INTRODUCTION TO AUTONOMOUS ROBOTS

MATTIAS WAHDE

Department of Applied Mechanics
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2016

Introduction to Autonomous Robots

MATTIAS WAHDE

Department of Applied Mechanics
Chalmers University of Technology
SE–412 96 Göteborg
Sweden
Telephone: +46 (0)31–772 1000

Contents

Chapter 1

Autonomous robots

Both animals and robots manipulate objects in their environment in order to achieve certain goals. Animals use their senses (e.g. vision, touch, smell) to probe the environment. The resulting information, in many cases also enhanced by the information available from internal states (based on short-term or long-term memory), is processed in the brain, often resulting in an action carried out by the animal, with the use of its limbs.

Similary, robots gain information of the surroundings, using their sensors. The information is processed in the robot's brain[1], consisting of one or several processors, resulting in motor signals that are sent to the actuators (e.g. motors) of the robot.

In this course, the problem of providing robots with the ability of making rational, intelligent decisions will be central. Thus, the development of robotic brains is the main theme of the course. However, a robotic brain cannot operate in isolation: It needs sensory inputs, and it must produce motor output in order to influence objects in the environment. Thus, while it is the author's view that the main challenge in contemporary robotics lies with the development of robotic brains, consideration of the actual hardware, i.e. sensors, processors, motors etc., is certainly very important as well.

This chapter gives a brief overview of **robotic hardware**, i.e. the actual frame (body) of a robot, as well as its sensors, actuators, processors etc. The

[1]The term *control system* is commonly used (instead of the term **robotic brain**). However, this term is misleading, as it leads the reader to think of classical control theory. Concepts from classical control theory *are* relevant in robots; For example, the low-level control of the motors of robots is often taken care of by PI- or PID-regulators. However, **autonomous robots**, i.e. freely moving robots that operate without direct human supervision, are expected to function in complex, unstructured environments, and to make their own decisions concerning which action to take in any given situation. In such cases, systems based *only* on classical control theory are simply insufficient. Thus, hereafter, the term **robotic brain** (or, simply, *brain*) will be used when referring to the system that provides an autonomous robot, however simple, with the ability to process information and decide upon which actions to take.

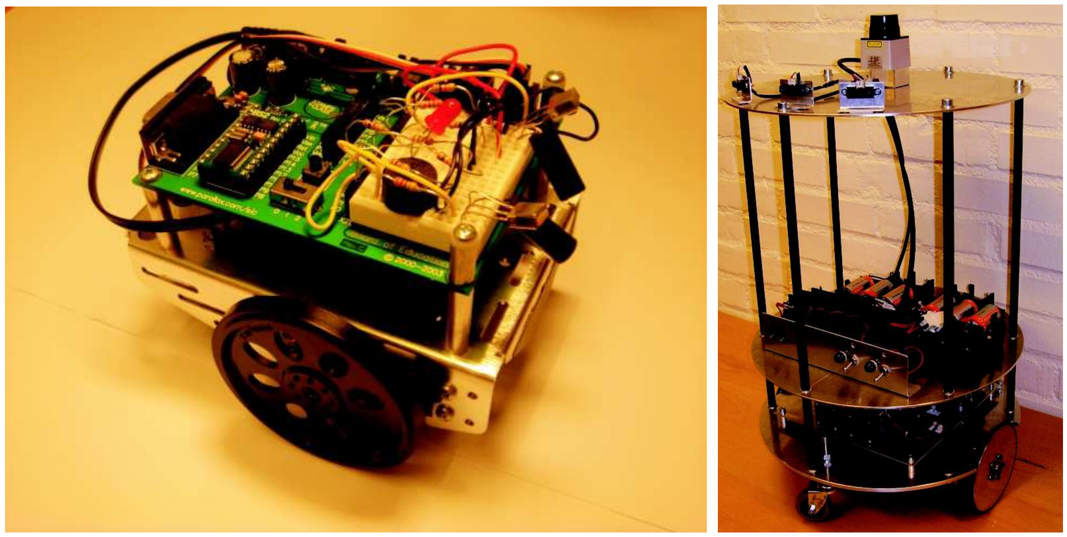
Figure 1.1: *Left panel: A Boe-bot. Right panel: A wheeled robot currently under construction in the Adaptive systems research group at Chalmers.*

various hardware-related issues will be studied in greater detail in the second half of the course, which will involve the construction of an actual robot of the kind shown in the left panel of Fig. 1.1.

1.1 Robot types

The are many different types of robots, and the taxonomy of such machines can be constructed in various ways. For example, one may classify different kinds of robots based on their complexity, their likeness to humans (or animals), their way of moving etc. In this course we shall limit ourselves to **mobile robots**, that is, robots that are able to move freely using, for example, wheels. The other main category of robots are stationary robotic arms, also referred to as **robotic manipulators**. Of course, as with any taxonomy, there are always examples that do not fit neatly into any of the available categories. For example, a **smart home** equipped with a central computer and, perhaps, some form of manipulation capabilities, can also be considered a robot, albeit of a different kind.

Robotic manipulators constitute a very important class of robots and they are used extensively in many industries, for example in assembly lines in the vehicle industry. However, such robots normally follow a predefined movement sequence and are not equipped with behaviors (such as collision avoidance) designed to avoid harming people. While there is nothing preventing the use of, for instance, sonar proximity sensors on a robotic manipulator, such options are rarely used. Instead, manipulators are confined to **robotic work cells**,

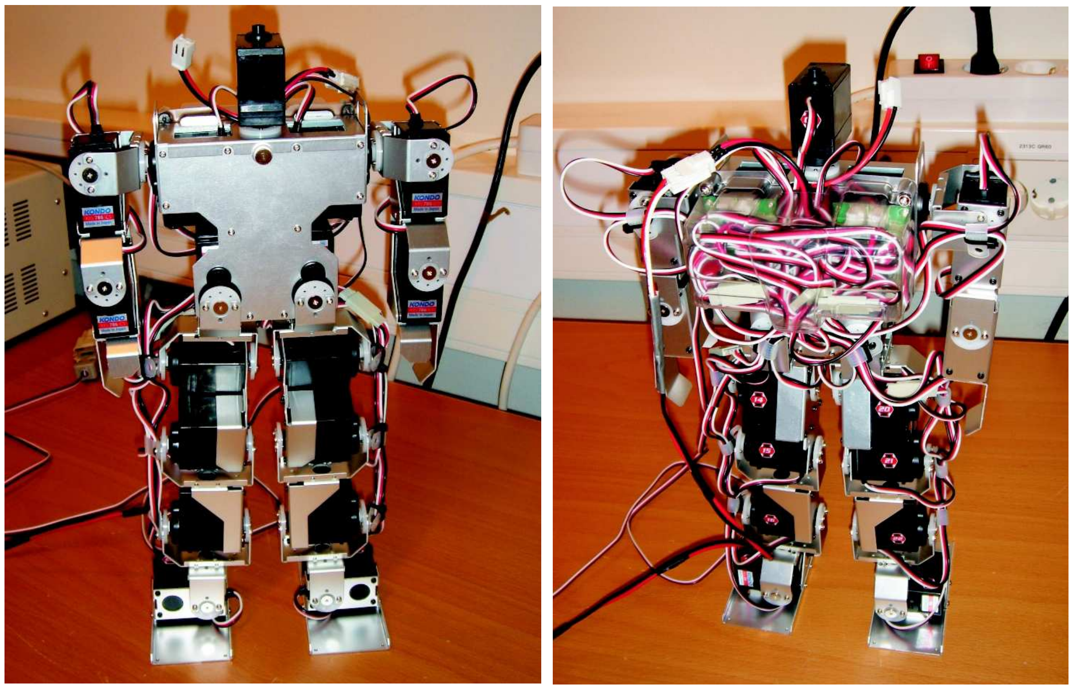

Figure 1.2: *A Kondo humanoid robot. Left panel: Front view. Right panel: Rear view.*

in which people are forbidden to enter while the manipulator is active.

By contrast, in this course, we shall consider **autonomous robots**, i.e. robots that are capable of making their own decisions (depending on the situation at hand) rather than merely executing a pre-defined sequence of motions. In fact, since most robots equipped with such decision-making capabilities are mobile, one may define an autonomous robot as a mobile robot with the ability to make decisions. Two examples of mobile robots can be seen in Fig. 1.1. The left panel shows a Boe-bot, which will be assembled and used in the second half of the course. Some of its main advantages are its small size (its length is around 0.14 m and its width 0.11 m) and its simplicity. Needless to say, the robot also has several limitations; for example, its onboard processor (microcontroller) is quite slow. However, on balance, the Boe-bot provides a good introduction to the field of mobile robots. The right panel of Fig. 1.1 shows a two-wheeled differentially steered robot which, although still under construction at the Adaptive systems research group at Chalmers, is already being used in several research projects. This robot has a diameter of 0.40 m and a height of around 1.00 m.

Robotic manipulators have long dominated the market for robots, but with the advent of low-cost mobile robots the situation is changing: In 2007, the number of mobile robots surpassed the number of manipulators for the first time, and the gap is expected to widen over the next decades.

The class of mobile robots can be further divided into subclasses, the most

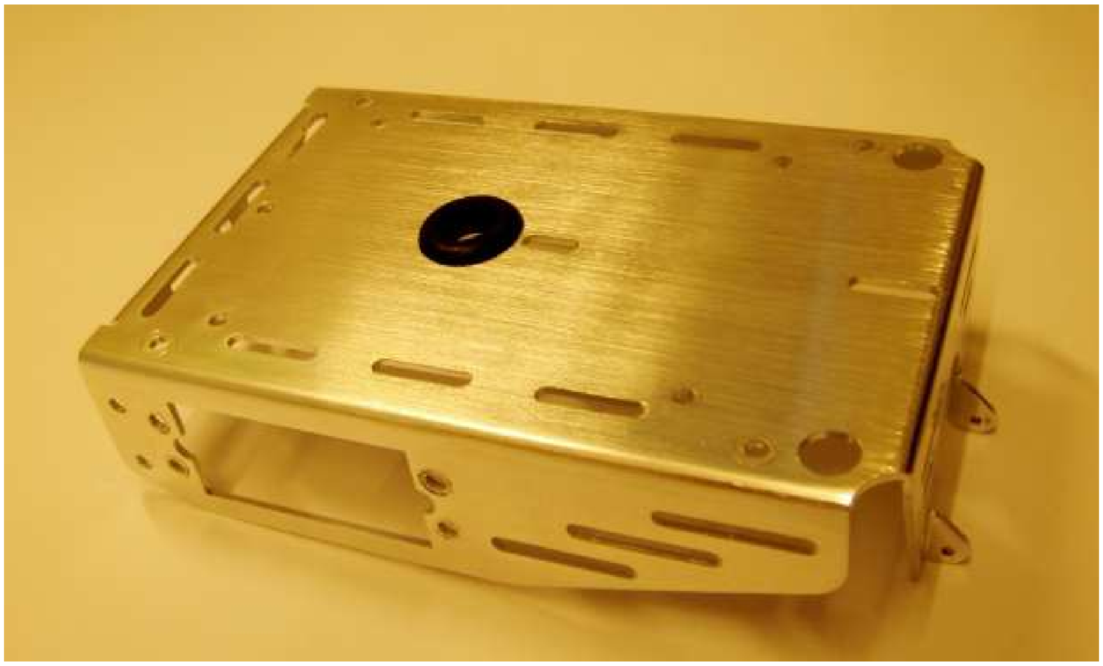

Figure 1.3: *The aluminium frame of a Boe-bot.*

important being **legged robots** and **wheeled robots**. Other kinds, such as flying robots, exist as well, but will not be considered in this course. The class of legged robots can be subdivided based on the number of legs, the most common types being **bipedal robots** (with two legs) and **quadrupedal robots** (with four legs). Most bipedal robots resemble humans, at least to some extent; such robots are referred to as **humanoid robots**. An example of a humanoid robot is shown in Fig. 1.2. Humanoid robots that (unlike the robot shown in Fig. 1.2) not only have the approximate shape of a human, but have also been equipped with more detailed human-like features, e.g. artificial skin, artificial hair etc., are called **androids**. It should be noted that the term *humanoid* refers to the shape of the robot, not its size; in fact, many humanoid robots are quite small. For example, the Kondo robot shown in Fig. 1.2 is approximately 0.35 m tall.

Some robots are *partly* humanoid. For example, the wheeled robot shown in the right panel of Fig. 1.1 is currently being equipped with a humanoid upper body. Unlike a fully humanoid robot, this robot need not be actively balanced, but will still exhibit many desirable features of humanoid robots, such as two arms for grasping and lifting objects, gesturing etc., as well as a head that will be equipped with two cameras for stereo vision and microphones providing capabilities for listening and speaking.

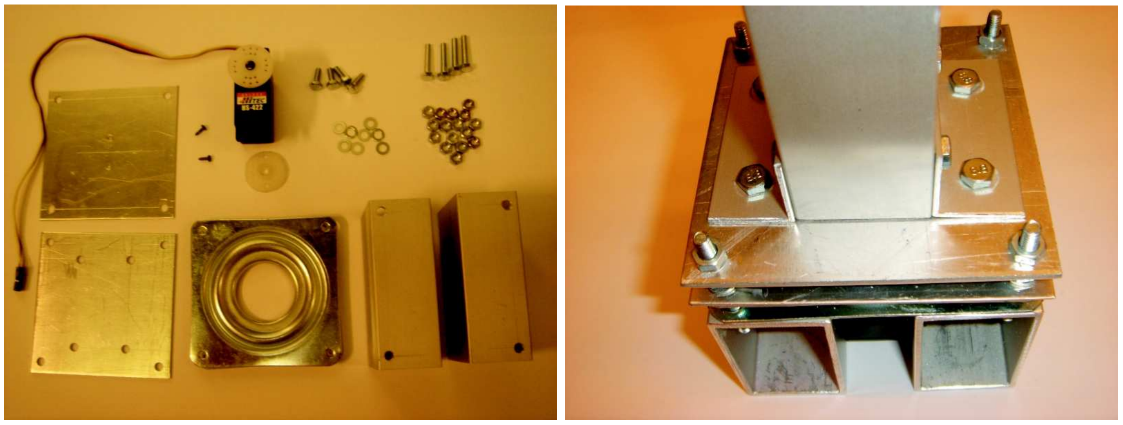

Figure 1.4: *Left panel: Aluminium parts used in the construction of a rotating base for a humanoid upper body. The servo motor used for rotating the base is also shown, as well as the screws, washers and nuts. Right panel: The assembled base.*

1.2 Robotic hardware

1.2.1 Construction material

Regarding the material used in the actual frame of the robot, several options are available, such as e.g. aluminium, steel, various forms of plastic etc. The frame of a robot should, of course, preferably be constructed using a material that is both sturdy and light and, for that reason, aluminium is often chosen. Albeit somewhat expensive, aluminium combines toughness with low weight in a near-optimal way, at least for small mobile robots. Steel is typically too heavy to be practical in a small robot, whereas many forms of plastic easily break. The frame of the robot used in this course (the Boe-bot) is made in aluminium, and is shown in Fig. 1.3. The left panel of Fig 1.4 shows the aluminium parts used in a rotating base for a humanoid upper body. The assembled base, which can rotate around the vertical axis, is shown in the right panel.

1.2.2 Sensors

The purpose of robotic sensors is to measure either some physical characteristic of the robot (for example, its acceleration) or some aspect of its environment (for example, the detected intensity of a light source). The raw data thus obtained must then, in most cases, be processed further before being used in the brain of the robot. For example, an infrared (IR) proximity sensor may provide a voltage (depending on the distance to the detected object) as its reading, which can then be converted to a distance, using the characteristics of the sensor available from its data sheet.

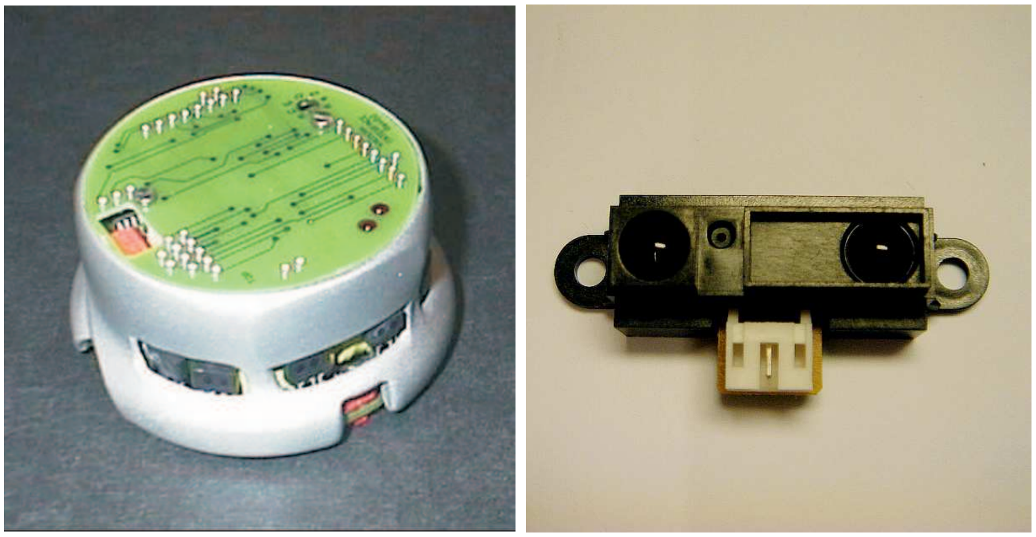

Figure 1.5: *Left panel: A Khepera II robot. Note the IR proximity sensors (small black rectangles around the periphery of the robot), consisting of an emitter and a detector. Right panel: A Sharp GP2D12 infrared sensor.*

Needless to say, there exists a great variety of sensors for mobile robots. Here, only a brief introduction will be given, focusing on a few fundamental sensor types.

Infrared proximity sensors

An **infrared proximity sensor** (or **IR sensor**, for short), consists of an emitter and a detector. The emitter, a light-emitting diode (LED), sends out infrared light, which bounces off nearby objects, and the reflected light is then measured by the detector (e.g. a phototransistor). Some IR sensors can also be used for measuring the ambient light level, i.e. the light observed by the detector when the emitter is switched off. As an example, consider the Khepera robot (manufactured by K-Team, www.k-team.com), shown in the left panel Fig. 1.5. This robot is equipped with eight IR sensors, capable of measuring both ambient and reflected light. The range of IR sensors is quite short, though. In the Khepera robot, reflected light measurements are only useful to a distance of around 0.050 m from the robot, i.e. approximately one robot diameter, even though other IR sensors have longer range. Another example is the Sharp GP2D12 IR sensor, shown in the right panel of Fig. 1.5. This sensor detects objects in the range $[0.10, 0.80]$ m. It operates using a form of triangulation: Light is emitted from the sensor and, if an object is detected, the reflected light is received at an angle that depends on the distance to the detected object. The raw signal from the sensor consists of a voltage that can be mapped to a distance. The mapping is non-linear, and for very short distances, the sensor cannot give

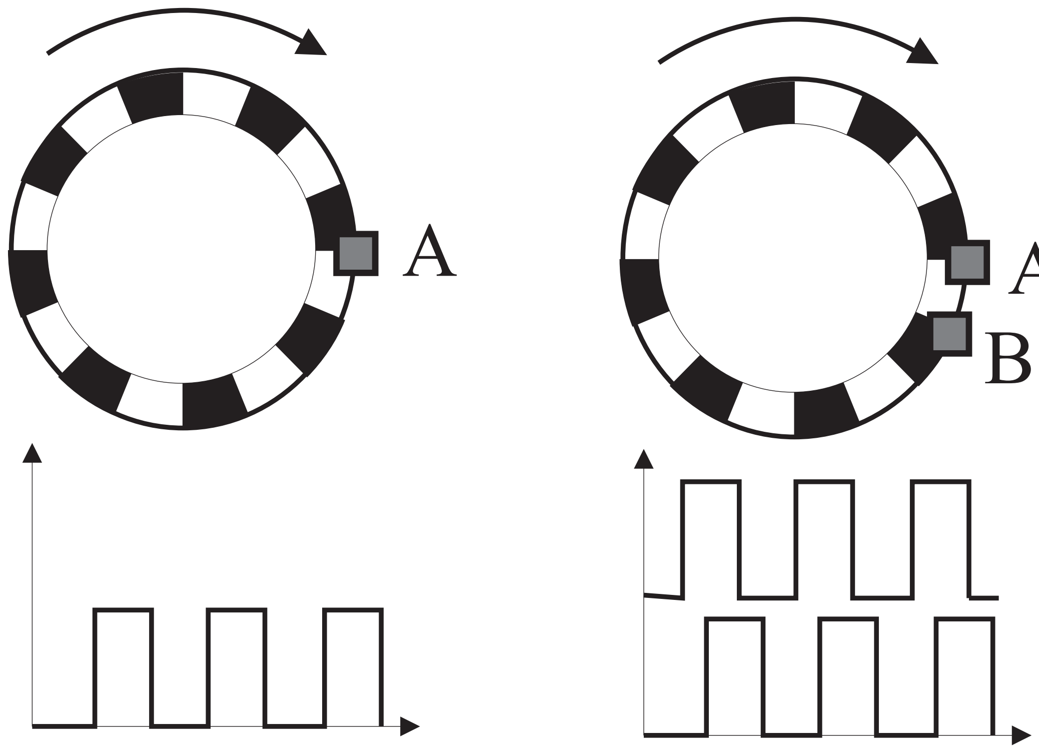

Figure 1.6: *The left panel shows a simple encoder, with a single detector (A), that measures the interruptions of a light beam, producing the curve shown below the encoder. In the right panel, two detectors are used, making it possible to determine also the* direction *of rotation.*

reliable readings (hence the lower limit of 0.10 m).

Digital optical encoders

In many applications, accurate position information is essential for a robot, and there are many different methods for positioning, e.g. inertial navigation, GPS navigation, landmark detection etc., some of which will be considered in a later chapter. One of the simplest forms of positioning, however, is **dead reckoning**, in which the position of a robot is determined based on measurements of the distance travelled by each wheel of the robot. This information, when combined with knowledge of the robot's physical properties (i.e. its kinematics, see Chapter 2) allows one to deduce the current position and heading. The process of measuring the rotation of the wheel of a robot is an example of **odometry**, and a sensor capable of such measurements is the **digital optical encoder** or, simply, **encoder**. Essentially, an encoder is a disc made of glass or plastic, with shaded regions that regularly interrupt a light beam. By counting the number of interruptions, the rotation of the wheel can be deduced, as shown in the left panel of Fig. 1.6. However, in order to determine also the *direction* of rotation, a second detector, placed at a quarter of a cycle out of phase

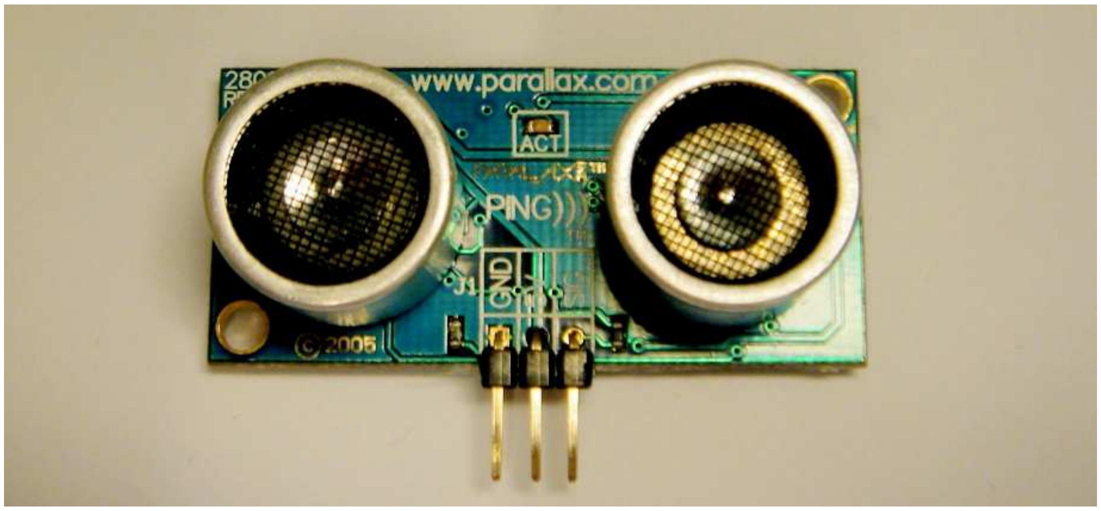

Figure 1.7: *A Ping ultrasonic distance sensor.*

with the first detector, is needed (such an arrangement is called **quadrature encoding**, and is shown in the right panel of Fig. 1.6).

Ultrasound (sonar) sensors

Ultrasound sensors, also known as **sonar sensors** or simply **sonars**, are based on time-of-flight measurement. Thus, in order to detect the distance to an object, a sonar emits a brief pulse of ultrasonic sound, typically in the frequency range 40-50 kHz[2]. The sensor then awaits the echo. Once the echo has been detected, the distance to the object can be obtained using the fact that sound travels at a speed of around 340 m/s. As in the case of IR sensors, there is both a lower and an upper limit for the detection range of a sonar sensor. If the distance to an object is too small, the sensor simply does not have enough time to switch from emission to listening, and the signal is lost. Similarly, if the distance is too large, the echo may be too weak to be detected.

Fig. 1.7 shows a Ping ultrasonic distance sensor, which is commonly used in connection with the Boe-bot. This sensor can detect distances to objects in the range $[0.02, 3.00]$ m.

Laser range finders

Laser range finders (LRFs) commonly rely, like sonar sensors, on time-of-flight measurements, but involve the speed of light rather than the speed of sound. Thus, a laser range finder emits pulses of laser light (in the form of thin beams),

[2]For comparison, a human ear can detect sounds in the range 20 hz to 20 kHz. Thus, the sound pulse emitted by a sonar sensor is not audible.

