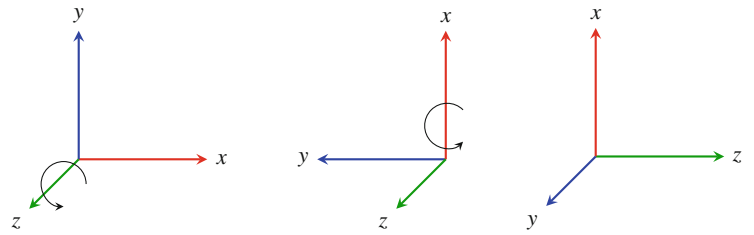



**Fig. 16.16** Rotation around the $z$-axis followed by rotation around the $x$-axis

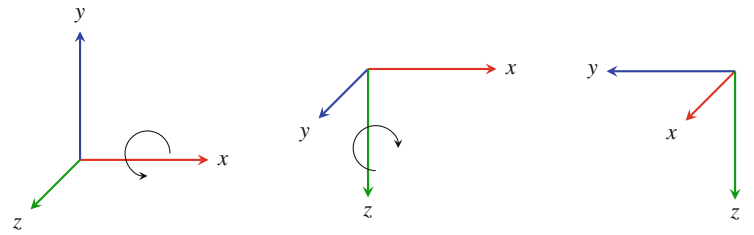



**Fig. 16.17** Rotation around the $x$-axis followed by rotation around the $z$-axis

is correct, redraw the diagram in Fig. 16.8b, substituting $z$ for $x$ and $x$ for $y$ and perform the trigonometric computation.

### *16.5.4 Multiple Rotations*

There is a caveat to composing rotations: like matrix multiplication, three-dimensional rotations *do not* commute. Let us demonstrate this by a simple sequence of two rotations. Consider a rotation of 90° around the $z$-axis, followed by a rotation of 90° around the (new position of the) $x$-axis (Fig. 16.16). The result can be expressed as (*up*, *out*, *right*).

Now consider the commuted operation: a rotation of 90° around the $x$-axis, followed by a rotation of 90° around the $z$-axis (Fig. 16.17). The result can be expressed as (*out*, *left*, *down*), which is not the same as the previous orientation.

### *16.5.5 Euler Angles*

An arbitrary rotation can be obtained by three individual rotations around the three axes, so the matrix for an arbitrary rotation can be obtained by multiplying the matrices for each single rotation. The angles of the rotations are called *Euler angles*.

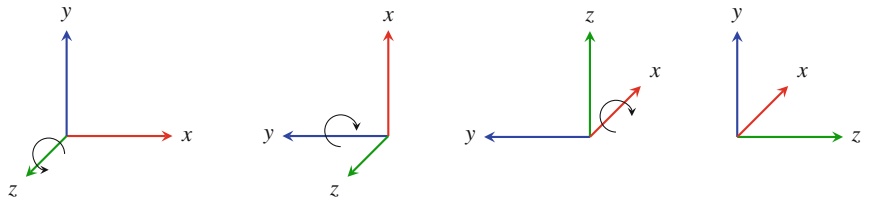



**Fig. 16.18** Euler angles $zyx$ of (90°, 90°, 90°)

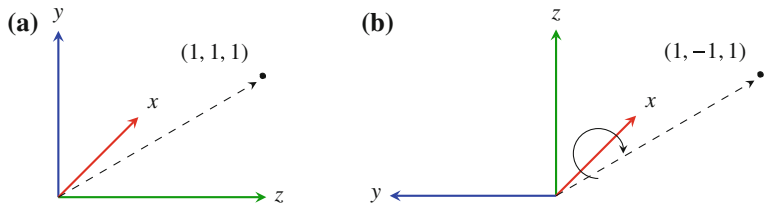



**Fig. 16.19** **a** Vector after final rotation. **b** Vector before rotating around the $x$-axis

The formulas are somewhat complex and can be found in the references listed at the end of the chapter. Here we demonstrate Euler angles with an example.