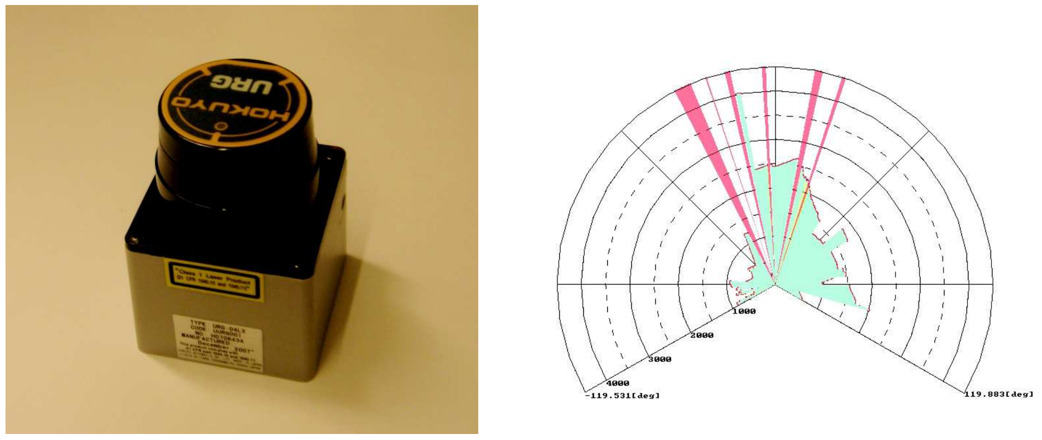

Figure 1.8: *Left panel: A Hokuyo URL-04LX laser range finder. Right panel: A typical reading, showing the distance to the nearest object in various directions. The pink rays indicate directions in which no detection is made. The maximum range of the sensor is 4 m.*

and measures the time it takes for the pulse to bounce off a target and return to the range finder. An LRF carries out a sweep over many directions[3] resulting in an accurate local map of distances to objects along the line-of-sight of each ray. LRFs are generally very accurate sensors, but they are also much more expensive than sonars sensors and IR sensors.

A Hokuyo URG-04LX LRF is shown in the left panel of Fig. 1.8. This sensor has a range of around four meters, with an accuracy of around 1 mm. It can generate readings in 683 different directions, with a frequency of around 10 Hz. As of the time of writing (Jan. 2010), a Hokuyo URG-04LX costs around 2,000 USD. The right panel of Fig. 1.8 shows a typical reading, obtained from the software delivered with the LRF.

Cameras

Cameras are used as the eyes of a robot. In many cases, two cameras are used, in order to provide the robot with binocular vision, allowing it to estimate the range to detected objects. There are many cameras available for robots, for example the CMUCam series which has been developed especially for use in mobile robots; The processor connected to the CMUCam is capable of basic image processing. At the time of writing (Jan. 2010), a CMUCam costs on the order of 150 USD. A low-cost alternative is to use ordinary webcams, for which prices start around 15 USD. Fig. 1.9 shows a simple robotic head consisting of two servo motors (see below) and a single webcam.

However, while the actual cameras may not be very costly, the use of cameras is *computationally* a very expensive procedure. Even at a low resolution,

[3]A typical angular interval for an LRF is around 180-240 degrees.

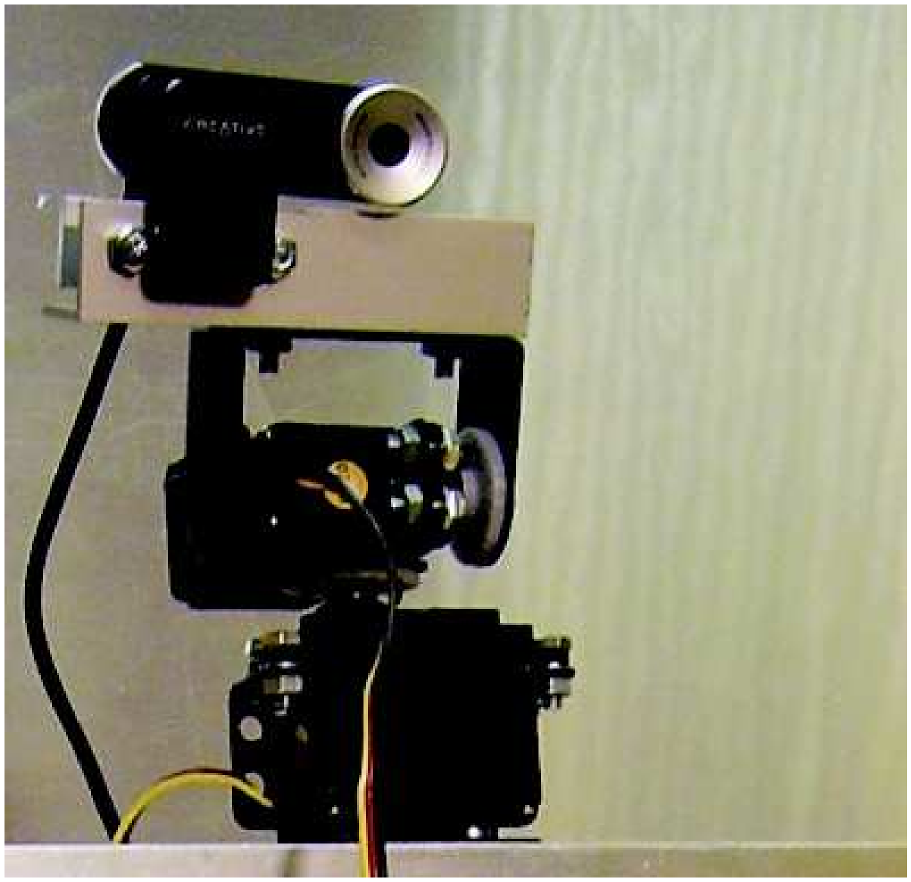

Figure 1.9: *A simple robotic head, consisting of two servo motors and a webcam.*

say 320×240 pixels, a webcam will deliver a flow of around 1.5 Mb/s, assuming a frame rate of 20 Hz and a single byte of data per pixel. The actual data transfer is easily handled by a Universal serial bus (USB), but the data must not only be transferred but also *analyzed*, something which is far from trivial. An introduction to image processing for robots will be given in a later chapter.

Other sensors

In addition to odometry based on digital optical encoders, robot positioning can be based on **inertial sensors**, i.e. sensors that measure the time derivatives of the position or heading angle of the robot. Examples of inertial sensors are **accelerometers**, measuring linear acceleration, and **gyroscopes**, measuring angular acceleration. Essentially, an accelerometer consists of a small object, with mass m, attached to a spring and damper, as shown in Fig. 1.10. As the system accelerates, the displacement z of the small object can be used to deduce the acceleration $\ddot{x}$ of the robot. Given continuous measurements of the acceleration, as a function of time, the position (relative to the starting position) can be

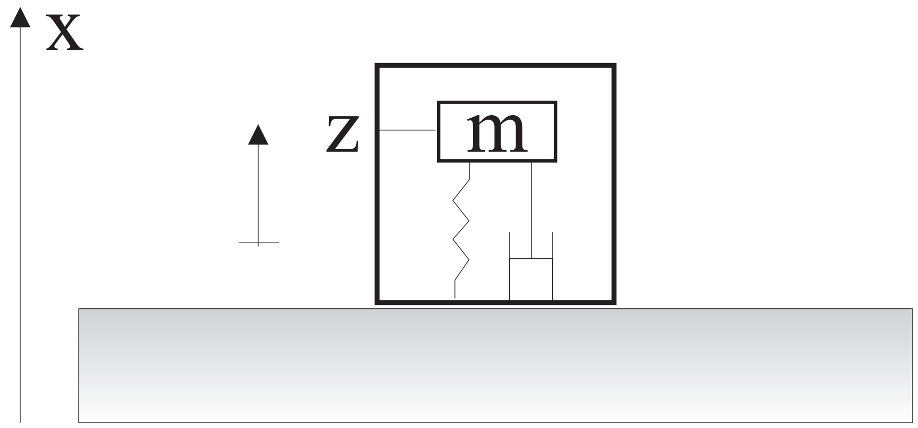

Figure 1.10: *An accelerometer. The motion of the small object (mass m) resulting from the acceleration of the larger object to which the accelerometer is attached can be used for deducing the acceleration.*

deduced. For robots operating in outdoor environments, positioning based on the **global positioning system (GPS)** is often a good alternative. The GPS relies on 24 satellites that transmit radio frequency signals which can be picked up by objects on Earth. Given the exact position of (at least) three satellites, relative to the position of e.g. a robot, the absolute position (latitude, longitude, and altitude) of the robot can be deduced.

Other sensors include **strain gauge sensors** (measuring deformation), **tactile (touch) sensors** measuring physical contact between a robot and objects in its environment, and **compasses**, measuring the direction of movement.

1.2.3 Actuators

An **actuator** is a device that allows a robot to take action, i.e. to move or manipulate the surroundings in some other way. Motors, of course, are very common types of actuators. Other kinds of actuation include, for example, the use of microphones (for human-robot interaction).

Movements can be generated in various ways, using e.g. electrical motors, pneumatic or hydraulic systems etc. In this course, we shall only consider electrical, direct-current (DC) motors and, in particular, servo motors. Thus, when referring to actuation in this course, the use of such motors is implied.

Note that actuation normally requires the use of a **motor controller** in connection with the actual motor. This is so, since the microcontroller (see below) responsible for sending commands to the motor cannot, in general, provide sufficient current to drive the motor. The issue of motor control will be considered briefly in connection with the discussion of servo motors below.

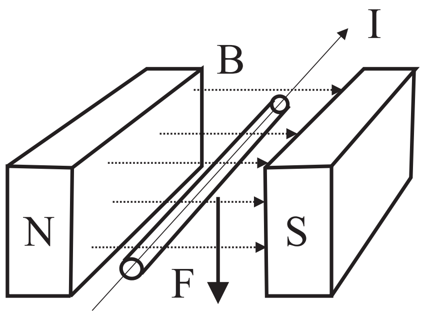

Figure 1.11: *A conducting wire in a magnetic field. B denotes the magnetic field strength and I the current through the wire. The Lorentz force* $\mathbf{F}$ *acting on the wire is given by* $\mathbf{F} = \mathbf{I} \times \mathbf{B}$.

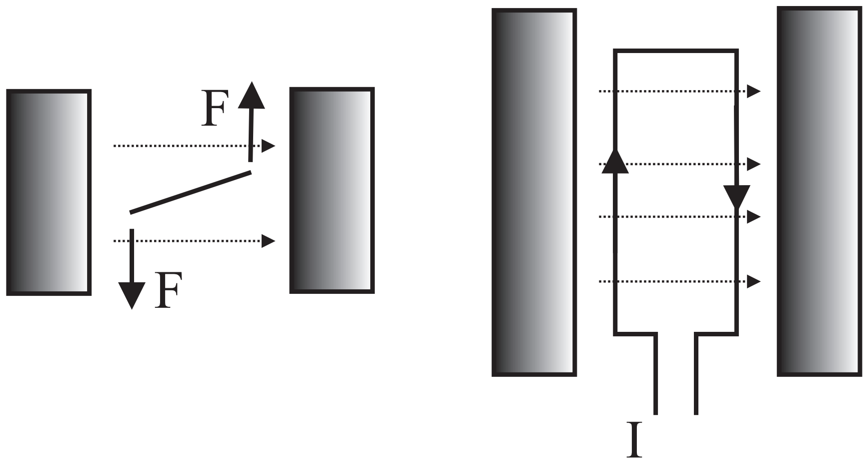

Figure 1.12: *A conducting loop of wire placed in a magnetic field. Due to the forces acting on the loop, it will begin to turn. The loop is shown from above in the right panel, and from the side in the left panel.*

DC motors

Electrical direct current (DC) motors are based on the principle that a force acts on a wire in a magnetic field if a current is passed through the wire, as illustrated in Fig. 1.11. If instead a current is passed through a closed loop of wire, as illustrated in Fig. 1.12, the forces acting on the two sides of the loop will point in opposite directions, making the loop turn. A standard DC motor consists of an outer stationary cylinder (the **stator**), providing the magnetic field, and an inner, rotating part (the **rotor**). From Fig. 1.12 it is clear that the loop will reverse its direction of rotation after a half-turn, unless the direction of the current is reversed. The role of the **commutator**, connected to the rotor of a DC motor, is to reverse the current through the motor every half-turn, thus allowing continuous rotation. Finally, carbon **brushes**, attached to the stator, complete the electric circuit of the DC motor. There are types of DC motors

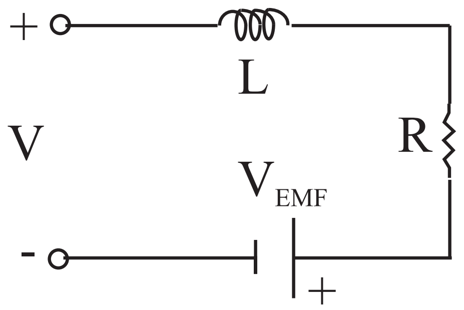

Figure 1.13: *The equivalent electrical circuit for a DC motor.*

that use electromagnets rather than a permanent magnet, and also types that are brushless. However, a detailed description of such motors are beyond the scope of this text.

DC motors are controlled by varying the applied voltage. The equations for DC motors can be divided into an electrical and a mechanical part. The motor can be modelled electrically by the equivalent circuit shown in Fig. 1.13. Letting V denote the applied voltage, and ω the angular speed of the motor shaft, the electrical equation takes the form

$$V = L\frac{\mathrm{d}i}{\mathrm{d}t} + Ri + V_{\mathrm{EMF}}, \tag{1.1}$$

where i is the current flowing through the circuit, L is the inductance of the motor, R its resistance, and V_{EMF} the voltage (the **back EMF**) counteracting V. The back EMF depends on the angular velocity, and can be written as

$$V_{\mathrm{EMF}} = c_e\omega, \tag{1.2}$$

where c_e is the **electrical constant** of the motor. For a DC motor, the generated torque τ_g is directly proportional to the current, i.e.

$$\tau_g = c_t i, \tag{1.3}$$

where c_t is the **torque constant** of the motor. Turning now to the mechanical equation, Newton's second law gives

$$I\frac{\mathrm{d}\omega}{\mathrm{d}t} = \sum\tau, \tag{1.4}$$

where I is the combined moment of inertia of the motor and its load, and $\sum\tau$ is the total torque acting on the motor. For the DC motor, the equation takes the form

$$I\frac{\mathrm{d}\omega}{\mathrm{d}t} = \tau_g - \tau_f - \tau, \tag{1.5}$$

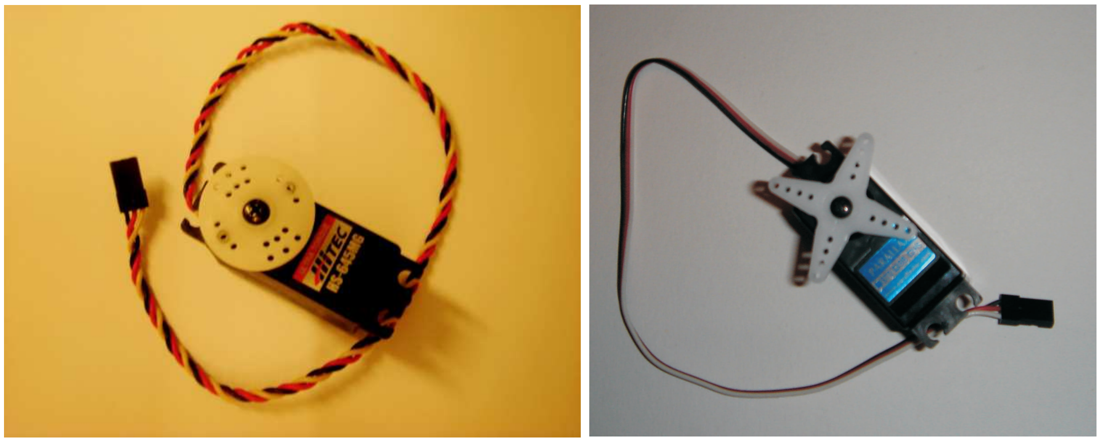

Figure 1.14: *Left panel: A HiTec 645MG servo. The suffix* MG *indicates that the servo is equipped with a metal gear train. Right panel: A Parallax servo, which has been modified for continuous rotation. Servos of this kind are used on the Boe-bot. The circular (left) and star-shaped (right) white plastic objects are the servo horns.*

where τ_f is the frictional torque opposing the motion and τ is the (output) torque acting on the load. The frictional torque can be divided into two parts, the **Coulomb friction** ($c_C\mathrm{sgn}(\omega)$) and the **viscous friction** ($c_v\omega$). Thus, the equations for the DC motor can now be summarized as

$$\tau_g = \frac{c_t}{R}V - \frac{c_t L}{R}\frac{\mathrm{d}i}{\mathrm{d}t} - \frac{c_e c_t}{R}\omega, \tag{1.6}$$

$$I\frac{\mathrm{d}\omega}{\mathrm{d}t} = \tau_g - c_C\mathrm{sgn}(\omega) - c_v\omega - \tau, \tag{1.7}$$

In many cases, the time constant of the electrical circuit is much shorter than that of the physical motion, so the inductance term can be neglected. Furthermore, for simplicity, the dynamics of the mechanical part can *also* be neglected under certain circumstances (e.g. if the moment of inertia of the motor and load is small). Thus, setting $\mathrm{d}i/\mathrm{d}t$ and $\mathrm{d}\omega/\mathrm{d}t$ to zero, the steady-state DC motor equations, determining the torque τ on the load for a given applied voltage V and a given angular velocity ω

$$\tau_g = \frac{c_t}{R}V - \frac{c_e c_t}{R}\omega, \tag{1.8}$$

$$\tau = \tau_g - c_C\mathrm{sgn}(\omega) - c_v\omega, \tag{1.9}$$

are obtained. In many cases, the axis of a DC motor rotates too fast and generates a torque that is too weak for driving a robot. Thus, a **gear box** is commonly used, which reduces the rotation speed taken out from the motor (on the **secondary drive shaft**) while, at the same time, increasing the torque. For an ideal (loss-free) gear box, the output torque and rotation speed are given by

$$\tau_{\mathrm{out}} = G\tau,$$

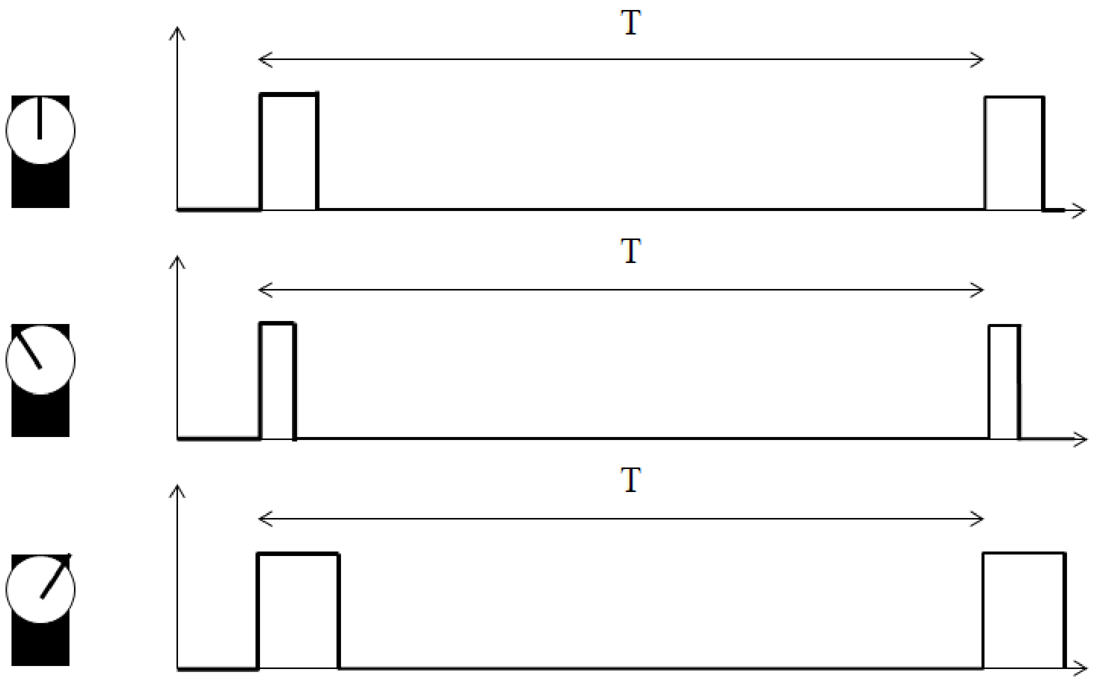

Figure 1.15: *Pulse width modulation control of a servo motor. The lengths of the pulses determine the requested position angle of the motor output shaft. The interval betwwn pulses (typically around 20 ms) is denoted T.*

$$\omega_{\text{out}} = \frac{1}{G}\omega, \tag{1.10}$$

where G is the **gear ratio**.

Servo motors

A **servo motor** is essentially a DC motor equipped with control electronics and a gear train (whose purpose is to increase the torque to the required level for moving the robot, as described above). The actual motor, the gear train, and the control electronics, are housed in a plastic container. A **servo horn** (either plastic or metal) makes it possible to connect the servo motor to a wheel or some other structure. Fig. 1.14 shows two examples of servo motors.

The angular position of a servo motor's output shaft is determined using a **potentiometer**. In a standard servo, the angle is constrained to a given range $[-\alpha_{max}, \alpha_{max}]$, and the role of the control electronics is to make sure that the servo rotates to a set position α (given by the user). A servo is fitted with a three-wire cable. One wire connects the servo to a power source (for example, a motor controller or, in some cases, a microcontroller board) and another wire connects it to *ground*. The third wire is responsible for sending signals to the servo motor. In servo motors, a technique called **pulse width modulation**

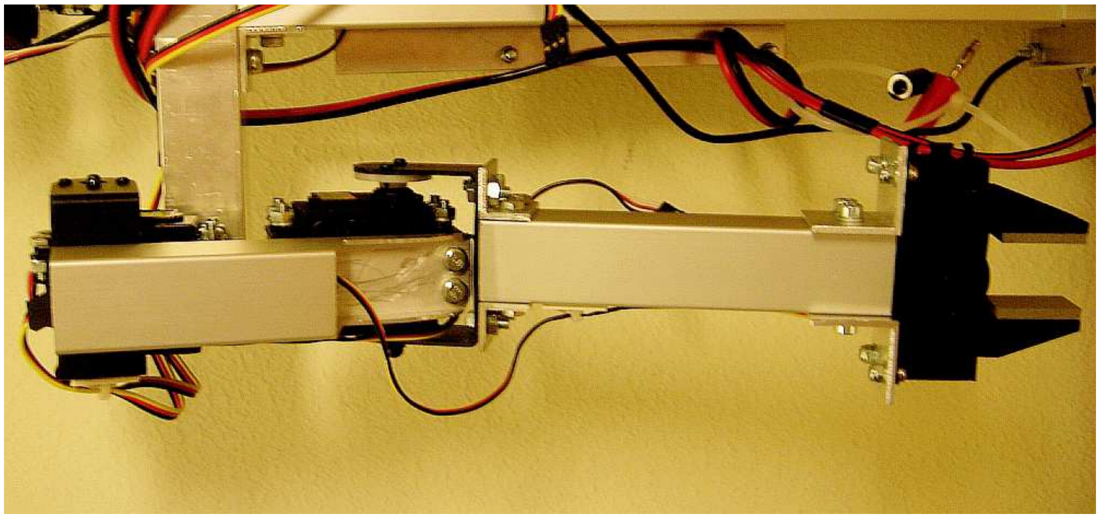

Figure 1.16: *An arm of a humanoid robot. The allowed rotation range of the elbow is around 100 degrees.*

(PWM) is used: Signals in the form of pulses are sent (e.g. from a microcontroller) to the control electronics of the servo motor. The duration of the pulses determine the required position, to which the servo will (attempt to) rotate, as shown in Fig. 1.15. For a walking robot (or for a humanoid upper body), the limitation to a given angular range poses no problem: The allowed rotation range of a servo is normally sufficient for, say, an elbow or a shoulder joint. As an example, an arm of a humanoid robot is shown in Fig. 1.16. For this particular robot, the rotation range for the elbow joint is around 100 degrees, i.e. easily within the range of a standard servo (around 180 degrees). The limitation is, of course, not very suitable for motors driving the wheels of a robot. Fortunately, servo motors can be modified to allow continuous rotation. The Boe-bot that will be built in the second half of the course uses **Parallax continuous rotation servos** (see the right panel of Fig. 1.14), rather than standard servos.

Other motors

There are many different types of motors, in addition to standard DC motors and servo motors. An example is the **stepper motor**, which is also a version of the DC motor, namely one that moves in fixed angular increments, as the name implies. However, in this course, only standard DC motors and servo motors will be considered.

1.2.4 Processors

Sensors and actuators are necessary for a robot to be able to perceive its environment and to move or manipulate the environment in various ways. How-

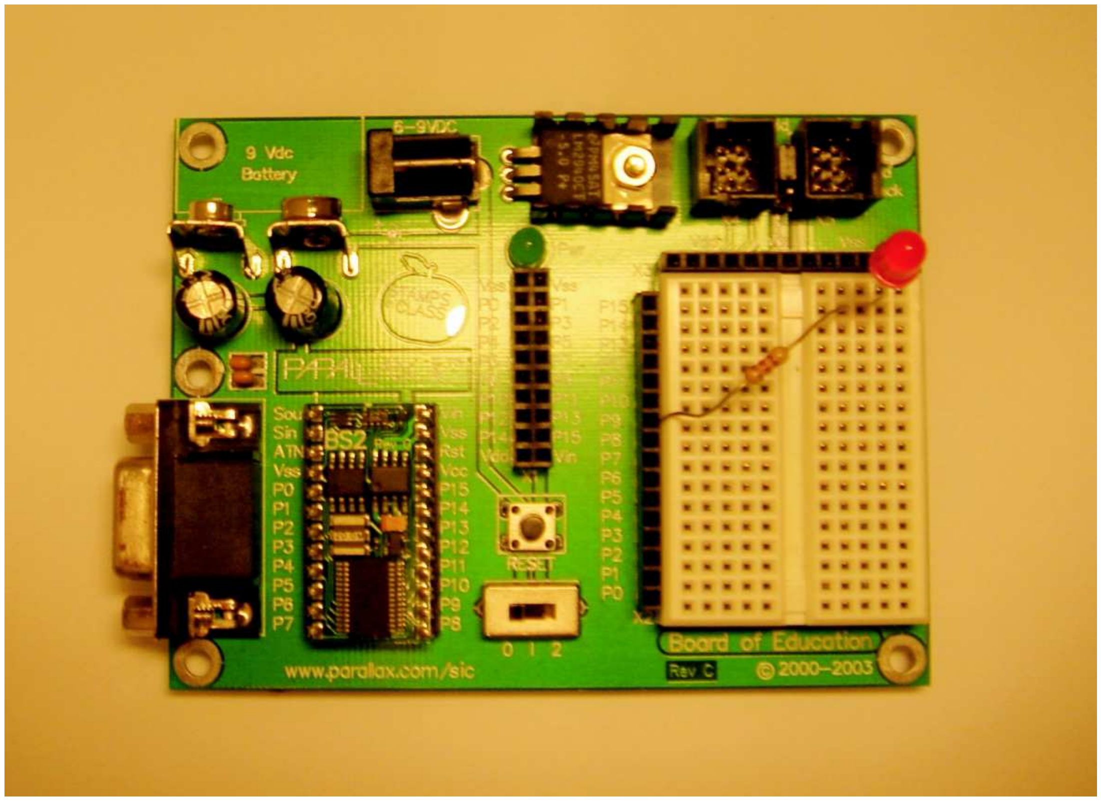

Figure 1.17: *A Board of Education (BOE) microcontroller board, with a Basic Stamp II (BS2) microcontroller attached. In addition to the microcontroller, the BOE has a serial port for communication with a PC (used, for example, when uploading a program onto the BS2), as well as sockets for attaching sensors and electronic circuits. In this case, a simple circuit involving a single LED, has been built on the BOE. The two black sockets in the upper right corner are used for connecting up to four servo motors.*

ever, in addition to sensors and actuators, there must also be a system for analyzing the sensory information, making decisions concerning what actions to take, and sending the necessary signals to the actuators.

In autonomous robots, it is common to use several processors to represent the brain of the robot. Typically, high-level tasks, such as decision-making, are carried out on a standard PC, for example a laptop computer mounted on the robot, whereas low-level tasks are carried out by microcontrollers, which will now be introduced briefly.

Microcontrollers

Essentially, a **microcontroller** is a single-chip computer, containing a **central processing unit** (CPU), read-only memory (ROM, for storing programs), random-access memory (RAM, for temporary storage, such as program variables), and several input-output (I/O) ports. There exist many different microcontrollers, with varying degrees of complexity, and different price levels, down to a few USD for the simplest ones. An example is the **Basic Stamp II**[4] (BS2) microcontroller, which costs around 50 USD.

While the BS2 is sufficient for the experimental work carried out in this course (in the next quarter), its speed is only around 4,000 operations per second (op/s) and it has a RAM memory (for program variables) of only 32 bytes and a ROM (for program storage) of 2 kilobytes (Kb).

However, many alternative microcontrollers are available for more advanced robots. Two examples, with roughly the same price as the BS2, are the BasicX and ZBasic microcontrollers, which are both compatible with the BOE microcontroller board used together with the BS2. The BasicX microcontroller has a RAM memory of 400 bytes and 32 Kb for program storage, whereas ZBasic has 4 Kb of RAM and 62 Kb for program storage. BasicX executes around 83,000 op/s, whereas (some versions of) ZBasic can reach up to 2.9 million op/s.

In many cases, microcontrollers are sold together with **microcontroller boards** (or **microcontroller modules**), containing sockets for wires connecting the microcontroller to sensors and actuators as well as control electronics, power supply etc. An example is the **Board of education** (BOE) microcontroller board. The BOE, shown in Fig. 1.17, is equipped with a **solderless breadboard**, on which electronic circuits can be built without any soldering, which is very useful for prototyping.

Since microcontrollers do not have human-friendly interfaces such as a keyboard and a screen, the normal operating procedure is to write and compile programs on an ordinary computer (using, of course, a compiler adapted for the microcontroller in question), and then upload the programs onto the microcontroller. In the case of the BS2 microcontroller, the language is a version of Basic called **PBasic**.

Robotic brain architectures

An autonomous robot must be capable of both high-level and low-level processing. The low-level processing consists, for example, of sending signals to motor controllers (see below) which, in turn, send (for example) PWM pulses to servo motors. Another low-level task is to retrieve raw data (e.g. a voltage value from an IR proximity sensor). The distinction between low-level and high-level tasks is a bit fuzzy. For example, the voltage value from an IR sensor (e.g. the Sharp GP2D12 mentioned above) can be mapped to a distance value, which of course normally is more relevant for decision-making than the raw voltage value. The actual conversion would normally be considered a low-level task but might as well also be carried out on the robot's onboard PC.

[4]Basic Stamp is a registered trademark of Parallax, inc., see www.parallax.com.

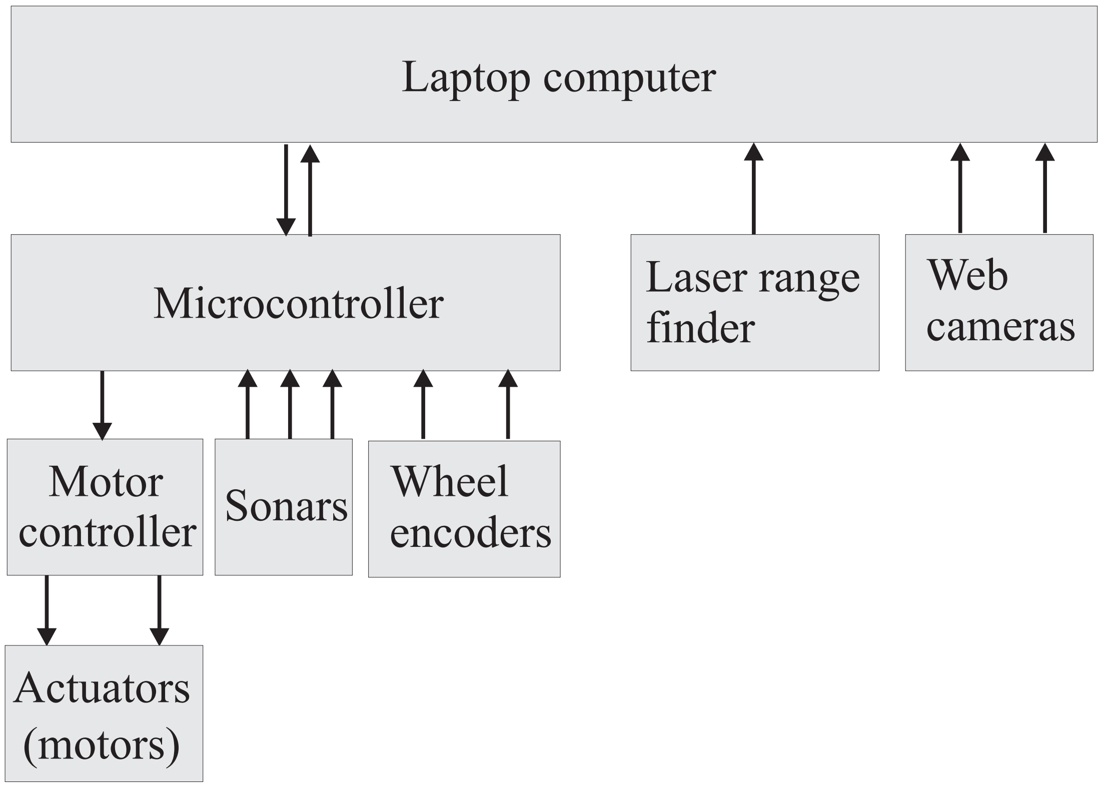

Figure 1.18: *An example of a typical robotic brain architecture, for a differentially steered two-wheeled robot equipped with wheel encoders, three sonar sensors, one LRF, and two web cameras.*

The hardware configuration providing a robot's processing capability is referred to as the **robotic brain architecture**. An example of a typical robotic brain architecture is shown in Fig. 1.18. The robotic brain shown in the figure would be used in connection with a two-wheeled differentially steered robot. As can be seen in the figure, the microcontroller would handle low-level processing, such as measuring the pulse counts of the wheel encoders, collecting readings from the three sonars, and sending motor signals (e.g. desired set speeds) to the **motor controller**[5], which, in turn, would send signals to the motors. However, the LRF and the web cameras would be directly connected, via USB (or, possibly, serial) ports, to the main processor (on the laptop), since most microcontrollers would not be able to handle the massive data flow from such sensors.

The main program (i.e. the robotic brain), running on the laptop, would process the data from the various sensors. For example, the pulse counts from

[5]A separate motor controller (equipped with its own power source) is often used for robotics applications, since the power source for the microcontroller may not be able to deliver sufficient current for driving the motors as well.

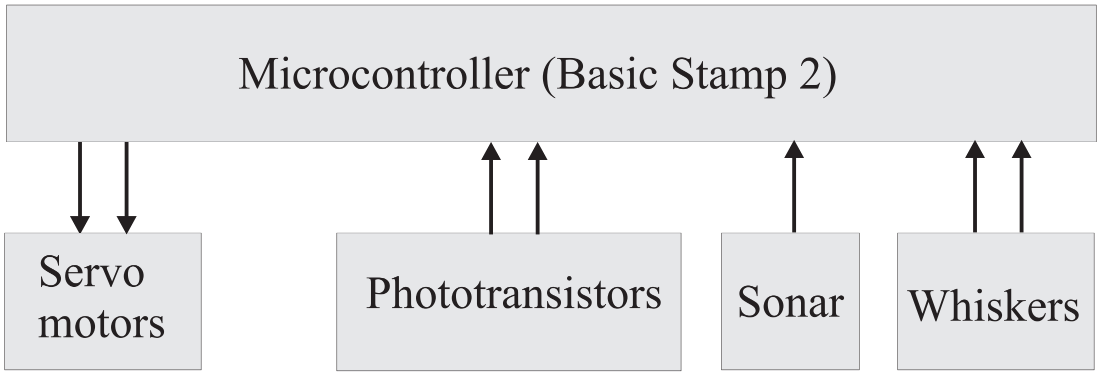

Figure 1.19: *An example of a robotic brain architecture for a Boe-bot.*

the wheel encoders would be translated to an estimate of position and heading, as described in Chapter 2. Given the processed sensory data, as well as information stored in the (long-term or short-term) memory of the robotic brain (for example, a map of the arena in which the robot operates), the main program would determine the next action to be carried out by the robot, compute the appropiate motor commands and send them to the microcontroller.

Note that the figure only shows an *example*: Many other configurations could be used as well. For example, there are cameras developed specifically for robotics applications that, unlike standard web cameras, are able to carry out much of the relevant image processing (e.g. detecting and extracting faces), and then only sending *that* information (rather than the raw pixel values) to the laptop computer.

The robotic brain architecture shown in Fig. 1.18 would be appropriate for a rather complex (and costly!) robot. Such robots are beyond the scope of the experimental work carried out in the second half of this course. The experimental work, which will be carried out using a Boe-bot (see the left panel of Fig. 1.1), involves a much simpler robotic brain architecture, illustrated in Fig. 1.19. As can be seen, in this case, the robot has a single processor, namely the BS2 microcontroller, which thus is responsible both for the low-level (signal) processing and the high-level decision-making.

The microcontroller sends signals to the two servo motors and receives input from the sensors attached to the robot, for example, two photo-resistors, a sonar sensor, and whiskers. The whiskers are simple touch sensors that give a reading of either 0 (if no object is touched) or 1 (if the whisker touches an object). Of course, other sensors (such as IR sensors or simple wheel encoders) can be added as well, but one should keep in mind that the processing capability of the BS2 is very limited. Note that no motor controller is used: The BOE is capable of generating sufficient current for up to four Parallax servo motors.

Chapter 2

Kinematics and dynamics

2.1 Kinematics

Kinematics is the process of determining the range of possible movements for a robot, without consideration of the forces acting on the robot, but taking into account the various constraints on the motion. The kinematic equations for a robot depend on the robot's structure, i.e. the number of wheels, the type of wheels used etc. Here, only the case of differentially steered two-wheeled robots will be considered. For balance, a two-wheeled robot must also have one or several supporting wheels (or some other form of ground contact, such as a ball in a material with low friction). The influence of the supporting wheels on the kinematics and dynamics will not be considered.

2.1.1 The differential drive

A schematic view of a differentially steered robot is shown in Fig. 2.1. The Boe-bot that will be considered in the second half of the course (see the left panel

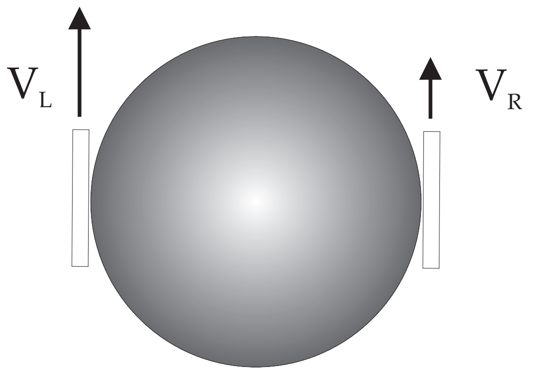

Figure 2.1: *A schematic representation of a two-wheeled, differentially steered robot.*

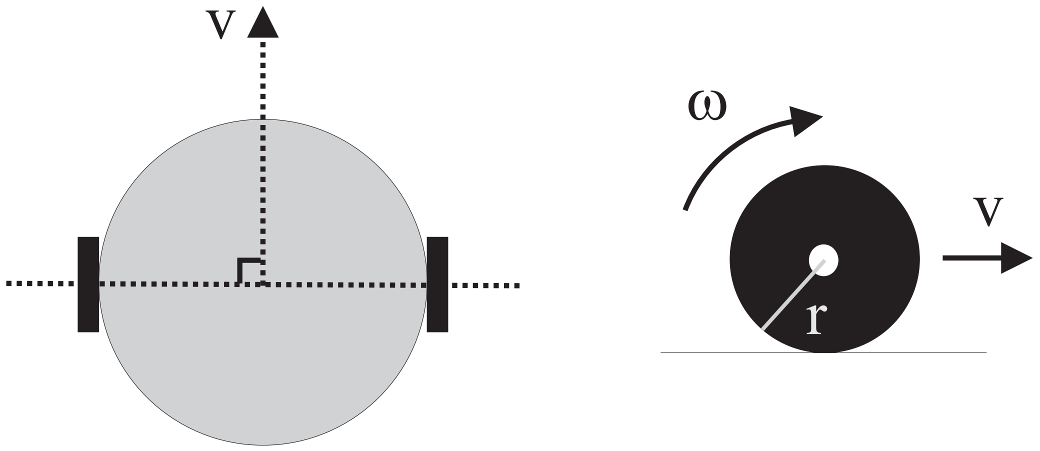

Figure 2.2: *Left panel: Kinematical constraints force a differentially steered robot to move in a direction perpendicular to a line through the wheel axes. Right panel: For a wheel that rolls without slipping, the equation $v = \omega r$ holds.*

of Fig. 1.1) is an example of such a robot.

A differentially steered robot is equipped with two independently steered wheels. The position of the robot is given by the two coordinates x and y, and its direction of motion is denoted φ.

It will be assumed that the wheels are only capable of moving in the direction perpendicular to the wheel axis (see the left panel of Fig. 2.2). Furthermore, it will be assumed that the wheels roll without slipping, as illustrated in the right panel of Fig. 2.2. For such motion, the forward speed v of the wheel is related to the angular velocity ω through the equation

$$v = \omega r, \tag{2.1}$$

where r is the radius of the wheel.

The **forward kinematics** of the robot, i.e. the variation of x, y and φ, given the speeds v_L and v_R of the left and right wheel, respectively, can be obtained by using the constraints on the motion imposed by the fact that the frame of the robot is a rigid body. For any values of the wheel speeds, the motion of the robot can be seen as a pure rotation, with angular velocity $\omega = \dot{\varphi}$ around the **instantaneous center of rotation** (ICR). Letting L denote the distance from the ICR to the center of the robot, the speeds of the left and right wheels can be written

$$v_L = \omega(L - R), \tag{2.2}$$

and

$$v_R = \omega(L + R), \tag{2.3}$$

where R is the radius of the robot's body (which is assumed to be circular and with a circularly symmetric mass distribution). The speed V of the center-of-mass of the robot is given by

$$V = \omega L. \tag{2.4}$$

Inserting Eq. (2.4) into Eqs. (2.2) and (2.3), L can be eliminated. V and ω can then be obtained in terms of v_L and v_R as

$$\begin{aligned} V &= \frac{v_L + v_R}{2}, & (2.5) \\ \omega &= -\frac{v_L - v_R}{2R}. & (2.6) \end{aligned}$$

Denoting the speed components of the robot V_x and V_y, and noting that $V_x = V\cos\varphi$, $V_y = V\sin\varphi$, the position of the robot at time t_1 is given by

$$\begin{aligned} X(t_1) - X_0 &= \int_{t_0}^{t_1} V_x(t)\mathrm{d}t = \int_{t_0}^{t_1} \frac{v_L(t) + v_R(t)}{2}\cos\varphi(t)\mathrm{d}t, & (2.7) \\ Y(t_1) - Y_0 &= \int_{t_0}^{t_1} V_y(t)\mathrm{d}t = \int_{t_0}^{t_1} \frac{v_L(t) + v_R(t)}{2}\sin\varphi(t)\mathrm{d}t, & (2.8) \\ \varphi(t_1) - \varphi_0 &= \int_{t_0}^{t_1} \omega(t)\mathrm{d}t = -\int_{t_0}^{t_1} \frac{v_L(t) - v_R(t)}{2R}\mathrm{d}t, & (2.9) \end{aligned}$$

where (X_0, Y_0) is the starting position of the robot (at time $t = t_0$), and φ_0 is its initial direction of motion. The position and heading together form the **pose** of the robot. Thus, if $v_L(t)$ and $v_R(t)$ are known, the position and orientation of the robot can be determined for any time t. Numerical integration is normally required, since the equations for X and Y can only be integrated analytically if φ has a rather simple form. Two special cases, for which the three equations can all be integrated analytically, are (check!) $(v_L(t), v_R(t)) = (v_1, v_2)$ and $(v_L(t), v_R(t)) = (v_0(t/t_1), v_0(t/t_2))$, where v_1, v_2, v_0, t_1 and t_2 are constants. In these cases, one can first find $\varphi(t)$ (for arbitrary t), and then obtain $X(t)$ and $Y(t)$.

Of course, in a real robot, the wheel speeds can never be determined with perfect accuracy. Instead, the integration must be based on *estimates* $\hat{v}_L(t)$ and $\hat{v}_R(t)$, which, in turn, are computed based on the pulse counts of the wheel encoders. There are many factors limiting the accuracy of the speed estimates. One such limitation concerns the number of pulses per revolution: For example, the wheel encoders supplied by Parallax (for the Boe-bot) use the eight holes in the robot's wheel for generating pulse counts, so that a complete revolution of a wheel corresponds to only eight pulses. Evidently, a speed estimate (which requires two different pulse readings, at different times, as well as an estimate of the time elapsed between the two readings) for such a robot would not be very accurate. By contrast, in more advanced robots, the encoders may be mounted *before* the gear box (in the case of a DC motor), and may also provide much more than eight pulses per revolution (of the motor shaft), so that a rather accurate wheel speed estimate can be obtained.

However, even if the speed estimates are very accurate, there are other sources of error as well. For example, the robot's wheels may slip occasionally. Furthermore, the kinematic model may not provide a completely accurate estimate of the robot's true kinematics (for example, no wheel is ever *perfectly* circular).

Once wheel speed estimates are available, the pose can be estimated, using a kinematic model as described above. The process of estimating a robot's position and heading based on wheel encoder data is called **odometry**. Due to the limited accuracy of velocity estimates, the estimated pose of the robot will be subject to an error, which grows with time. Thus, in order to maintain a sufficiently accurate pose estimate for navigation over long distances, an independent method of **odometric recalibration** must be used. This issue will be considered in a later chapter.

Normally, the wheel speeds are not given *a priori*. Instead, the signals sent to the motors by the robotic brain (perhaps in response to external events, such as detection of obstacles) will determine the torques applied to the motor axes. In order to determine the motion of the robot one must then consider not only its kinematics but also its **dynamcs**. This will be the topic of the next section.

2.2 Dynamics

The kinematics considered in the previous section determines the range of possible motions for a robot, given the constraints which, in the case of the two-wheeled differential robot, enforce motion in the direction perpendicular to the wheel axes. However, kinematics says nothing about the way in which a particular motion is achieved. **Dynamics**, by contrast, considers the motion of the robot in response to the forces (and torques) acting on it. In the case of the two-wheeled, differentially steered robot, the two motors generate torques (as described above) that propel the wheels forward, as shown in Fig. 2.3. The frictional force at the contact point with the ground will try to move the ground backwards. By Newton's third law, a reaction force of the same magnitude will attempt to move the wheel forward. In addition to the torque τ from the motor (assumed to be known) and the reaction force F from the ground, a reaction force ρ from the main body of the robot will act on the wheel, mediated by the wheel axis (the length of which is neglected in this derivation). Using Newton's second law, the equations of motion for the wheels take the form

$$m\dot{v}_L = F_L - \rho_L, \tag{2.10}$$

$$m\dot{v}_R = F_R - \rho_R, \tag{2.11}$$

$$\overline{I}_{\mathrm{w}}\ddot{\phi}_L = \tau_L - F_L r, \tag{2.12}$$

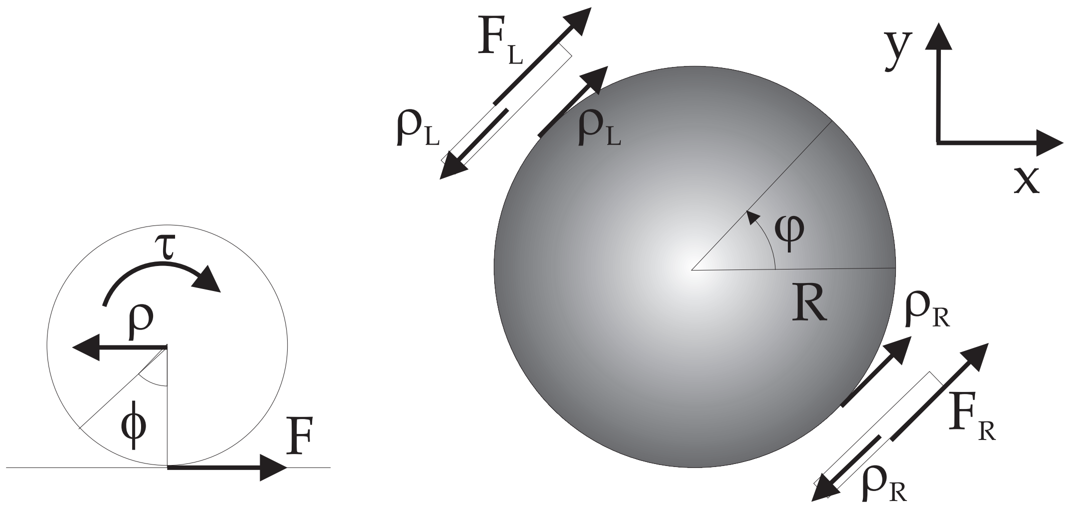

Figure 2.3: *Left panel: Free-body diagram showing one of the two wheels of the robot. Right panel: Free-body diagram for the body of the robot and for the two wheels. Only the forces acting in the horizontal plane are shown.*

and

$$\overline{I}_{\mathrm{w}}\ddot{\phi}_R = \tau_R - F_R r, \tag{2.13}$$

where m is the mass of the wheel, $\overline{I}_{\mathrm{w}}$ is its moment of inertia, and r its radius. It is assumed that the two wheels have identical properties. The right panel of Fig. 2.3 shows free-body diagrams of the robot and the two wheels, seen from above. Newton's equations for the main body of the robot (mass M) take the form

$$M\dot{V} = \rho_L + \rho_R \tag{2.14}$$

and

$$\overline{I}\ddot{\varphi} = \left(-\rho_L + \rho_R\right) R, \tag{2.15}$$

where $\overline{I}$ is its moment of inertia.

In the six equations above there are 10 unknown variables, namely v_L, v_R, F_L, F_R, ρ_L, ρ_R, ϕ_L, ϕ_R, V, and φ. Four additional equations can be obtained from kinematical considerations. As noted above, the requirement that the wheels should roll without slipping leads to the equations

$$v_L = r\dot{\phi}_L \tag{2.16}$$

and

$$v_R = r\dot{\phi}_R. \tag{2.17}$$

Furthermore, the two kinematic equations (see Sect. 2.1)

$$V = \frac{v_L + v_R}{2}, \tag{2.18}$$

and

$$\dot{\varphi} = -\frac{v_L - v_R}{2R}. \tag{2.19}$$

complete the set of equations for the dynamics of the differentially steered robot. Combining Eq. (2.10) with Eq. (2.12) and Eq. (2.11) with Eq. (2.13), the equations

$$m\dot{v}_L = \frac{\tau_L - \overline{I}_{\text{w}}\ddot{\phi}_L}{r} - \rho_L, \tag{2.20}$$

$$m\dot{v}_R = \frac{\tau_R - \overline{I}_{\text{w}}\ddot{\phi}_R}{r} - \rho_R \tag{2.21}$$

are obtained. Inserting the kinematic conditions from Eqs. (2.16) and (2.17), ρ_L and ρ_R can be expressed as

$$\rho_L = \frac{\tau_L}{r} - \left(\frac{\overline{I}_{\text{w}}}{r^2} + m\right)\dot{v}_L, \tag{2.22}$$

and

$$\rho_R = \frac{\tau_R}{r} - \left(\frac{\overline{I}_{\text{w}}}{r^2} + m\right)\dot{v}_R. \tag{2.23}$$

Inserting Eqs. (2.22) and (2.23) in Eq. (2.14), one obtains the acceleration of the center-of-mass of the robot body as

$$\begin{aligned} M\dot{V} &= \rho_L + \rho_R = \frac{(\tau_L + \tau_R)}{r} - \left(\frac{\overline{I}_{\text{w}}}{r^2} + m\right)(\dot{v}_L + \dot{v}_R) = \\ &= \frac{(\tau_L + \tau_R)}{r} - 2\left(\frac{\overline{I}_{\text{w}}}{r^2} + m\right)\dot{V}, \end{aligned} \tag{2.24}$$

where, in the last step, the derivative with respect to time of Eq. (2.18) has been used. Rearranging terms, one can write Eq. (2.24) as

$$M\dot{V} = A\left(\tau_L + \tau_R\right), \tag{2.25}$$

where

$$A = \frac{1}{r\left(1 + 2\left(\frac{\overline{I}_{\text{w}}}{Mr^2} + \frac{m}{M}\right)\right)}. \tag{2.26}$$

For the angular motion, using Eqs. (2.22) and (2.23), Eq. (2.15) can be expressed as

$$\overline{I}\ddot{\varphi} = \left(-\rho_L + \rho_R\right)R = \left(-\tau_L + \tau_R\right)\frac{R}{r} + R\left(\frac{\overline{I}_{\text{w}}}{r^2} + m\right)\left(\dot{v}_L - \dot{v}_R\right), \tag{2.27}$$

Differentiating Eq. (2.19) with respect to time, and inserting the resulting expression for $\dot{v}_R - \dot{v}_L$ in Eq. (2.27), one obtains the equation for the angular motion as

$$\overline{I}\ddot{\varphi} = \left(-\tau_L + \tau_R\right)\frac{R}{r} - 2R^2\left(\frac{\overline{I}_{\text{w}}}{r^2} + m\right)\ddot{\varphi}. \tag{2.28}$$

Rearranging terms, this equation can be expressed in the form

$$\overline{I}\ddot{\varphi} = B\left(-\tau_L + \tau_R\right), \tag{2.29}$$

where

$$B = \frac{1}{\frac{r}{R} + 2\left(\frac{\overline{I}_{\mathrm{w}}R}{\overline{I}r} + \frac{mRr}{\overline{I}}\right)}. \tag{2.30}$$

Due to the limited strength of the motors and to friction, as well as other losses (for instance in the transmission), there are of course limits on the speed and rotational velocity of the robot. Thus, the differential equations for V and φ should also include damping terms. In practice, for any given robot, the exact form of these terms must be determined through experiments (i.e. through **system identification**). A simple approximation is to use linear damping terms, so that the equations of motion for the robot become

$$M\dot{V} + \alpha V = A\left(\tau_{\mathrm{L}} + \tau_{\mathrm{R}}\right), \tag{2.31}$$

and

$$\overline{I}\ddot{\varphi} + \beta\dot{\varphi} = B\left(-\tau_{\mathrm{L}} + \tau_{\mathrm{R}}\right), \tag{2.32}$$

where α and β are constants. Note that, if the mass m and moment of inertia $\overline{I}_w$ of the wheels are small compared to the mass M and moment of inertia I of the robot, respectively, the expression for A can be simplified to

$$A = \frac{1}{r}. \tag{2.33}$$

Similarly, the expression for B can be simplified to

$$B = \frac{R}{r}. \tag{2.34}$$

Given the torques τ_L and τ_R generated by the two motors in response to the signals sent from the robotic brain, the motion of the robot can thus be obtained by integration of Eqs. (2.31) and (2.32).