**Example** Figure 16.18 shows a coordinate frame rotated sequentially 90° around the $z$-axis, then the $y$-axis and finally the $x$-axis. This is called a $zyx$ Euler angle rotation. The final orientation is (*in*, *up*, *right*).

Let us consider a robotic manipulator that consists of a single joint that can rotate around all three axes. A sequence of rotations is performed as shown in Fig. 16.18. Consider the point at coordinates (1, 1, 1) relative to the joint (Fig. 16.19a). After the rotations, what are the coordinates of this point in the original fixed frame?

This can be computed by leaving the vector fixed and considering the rotations of the coordinate frames. To reach the final position shown in Fig. 16.19a the frame was rotated around the $x$-axis from the orientation shown in Fig. 16.19b. By examining the figure we see that the coordinates in this frame are (1, −1, 1). Proceeding through the previous two frames (Fig. 16.20a, b), the coordinates are (1, −1, −1) and (1, 1, −1).

These coordinates can be computed from the rotation matrices for the rotations around the three axes. The coordinates of the final coordinate frame are (1, 1, 1), so in the frame before the rotation around the $x$-axis the coordinates were:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix}.$$

The coordinates in the frame before the rotation around the $y$-axis were:

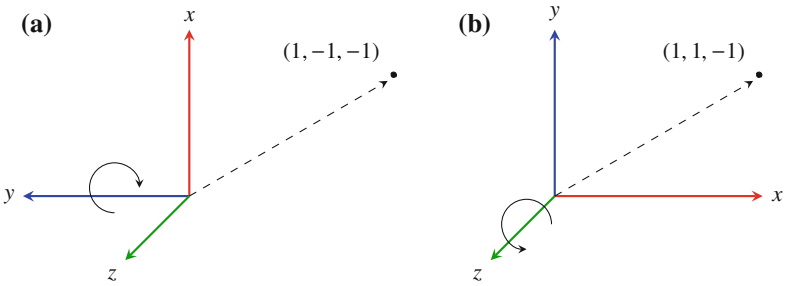



**Fig. 16.20** **a** Vector before rotating around the $y$-axis. **b** Vector in the fixed frame before rotating around the $z$-axis

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} .$$

Finally, the coordinates in the fixed frame before the rotation around the $z$-axis were:

$$\begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} .$$

For three arbitrary $zyx$ Euler angle rotations: $\psi$ around the $z$-axis, then $\theta$ around the $y$-axis and finally $\phi$ around the $x$-axis the rotation matrix is:

$$R = R_{z(\psi)} R_{y(\theta)} R_{x(\phi)} .$$

It may seem strange that the order of the matrix multiplication (which is always from right to left) is opposite the order of the rotations. This is because we are taking a vector in the final coordinate frame and transforming it back into the fixed frame to determine its coordinates in the fixed frame.

**Activity 16.5: Multiple Euler angles**

- Multiply the three matrices to obtain a single matrix that directly transforms the coordinates from (1, 1, 1) to (1, 1, −1).
- Perform the same computation for other rotations, changing the sequence of the axes and the angles of rotation.

### 16.5.6 The Number of Distinct Euler Angle Rotations

There are three axes so there should be $3^3 = 27$ sequences of Euler angles. However, there is no point in rotating around the same axis twice in succession because the same result can be obtained by rotating once by the sum of the angles, so there are only $3 \cdot 2 \cdot 2 = 12$ different Euler angles sequences. The following activity asks you to explore different Euler angle sequences.

**Activity 16.6: Distinct Euler angles**

- To experiment with three-dimensional rotations, it is helpful to construct a coordinate frame from three mutually perpendicular pencils or straws.
- Draw the coordinate frames for a $zyz$ Euler angle rotation, where each rotation is by 90°.
- What $zyz$ rotation gives the same result at the $zyx$ rotation shown in Fig. 16.18?
- Experiment with other rotation sequences and with angles other than 90°.

## 16.6 Advanced Topics in Three-Dimensional Transforms

Now that you have tasted three-dimensional rotations, we survey the next steps in learning this topic which you can study in the textbooks listed in the references.