Chapter 3

Simulation of autonomous robots

Simulations play an important role in research on (and development of) autonomous robots, for several reasons. First of all, testing a robot in a simulated environment can make it possible to detect whether or not the robot is prone to catastrophic failure in certain situations, so that the behavior of the robot can be altered before it is unleashed in the real world. Second, building a robot is often costly (for example, most laser range finders cost several thousand USD). Thus, through simulations, it is possible to test several designs before constructing an actual robot. Furthermore, it is common to use stochastic optimization methods, such as evolutionary algorithms, in connection with the development of autonomous robots. Such methods require that many different robotic brains be evaluated, which is *very* time-consuming if the work must be carried out in an actual robot. Thus, in such cases, simulations are often used, even though the resulting robotic brains must, of course, be thoroughly tested in real robots, a procedure which often requires several iterations involving simulated and actual robots. In this chapter, an introduction to some of the general issues pertaining to robotic simulations will be given, along with a brief description of (some of) the features of two particular simulators for mobile robots, namely GPRSim and ARSim. GPRSim is an advanced 3D simulator for automomous robots, which is used in certain research projects within the Adaptive systems group. ARSim is a simplified (2D) Matlab simulator used in this course.

3.1 Simulators

Over the years, several different simulators for mobile robots have appeared, with varying degrees of complexity. One of the most ambitious simulators to date is **Robotics studio** from Microsoft, which allows the user to simulate many of the commercially available mobile robots, or even to assemble a (virtual) robot using generic parts.

Some simulators include not only general simulation of the kinematic and dynamics of robots, but also procedures for stochastic optimization. Some examples of such simulators are `Webots`, which is manufactured by Cyberbotics (`www.cyberbotics.com`) and the open source package `Darwin2K`, which can be found at `darwin2k.sourceforge.net`.

The Adaptive systems research group at Chalmers has developed a simulator called the **General-purpose robotic simulator** (GPRSim), which is extensively used in our research projects. Unlike the other simulators mentioned above, GPRSim features, as an integral part of the simulator, an implementation of the general-purpose robotic brain structure (GPRBS) (also developed in the Adaptive systems research group). The GPRBS, in turn, consists of a standardized representation of a robotic brain, consisting of a set of so called brain processes as well as a decision-making system. This structure allows researchers to build complex robotic brains involving many different behavioral aspects and also to export the resulting robotic brain for use in real (physical) robots. The existence of a standardized representation for robotic brains also makes it possible, for example, to reuse parts of a previously developed robotic brain in other applications than the original one.

However, GPRSim is primarily a research tool and, as such, it is not very user-friendly. Moreover, the underlying code is quite complex. Thus, in this course, a different simulator will be used, namely the **Autonomous robot simulator** (ARSim), which is a 2D simulator written in Matlab. This simulator is generally too slow to be useful in research projects, but it is perfectly suited to most of the tasks considered in this course. Note also that, even though ARSim is greatly simplified, many parts of the code (for example the simulation of DC motors, IR sensors etc.) are essentially the same in GPRSim and ARSim

3.2 General simulation issues

In Fig. 3.1, the general flow of a single-robot simulation is shown. Basically, after initialization, the simulation proceeds in a stepwise fashion. In each step, the simulator reads the sensors of the robot, and the resulting signals are sent to the robotic brain, which computes appropriate motor signals that, finally, are sent to the motors. Given the motor signals, the acceleration of the robot can be updated, and new velocities and positions can be computed. Changes to the arena (if any) are then made, and the termination criteria are checked.

3.2.1 Timing of events

As mentioned earlier, simulation results in robotics must be validated in an actual robot. However, in order for this to be possible, some care must be

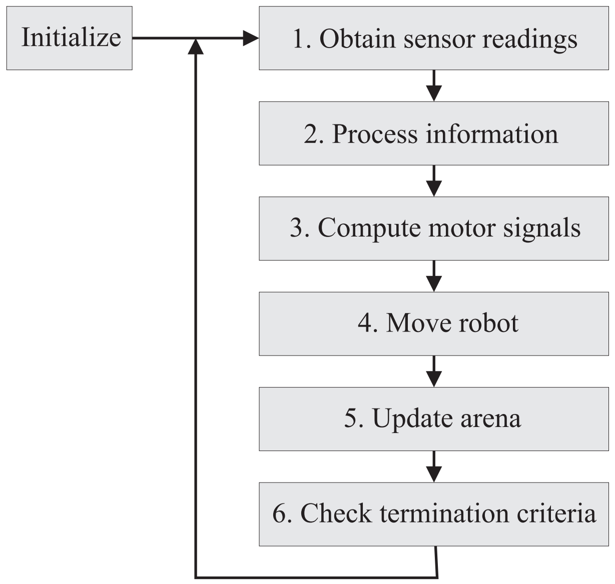

Figure 3.1: *The flow of a single-robot simulation. Steps 1 through 6 are carried out in each time step of the simulation.*

taken, particularly regarding steps 1-3. To be specific, one must make sure that these steps can be executed (on the real robot) in a time which does not exceed the time step length in the simulation. Here, it is important to distinguish between two different types of events, namely (1) those events that take a long time to complete in simulation, but would take a very short time in a real robot, and (2) those events that are carried out rapidly in simulation, but would take a long time to complete in a real robot.

An example of an event of type (1) is collision-checking. If performed in a straight-forward, brute-force way, the possibility of a collision between the (circular, say) body of the robot and an object must be checked by going through *all* lines in a 2D-projection of the arena. A better way (used, for example, in GPRSim) is to introduce an invisible grid, and only check for collisions between the robot and those objects that (partially) cover the grid cells that are also covered by the robot. However, even when such a procedure is used, collision-checking may nevertheless be very time-consuming in simulation whereas, in a real robot, it amounts simply to reading a bumper sensor (or, as on the Boe-bot, a whisker), and transferring the signal (which, in this case, is binary, i.e. a single bit of information) from the sensor to the brain of the robot. Events of this type cause no (timing) problems at the stage of transferring the results to a real robot, even though they may slow down the simulation considerably.

An example of an event of type (2) is the reading of sensors. For example, an IR sensor can be modelled using simple ray-tracing (see below) and, provided that the number of rays used is not too large, the update can be car-

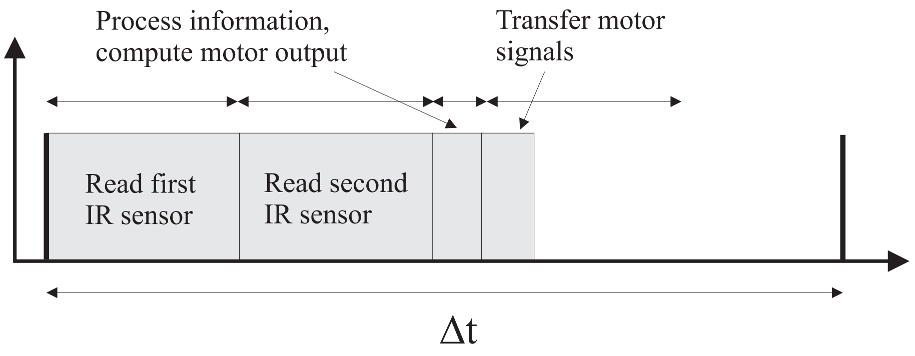

Figure 3.2: *A timing diagram. The boxes indicate the time required to complete the corresponding event in hardware, i.e. a* real *robot. In order for the simulation to be realistic, the time step Δt used in the simulation must be longer than the total duration (in hardware) of all events taking place within a time step.*

ried out in a matter of microseconds in a simulator. However, in a real robot it might take longer time. While the reading of an IR sensor involves a very limited signal flow compared to the reading of a camera with, say, 640×480 pixels, the *transfer* of the reading from the sensor to the robotic brain is a potential bottleneck. A common setup is to have a microcontroller (see Chapter 1) handling the low-level communication, i.e. obtaining sensor readings and sending signals to actuators, and a PC (for example, a laptop placed on the robot) handling high-level issues, such as decision-making, motion planning etc. Very often, the communication between the laptop and the microcontroller takes place through a serial port, operating with a speed of, say, 9600 or 38400 bits/s. If the onboard PC must read, for example, four proximity sensors (assuming one byte per reading) and send signals to two motors (again assuming that each signal requires one byte), a total of $6 \times 8 = 48$ bits is needed, limiting the number of interactions between the PC and the microcontroller to $9600/48 = 200$ per second in the case of a serial port speed of 9600 bits/s. As another, more specific, example, consider the small mobile robot Khepera, shown in the left panel of Fig. 1.5. In its standard configuration, it is equipped with eight IR sensors, which are read in a sequential way every 2.5 ms, so that the processor of the robot receives an update of a given IR sensor's reading every 20 ms. The updating frequency of the sensors is therefore limited to 50 Hz. Thus, a simulation of a Khepera robot in which the simulated sensors are updated with a frequency of, say, 100 Hz would be unrealistic.

In practice, the problem of limited updating frequency in sensors can be solved by introducing a Boolean **readability state** for each (simulated) sensor. Thus, in the case of a Khepera simulation with a time step of 0.01s, the sensor values would be updated only every other time step. Step 2, i.e. the processing of information by the brain of the robot, must also, in a realistic simulation, be of limited complexity so that the three steps (1, 2, and 3) *together* can be carried out within the duration Δt (the simulation time step) when transferred to the real robot. An example of a timing diagram for a generic robot (not Khepera) is shown in Fig. 3.2. In the case shown in the figure, two IR proximity sensors are read, the information is processed (for example, by being passed through an artificial neural network), and the motor signals (voltages, in the case of standard DC motors) are then transferred to the motors. The figure shows a case which could be realistically simulated, with the given time step length Δt. However, if two additional IR sensors were to be added, the simulation would become unrealistic: The real robot would not be able to complete all steps during the time Δt.

For the simple robotic brains considered in this course, step 2 would generally be carried out almost instantaneously (compared to step 1) in a real robot. Similarly, the transfer of motor signals to a DC motor is normally very rapid (note, however, that the dynamics of the *motors* may be such that it is pointless to send commands with a frequency exceeding a certain threshold).

To summarize, a sequence of events that takes, say, several seconds per time step to complete in simulation (e.g. the case of collision-checking in a very complex arena) may be perfectly simple to transfer to a real robot, whereas a sequence of events (such as the reading of a large set of IR sensors) that can be completed almost instantaneously in a simulated robot, may simply not be transferable to a real robot, unless a dedicated processor for signal processing and signal transfer is used.

3.2.2 Noise

Another aspect that should be considered in simulations is *noise*. Real sensors and actuators are invariably noisy, on several levels. Furthermore, even sensors that are supposed to be identical often show very different characteristics in practice. In addition, regardless of the noise level of a particular sensor, the frequency with which readings can be updated is limited, thus introducing another source of noise, in certain cases. For example, the limited sampling frequency of wheel encoders implies that, even in the (unrealistic) case where the kinematic model is perfect and there are no other sources of noise, the integrals in the kinematic equations (Eqs. (2.7)-(2.9)) can only be approximately computed.

Thus, in any realistic robot simulation, noise must be added, at all relevant levels. Noise can be added in several different ways. A common method (used in GPRSim and ARSim) is to take the original reading S of a sensor and add noise to form the actual reading $\hat{S}$ as

$$\hat{S} = SN(1, \sigma), \tag{3.1}$$

where $N(1, \sigma)$ denotes the normal (Gaussian) distribution with mean 1 and standard deviation σ. Of course, other distributions (e.g. a uniform distribution) can be used as well.

An alternative method is to take some measurements of a *real* sensor and store the readings in a lookup table, which is then used by the simulated robot. For example, in the case of an IR sensor with a range of, say, 0.5 m, one may, for example, take 10 readings each at distances of $0.05, 0.10, \ldots, 0.50$ m, and store those readings in a matrix. In the simulator, when the IR sensor is used, the distance L to the nearest obstacle is determined, and the reading is then obtained by interpolating linearly between two samples from the lookup table. For example, if $L = 0.23$ m, a randomly chosen sample $\hat{s}_{20}$ is taken from the 10 readings stored for $L = 0.20$ m, and another randomly chosen sample $\hat{s}_{25}$ is taken from the readings stored for $L = 0.25$ m. The reading of the simulated sensor is then taken as

$$\hat{S} = \hat{s}_{20} + \frac{0.23 - 0.20}{0.25 - 0.20}(\hat{s}_{25} - \hat{s}_{20}) \tag{3.2}$$

This method has the advantage of forming simulated readings from actual sensor readings, rather than introducing a model for the noise. Furthermore, using lookup tables, it is straightforward to account for the individual nature of supposedly identical sensors. However, a clear disadvantage is the need for generating the lookup tables, which often must contain a very large number of samples taken not only at various distances, but also, perhaps, at various *angles* between the forward direction of the sensor and the surface of the obstacle. Thus, the first method, using a specific noise distribution, is normally used instead.

3.2.3 Sensing

In addition to correct timing of events and the addition of noise in sensors and actuators, it is necessary to make sure that the sensory signals received by the simulated robot do not contain more information than could be provided by the sensors of the corresponding real robot. For example, in the simulation of a robot equipped only with wheel encoders (for odometry), it is not allowed to provide the simulated robot with continuously updated and error-free position measurements. Instead, the simulated wheel encoders, including noise and other inaccuracies, should be the only source of information regarding the position of the simulated robot.

In both GPRSim and ARSim, several different sensors have been implemented, namely (1) wheel encoders, (2) IR proximity sensors, and (3) compasses. In addition, GPRSim (but not ARSim) also features (4) sonar sensors and (5) laser range finders (LRFs). An important subclass of (simulated) sensors are **ray-based sensors**, which use a simple form of ray tracing in order to form their reading(s). Examples of ray-based sensors are IR proximity sensors, sonar sensors, and laser range finders.

Now, the different natures of, say, an IR sensor, which gives a fuzzy reading based on infrared light, and an LRF, which gives very accurate readings (in many directions) based on laser light, imply that slightly different procedures must be used when forming the (simulated) sensor readings of those two sensor types. However, in both cases, the simulation of the sensor requires ray tracing, which will now be considered.

Ray-based sensors In ray-based sensors, the formation of sensor readings is based on the concept of **sensor rays**. Basically, a number of rays are sent out from a sensor, in various directions (depending on the opening angle of the sensor), and the distance to the nearest obstacle is determined. If no obstacle is available within the range of the sensor, the ray in question provides no reading. Of course, in order to obtain any ray reading, not only the robot must be available, but also the objects (e.g. walls and furniture) located in the arena in which the robot is operating. In GPRSim, objects are built from boxes and cylinders. Boxes are represented as a sequence of six planes, whereas (the mantle surface of) cylinders are represented by a sufficient number of planes (usually around 10-20) to approximate the circular cross section of the cylinder. The ray readings are thus obtained using general equations for line-plane intersections[1]. Here, however, we shall only consider the simpler two-dimensional case, in which all surfaces are vertical and where the sensors are oriented such that all emitted rays are parallel to the ground. In such cases, the arena objects can be represented simply as a sequence of lines in two dimensions. Indeed, this is how objects are represented in ARSim.

An example of such a configuration is shown in Fig. 3.3. The left panel shows a screenshot from GPRSim, in which an LRF mounted on top of a robot takes a reading in an arena containing only walls. The right panel shows a two-dimensional representation of the arena and the LRF (the body of the robot is not shown). Given the exact position of a ray's starting point, as well as the range of the corresponding sensor, it is possible to determine the distance between the ray and the nearest obstacle using general equations for line-line intersection, which will be described next. However, it should first be noted that, even though the *simulator* of course uses the exact position of the robot and its sensors in order to compute sensor readings, the *robot* (or, more specifically, its brain) is *only* provided with information regarding the actual sensor readings.

Consider now a single sensor ray. Given the start and end points of the

[1]In order to speed up the simulator, a grid (also used in collision checking) is used, such that only those obstacles that are (partially) located in the grid cells currently covered by the sensor are considered.

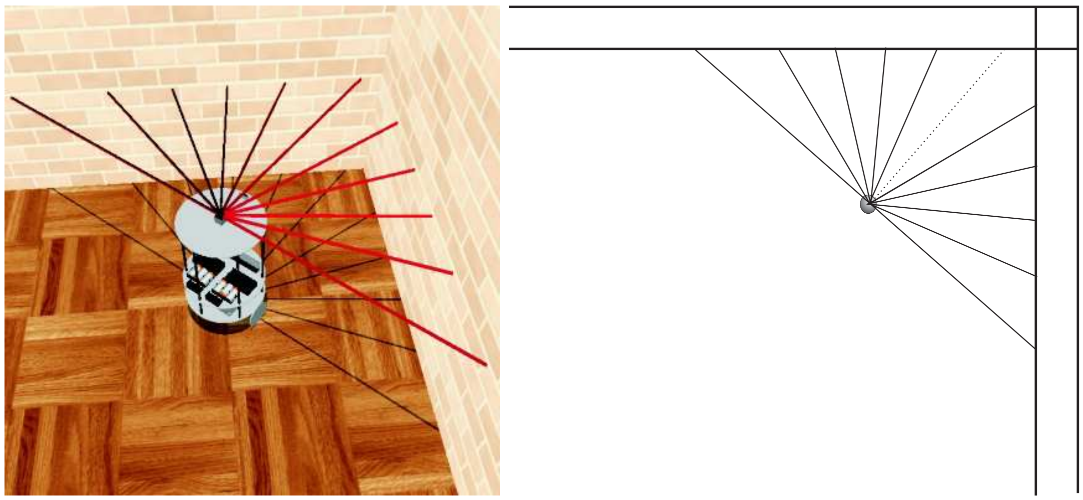

Figure 3.3: *Left panel: A screenshot from GPRSim, showing an LRF taking a reading in an arena containing only walls. Right panel: A two-dimensional representation of the sensor reading. The dotted ray points in the forward direction of the robot which, in this case, coincides with the forward direction of the LRF.*

ray, its equation can be determined. Let $(x_{\rm a}, y_{\rm a})$ denote the start point for the ray (which will be equal to the position of the sensor, if the size of the latter can be neglected). Once the absolute direction (β_i) of the sensor ray has been determined, the end point $(x_{\rm b}, y_{\rm b})$ of an unobstructed ray (i.e. one that does not hit any obstacle) can be obtained as

$$(x_{\rm b}, y_{\rm b}) = (x_{\rm a} + D\cos\beta_i, y_{\rm a} + D\sin\beta_i), \tag{3.3}$$

where D denotes the sensor range. Similarly, any line corresponding to the side of an arena object can be defined using the coordinates of its start and end points. Note that, in Fig. 3.3, all lines defining arena objects coincide with coordinate axes, but this is, of course, not always the case. Now, in the case of two lines of infinite length, defined by the equations $y_k = c_k + d_k x,\ k = 1, 2$, it is trivial to find the intersection point (if any) simply by setting $y_1 = y_2$. However, here we are dealing with line segments of finite length. In this case, the intersection point can be determined as follows: Letting $\mathbf{P}_i^{\rm a} = (x_i^{\rm a}, y_i^{\rm a})$ and $\mathbf{P}_i^{\rm b} = (x_i^{\rm b}, y_i^{\rm b})$, denote the start and end points, respectively, of line $i,\ i = 1, 2$, the equations for an arbitrary point $\mathbf{P}_i$ along the two line segments can be written

$$\mathbf{P}_1 = \mathbf{P}_1^{\rm a} + t\left(\mathbf{P}_1^{\rm b} - \mathbf{P}_1^{\rm a}\right), \tag{3.4}$$

and

$$\mathbf{P}_2 = \mathbf{P}_2^{\rm a} + u\left(\mathbf{P}_2^{\rm b} - \mathbf{P}_2^{\rm a}\right), \tag{3.5}$$

where $(t, u) \in [0, 1]$. Solving the equation $\mathbf{P}_1 = \mathbf{P}_2$ for t and u gives, after some algebra,

$$t = \frac{(x_2^{\rm b} - x_2^{\rm a})(y_1^{\rm a} - y_2^{\rm a}) - (y_2^{\rm b} - y_2^{\rm a})(x_1^{\rm a} - x_2^{\rm a})}{(y_2^{\rm b} - y_2^{\rm a})(x_1^{\rm b} - x_1^{\rm a}) - (x_2^{\rm b} - x_2^{\rm a})(y_1^{\rm b} - y_1^{\rm a})} \tag{3.6}$$

and

$$u = \frac{(x_1^{\rm b} - x_1^{\rm a})(y_1^{\rm a} - y_2^{\rm a}) - (y_1^{\rm b} - y_1^{\rm a})(x_1^{\rm a} - x_2^{\rm a})}{(y_2^{\rm b} - y_2^{\rm a})(x_1^{\rm b} - x_1^{\rm a}) - (x_2^{\rm b} - x_2^{\rm a})(y_1^{\rm b} - y_1^{\rm a})} \tag{3.7}$$

An intersection occurs if *both* t and u are in the range $[0,1]$. Assuming that the first line (with points given by $\mathbf{P}_1$) is the sensor ray, the distance d between the sensor and the obstacle, along the ray in question, can then easily be formed by simply determining $\mathbf{P}_1$ using the t value found, and computing

$$d = |\mathbf{P}_1 - \mathbf{P}_1^{\rm a}| = |t(\mathbf{P}_1^{\rm b} - \mathbf{P}_1^{\rm a})|. \tag{3.8}$$

If the two lines happen to be parallel, the denominator becomes equal to zero[2]. Thus, this case must be handled separately.

In simulations, for any time step during which the readings of a particular sensor are to be obtained, the first step is to determine the origin of the sensor rays (i.e. the position of the sensor), as well as their directions. An example is shown in Fig. 3.4. Here, a sensor is placed at a point $\mathbf{p}_s$, relative to the center of the robot. The absolute position $\mathbf{P}_{\rm s}$ (relative to an external, fixed coordinate system) is given by

$$\mathbf{P}_{\rm s} = \mathbf{X} + \mathbf{p}_{\rm s}, \tag{3.9}$$

where $\mathbf{X} = (X, Y)$ is the position of the (center of mass of the) robot. Assuming that the front direction of the sensor is at an angle α relative to the direction of heading (φ) of the robot, and that the sensor readings are to be formed using N equally spaced rays over an **opening angle** of γ, the absolute direction β_i of the $i^{\rm th}$ ray equals

$$\beta_i = \varphi + \alpha - \frac{\gamma}{2} + (i-1)\delta\gamma, \tag{3.10}$$

where $\delta\gamma$ is given by

$$\delta\gamma = \frac{\gamma}{N-1}. \tag{3.11}$$

Now, the use of the ray readings differs between different simulated sensors. Let us first consider a simulated IR sensor. Here, the set of sensor rays is used *only as an artificial construct* needed when forming the rather fuzzy reading of such a sensor. In this case, the rays themselves are merely a convenient computational tool. Thus, for IR sensors, the robotic brain is not given information regarding the individual rays. Instead, only the complete reading S is provided, and it is given by

$$S = \frac{1}{N}\sum_{i=1}^{N} \rho_i, \tag{3.12}$$

[2]In case the two lines are not only parallel but also *coincident*, both the numerators and the denominators are equal to zero in the equations for t and u.

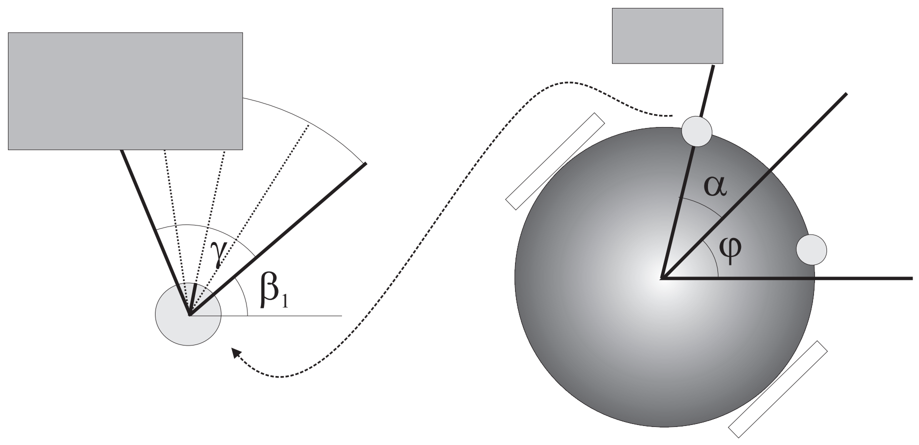

Figure 3.4: *The right panel shows a robot equipped with two IR sensors, and the left panel shows a blow-up of the left sensor. In this case, the number of rays (N) was equal to 5. The leftmost and rightmost rays, which also indicate the opening angle γ of the IR sensor are shown as solid lines, whereas the three intermediate rays are shown as dotted lines.*

where ρ_i is the **ray reading** of ray i. Ideally, the value of N should be very large for the simulated sensor to represent accurately a real IR sensor. However, in practice, rather small values of N (3-5, say) is used in simulation, so that the reading can be obtained quickly. The loss of accuracy is rarely important, since the (fuzzy) reading of an IR sensor is normally used only for proximity detection (rather than, say, mapping or localization). An illustration of a simulated IR sensor is given in Fig. 3.4.

A common phenomenological model for IR sensor readings (used in GPRSim and ARSim) defines ρ_i as

$$\rho_i = \min\left(\left(\frac{c_1}{d_i^2} + c_2\right)\cos\kappa_i, 1\right), \tag{3.13}$$

where c_1 and c_2 are non-negative constants, $d_i > 0$ is the distance to the nearest object along ray i, and

$$\kappa_i = -\frac{\gamma}{2} + (i-1)\delta\gamma, \tag{3.14}$$

is the **relative ray angle** of ray i. If $d_i > D$ (the range of the sensor), $\rho_i = 0$. Note that it is assumed that $\kappa_i \in [-\pi/2, \pi/2]$, i.e. the opening angle cannot exceed π radians. Typical opening angles are $\pi/2$ or less. It should also be noted that this IR sensor model has limitations; for example, the model does not take into account the orientation of the obstacle's surface (relative to the direction of the sensor rays) and neither does it account for the different IR reflectivity of different materials.