There are 12 Euler angles and the choice of which to use depends on the intended application. Furthermore, there is a different way of defining rotations. Euler angles are *moving axes* transforms, that is, each rotation is around the *new* position of the axis after the previous rotation. In Fig. 16.18, the second rotation is around the $y$-axis that now points left, not around the original $y$-axis that points up. It is also possible to define *fixed axes* rotations in which subsequent rotations are around the original axes of the coordinate system. In three dimensions, homogeneous transforms that include translations in addition to rotations can be efficiently represented as $4 \times 4$ matrices.

Euler angles are relatively inefficient to compute and suffer from computational instabilities. These can be overcome by using *quaternions*, which are a generalization of complex numbers. Quaternions use three "imaginary" numbers $i$, $j$, $k$, where:

$$i^2 = j^2 = k^2 = ijk = -1\,.$$

Recall that a vector in the two-dimensional plane can be expressed as a complex number $x + \mathbf{i}\,y$. Rotating the vector by an angle $\theta$ can be performed by multiplying by the value $\cos\theta + \mathbf{i}\sin\theta$. Similarly, in three dimensions, a vector can be expressed as a *pure quaternion* with a zero real component: $p = 0 + x\,\mathbf{i} + y\,\mathbf{j} + z\,\mathbf{k}$. Given

an axis and an angle, there exists a quaternion $q$ that rotates the vector around the axis by this angle using the formula $qpq^{-1}$. This computation is more efficient and robust than the equivalent computation with Euler angles and is used in a variety of contexts such as aircraft control and computer graphics.

## 16.7 Summary

Kinematics is the description of the motion of a robot. In forward kinematics, we are given a set of commands for the robot and we need to compute its final position relative to its initial position. In inverse kinematics, we are given a desired final position and need to compute the commands that will bring the robot to that position. This chapter has demonstrated kinematic computations for a simple two-dimensional robotic manipulator arm. In practice, manipulators move in three-dimensions and the computations are more difficult. Exact solutions for computing inverse kinematics usually cannot be found and approximate numerical solutions are used.

There are many ways of defining and computing arbitrary rotations. We mentioned the Euler angles where an arbitrary rotation is obtained by a sequence of three rotations around the coordinate axes. Quaternions, a generalization of complex numbers, are often used in practice because they are computationally more efficient and robust.

## 16.8 Further Reading

Advanced textbooks on robotic kinematics and related topics are those by Craig [2] and Spong et al. [3]. See also Chap. 3 of Correll [1]. Appendix B of [2] contains the rotation matrices for all the Euler angle sequences. The video lectures by Angela Sodemann are very helpful:
https://www.youtube.com/user/asodemann3,
http://www.robogrok.com/Flowchart.html.

Although not a book on robotics, Vince's monograph on quaternions [4] gives an excellent presentation of the mathematics of rotations.

## References

1. Correll, N.: Introduction to Autonomous Robots. CreateSpace (2014). https://github.com/correll/Introduction-to-Autonomous-Robots/releases/download/v1.9/book.pdf
2. Craig, J.J.: Introduction to Robotics: Mechanics and Control, 3rd edn. Pearson, Boston (2005)
3. Spong, M.W., Hutchinson, S., Vidyasagar, M.: Robot Modeling and Control. Wiley, New York (2005)
4. Vince, J.: Quaternions for Computer Graphics. Springer, Berlin (2011)

# Appendix A
# Units of Measurement

Tables A.1 and A.2 show the units of measurement and their abbreviations.

**Table A.1** Units of measurement

| Property | Variable | Unit | Abbreviation |
|---|---|---|---|
| Distance | $s$ | Meter | m |
| Time | $t$ | Second | s |
| Velocity | $v$ | Meter/second | m/s |
| Acceleration | $a$ | Meter/second$^2$ | m/s$^2$ |
| Frequency | $f$ | Hertz | Hz |
| Angle | $\theta$ | Radian | rad |
| | | Degree | ° |

**Table A.2** Prefixes

| Prefix | Meaning | Abbreviation |
|---|---|---|
| kilo- | Thousands | k |
| centi- | Hundredths | c |
| milli- | Thousandths | m |
| micro- | Millionths | μ |