For simulated *sonar* sensors (which are included in GPRSim but not in ARSim), the rays are also only used as a convenient computational tool, but the final reading S is formed in a different way. Typically, sonar sensors give rather accurate distance measurements in the range $[D_{\rm min}, D_{\rm max}]$, but sometimes fail to give a reading at all. Thus, in GPRSim, the reading of a sonar sensor is formed as $S = \min_i d_i$ with probability p and $D_{\rm max}$ (no detection) with probability $1-p$. Also, if $S < D_{\rm min}$ the reading is set to $D_{\rm min}$. Typically, the value of p is very close to 1. The number of rays (N) is usually around 3 for simulated sonars.

A simulated LRF, by contrast, gives a vector-valued reading, $\mathbf{S}$, where each component S_i is obtained simply as the distance d_i to the nearest obstacle along the ray. Thus, for LRFs, the sensor rays have a specific physical interpretation, made possible by the fact that the laser beam emitted by an LRF is very narrow. In GPRSim, if $d_i > D$, the corresponding laser ray reading is set to -1, to indicate the absence of any obstacle within range of the ray in question. Note that LRFs are only implemented in GPRSim. It would not be difficult to add such a sensor to ARSim, but since an LRF typically takes readings in 1,000 different directions (thus requiring the same number of rays), such sensors would make ARSim run very slowly.

As a final remark regarding ray-based sensors, it should be noted that a given sensor ray i may intersect several arena object lines (see, for example, Fig. 3.3) In such cases, d_i is taken as the *shortest* distance obtained for the ray.

3.2.4 Actuators

A commonly used actuator in mobile robots is the DC motor. The equations describing such motors are given in Chapter 1.

In both GPRSim and ARSim, a standard DC motor has been implemented. In this motor, the input signal is the applied voltage. Both the electrical and mechanical dynamics of the motors are neglected. Thus the torque acting on the motor shaft axis is given by Eqs. (1.8) and (1.9). Gears are implemented in both simulators, so that the torques acting on the wheels are given by Eqs. (1.10). However, the simulators also include the possibility of setting a maximum torque $\tau_{\rm max}$ which cannot be exceeded, regardless of the output torque $\tau_{\rm out}$ obtained from Eqs. (1.10).

In addition, GPRSim (but not ARSim) also allows simulation of **velocity-regulated motors**. Unlike the voltage signal used in the standard DC motor, a velocity-regulated motor takes as input a desired **reference speed** $v_{\rm ref}$ for the wheel attached to the motor axis. The robot then tries to reach this speed value, using proportional control. The actual output torque of a velocity-regulated motor is given by

$$\tau = K\left(v_{\rm ref} - v\right) \tag{3.15}$$

In this model, a change in $v_{\rm ref}$ generates an immediate change in the torque. In a real motor, the torques cannot change instantaneously. However, Eq. (3.15)

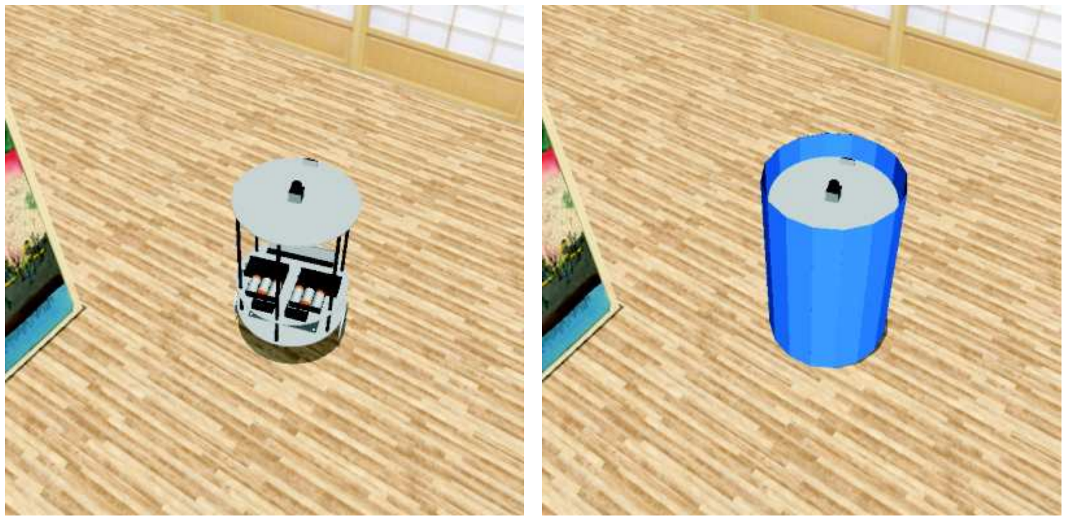
Figure 3.5: *Left panel: A simulated robot (from GPRSim), consisting of more than 100 objects. Right panel: An example (in blue) of a collision geometry.*

usually provides a sufficiently accurate estimate of the torque. As in the case of the standard DC motor, there is also a maximum torque $\tau_{\max}$ for velocity-regulated motors.

Note that, if velocity-regulated motors are to be used, the robot must be equipped with wheel encoders to allow the computation of odometric estimates of the wheel speeds.

3.2.5 Collision checking

A real robot should normally be very careful not to collide with an obstacle (or, worse, a person). In simulations, however, one may allow collisions, for example during simulations involving stochastic optimization, where the robotic brains in the early stages of an optimization run may be unable to avoid collisions. In any case, collisions should, of course, be detected.

In GPRSim the concept of a **collision geometry** is used when checking for collisions. The collision geometry is a set of vertical planes in which the body of the robot should be contained. It would be possible to check collisions between the boxes and cylinders constituting the (simulated) body of the robot. However, it is common that the robotic body consists of a very large number of objects, making collision-checking very slow indeed. Thus, instead, a simpler collision geometry is used. An example is given in Fig. 3.5. The left panel shows a simulated robot (consisting of more than 100 separate objects), and the right panel shows (in blue) a collision geometry for the same robot.

By contrast, in ARSim the simulated robot is always represented as a circular disc. Thus, the collision detection method simply checks for intersections between the circular body of the robot and any line representing a side of an arena object.

3.2.6 Motion

Once the torques acting on the wheels have been generated, the motion of the robot is obtained through numerical integration of Eqs. (2.31) and (2.32). In both GPRSim and ARSim, the integration is carried out using simple first-order (Euler) integration. For each time step, $\dot{V}$ and $\ddot{\varphi}$ are computed using Eqs. (2.31) and (2.32), respectively. The new values V' and $\dot{\varphi}'$ of V and $\dot{\varphi}$ are then computed as

$$V' = V + \dot{V}\Delta t, \tag{3.16}$$

$$\dot{\varphi}' = \dot{\varphi} + \ddot{\varphi}\Delta t, \tag{3.17}$$

where Δt is the time step length (typically set to 0.01 s). The value of φ is then updated, using the equation

$$\varphi' = \varphi + \dot{\varphi}'\Delta t, \tag{3.18}$$

The cartesian components of the velocity are then obtained as

$$V'_x = V' \cos\varphi, \tag{3.19}$$

$$V'_y = V' \sin\varphi. \tag{3.20}$$

Finally, given V'_x and V'_y, the new positions X' and Y' can be computed as

$$X' = X + V'_x\Delta t, \tag{3.21}$$

$$Y' = Y + V'_y\Delta t. \tag{3.22}$$

In addition, if wheel encoders are used, both GPRSim and ARSim also keep track of the rotation of each wheel, for possible use in odometry (if available).

3.2.7 Robotic brain

While the physical components of a robot, such as its sensors and motors, often remain unchanged between simulations, the robotic brain must, of course, be adapted to the task at hand. Robotic brains can be implemented in many different ways.

In **behavior-based robotics** (BBR) the brain of a robot is built from a repertoire (i.e. a set) of basic behaviors, as well as a decision-making procedure, selecting which behavior(s) to activate at any given time. In the **General-purpose robotic brain structure (GPRBS)**, developed in the author's research group, the robotic brain is built from a set of **brain processes**, some of which are **motor behaviors** (that make use of the robot's motors) and some of which are **cognitive processes**, i.e. processes that do not make use of any motors. In addition, GPRBS features a decision-making system based on the concept of utility. One of the main properties of GPRBS is that this structure allows several processes to run in parallel, making it possible to build complex robotic brains. In fact, the specific aim of the development of GPRBS is to move beyond the often very simple robotic brains defined within standard BBR.

In GPRBS, all brain processes are specified in a standardized format, which simplifies the development of new brain processes, since many parts of an already existent process often can be used when writing a new process. However, at the same time, GPRBS (as implemented in GPRSim) is a bit complex to use, especially since it is intended for use in research, rather than as an educational tool. Thus, in this course, ARSim will be used instead. This simulator allows the user to write simple brain processes (as well as a basic decision-making system) in any desired format (i.e. without using GPRBS). Methods for writing brain processes will be described further in a later chapter.

3.3 Brief introduction to ARSim

The simplest way to acquaint oneself with `ARSim` is to run and analyze the test program distributed with the program. In order to do so, start Matlab, move to the right directory and write

```
>> TestRunRobot
```

and press return. The robot appears in a quadratic arena with four walls and two obstacles, as shown in Fig. 3.6. The robot is shown as a circle, and its direction of motion is indicated by a thin line. The IR sensors (of which there are two in the default simulation) are shown as smaller circles. The rays used for determining the sensor readings (three per sensor, per default) are shown as lines emanating from the sensors. In the default simulation, the robot executes 1,000 time steps of length 0.01 s, unless it is interrupted by a collision with an obstacle or a wall.

The flow of the simulation basically follows the structure given in Fig. 3.1. The first lines of code in the `TestRunRobot.m` file are devoted to adding the various `ARSim` function libraries to Matlab's search path. The arena objects are then created and added to the arena. Next, the brain of the robot is created (by a call to `CreateBrain`), and the setup is completed by creating the sensors and motors, and adding them to the robot.

Before the actual simulation starts, the robot's position, heading, velocity, and angular speed are set, and the plot of the arena (including the robot) is created. Optionally, a variable `motionResults`, storing information about

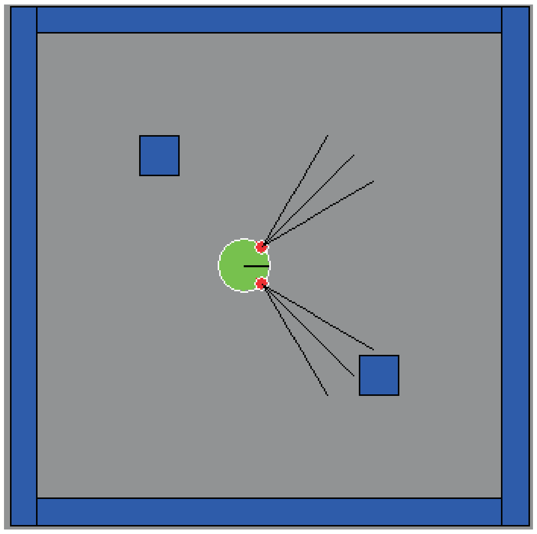

Figure 3.6: *A typical screenshot from an ARSim simulation. The black lines emanating from the two IR proximity sensors of the robot are the rays used for determining sensor readings.*

the robot's motion, can be created.

`ARSim` then executes the actual simulation. Each time step begins with the sensors being read. First, the readings of all ray-based sensors (a category in which only IR sensors have been implemented in ARSim, so far) are obtained. Next, the odometer and compass readings are obtained (provided, of course, that the robot is equipped with those sensors. Next, the robotic brain processes the sensory information (by executing the `BrainStep` function), producing motor signals, which are used by the `MoveRobot` function. Finally, a collision check is carried out.

In normal usage, only a few of `ARSim`'s functions need be modified, namely `CreateBrain`, in which the parameters of the brain are set, `BrainStep`, which determines the processing carried out by the robotic brain, and, of course, the main file (i.e. `TestRunRobot` in the default simulation), where the setup of the arena and the robot are carried out. Normally, no other Matlab functions should be modified unless, for example, one wants to modify the plot procedure.

Note that, by default, the rays involved in the computation of the IR sensor readings are not plotted. In order to plot the sensor rays, one must set the parameter `ShowSensorRays` to `true`. If the robot is equipped with an odometer, one can plot also the position and heading estimated by the odometer, by setting the parameter `ShowOdometricGhost` to true. A brief description of the Matlab functions contained in `ARSim` is given in Appendix A.

Chapter 4

Animal behavior

The behavior-based approach to robotics is strongly influenced by animal behavior. Before studying robotics, it is therefore appropriate to learn some of the basics of this topic. Of course, animal behavior is a vast topic, and in this chapter we shall only study a few examples.

Two important examples are decision-making, which will be introduced briefly in Subsect. 4.4.2 below, and navigation, which is considered in Subsect. 4.4.4.

4.1 Introduction and motivation

Animal behavior is important as a source of inspiration for all work involving autonomous robots. Animals are able to function more or less perfectly in their environment, and to adapt to changes in it. Models of animal behavior, both low-level models involving individual neurons, and high-level phenomenological models, can serve as an inspiration for the development of the corresponding behavior in robots. Furthermore, animals are generally experts in allocating time in an optimal or near-optimal fashion to the many activities (such as eating, sleeping, drinking, foraging etc.) that they must carry out in various circumstances, and lessons concerning behavior selection in animals can give important clues to the solution of similar problems in robotics.

It should be noted that, in the behavior-based approach to robotics (introduced in detail in Chapter 5) one uses a more generous definition of intelligent behavior than in classical artificial intelligence, which was strongly centered on high-level behavior (e.g. reasoning about abstract things) in humans. By contrast, in behavior-based robotics, simple behaviors in animals, such as reflexes and gradient-following (taxis), play a very important role, as will be seen during this course.

4.2 Bottom-up approaches vs. top-down approaches

As is the case with many different topics in science, animal behavior can be studied using either a **bottom-up approach** or a **top-down approach**. The bottom-up approach can, in principle, lead to a more complete and detailed understanding of the objects or organisms under study. However, in sufficiently complex systems, the bottom-up approach may fail to give important insights. For example, when using a computer, it is not necessary to know exactly how the computer manipulates and stores information down to the level of individual electrons. Even without such detailed knowledge, it is certainly possible to use the computer, if only one has information regarding how to program it.

Similarly, in animal behavior, even very simple, top-down models can lead to a good understanding of seemingly complex behavior, as will be shown below in the example of the orientation of bacteria.

On the other hand, a bottom-up study (on the level of individual neurons) can reveal many important aspects of relatively simple animals, such as e.g. the much-studied worm *C. Elegans* or the sea-slug *Aplysia*. The neural level is relevant also in the field of autonomous robotics, where simple behaviors are often implemented using neural network architectures. However, in such cases, the networks are most often used as **black-box models** (obtained, for example, by means of artificial evolution).

4.3 Nervous systems of animals

In essence, the brain of vertebrates consists of three structures namely, the **forebrain**, the **midbrain**, and the **hindbrain**. The **central nervous system** (CNS) consists of the brain and the spinal cord. In addition to the CNS, there is the **peripheral nervous system**, which consists of sensory neurons that carry information to the CNS and motor neurons that carry motor signals from the CNS to muscles and glands (see below). The peripheral nervous system can be sub-divided into the **somatic nervous system**, that deals with the external environment (through sensors and muscles) and the **autonomic nervous system** which provides the control of internal organs such as the heart and lungs. The autonomic nervous system is generally associated with involuntary actions, such heart beat and breathing.

It is interesting to note that the embryological development of different vertebrates is quite similar: During development, a neural tube is differentiated into a brain and a spinal cord.

Note that the presence of a nervous system is not a prerequisite for *all* forms of intelligent behavior: Even single-celled organisms (which, clearly, cannot contain a CNS: Neurons are cells), are able to exhibit rudimentary intelligent behavior. An example involving bacteria will be given below.

In addition to the nervous system, there is a parallel system for feedback in the body of animals, namely the **endocrine system**. The **glands** of the endocrine system release **hormones** (into the blood stream) that influence body and behavior. For example, elevated levels of the hormone **angiotensin** (whose source is the kidney) lead to a feeling of thirst, whereas **adrenaline** is involved in **fight-or-flight** reactions (fear, anxiety, aggression). Hormone release by the endocrine system is controlled either directly by the brain or by (the levels of) other hormones.

Emotions such as fear, and the resulting survival-related reactions, such as fleeing from a predator are, of course, very important for the survival of animals and it is therefore perhaps not surprising that robotics researchers have begun considering artificial emotions in robots. Furthermore, the use of artificial hormones in the modulation of behavior and for behavior selection in autonomous robots has been studied as well.

4.4 Ethology

Historically, different approaches to animal behavior were considered in Europe and the USA: European scientists, such as the winners of the 1972 Nobel prize for medicine or physiology, Lorenz, Tinbergen, and von Frisch, generally were concerned with the study of the behavior of animals in their natural environment. Indeed, the term **ethology** can be defined as *the study of animals in their natural environment*. By contrast, American scientists working with animal behavior generally performed experiments in controlled environments (e.g. a laboratory). This field of research is termed **comparative psychology**.

Both approaches have advantages and disadvantages: The controlled experiments carried out within comparative psychology allow more rigor than the observational activities of ethologists, but the behaviors considered in such experiments may, on the other hand, differ strongly from the behaviors exhibited by animals in their natural environment.

However, in both approaches, **phenomenological models** are used, i.e. models which can describe (and make predictions) concerning, for example, a certain behavior, without modelling the detailed neural activities responsible for the behavior. Indeed, many ethological models introduce purely artificial concepts (such as **action-specific energy** in Lorenz' model for animal motivation), which, nevertheless, may offer insight into the workings of a behavior.

On the following pages, the major classes of animal behavior will be introduced and described.

4.4.1 Reflexes

Reflexes, the simplest forms of behavior, are involuntary reactions to external stimuli. An example is the withdrawal reflex, which is present even in very simple animals (and, of course, in humans as well). However, even reflexes show a certain degree of modulation. For example, some reflexes exhibit **warm-up**, meaning that they do not reach their maximum intensity instantaneously (an example is the scratch reflex in dogs). Also, reflexes may exhibit **fatigue**, by which is meant a reduced, and ultimately disappearing, intensity even if the stimulus remains unchanged. Two obvious reasons for fatigue may be muscular or sensory exhaustion, i.e. either an inability to move or an inability to sense. However, these explanations are often wrong, since the animal may be perfectly capable of carrying out other actions, involving both muscles and sensors, even though it fails to show the particular reflex response under study. An alternative explanation concerns *neural* exhaustion, i.e. an inability of the nervous system to transmit signals (possibly as a result of neurotransmitter depletion). An example[1] is the behavior of *Sarcophagus* (don't ask - you don't want to know) larvae. These animals generally move away from light. However, if placed in a tunnel, illuminated at the entrance, and with a dead end (no pun intended), they move to the end of the tunnel, turn around, and move *towards* the light, out of the tunnel. This is interesting, since these larvae will (if not constrained) always move away from light. However, this is neither a case of muscular exhaustion nor sensory exhaustion. Instead, the larvae have simply exhausted their neural circuits responsible for the turning behavior.

4.4.2 Kineses and taxes

Another form of elementary behavior is orientation of motion, either towards an object, substance, or other stimulus, or away from it. In **taxis**, the animal follows a gradient in a stimulus such as a chemical (**chemotaxis**) or a light source (**phototaxis**). Typical examples are pheromone trail following in (some) ants, an example of chemotaxis, and the motion towards a light source by fly maggots. It is easy to understand how such phototaxis occurs: the maggots compare the light intensity on each side of their bodies, and can thus estimate the light gradient. Motion towards a higher concentration (of food, for example), is exhibited even by very simple organisms, such as bacteria. One may be tempted to use the same explanation, i.e. comparison of concentrations on different sides of the body, for this case as well. However, bacteria are simply too small for the gradient (across their minuscule bodies) to be measurable. In the case of the common *E. Coli* bacterium, concentration differences as small as one part in 10,000 would have to be detectable in order for the organism to

[1]See *Essentials of animal behavior*, by P.J.B. Slater.

follow the gradient in the same way as the fly maggots do. Interestingly, *E. Coli* bacteria are nevertheless able to move towards, and accumulate in, regions of high food concentration, an ability which is exploited by another predatory bacterium, *M. Xanthus*, which secretes a substance that attracts *E. Coli* in what Shi and Zusman[2] has called "fatal attraction". The fact that the *M. Xanthus* are able to feed on *E. Coli* is made all the more interesting by the fact that the latter move around 200 times faster than the former. Now two questions arise: How do the *E. Coli* find regions of higher food concentration, and how do the *M. Xanthus* catch them?

Case study: Behavior selection in E. Coli

Interestingly, a very simple model can account for the ability of *E. Coli* to move towards regions of higher concentration. Essentially, the *E. Coli* bacteria have two behaviors, *straight-line movement*, and *random-walk tumbling*. It can be shown experimentally that, at any given absolute concentration of an attractant substance, the bacteria generally exhibit the tumbling behavior, at least after some time. However, if the bacteria are momentarily placed in a region of higher concentration, they begin moving in straight lines. Now, this cannot be due to normal gradient following, since there is no (spatial) gradient. However, there is a *temporal* gradient, i.e. a difference in concentration over time, and this provides the explanation: While unable to detect a spatial gradient, the *E. Coli* bacteria are equipped with a rudimentary **short-term memory**, allowing them to detect a temporal gradient. The behavior of the *E. Coli* is a simple example of chemotaxis. It is *because* of its slow motion that the *M. Xanthus* bacterium is able to catch the *E. Coli*: By releasing an attractant and staying in the same area, the *M. Xanthus* is able to lure the *E. Coli* to the region, and to keep them tumbling there, ending up as food for the *M. Xanthus* - indeed a fatal attraction.

A simple mathematical model of bacterial chemotaxis can now be formulated. Consider a bacterium faced with the choice of activating its straight-line movement behavior (hereafter: B_1) or its tumbling behavior (hereafter: B_2), and introduce a variable U such that B_1 is activated if $U > T$ (where T is the threshold), and B_2 otherwise. Let X be the value of the stimulus (i.e. the concentration of the attractant). Consider now a **leaky integrator**, given by the equation

$$\frac{\mathrm{d}V(t)}{\mathrm{d}t} + aV(t) = bX(t). \tag{4.1}$$

Now, consider the difference $U(t) = X(t) - V(t)$, and set $T = 0$. In case the bacterium experiences an increase $X(t)$ in the concentration of the attracant, $U(t)$ becomes positive, thus activating B_1. If X remains constant, $U(t)$ slowly

[2]See Shi, W. and Zusman, D.R. *Fatal attraction*, Nature **366**, pp. 414-415 (1993).

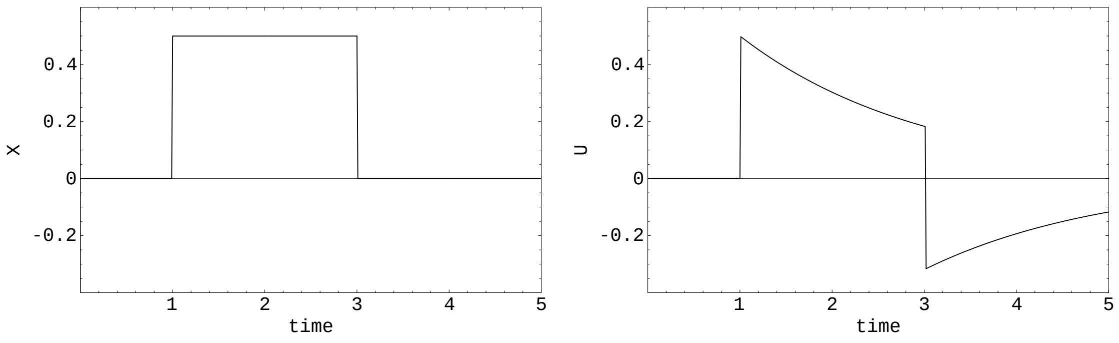

Figure 4.1: *An illustration of the switch between straight-line swimming and tumbling in* E. Coli *bacteria, based on a model with a single leaky integrator given in Eq. (4.1). The left panel shows the variation of the attractant concentration $X(t)$, and the right panel shows $U(t)$. The straight-line swimming behavior (B_1) is active between $t = 1$ and $t = 3$.*

falls towards zero (and eventually becomes negative, if $b > a$). However, if there is a decrease in X, i.e. if the bacterium begins to leave the region of high attractant concentration, $U(t)$ becomes negative, and B_2 is activated, keeping the bacterium approximately stationary. Thus, the chemotaxis of *E. Coli* can be modelled with a single leaky integrator.

Finally, note the importance of taxes in simple behaviors for autonomous robots. For example, it is easy (using, for example, two IR sensors) to equip a robot with the ability to follow a light gradient. Thus, for example, if a light source is placed next to, say, a battery charging station, the robot may achieve long-term autonomy, by activating a phototactic behavior whenever the battery level drops below a certain threshold. Problems involving such behavior selection in autonomous robots will be studied further in later chapters in connection with the topic of **utility**. The simple chemotaxis of the *E. Coli* can be considered as a case of utility-based behavior selection, in which one behavior, B_2, has a fixed utility T, and the utility of the other, B_1, is given by $U(t)$.

Kinesis is an even simpler concept, in which the level of activity (e.g. movement) of an organism depends on the level of some stimulus, but is undirected. An example is the behavior of wood lice: If the ambient humidity is high (a condition favored by wood lice), they typically move very little. If the humidity drops, they will increase their rate of movement.