**Examples**:

- $20\,\text{kHz} = 20\,\text{kilohertz} = 20{,}000\,\text{hertz}$
- $15\,\text{cm} = 15\,\text{centimeters} = \frac{15}{100}\,\text{meters}$
- $50\,\text{ms} = 50\,\text{milliseconds} = \frac{50}{1000}\,\text{seconds}$
- $10\,\mu\text{s} = 10\,\text{microseconds} = \frac{10}{1000}\,\text{milliseconds} = \frac{10}{1{,}000{,}000}\,\text{seconds}$



M. Ben-Ari and F. Mondada, *Elements of Robotics*,
https://doi.org/10.1007/978-3-319-62533-1

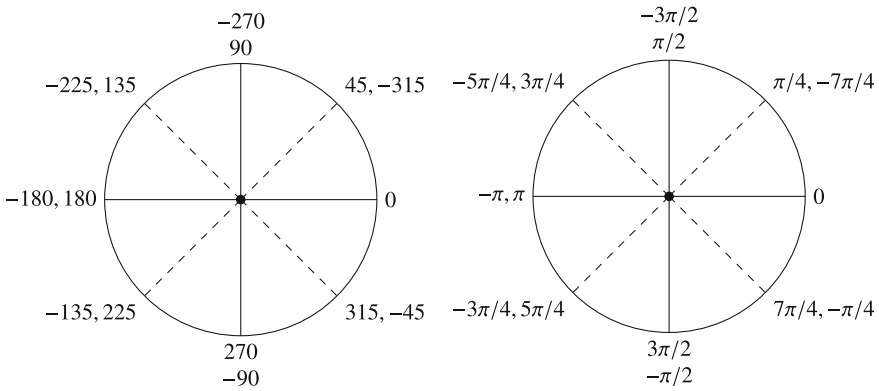



**Fig. A.1** Angles in degrees (*left*) and radians (*right*)

Angles, such as the heading of a robot and the direction to an object, are measured in degrees or radians (Fig. A.1). By convention, angles are positive in the counter-clockwise direction and negative in the clockwise direction as measured from the front of the robot.

# Appendix B
# Mathematical Derivations and Tutorials

This appendix collects mathematical derivations used in the text as well as short tutorials of concepts which may not be familiar.

## B.1 Conditional Probability and Bayes Rule

Given a reading $z$ of the sensor, what is the probability that we are at position $x_i$? This is expressed as a *conditional probability* $p(x_i \mid z)$. The data we have available are the *current* probability $p(x_i)$ that we are at $x_i$, and $p(z \mid x_i)$, the conditional probability that the sensor reads $z$ if we are in fact at $x_i$. Let us multiply these two probabilities:

$$p(z \mid x_i)\, p(x_i)\,.$$

What does this mean? The event $x_i$ occurs with probability $p(x_i)$ and once it occurs the event $z$ occurs with probability $p(z \mid x_i)$. Therefore, this is the probability that both $z$ and $x_i$ occur, called the *joint probability* of the two events:

$$p(z \cap x_i) = p(z \mid x_i)\, p(x_i)\,.$$

The joint probability can also be obtained by multiplying the conditional probability for $x_i$ given $z$ by the probability of $z$:

$$p(x_i \cap z) = p(x_i \mid z)\, p(z)\,.$$

The joint probability is commutative so by equating the two expressions we have:

$$p(x_i \mid z)\, p(z) = p(x_i \cap z) = p(z \cap x_i) = p(z \mid x_i)\, p(x_i)\,.$$



M. Ben-Ari and F. Mondada, *Elements of Robotics*,
https://doi.org/10.1007/978-3-319-62533-1

Dividing by $p(z)$ gives:

$$p(x_i \mid z) = \frac{p(z \mid x_i)\, p(x_i)}{p(z)}\,,$$

which is known as *Bayes rule*.