4.4.3 Fixed action patterns

The concept of **fixed action patterns** (FAPs) was introduced to describe more complex behaviors that are extended over time (beyond the temporal extension of the stimulus) and may involve a sequence of several actions. It should be noted, however, that the term FAP is used less frequently today, since it has been observed that several aspects of such behaviors are not at all fixed. Some

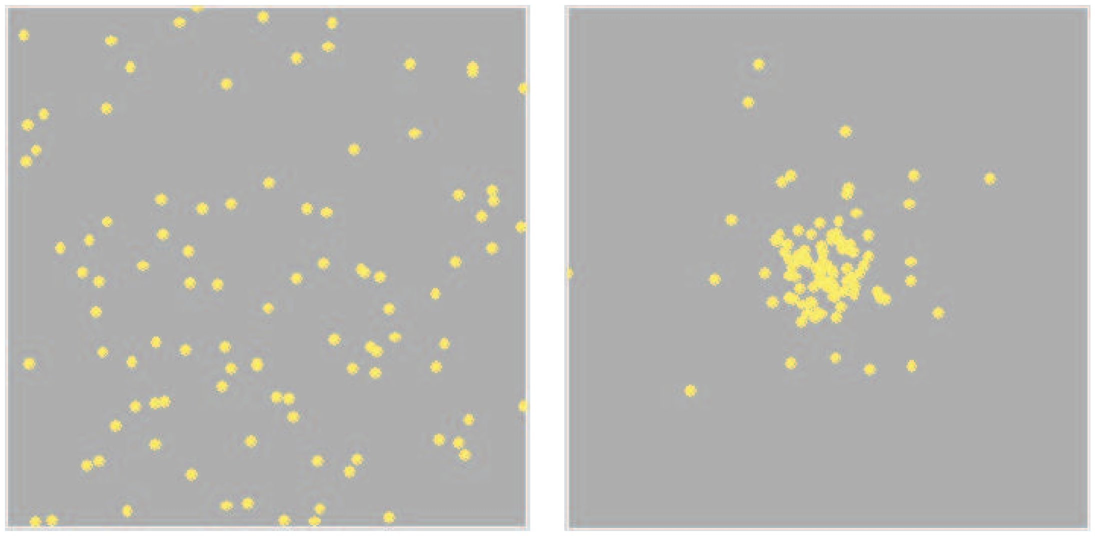

Figure 4.2: *The motion of simulated* E. Coli *bacteria based on the behavior switch defined in the main text. 100 bacteria were simulated, and the parameters a and b were both equal to 1. The attractant had a gaussian distribution, with its peak at the center of the image. The threshold was set to 0. The left panel shows the initial distribution of bacteria, and the right panel shows the distribution after 10 seconds of simulation, using a time step of 0.01.*

behaviors, such as courtship behaviors are, however, quite stereotyped, since they are required to be strongly indicative of the species in question.

An example of an FAP is the egg-retrieving behavior of geese, which is carried out to completion (i.e. the goose moves its beak all the way towards its chest) even if the egg is removed.

Another example is the completely stereotyped attack behavior of the praying mantis (an insect). Once started, the behavior is always carried out to completion regardless of the feedback from the environment (in the case of the praying mantis, the attack occurs with terrifying swiftness, making it essentially impossible for the animal to modulate its actions as a result of sensory feedback).

In addition to FAPs, another concept which has also fallen out of fashion, is the **innate releasing mechanism** (IRM). An IRM was considered a built-in mechanism, characteristic of the species in question, inside the animal which caused it to perform some action based on a few salient features of a stimulus. An example of such a **sign stimulus** is the red belly of male stickleback fish in breeding condition. When competing for a female, the male stickleback must identify and chase away other males. It has been shown in experiments involving the presentation of various crude models of fish to a male stickleback, that the detection of the red color of the belly of other males causes the fish to assume an aggressive posture (rather than, say, the detailed shape of the model, which seems to be more or less irrelevant). Note that several aspects of IRMs, e.g. the question of whether they really are inborn mechanisms, have been called into question.

4.4.4 Complex behaviors

As indicated above, many action sequences that were originally described as FAPs have been found to have a much more complex dynamics than originally thought. Furthermore, many behaviors, such as prey tracking by various mammals, are highly adaptive or involve many different aspects (see the case study below), and can hardly be called FAPs.

Animals generally do not simply react to the immediate stimuli available from the environment, but maintain also an **internal state**, which *together* with the external (sensory) stimuli determine which action to take. Behaviors which depend on an internal state are said to be **motivated**, and the study of animal motivation is an important part of ethology. In early models of motivation, the concept of **drive** was central. A simple model of motivation, based on drives, is the so called **Lorenz' psychohydraulic model**, which will not be studied in the detail here, however. While Lorenz' model is simple, intuitive, and pedagogical, alas it does not fit observations of animal behavior very well. In the modern view of motivation, a given internal state is maintained through a combination of several regulatory mechanisms, and rather than postulating the concept of drives for behaviors, the tendency to express a given behavior is considered as a combination of several factors, both internal and external. The **physiological state** of the animal (e.g. its temperature, amount of water in the body, amount of different nutrients etc.) can be represented as a point in a multi-dimensional **physiological space**. In this space, lethal boundaries can be introduced, i.e. levels which the particular variable (for example, body temperature) may no exceed or fall below.

The **motivational state** of the animal is, in this view, generated by the combination of the physiological state and the **perceptual state** (i.e. the signals obtained through the sensory organs of the animal), and can be represented as a point in a **motivational space**.

As an example of a complex animal behavior, we shall end this chapter with a discussion of desert ant navigation.

Case study: Desert ant navigation

As mentioned above, many species of ants use pheromone trails during navigation. However, for desert ants, such as *Cataglyphis Fortis*, pheromones would not be very useful, due to the rapid evaporation in the desert heat. However, when searching for food, *Cataglyphis* is nevertheless capable of navigating over very large distances (many thousands of body lengths), and then to return to (and locate) the next on an essentially straight line. Locating the nest is no

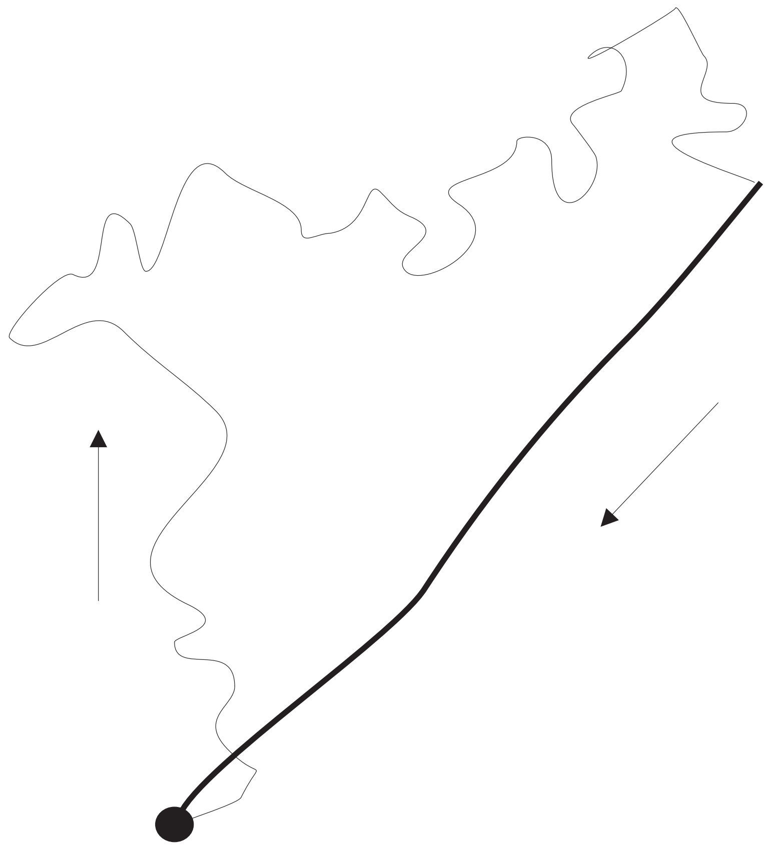

Figure 4.3: *A schematic illustration of a typical* Cataglyphis *trajectory. On the outbound journey, away from the nest (shown as a filled disc) the ant moves on a rather irregular path. However, when returning, the ant follows a more or less straight trajectory (shown as a thick line), following the vector obtained by path integration.*

small feat, keeping in mind that the entrance to the nest is a small hole in the ground.

How does *Cataglyphis* achieve such accurate navigation? This issue has been studied meticulously by Wehner and his colleagues and students[3]. Their experiments have shown that *Cataglyphis* has the capability of computing distance travelled and also to combine the position information thus obtained with heading information obtained using an ingenious form of compass, based on the pattern of light polarization over the sky. Combining the odometric information with the compass information, *Cataglyphis* is able to carry out path integration, i.e. to determine, at any time, the direction vector (from the nest to its current position), regardless of any twists and turns during the outbound section of its movement. Once a food item has been found, the ant will use the stored direction vector in order to return to the nest on an almost straight line.

[3]See e.g. Wehner, R. *Desert ant navigation: how miniature brains solve complex tasks*, J Comp Physiol, **189**, pp. 579-588, 2003.

It should be noted that the light polarization patterns varies with the movement of the sun in the sky. Thus, in order to use its compass over long periods of time, the ant also needs an **ephemeris function**, i.e. a function that describes the position of the sun during the day. Experiments have shown that *Cataglyphis* indeed has such a function.

Even with the path integration, finding the exact spot of the nest is quite difficult: As in the case of robotics, the odometric information and the compass angle have limited accuracy. However, the tiny brain of *Cataglyphis* (weighing in at around 0.1 mg, in a body weighing around 10 mg), is equipped with yet another amazing skill, namely pattern matching: Basically, when leaving its nest, *Cataglyphis* takes (and stores) a visual snapshot (using its eyes) of the scenery around the nest. Then, as the ant approaches the nest (as determined by the vector obtained from path integration), the ant will match its current view to the stored snapshot, and thus find the nest.

It is interesting to note the similarities between robot navigation (which will be described in detail in a later chapter) and *Cataglyphis* navigation: In both cases, the agent (robot or ant) combines path integration with an independent calibration method based on landmark recognition in order to maintain accurate estimates of its pose.

In fact, the description of *Cataglyphis* navigation above is greatly simplified. For example, the interplay between path integration and landmark detection is quite a complex case of decision-making. Furthermore, recent research[4] has shown that, in the vicinity of the nest, *Cataglyphis* uses not only visual but also *olfactory* landmarks (that is, landmarks based on odour). This illustrates another important principle, namely that of *redundancy*. If one sensory modality, or a procedure (such as pattern matching) derived from it, fails, the agent (robot or ant) should be able to use a different sensory modality to achieve its objective.

[4]See Steck, K. *et al. Smells like home: Desert ants, Cataglyphis fortis, use olfactory landmarks to pinpoint the nest*, Frontiers in Zoology **6**, 2009.

Chapter 5

Approaches to machine intelligence

The quest to generate intelligent machines has now (2011) been underway for about a half century. While much progress has been made during this period of time, the intelligence of most mobile robots in use today reaches, at best, the level of insects. Indeed, during the last twenty years, many of the efforts in robotics research have been inspired by rather simple biological organisms, with the aim of understanding and implementing basic, survival-related behaviors in robots, before proceeding with more advanced behaviors involving, for example, high-level reasoning. These efforts have been made mainly within the paradigm of **behavior-based robotics** (BBR), an approach to machine intelligence which differs quite significantly from traditional **artificial intelligence** (AI). However, researchers have not (yet) succeeded in generating truly intelligent machines using either BBR, AI or a combination thereof.

This chapter begins with a brief discussion of the paradigms introduced above. Next, a more detailed introduction to BBR will be given. Finally, the topic of generating basic robotic (motor) behaviors will be considered.

5.1 Classical artificial intelligence

The field of **machine intelligence** was founded in the mid 1950s, and is thus a comparatively young scientific discipline. It was given the name **artificial intelligence** (AI), as opposed to the natural intelligence displayed by certain biological systems, particularly higher animals. The goal of AI was to generate machines capable of displaying human–level intelligence. Such machines are required to have the ability to reason, make plans for their future actions, and also, of course, to carry out these actions.

However, as noted above, despite a half–century of activity in this area, no machines displaying human-level intelligence are, as yet, available. True, there are machines that display a limited amount of intelligent behavior, such

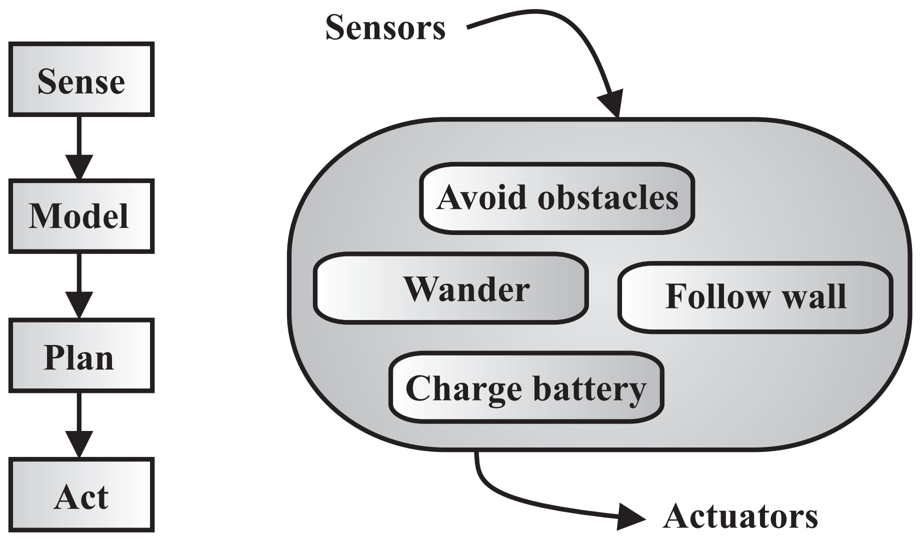

Figure 5.1: *A comparison of the information flow in classical AI (left panel) and in BBR (right panel). For BBR, any number of behaviors may be involved, and the figure only shows an example involving four behaviors.*

as vacuum-cleaning and lawn-mowing robots, and even elevators, automatic trains, TV sets and other electronic equipment. However, the intelligence of these machines is very far from the human–level intelligence originally aimed at by AI researchers. To put it mildly, the construction of artificial systems with human–level intelligence has turned out to be difficult. Human–level intelligence is, of course, extremely complex, and therefore hardly the best starting point. The complex nature of human brains is difficult both to understand and to implement, and one may say that the preoccupation with *human*–level intelligence in AI research has probably been the most important obstacle to progress.

In classical AI, the flow of information is as shown in the left panel of Fig. 5.1. First, the sensors of the robot sense the environment. Next, a (usually very complex) **world model** is built, and the robot reasons about the effects of various actions within the framework of this world model, *before* finally deciding upon an action, which is executed in the real world. Depending on the complexity of the task at hand, the modelling and planning phases can be quite time-consuming. Now, this procedure is very different from the distributed form of computation found in the brains of biological organisms, and, above all, it is generally very *slow*, strongly reducing its survival value. This is not the way biological organisms function. As a good counterexample, consider the evasive maneuvers displayed by noctuid moths, as they attempt to escape from a pursuing predator (for instance, a bat). A possible way of achieving evasive behavior would be to build a model of the world, considering many different bat trajectories, and calculating the appropriate response. However, even if the brain of the moth were capable of such a feat (it is not), it would most likely find itself eaten before deciding what to do. Instead, moths use a much simpler procedure: Their evasive behavior is in fact based on only a few neurons and an ingenious positioning of the ears on the body. This simple system enables the moth to fly away from an approaching bat and, if it is unable to shake off the pursuer, start to fly erratically (and finally dropping toward the ground) to confuse the predator.

As one may infer from the left panel of Fig. 5.1, classical AI is strongly focused on high-level **reasoning**, i.e. an advanced cognitive procedure displayed in humans and, perhaps, some other mammals. Attempting to emulate such complex biological systems has proven to be too complex as a starting-point for research in robotics: Classical AI has had great success in many of the subfields it has spawned (e.g. pattern recognition, path planning etc.), but has made little progress toward the goal of generating truly intelligent machines, capable of autonomous operation.

5.2 Behavior-based robotics

The concept of BBR was introduced in the mid 1980s, and was championed by Rodney Brooks[1] and others. Nowadays, the behavior-based approach is used by researchers worldwide, and it is often strongly influenced by ethology (see Chapter 4).

BBR approaches intelligence in a way that is very different from the classical AI approach, as can be seen in Fig. 5.1. BBR, illustrated in the right panel of Fig. 5.1 is an alternative to classical AI, in which intelligent behavior is built from a set of basic **behaviors**. This set is known as the **behavioral repertoire**. Many behaviors may be running simultaneously in a given robotic brain, giving suggestions concerning which actions the robot ought to take. An attempt should be made to define the concepts of **behaviors** and **actions**, since they are used somewhat differently by different authors. Here, a behavior will be defined simply as a sequence (possibly including feedback loops) of actions performed in order to achieve some goal. Thus, for example, an obstacle avoidance behavior may consist of the actions of stopping, turning, and starting to move again in a different direction.

The construction of a robotic brain (in BBR) can be considered a two-stage process: First the individual behaviors must be generated. Next, a system for selecting which behavior(s) to use in any given situation must be constructed as well: In any robot intended for complex applications, the **behavior selection system** is just as important as the individual behaviors themselves. Be-

[1]See e.g. Brooks, R. *A Robust Layered Control System for a Mobile Robot*, IEEE Journal of Robotics and Automation, **RA–2**, No. 1, pp. 14–23, 1986.

havior selection or, more generally, **decision-making** will be considered in a later chapter.

The example of the moth above shows that **intelligent behavior** does *not* (always) require reasoning, and in BBR one generally uses a more generous definition of intelligent behavior than that implicity used in AI. Thus, in BBR, one may define intelligent behavior as *the ability to survive, and to strive to reach other goals, in an unstructured environment*. This definition is more in tune with the fact that most biological organisms are capable of highly intelligent behavior in the environment where they normally live, even though they may fail quite badly in novel environments (as illustrated by the failure of, for example, a fly caught in front of a window). An **unstructured environment** changes rapidly and unexpectedly, so that it is impossible to rely *completely* on static maps: Even though such maps are highly relevant during, say, navigation, they must also be complemented with appropriate behaviors for obstacle avoidance and other tasks.

The BBR approach has been criticized for its inability to generate solutions to anything other than simple toy problems. In BBR, one commonly ties action directly to sensing; in other words, not much (cognitive) processing occurs. Furthermore, BBR is a rather loosely defined paradigm, in which many different representations of behaviors and behavior selection systems have been developed. The absence of a clearly defined, universal representation of behaviors and behavior selection makes it difficult, say, to combine (or compare) results obtained by different authors and to transfer a robotic brain (or a part thereof) from one robotic platform to another.

For these (and other) reasons, the Adaptive systems research group at Chalmers has, over the last few years, developed the General-purpose robotic brain structure (GPRBS), which is also implemented in GPRSim, with the specific aim of applying the strong sides of BBR (e.g. the connection to biological systems, allowing the development of robust, survival-related behaviors) as well as the strong sides of classical AI (for implementing high-level cognitive processes). Furthermore, GRPBS implements a standardized method for decision-making. This structure will not be described further here, however. Instead, the topic of generating simple behaviors will be considered.

5.3 Generating behaviors

As indicated above, a robot intended for operation in the real world should first be equipped with the most fundamental of all behaviors, namely those that deal with survival. In animals, survival obviously takes precedence over any other activity: Whatever an animal is doing, it will suspend that activity if its life is threatened. What does survival involve in the case of a robot? In order to function, a robot must, of course, be structurally intact, and have a non-zero energy level in its batteries. Thus, examples of survival-related behaviors are *Collision avoidance* and *Homing* (to find, say, a battery charging station). However, even more important, particularly for large robots, is to avoid harming *people*. Thus, the purpose of collision avoidance is often to protect people in the surroundings of the robot, rather than protecting the robot itself. Indeed, one could imagine a situation where a robot would be required to sacrifice itself in defense of a human (this is what robots used by bomb squads do already today). These ideas have been summarized beautifully by the great science fiction author Isaac Asimov in his three laws of robotics, which are stated as follows

> First law: A robot may not injure a human being, or, through inaction, allow a human being to come to harm.
>
> Second law: A robot must obey orders given it by human beings, except where such orders would conflict with the first law.
>
> Third law: A robot must protect its own existence as long as such protection does not conflict this the first or second law

While Asimov's laws certainly can serve as an inspiration for researchers working on autonomous robots, a full implementation of those laws would be a daunting task, requiring reasoning and deliberation by the robot on a level way beyond the reach of the current state-of-the-art. However, in a basic sense, BBR and GPRBS clearly deal with behaviors related to Asimov's laws.

5.3.1 Basic motor behaviors in ARSim

Writing *reliable* behaviors for autonomous robots is more difficult than it might seem at a first glance, particularly for robots operating in realistic (i.e. noisy and unpredictable) environments. In GPRBS, as has been mentioned before, robotic brains consist of several brain processes which together define the overall behavior of the robot. However, there is no well-defined method for deciding the exact composition of a brain process: For example, in some applications, navigation and obstacle avoidance may be parts of the same (motor) behavior whereas, in other applications, navigation and obstacle avoidance may be separate behaviors, as illustrated beautifully by the phenomenological model for the chemotaxis of *E. Coli* in Chapter 4, an example that also illustrates an important principle, namely that of keeping individual behaviors simple, if possible.

A common approach to writing behaviors is to implement them in the form of a sequence of `IF-THEN-ELSE` rules. Such a sequence can also be interpreted as a finite-state machine (FSM), i.e. a structure consisting of a finite number of states and (for each state) a set of transition conditions. In each state, the robot can carry out some action (or a sequence of actions), for example setting its motor signals (left and right, for a differentially steered robot) to particular values.

In ARSim, the program flow essentially follows the diagram shown in Fig. 3.1. Thus, in each time step of the simulation, (1) the robot probes the state of the enviroment using its sensors. With the updated sensor readings, the robot then (2) selects an action and (3) generates motor signals (one for each motor), which are then sent to the motors. Next, (4) the position and velocity are updated, as well as (5) the arena (if needed). Finally, (6) the termination criteria (for example, collisions) are considered.

Note that, as mentioned in Chapter 3, for such a simulation to be realistic (i.e. implementable in a real robot), the time required, in a corresponding real robot, for the execution of steps (1) - (3) must be *shorter* than the simulation time step. By default, ARSim uses an updating frequency of 100 Hz (i.e. a time step of 0.01 s), which is attainable by the simple IR sensors used in the default setup. Furthermore, in the simple behaviors considered here, the deliberations in step (2) usually amount to checking a few `if-then-else`-clauses, a procedure that a modern processor completes within a few microseconds.

The writing of basic behaviors for autonomous robots will be exemplified (1) in the form of a *wandering* behavior, which allows a robot to explore its surroundings, provided that no obstacles are present, and (2) using a simple *navigation* behavior which makes the robot move a given distance (in a straight line), using odometry. Other, more complex behaviors, will be considered in the home problems.

5.3.2 Wandering

The task of robot navigation, in a general sense, is a very complex one, since it normally requires that the robot should know its position at all times which, in turn, requires accurate **positioning** (or **localization**), a procedure which will be considered briefly in the next example. Simpler aspects of the motion of a robot can be considered without the need to introduce localization. For example *wandering* is an important behavior in, for instance, an exploration robot or a guard robot that is required to cover an area as efficiently as possible. In order to implement a specific behavior in ARSim, one must modify the `CreateBrain` and `BrainStep` functions. For the *wandering* behavior, they take the shape shown in code listings 5.1 and 5.2.

As can be seen in code listing 5.1, the CreateBrain function defines all the relevant variables and parameters of the robotic brain. The parameter values remain constant during the simulation, whereas the variables, of course, do not. Even though the variables are given values in `CreateBrain`, those values are typically modified in the first state (initialization) of the behavior; see

Code listing 5.1: *The CreateBrain function for the* wandering *example.*

```
function b = CreateBrain;

%% Variables
leftMotorSignal = 0;
rightMotorSignal = 0;
currentState = 0;

%% Parameters:
forwardMotorSignal = 0.5;
turnMotorSignal = 0.7;
turnProbability = 0.01;
stopTurnProbability = 0.03;
leftTurnProbability = 0.50;


b = struct('LeftMotorSignal',leftMotorSignal,...
           'RightMotorSignal',rightMotorSignal,...
           'CurrentState',currentState,...
           'ForwardMotorSignal',forwardMotorSignal,...
           'TurnMotorSignal',turnMotorSignal,...
           'TurnProbability',turnProbability,...
           'StopTurnProbability',stopTurnProbability,...
           'LeftTurnProbability',leftTurnProbability);
```

code listing 5.2. Note the capitalization used when defining a Matlab `struct`.

Code listing 5.2: *The BrainStep function for the* wandering *example.*

```
function b = BrainStep(robot, time);

b = robot.Brain;

%%%%%%%%%%%%%%% FSM: %%%%%%%%%%%%%%%%%%
if (b.CurrentState == 0) % Forward motion
 b.LeftMotorSignal = b.ForwardMotorSignal;
 b.RightMotorSignal = b.ForwardMotorSignal;
 b.CurrentState = 1;
elseif (b.CurrentState == 1) % Time to turn?
 r = rand;
 if (r < b.TurnProbability)
  s = rand;
  if (s < b.LeftTurnProbability)
   b.LeftMotorSignal  =  b.TurnMotorSignal;
   b.RightMotorSignal = -b.TurnMotorSignal;
  else
   b.LeftMotorSignal  = -b.TurnMotorSignal;
   b.RightMotorSignal =  b.TurnMotorSignal;
  end
  b.CurrentState = 2;
 end
elseif (b.CurrentState == 2) % Time to stop turning?
 r = rand;
 if (r < b.StopTurnProbability)
  b.CurrentState = 0;
 end
end
```

In this case, the `BrainStep` function is implemented as an FSM with three states. In the first state (State 0), the motor signals are set to equal values, making the robot move forward in an (almost) straight line, depending on the level of actuator noise. The FSM then jumps to State 1. In this state, the FSM checks whether it should begin turning. If yes, it decides (randomly) on a turning direction, and then jumps to State 2. If no, the FSM will remain in State 1. Note that the `BrainStep` function is executed 100 times per second (with the default time step of 0.01 s). In State 2, the FSM checks whether it should stop turning. If yes, it returns to State 0.

Code listing 5.3: *The CreateBrain function for the* navigation *example.*

```
function b = CreateBrain;

%% Variables:

leftMotorSignal = 0;
rightMotorSignal = 0;
currentState = 0;
initialPositionX = 0; % Arbitrary value here - set in state 0.
initialPositionY = 0; % Arbitrary value here - set in state 0.

%% Parameters:
desiredMovementDistance = 1;
motorSignalConstant = 0.90;
atDestinationThreshold = 0.02;


b = struct('LeftMotorSignal',leftMotorSignal,...
           'RightMotorSignal',rightMotorSignal,...
           'CurrentState',currentState,...
           'InitialPositionX',initialPositionX,...
           'InitialPositionY',initialPositionY,...
           'DesiredMovementDistance',desiredMovementDistance,...
           'MotorSignalConstant',motorSignalConstant,...
           'AtDestinationThreshold',atDestinationThreshold);
```

5.4 Navigation

Purposeful navigation normally requires estimates of position and heading. In ARSim, the robot can be equipped with wheel encoders, from which odometric estimates of position and heading can be obtained. Note that the odometer is calibrated upon initialization, i.e. the estimate is set equal to the true pose. However, when the robot is moving, the odometric estimate will soon deviate from the true pose. In ARSim, the odometric estimate (referred to as the *odometric ghost*) can be seen by setting the variable `ShowOdometricGhost` to `true`.