If we know $p(z \mid x_i)$ and $p(x_i)$ for each $i$, $p(z)$, the *total probability* of the event $z$, can be computed by summing the individual known probabilities:

$$p(z) = \sum_i p(z \mid x_i)\, p(x_i)\,.$$

**Example** Let us do the computation for the example in Sect. 8.4. Let $x_i$ be the event that we are at position $i$ and let $z$ be the event that the robot detects a door. Initially, $p(x_i) = 0.125$ for all positions $x_i$, and, if the robot is at a door, the probability that the sensor detects this correctly is 0.9, while the probability that it incorrectly detects a door is $1 - 0.9 = 0.1$. The probability of detecting a door, $p(z)$, is obtained by summing the probabilities at each position, where the probability is $0.125 \times 0.9 = 0.1125$ at a position with a door and $0.125 \times 0.1 = 0.0125$ at a position with no door:

$$p(z) = 0.1125 + 0.1125 + 0.0125 + 0.0125 + 0.1125 + 0.1125 + 0.1125 + 0.0125 = 0.575\,.$$

By Bayes rule, the probability of being at position $i$ with a door *if* a door is detected is:

$$p(x_i \mid z) = \frac{p(z \mid x_i)\, p(x_i)}{p(z)} = \frac{0.9 \times 0.125}{0.575} = 0.196\,,$$

while the probability of being at position $i$ with no door, but incorrectly a door is detected is:

$$p(x_i \mid z) = \frac{p(z \mid x_i)\, p(x_i)}{p(z)} = \frac{0.1 \times 0.125}{0.575} = 0.022\,.$$

## B.2 Normalization

The set of probabilities of the possible outcomes of an event must add up to 1 since one of the outcomes must occur. If a door is detected, the robot must be at one of the 8 possible positions, but the sum over all positions $i$ of the probability that the robot is at position $i$ was shown above to be:

$$0.1125 + 0.1125 + 0.0125 + 0.0125 + 0.1125 + 0.1125 + 0.1125 + 0.0125 = 0.575\,.$$

The probabilities must be *normalized* by dividing by the sum 0.575 so that the sum will be 1. The normalized probabilities are $0.1125/0.575 \approx 0.19$ and $0.0125/0.575 \approx 0.02$, which do add up to 1:

$$0.19 + 0.19 + 0.02 + 0.02 + 0.19 + 0.19 + 0.19 + 0.02 \approx 1\,.$$

## B.3 Mean and Variance

The *mean*[1] $\mu$ of a set of values $\{x_1, \ldots, x_n\}$ is:

$$\mu = \frac{1}{n} \sum_{i=1}^{n} x_i\,.$$

Consider five people earning 8, 9, 10, 11, 12 thousand Euros per year, respectively. Their mean salary is:

$$\mu = \frac{8 + 9 + 10 + 11 + 12}{5} = \frac{50}{5} = 10\,.$$

The mean doesn't tell us very much because the same mean can be obtained from very different data:

$$\mu = \frac{5 + 6 + 10 + 14 + 15}{5} = \frac{50}{5} = 10\,.$$

The mean is significantly influenced by *outliers*: values that are much higher or lower than the rest of the values. If the person earning 10 thousand Euros suddenly received a bonus of 90 thousand Euros, the mean salary is now:

$$\mu = \frac{8 + 9 + 100 + 11 + 12}{5} = \frac{140}{5} = 28\,.$$

A politician would jump at the opportunity of claiming that during his term of office the average salary had risen by 180%!

*Variance* is a measure of the spread of a set of values. The closer together the values are, the lower the variance. Measures like the mean are more reliable if the values are clustered together and thus have a low variance. The formula for the variance of a set of values $\{x_1, \ldots, x_n\}$ is[2]:

[1] *Mean* is the technical term for average.

[2] $n - 1$ counts the *degrees of freedom*. Since the mean is computed before the variance, we can't choose the $n$ values arbitrarily; the last value chosen is constrained to be the value that causes the computation to produce the given mean.

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \mu)^2 .$$

Each term $x_i - \mu$ measures the distance of the value $x_i$ from the mean; the variance is the average of the squares of these distances. The distances are squared so that values on either side of the mean don't cancel each other out. For example, given values 100 and 300, their mean is 200; if we computed the variance as $(100-200)+(200-300)$, the result would be 0 even though the values are spread out. Using the definition above, the variance is $(100-200)^2 + (200-300)^2 = 20{,}000$.