A simple example of navigation, in which a robot is required to move 1 m in its initial direction of heading, is given in code listings 5.3 and 5.4. As in the previous example, the variables and parameters are introduced in the `CreateBrain` function. The `BrainStep` function is again represented as an FSM. In State 0, the initial position and heading are stored and the FSM then jumps to State 1, in which the motor signal s (range $[-1, 1]$) is set as

$$s = a\frac{D - d}{D}, \tag{5.1}$$

where a is a constant, D is the desired movement distnace (in this case, 1 m) and d is the actual distance moved.

Code listing 5.4: *The BrainStep function for the* navigation *example.*

```
function b = BrainStep(robot, time);

b = robot.Brain;

if (b.CurrentState ~= 0)
 deltaX = robot.Odometer.EstimatedPosition(1) - b.InitialPositionX;
 deltaY = robot.Odometer.EstimatedPosition(2) - b.InitialPositionY;
 distanceTravelled = sqrt(deltaX*deltaX + deltaY*deltaY);
end

%%%%%%%%%%%%%%% FSM: %%%%%%%%%%%%%%%%%%%
if (b.CurrentState == 0) % Initialization
 b.InitialPositionX = robot.Odometer.EstimatedPosition(1);
 b.InitialPositionY = robot.Odometer.EstimatedPosition(2);
 b.CurrentState = 1;
elseif (b.CurrentState == 1) % Adaptive motion
 motorSignal = b.MotorSignalConstant*(b.DesiredMovementDistance-...
               distanceTravelled)/b.DesiredMovementDistance;
 b.LeftMotorSignal = motorSignal;
 b.RightMotorSignal = motorSignal;
 if (abs(b.DesiredMovementDistance - distanceTravelled) < ...
         b.AtDestinationThreshold*b.DesiredMovementDistance)
  b.CurrentState = 2;
  % 'At destination'  % Output for debugging
 end
elseif (b.CurrentState == 2) % At destination
 b.LeftMotorSignal = 0;
 b.RightMotorSignal = 0;
end
```

The motor signal s is then applied to both wheels of the differentially steered robot. As can be seen, the motor signals will gradually drop from a to (almost) zero. However, when $|D - d|$ drops below bD, where $b \ll 1$ is a constant, the FSM jumps to State 2, in which the robot stops.

Chapter 6

Exploration, navigation, and localization

In the previous chapter, the concept of robotic behaviors was introduced and exemplified by means of some basic motor behaviors. Albeit very simple, such behaviors can be tailored to solve a variety of tasks such as, for example, wandering, wall following and various forms of obstacle avoidance. However, there are also clear limitations. In this chapter, some more advanced motor behaviors will be studied. First, behaviors for exploration and navigation will be considered. Both of these two types of behavior require accurate pose estimates for the robot. It is assumed that the robot is equipped with a (cognitive) *Odometry* brain process, providing continuous pose (and velocity) estimates. As mentioned earlier, such estimates are subject to odometric drift, and therefore an independent method for localization (i.e. odometric recalibration) is always required in realistic applications. Such a method will be studied in the final section of this chapter. However, exploration and navigation are important problems in their own right and, in order to first concentrate on *those* problems, it will thus (unrealistically) be assumed, in the first two sections of the chapter, that the robot obtains perfect, noise-free pose estimates using odometry only.

6.1 Exploration

Purposeful navigation requires some form of **map** of the robot's environment. In many cases, however, no map is available *a priori*. Instead, it is the robot's task to *acquire* the map, in a process known as **simultaneous localization and mapping** (SLAM). In (autonomous) SLAM, a robot is released in an unknown arena, and it is then supposed to move in such a way that, during its motion, its long-range sensors (typically an LRF) covers every part of the arena, so that the sensor readings can be used for generating a map. This is a rather difficult task since, during exploration and mapping, the robot must keep track of its position using, for odometric recalibration, the (incomplete, but growing) map that it is currently generating. SLAM is an active research topic, for which many different methods have been suggested. A currently popular approach is **probabilistic robotics**, in which the robot maintains a probability density function from which its position is inferred. However, SLAM is beyond the scope of this text. Instead, the simpler, but still challenging, topic of exploration *given* perfect positioning (as mentioned in the introduction to this chapter) will be considered.

Exploration can be carried out for different reasons. In some applications, such as lawn mowing, vacuum cleaning, clearing mine fields etc., the robot must physically cover as much as possible of the floor or ground in its environment. Thus, the robot must carry out **area coverage**. In some applications, e.g. vacuum cleaning, it is often sufficient that the robot carries out a more or less aimless wandering that, eventually, will make it cover the entire floor. In other applications, such as mapping, it is unnecessary for the robot to physically visit every spot in the arena. Instead, what matters is that its long-range sensor, typically an LRF (or a camera), is able to sense every place in the arena at some point during the robot's motion. The problem of exploring an arena such that the long-range sensor(s) reach all points in the arena will here be referred to as **sensory area coverage**.

Exploring an arena, without any prior knowledge regarding its structure, is far from trivial. However, a motor behavior (in the GPRBS framework) for sensory area coverage has recently (2009) been implemented[1]. This *Exploration* behavior has been used both in the simulator GPRSim and in a real robot (as a part of SLAM). In both cases, the robot is assumed to be equipped with an LRF. The algorithm operates as follows: A **node** is placed at the current (estimated) position of the robot. Next, the robot generates a set of nodes at a given distance (D) from its current position (estimated using the *Odometry* process). Before any nodes are placed, the robot used the LRF (with an opening (sweep) angle α) to find feasible angular intervals for node placement, i.e. angular intervals in which the distance to the nearest obstacle exceeds $D + \Delta$, where Δ is a parameter measuring the margin between a node and the nearest obstacle behind the node. The exact details of the node placement procedure will not be given here. Suffice it to say that, in order to be feasible, an angular interval must have a width γ exceeding a lower limit γ_{min} in order for a node to be placed (at the center of the angular interval). Furthermore, if the width of a feasible angular interval is sufficiently large, more than one node may be placed in the interval. An illustration of feasible angular intervals and node

[1]See Wahde, M. and Sandberg, D. *An algorithm for sensory area coverage by mobile robots operating in complex arenas*, Proc. of AMiRE 2009, pp. 179-186, 2009.

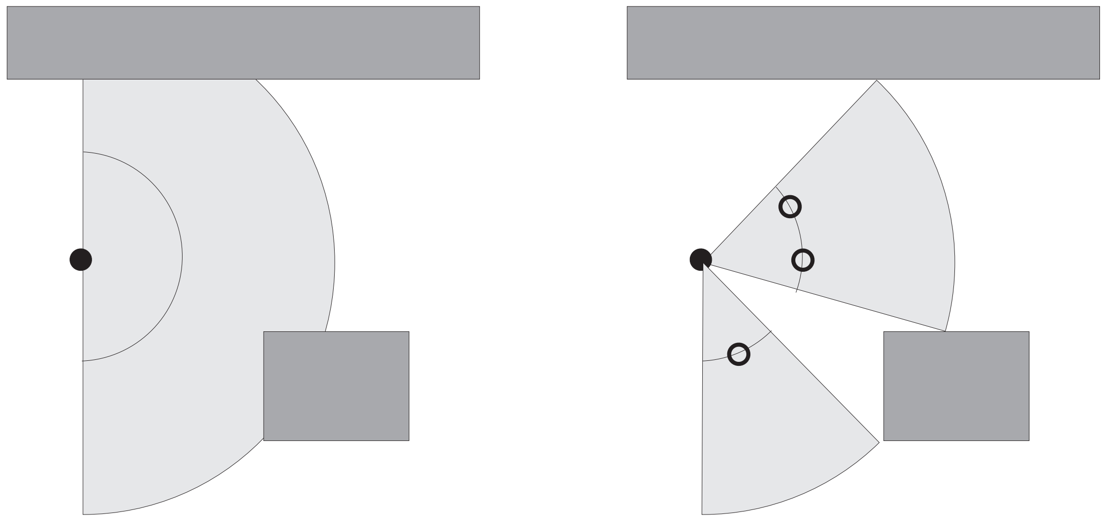

Figure 6.1: *An illustration of the node placement method in the* Exploration *behavior. The left panel shows the distances obtained over the 180 degree opening angle of the LRF (note that individual rays are not shown). The inner semi-circle has a radius of D (the node placement distance) whereas the radius of the outer semi-circle is $D + \Delta$. The right panel shows the resulting distribution of nodes. Note that one of the two feasible angular intervals is sufficiently wide to allow* two *nodes to be placed.*

placement is given in Fig. 6.1.

At this point, the reader may ask why nodes are placed at a distance D from the current node, rather than as far away as possible (minus the margin Δ). The reason is that, in practical use, one cannot (as is done here) assume that the odometry provides perfect pose estimates. Since the *Exploration* behavior is normally used in connection with SLAM, for which accurate positioning is crucial when building the map (a process involving alignment of consecutive laser scans), one cannot move a very large distance between consecutive laser snapshots. Thus, even though the typical range R of an LRF is around 4-10 m or more, the distance D is typically only around 1 m.

An additional constraint on node placement regards the separation (concerning *distances*, not angles) between nodes. A minimum distance of d (typically set to 0.75 m or so) is enforced. The requirement that nodes should be separated by a distance of at least d makes the algorithm finite: At some point, it will no longer be possible to place new nodes without violating this constraint. Thus, when all nodes have been processed (i.e. either having been visited or deemed unreachable, see below), and no further nodes can be added, the exploration of the arena is complete.

Returning to the algorithm, note that the initial node, from which the robot starts its exploration, is given the status *completed* (implying that this node has been reached) and is referred to as the *active* node. All newly generated nodes are given the status *pending*. The robot also generates paths to the pending

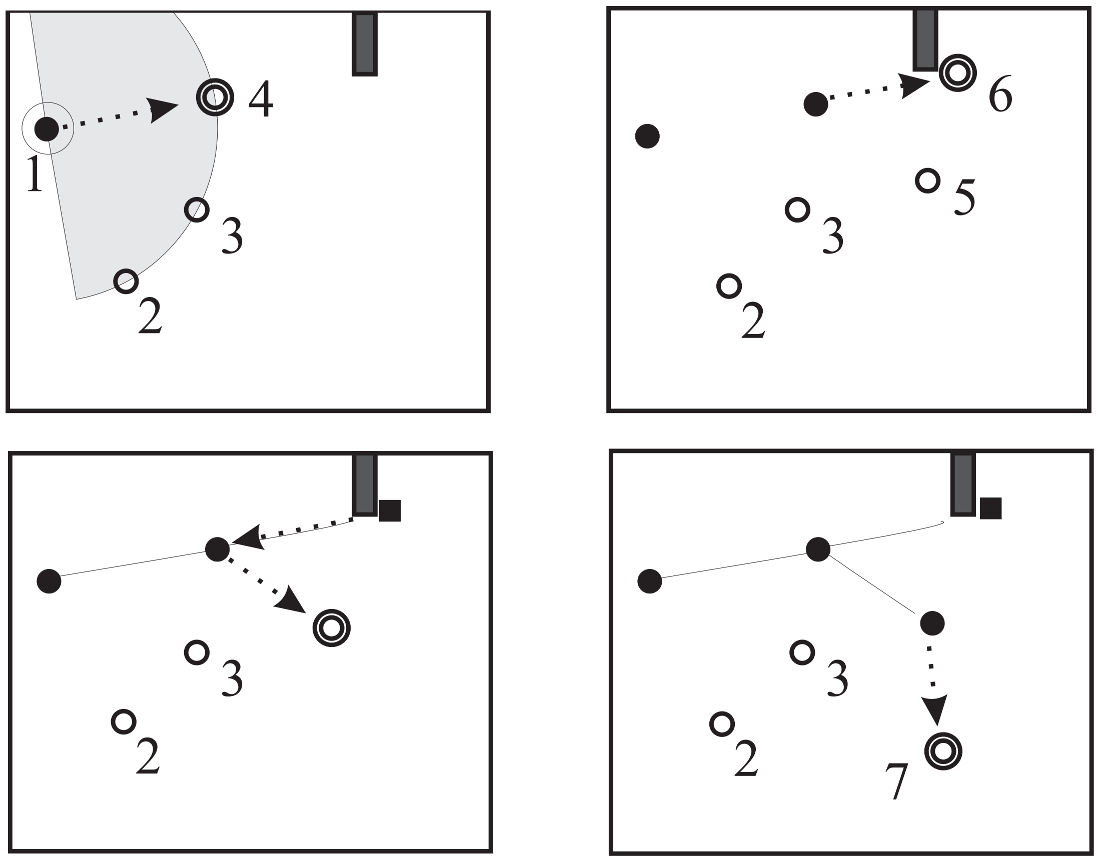

Figure 6.2: *The early stages of a run using the exploration algorithm, showing a robot exploring a single rectangular room without a door. The arena contains a single, low obstacle, which cannot be detected using the LRF (since it is mounted above the highest point of the obstacle). In each panel, the target node is shown with two thick circles, pending nodes are shown as a single thick circle, completed nodes as a filled disc, and unreachable nodes as a filled square. Upper left panel: The robot, whose size is indicated by a thin open circle, starts at node 1, generating three new pending nodes (2, 3, and 4). Upper right panel: Having reached node 4, the robot sets the status of that node to* completed*, and then generates new pending nodes. Lower left panel: Here, the robot has concluded (based on IR proximity readings) that node 6 is unreachable, and it therefore selects the nearest pending node (5, in this case) based on path distance, as the new target node. Lower right panel: Having reached node 5, due to the minimum distance requirement (between nodes) the robot can only generate one new node (7). It rotates to face that node, and then moves towards it etc.*

nodes. For example, if the robot is located at node 1 and generates three pending nodes (2,3 and 4), the paths will be (1, 2), (1, 3) and (1, 4). The robot next selects the nearest node, based on the *path length* as the **target node**. In many cases (e.g. when more than one node can be placed), several nodes are at the same (estimated) distance from the robot. In such cases, the robot (arbitrarily) selects one of those nodes as the target node. For the paths just described, the path length equals the cartesian distance between the nodes. If a path contains more than two elements, however, the path length will differ from the cartesian distance, unless all the nodes in the path lie along a straight line. The path length is more relevant since, when executing the exploration algorithm described here, the robot will generally follow the path, even though direct movement between the active node and a target node is also possible under certain circumstances; see below.

Next, the robot rotates to face the target node, and then proceeds towards it; see the upper left panel of Fig. 6.2. During the motion, one of two things can happen: Either (i) the robot reaches the target node or, (ii) using the output from a *Proximity detection* brain process (assumed available), it concludes that the target node cannot be reached along the current path. Note that, in order for the *Proximity detection* brain process to be useful, the sensors it uses should be mounted at a different (smaller) height compared to the LRF.

In case (i), illustrated in the upper right panel of Fig. 6.2, once the target node has been reached, it is given the status *completed* and is then set as the new active node. At this point, the paths to the remaining pending nodes are updated. Continuing with the example above, if the robot moves to node 4, the paths to the other pending nodes (2 and 3) will be updated to (4, 1, 2) and (4, 1, 3). Furthermore, having reached node 4, the robot generates new pending nodes. Note that the robot need not be located exactly *on* node 4; instead, a node is considered to be reached when the robot passes within a distance a from it. The new nodes are added as described above. The minimum distance requirement between added nodes (see above) is also enforced. Proceeding with the example, the robot might, at this stage, add nodes 5 and 6, with the paths (4, 5) and (4 ,6). Again, the robot selects the nearest node based on the path length, rotates to face that node, and then starts moving towards it etc. Note that the robot can (and will) visit completed nodes more than once. However, by the construction described above, only pending nodes can be target nodes.

In case (ii), i.e. when the target node cannot be reached, the node is assigned the status *unreachable*, and the robot instead selects another target node and proceeds towards it, along the appropriate path. This situation is illustrated in the lower left panel of Fig. 6.2: Here, using its *Proximity detection* brain process, the robot concludes that it cannot reach node 6. It therefore marks this node *unreachable*, sets it as the active node, and then sets the nearest pending node as the new target, in this case node 5. One may wonder why case (ii) can occur, since the robot uses the LRF before assigning new nodes. The reason, of course, is that the LRF (which is assumed to be two-dimensional) only scans the arena at a given height, thus effectively only considering a horizontal slice of the arena. A low obstacle may therefore be missed, until the robot comes sufficiently close to it, so that the *Proximity detection* brain process can detect it.

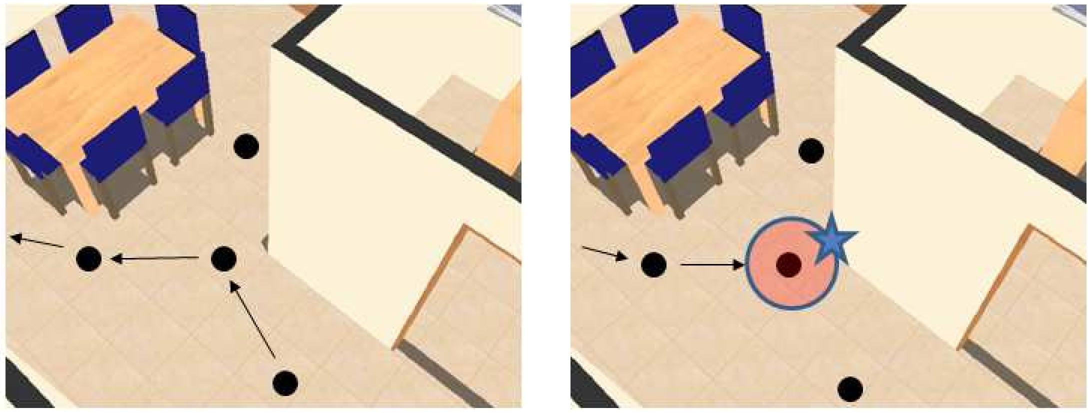

Figure 6.3: *An illustration of a problem that might occur during exploration. Moving in one particular direction (left panel) the robot is able to place and follow the nodes shown. However, upon returning (right panel), the robot may conclude that it will be unable to pass the node near the corner, due to the proximity detection triggered as the robot approaches the node, with the wall right in front of it.*

Note that unreachable nodes are exempt from the minimum distance requirement. This is so, since a given node may be unreachable from *one* direction but perhaps reachable from some other direction. Thus, the exploration algorithm is allowed to place new pending nodes arbitrarily close to unreachable nodes.

One should note that robust exploration of any arena is more difficult than it might seem. An example of a problem that might occur is shown in Fig. 6.3. Here, the robot passes quite near a corner on its outbound journey (left panel), but no proximity detection is triggered. By contrast, upon returning (right panel) a proximity detection *is* triggered which, in turn, may force the robot to abandon its current path. In fact, the *Exploration* behavior contains a method (which will not be described here) for avoiding such deadlocks. In the (very rare) cases in which even the deadlock avoidance method fails, the robot simply stops, and reports its failure.

Because of the path following strategy described above, the robot may sometimes take an unnecessarily long path from the active node to the target node. However, this does not happen so often since, in most cases, the robot will proceed directly to a newly added node, for which the path length is the same as the cartesian distance. However, when the robot cannot place any more nodes (something that occurs, for example, when it reaches a corner), a distant node may become the target node. Therefore, in cases where the path to the target node differs from the direct path, the robot rotates to face the target node (provided that it is located within a distance L, where L should be smaller than or equal to the range R of the LRF). Next, if the robot concludes (based on its LRF readings) that it can reach the target node, it then proceeds directly towards it, rather than following the path. However, also in this case,

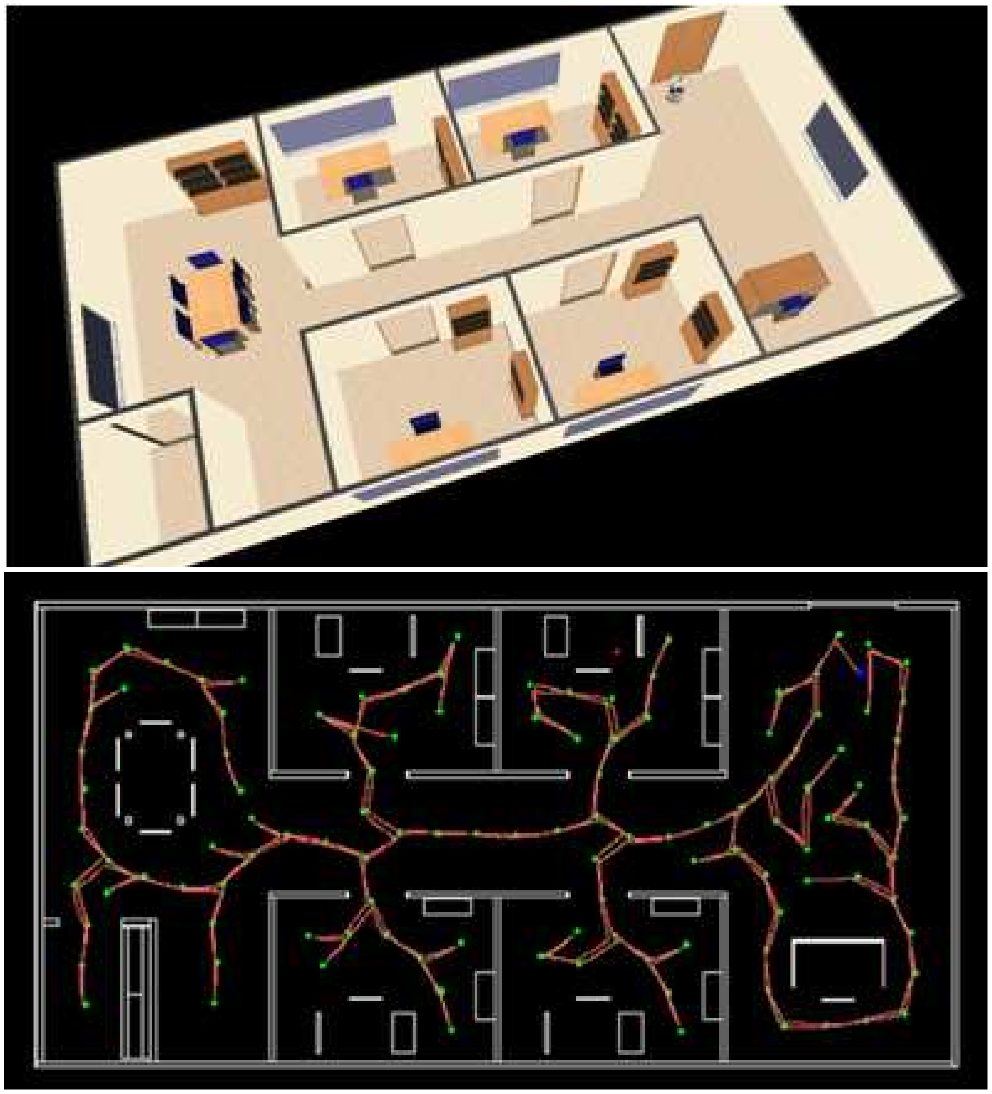

Figure 6.4: *Left panel: The robot in its initial position in an unexplored arena. Right panel: The final result obtained after executing the* Exploration *behavior. The completed (visited) nodes are shown as green dots, whereas the (single) unreachable node is shown as a red dot. The final target node (the last node visited) is shown as a blue dot. In this case, the robot achieved better than 99.5% sensory area coverage of the arena.*

it is possible that a (low) obstacle prevents the robot from reaching its target, in which case the robot instead switches to following the path as described above.

The robot continues this cycle of node placement and movement between nodes, until all nodes have been processed (i.e. either having been visited or deemed unreachable), at which point the exploration of the arena is complete. The *Exploration* behavior consists of an FSM with 17 states, which will not be

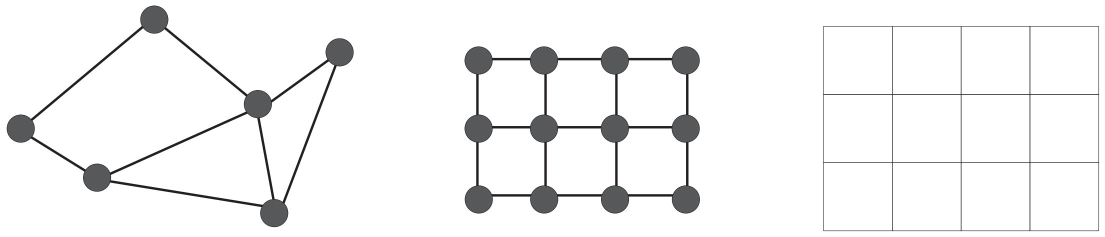

Figure 6.5: *Three examples of grids that can be used in connection with grid-based navigation methods. In the left panel, the nodes are not equidistant, unlike the middle panel which shows a regular lattice with equidistant nodes. The regular lattice can also be represented as grid cells, as shown in the right panel. Note that the right and middle panels show equivalent grids.*

described in detail here. A performance example is shown in Fig. 6.4. The left panel shows the robot at its starting point in a typical office arena. The right panel shows the final result, i.e. the path generated by the robot. The completed (visited) exploration nodes are shown as green dots, whereas the unreachable nodes (only one in this case) are shown as red dots. The final target node is shown as a blue dot. Note that the robot achieved a *sensory* area coverage (at the height of its LRF) of more than 99.5% during exploration.

6.2 Navigation

In this section, it will again be assumed that the robot has access to accurate estimates of its pose (from the *Odometry* brain process), and the question that will be considered is: Given that the robot knows its pose and velocity, how can it navigate between two arbitrary points in an arena? In the robotics literature, many methods for navigation have been presented, three of which will be studied in detail in this section.

6.2.1 Grid-based navigation methods

In **grid-based navigation methods**, the robot's environment must be covered with an (artificial) grid, consisting of **nodes** (vertices) and **edges** connecting the nodes. The grid may have any shape, as illustrated in the left panel of Fig. 6.5, i.e. it need not be a rectangular lattice of the kind shown in the middle panel. However, if the grid happens to be a rectangular lattice, it is often represented as shown in the right panel of the figure, where the nodes have been replaced by cells, and the edges are not shown[2]. Furthermore, the edges must be associated with a (non-negative) cost, which, in many cases is simply taken

[2]Note that in the cell representation in the right panel, the sides of each cell are *not* edges: The edges connect the centers of the grid cells to each other, as shown in the middle panel.