For the data set {8, 9, 10, 11, 12} the variance is:

$$s^2 = \frac{(-2)^2 + (-1)^2 + 0 + 1^2 + 2^2}{5-1} = \frac{10}{4} = 2.5 ,$$

while for the data set {5, 6, 10, 14, 15} the variance is:

$$s^2 = \frac{(-5)^2 + (-4)^2 + 0 + 4^2 + 5^2}{4} = 20.5 .$$

Since 20.5 is much larger than 2.5 the data in the second set are spread over a wider range than the data of the first set. After the bonus is received the variance is:

$$s^2 = \frac{20^2 + 19^2 + 72^2 + 17^2 + 16^2}{4} = \frac{6490}{4} = 1622.5 .$$

Clearly, one shouldn't interpret the mean salary as meaningful if there are outliers.

## B.4 Covariance

Consider a group of ten people earning the following salaries:

$$x_1 = \{11, 12, 13, 14, 15, 16, 17, 18, 19, 20\} ,$$

in thousands of Euros. The average salary is:

$$\mu_1 = \frac{1}{10}(11 + 12 + 13 + 14 + 15 + 16 + 17 + 18 + 19 + 20) = 15.5 .$$

We conjecture that people with higher salaries buy more expensive cars than those with low salaries. Suppose that there are two models of cars being sold in that area, one for 10 thousand Euros and one for 20 thousand Euros. The following data set shows the cars bought by this group of people, where the $i$th element is the cost of the car bought by the $i$th person:

$$x_2 = \{10, 10, 10, 20, 10, 20, 10, 10, 20, 20\} .$$

To see if there is any connection between the salaries and the costs of the cars, the *covariance* $cov(x_1, x_2)$ between the data sets $x_1$ and $x_2$ is calculated. The computation is similar to that of the variance, except that instead of squaring the difference between a value in a single set and the mean of that set, we multiply the difference between a value from the first set and its mean by the difference between a value from the second set and its mean:

$$cov(x_1, x_2) = \frac{1}{n-1} \sum_{i=1}^{n} (x_{1,i} - \mu_1)(x_{2,i} - \mu_2) \,.$$

The covariance of the sets of value $x_1$ and $x_2$ is 7.8, a positive value, which indicates that salaries and car cost increase together, that is, people making more money tend to buy more expensive cars. If the first five people buy cars worth 10 and the next five people buy cars worth 20, the covariance becomes 13.9, indicating a stronger connection between salary and the cost of a car. Conversely, if the first five buy expensive cars and the next five buy cheap cars, the covariance is $-13.9$, so that as the salary goes up, the cost of a car goes down. Finally, if everyone buys the same car, the covariance is 0 and we conclude as expected that there is no connection between one's salary and the car one buys.

Covariance is symmetric because multiplication of real numbers is commutative:

$$\begin{aligned} cov(x_1, x_2) &= \frac{1}{n-1} \sum_{i=1}^{n} (x_{1,i} - \mu_1)(x_{2,i} - \mu_2) \\ &= \frac{1}{n-1} \sum_{i=1}^{n} (x_{2,i} - \mu_2)(x_{1,i} - \mu_1) \\ &= cov(x_2, x_1) \,. \end{aligned}$$

The covariance matrix combines the variances and the covariances:

$$\begin{bmatrix} s^2(x_1) & cov(x_1, x_2) \\ cov(x_2, x_1) & s^2(x_2) \end{bmatrix} .$$

$cov(x_1, x_2) = cov(x_2, x_1)$, so there are only three different values in the matrix.

## B.5 Multiplication of Vectors and Matrices

Multiplication of a (two-dimensional) matrix $\mathbf{M}$ by a vector $\mathbf{v}$ gives a new vector:

$$\mathbf{Mv} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + by \end{bmatrix} .$$

Multiplication of two matrices is performed by multiplying the rows of left matrix separately with each column vector of the right matrix to get the column vectors for the resulting matrix:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x & u \\ y & v \end{bmatrix} = \begin{bmatrix} ax + by & au + bv \\ cx + dy & cu + dv \end{bmatrix}.$$

The identity matrix is:

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

and it is easy to check that for any matrix $\mathbf{M}$, $\mathbf{M}\,\mathbf{I} = \mathbf{I}\,\mathbf{M} = \mathbf{M}$. For a matrix $\mathbf{M}$, its inverse $\mathbf{M}^{-1}$ is the matrix that results in $\mathbf{I}$ when multiplied by $\mathbf{M}$:

$$\mathbf{M} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \quad \mathbf{M}^{-1} = \frac{1}{det\,(\mathbf{M})} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix},$$

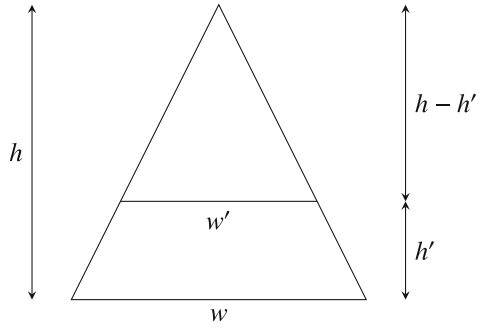
where *det* (**M**), the *determinant* of **M**, is $ad - bc$. We can check this by multiplying:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} = \begin{bmatrix} ad - bc & -ab + ba \\ cd - dc & -bc + da \end{bmatrix} = \begin{bmatrix} ad - bc & 0 \\ 0 & ad - bc \end{bmatrix}.$$

This is valid only for matrices whose determinant is non-zero, because *singular* matrices—those whose determinant is zero—do not have an inverse.

## B.6 The Area of a Trapezoid at the Base of a Triangle

The following diagram shows a triangle of width $w$ and height $h$ with a parallel line at height $h'$ that creates a trapezoid:



We want to find a formula for the area of the trapezoid using the values $w, h, h'$. The area $a$ is the difference between the the areas of the two triangles:

$$a = \frac{wh}{2} - \frac{w'(h-h')}{2}\,.$$

By similar triangles:

$$\frac{h}{h-h'} = \frac{w}{w'}\,,$$

so:

$$w' = \frac{w(h-h')}{h}\,.$$

Substituting:

$$\begin{aligned} a &= \frac{wh}{2} - \frac{w(h-h')(h-h')}{2h} \\ &= \frac{w(h^2 - (h-h')^2)}{2h} \\ &= \frac{w(h^2 - h^2 + 2hh' - h'^2)}{2h} \\ &= \frac{w(2hh' - h'^2)}{2h} \\ &= wh'(1 - \frac{h'}{2h})\,. \end{aligned}$$

## B.7 Algebraic Formulas for cos 15°

Equation 16.1 claims that:

$$\cos^{-1}\left(\frac{\sqrt{2+\sqrt{3}}}{2}\right) = \pm 15^\circ\,.$$

Using the formula for the cosine of the difference of two angles, we have:

$$\begin{aligned} \cos 15^\circ &= \cos(45^\circ - 30^\circ) \\ &= \cos 45^\circ \cos 30^\circ + \sin 45^\circ \sin 30^\circ \\ &= \frac{\sqrt{2}}{2} \cdot \frac{1}{2} + \frac{\sqrt{2}}{2} \cdot \frac{\sqrt{3}}{2} \end{aligned}$$

$$= \frac{\sqrt{2}+\sqrt{6}}{4}.$$

We now compute:

$$\left(\frac{\sqrt{2}+\sqrt{6}}{4}\right)^2 = \left(\frac{8+2\sqrt{2}\sqrt{6}}{16}\right) = \frac{2+\sqrt{3}}{4} = \left(\frac{\sqrt{2+\sqrt{3}}}{2}\right)^2.$$

# Index




M. Ben-Ari and F. Mondada, *Elements of Robotics*,
https://doi.org/10.1007/978-3-319-62533-1