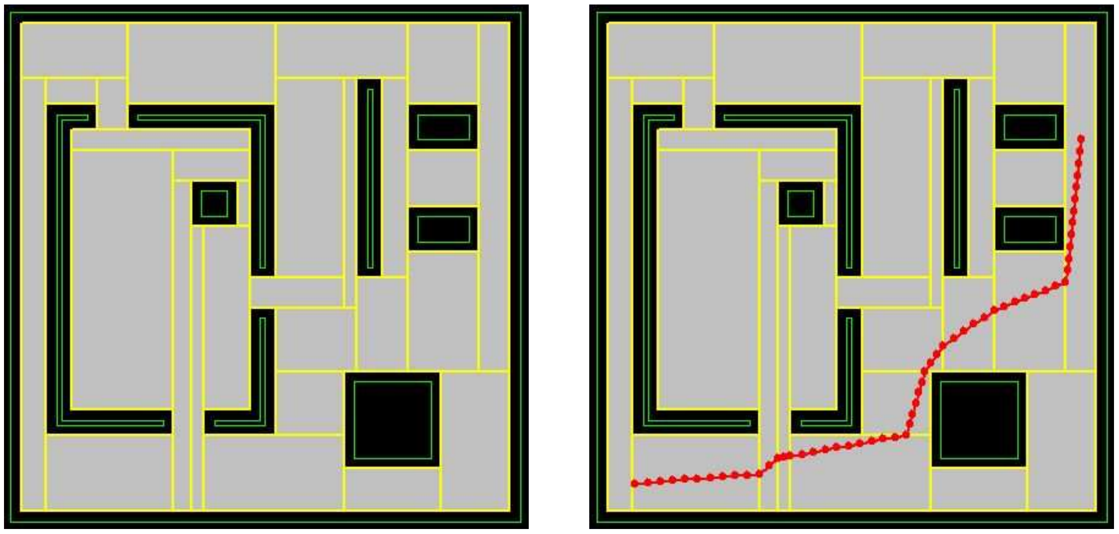

Figure 6.6: *Left panel: An example of automatic grid generation. The walls of the arena are shown as green thin lines. The black regions represent the forbidden parts of the arena, either unreachable locations or positions near walls and other obstacles. The grid cell boundaries are shown as thick yellow lines. Right panel: An example of a path between two points in the arena. The basic path (connecting grid cells) was generated using Dijkstra's algorithm (see below). The final path, shown in the figure, was adjusted to include changes of directions within grid cells, thus minimizing the length of the path. Note that all cells are convex, so that the path segments within a cell can safely be generated as straight lines between consecutive waypoints.*

as the euclidean distance between the nodes. Thus, for example, in the grids shown in the middle and right panels of Fig. 6.5, the cost of moving between adjacent nodes would be equal to 1 (length unit), whereas, in the grid shown in the left panel the cost would vary depending on which nodes are involved.

An interesting issue is the *generation* of a navigation grid, given a two-dimensional map of an arena. This problem is far from trivial, especially in complex arenas with many walls and other objects. Furthermore, the grid generation should take the robot's size (with an additional margin) into account, in order to avoid situations where the robot must pass very close to a wall or some other object. The grid-based navigation methods described below generate paths *between* grid cells. On a regular grid with small, quadratic cells (as in the examples below) it is sometimes sufficient to let the robot move on straight lines between the cell centers. However, the generated path may then become somewhat ragged. Furthermore, in more complex grids, where the cells are of different size, following a straight line between cell centers may result in an unnecessarily long path. Thus, in such cases, the robot must normally modify its heading *within* a cell, in order to find the shortest path.

When generating a grid, one normally requires the grid cells to be **convex**,

1. Place the robot at the start node, which then becomes the current node. Assign the status *unvisited* to all nodes.

2. Go through each of the cells a_i that are (i) unvisited and (ii) directly reachable (via an edge) from the current node c. Such nodes are referred to as **neighbors** of the current node. Compute the cost of going from a_i to the target node t, using the heuristic $f(a_i)$.

3. Select the node $a_{\min}$ associated with the lowest cost, based on the cost values computed in Step 2.

4. Set the status of the current node c as *visited*, and move to $a_{\min}$ which then becomes the current node.

5. Return to Step 2.

Figure 6.7: *The best-first search algorithm.*

so that all points on a straight line between any two points in the cell also are part of the cell. One way of doing so is to generate a grid consisting of triangular cells, which will all be convex. However, such grids may not be optimal: The pointiness of the grid cells may force the robot to make many unnecessary (and sharp) turns. An algorithm for constructing, from a map, a general grid consisting of convex cells with four or more sides (i.e. non-triangular) exists as well[3]. Fig. 6.6 shows an example of a grid generated with this algorithm. Because of its complexity, the algorithm will not be considered in detail here. Instead, in the examples below, we shall consider grids consisting of small quadratic cells, and we will neglect changes of direction within grid cells.

Best-first search algorithm

In **best-first search** (BFS) algorithm the robot moves greedily towards the target, as described in Fig. 6.7. As can be seen, the BFS method chooses the next node based on the (estimated) cost of going from that node n to the goal, which is estimated using a **heuristic** function $f(n)$. $f(n)$ can be chosen in different ways, the simplest being to use the euclidean distance between the node under consideration and the target. However, in that case, the BFS method may, in fact, get stuck. A more sophisticated heuristic function may, for example, add a penalty for each obstacle encountered on a straight-line path from the node under consideration to the target node.

The path can be generated by simply storing the list of visited nodes during

[3]See Wahde, M., Sandberg, D., and Wolff, K. *Reliable long-term navigation in indoor environments*, In: Topalov, A.V. (Ed.), Recent advances in Mobile Robots, InTech, 2011, pp. 261–286.

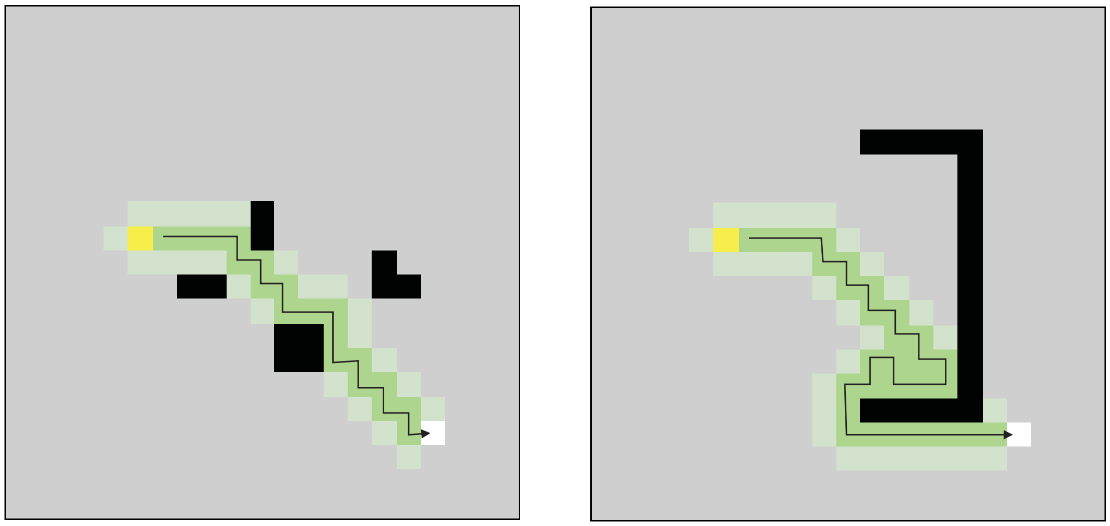

Figure 6.8: *Two examples of paths generated using the BFS algorithm. The cells (nodes) that were checked during path generation are shown in light green, whereas the actual path is shown in dark green and with a solid line. The yellow cell is the start node and the white cell is the target node.*

path generation. The BFS method is very efficient in the absence of obstacles or when the obstacles are few, small, and far apart. An example of such a path generated with BFS is shown in the left panel of Fig. 6.8. As can be seen, the robot quickly moves from the start node to the target node. However, if there are extended obstacles between the robot's current position and the target node, the BFS algorithm will not find the shortest path, as shown in the right panel of Fig. 6.8. Because of its greedy approach to the target, the robot will find itself in front of the obstacle, and must then make a rather long detour to arrive at the target node.

Dijkstra's algorithm

Like BFS, **Dijkstra's algorithm** also relies on a grid in which the edges are associated with non-negative costs. Here, the cost will simply be taken as the euclidean distance between nodes. Instead of focusing on the (estimated) cost of going from a given node to the target note, Dijkstra's algorithm considers the distance between the *start* node and the node under consideration, as described in Fig. 6.9. In Step 2, the distance from the start node s to any node a_i is computed using the (known) distance from the initial node to the current node c and simply adding the distance between c and a_i. This algorithm will check a large number of nodes, in an expanding pattern from the start node, as shown in Fig. 6.10. In order to determine the actual path to follow, whenever a new node a is checked, a note is made regarding the predecessor node p, i.e. the

1. Place the robot at the start node s, which then becomes the current node. Assign the distance value 0 to the start node, and infinity to all other nodes (in practice, use a very large, finite value). Set the status of all nodes to *unvisited*.

2. Go through all the unvisited, accessible (i.e. empty) neighbors a_i of the current node c, and compute their distance d from the *start* node s. If d is smaller than the previously stored distance d_i (initially infinite, see Step 1), then (i) update the stored distance, i.e. set $d_i = d$ and (ii) assign the current node as the predecessor node of a_i.

3. After checking all the neighbors of the current node, set its status to visited.

4. Select the node (among *all* the unvisited, accessible nodes in the grid) with the smallest distance from the start node, and set it as the new current node.

5. Return to Step 2, unless the target has been reached.

6. When the target has been reached, use the predecessor nodes to trace a path from the target node to the start node. Finally, reverse the order of the nodes to find the path from the start node to the target node.

Figure 6.9: *Dijkstra's algorithm.*

node that was the current node when checking node a. When the target has been found, the path connecting it to the initial node can be obtained by going through the predecessor nodes backwards, from the target node to the initial node.

Unlike the BFS algorithm, Dijkstra's algorithm is guaranteed to find the shortest path[4] from the start node to the target node. However, a drawback with Dijkstra's algorithm is that it typically searches many nodes that, in the end, turn out to be quite irrelevant. Looking at the search patterns in Figs. 6.8 and 6.10, one may hypothesize that a *combination* of the two algorithms would be useful. Indeed, there is an algorithm, known as **A*** that combines the BFS and Dijkstra algorithms. Like Dijkstra's algorithm, A* is guaranteed to find the shortest path. Moreover, it does so more efficiently than Dijkstra's algorithm. However, A* is beyond the scope of this text.

[4]There may be more than one such path: Dijkstra's algorithm will select one of them.

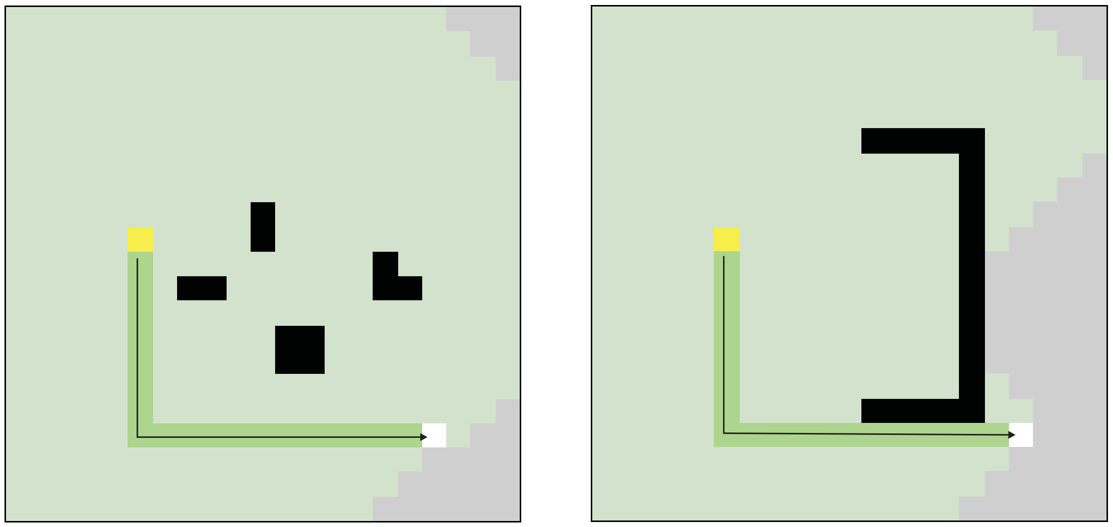

Figure 6.10: *Two examples of paths generated using Dijkstra's algorithm. The cells (nodes) that were checked during path generation are shown in light green, whereas the actual path is shown in dark green and with a solid line. The yellow cell is the start node and the white cell is the target node.*

6.2.2 Potential field navigation

Unlike the algorithms described above, the **potential field method** does not require a grid. In the potential field method, a robot obtains its desired direction of motion as the negative gradient of an artificial potential field, generated by potentials assigned to the navigation target and to objects in the arena.

Potential fields

As shown in Fig. 6.11, a potential field can be interpreted as a landscape with hills and valleys, and the motion of a robot can be compared to that of a ball rolling through this landscape. The navigation target is assigned a potential corresponding to a gentle downhill slope, whereas obstacles should generate potentials corresponding to steep hills.

In principle, a variety of different equations could be used for defining different kinds of potentials. An example, namely a potential with ellipsoidal equipotential surfaces, and exponential variation with (ellipsoidal) distance from the center of the potential, takes the mathematical form

$$\phi(x, y; x_p, y_p, \alpha, \beta, \gamma) = \alpha \mathrm{e}^{-\left(\frac{x-x_p}{\beta}\right)^2 - \left(\frac{y-y_p}{\gamma}\right)^2}, \tag{6.1}$$

where (x, y) is the current (estimated) position at which the potential is calculated, (x_p, y_p) is the position of the object generating the potential, and α, β and γ are constants (not to be confused with the constants defined in connection

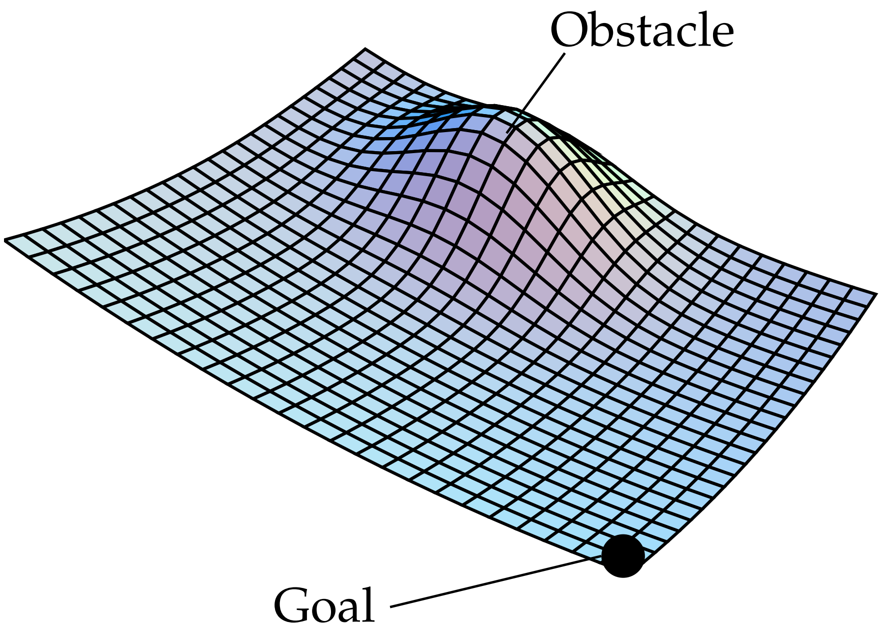

Figure 6.11: *A potential field containing a single obstacle and a navigation goal.*

with the equations of motion in Chapter 2 and the sensor equations in Chapter 3). Now, looking at the mathematical form of the potentials, one can see that an attractive potential (a valley) is formed if α is negative, whereas a positive value of α will generate a repulsive potential (a hill).

Normally, the complete potential field contains many potentials of the form given in Eq. (6.1), so that the total potential becomes

$$\Phi(x, y) = \sum_{i=1}^{k} \phi_i(x, y; x_{p_i}, y_{p_i}, \alpha_i, \beta_i, \gamma_i), \tag{6.2}$$

where k is the number of potentials. An example of a potential field, for a simple arena with four central pillars, is shown in Fig. 6.12.

Navigating in a potential field

Once the potential field has been defined, the desired direction of motion $\hat{\mathbf{r}}$ of the robot can be computed as the negative of the normalized gradient of the field

$$\hat{\mathbf{r}} = -\frac{\nabla\Phi}{|\nabla\Phi|} \equiv -\frac{\left(\frac{\partial\Phi}{\partial x}, \frac{\partial\Phi}{\partial y}\right)}{\sqrt{\left(\frac{\partial\Phi}{\partial x}\right)^2 + \left(\frac{\partial\Phi}{\partial y}\right)^2}} \tag{6.3}$$

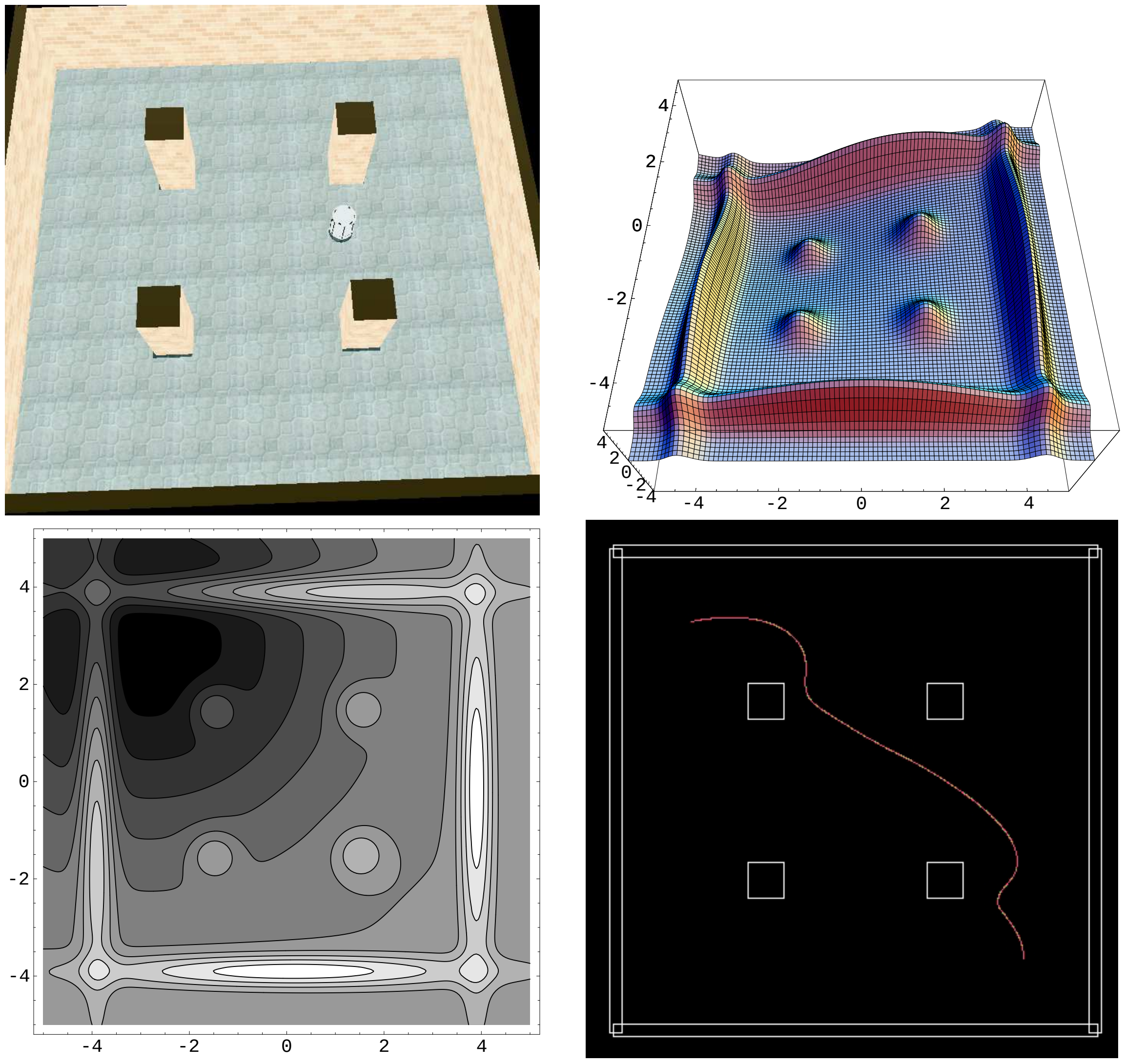

Figure 6.12: *An illustration of potential field navigation in GPRSim. Upper left panel: A simple arena, with a robot following a potential field toward a target position in the upper left corner of the arena. Upper right panel: The corresponding potential field, generated by a total of nine potentials (one for the target, one for each of the walls, and one for each pillar). Lower left panel: A contour plot of the potential field, in which the target position can be seen in the upper left corner. Lower right panel: The trajectory followed by the robot. Note that, in this simulation, the odometric readings were (unrealistically) noise-free.*

In order to integrate the equations of motion of the robot, it is not sufficient only to know the desired direction: The magnitude of the force acting on the robot must also be known. In principle, the negative gradient of the potential field could be taken (without normalization) as the *force* acting on the robot, providing both magnitude and direction. However, in that case, the magnitude of the force would vary quite strongly with the position of the robot, making the robot a dangerous moving object (if it is large). Thus, the potential field is only used for providing the direction, as in Eq. (6.3). The robot's speed v (i.e. the magnitude of its velocity vector $\mathbf{v}$) can be assigned in various ways. For example, one may use proportional control to try to keep the speed constant[5].

An example of a trajectory generated during potential field navigation is shown in the lower right panel of Fig. 6.12. In the experiment in which this figure was generated, the noise in the odometric readings was (unrealistically) set to zero, since the aim here is simply to illustrate potential field navigation. However, in a realistic application, one would have to take into account the fact that the robot's estimate of its pose will never be error-free. Thus, when setting up a potential field, it is prudent to make the potentials slightly larger[6] than the physical objects that they represent. At the same time, in narrow corridors, one must be careful not to make the potentials (for walls on opposite sides of the corridor, say) so wide that the robot will be unable to pass.

In fact, the definition of a potential field for a given arena is something of an art. In addition to the problem of determining the effective extension of the potentials, one also has to decide whether a given object should be represented by one or several potentials (for instance of the form given in Eq. (6.1)). For example, an extended object (for example, a long wall) *can* be represented as a single potential (typically with very different values of the parameters β and γ), but it can also be represented as a sequence of potentials. In complex environments, one may resort to stochastic optimization of the potential field, as well as the details of the robot's motion in the field[7].

Aspects of potential field navigation

A gradient-following method, such as the potential field method, always suffers the risk of encountering local minima in the field. Of course, in potential

[5]The procedure for assigning the robot's speed in potential field navigation will be described below.

[6]Of course, since the exponential potentials defined in Eq. (6.1) have infinite extension, the corresponding force never drops exactly to zero, but beyond a distance of a few d, where $d = \max(\beta, \gamma)$, the force is negligible.

[7]For an example of such an approach, see Savage *et al.*, *Optimization of waypoint-guided potential field navigation using evolutionary algorithms*, Proceedings of the 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2004), 3463-3468, 2004.

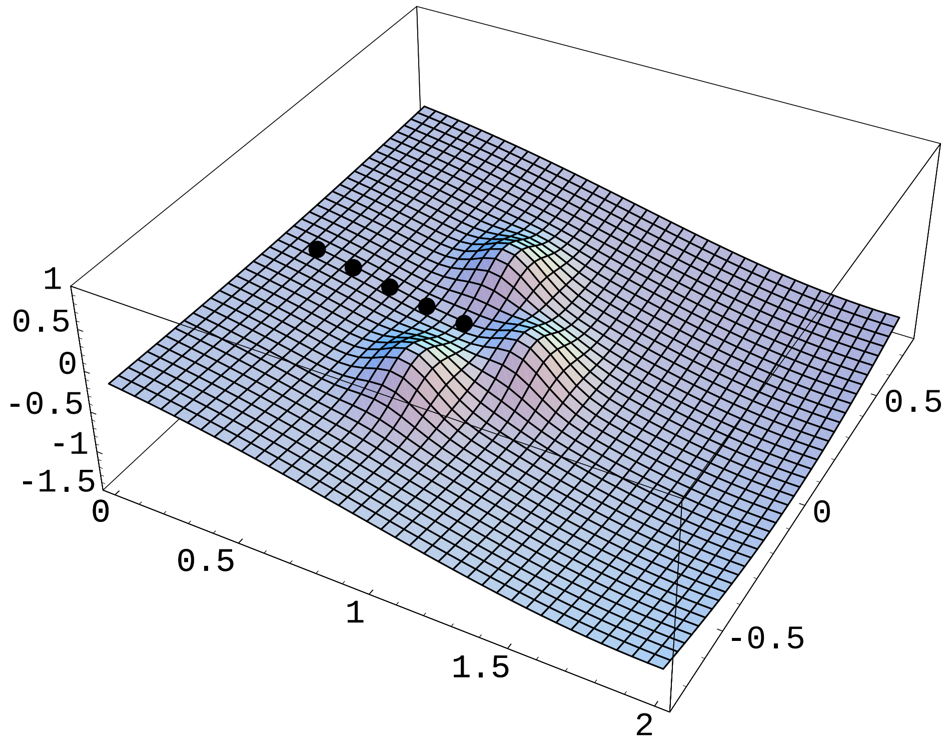

Figure 6.13: *The locking phenomenon. Following the gradient of the potential field the robot, whose trajectory is shown as a sequence of black dots, moves along the x-axis toward the goal, located at $(2, 0)$. However, because of the local minimum in the potential field, the robot eventually gets stuck.*

field navigation, the goal is to reach the local minimum represented by the navigation target. However, depending on the shape of the arena (and therefore the potential field), there may also appear one or several unwanted local minima in the field, at which the robot may become trapped.

This is called the **locking phenomenon** and it is illustrated in Fig. 6.13. Here, a robot encounters a wedge-shaped obstacle represented by three potentials. At a large distance from the obstacle, the robot will be directed toward the goal potential, which is located behind the obstacle as seen from the starting position of the robot. However, as the robot approaches the obstacles their repulsive potentials will begin to be noticeable. Attracted by the goal, the robot will thus eventually find itself stuck inside the wedge, at a local minimum of the potential.

In order to avoid locking phenomena, the path between the robot and the goal can be supplied with **waypoints**, represented by attractive potentials (for example, of the form given in Eq. (6.1)) with rather small extension. Of course, the introduction of waypoints leads to the problem of determining where to put them. An analysis of such methods will not be given here[8]. Suffice it to say that the problem of waypoint placement can be solved in various ways to aid the robot in its navigation. A waypoint should be removed once the robot has passed within a distance L from it, to avoid situations in which the robot finds itself stuck at a *waypoint*.

The potential field method also has several advantages, one of them being that the direction of motion is obtained simply by computing the gradient of the potential field at the current position, without the need to generate an entire path from the current position to the navigation target. Furthermore, the potential field is defined for all points in the arena. Thus, if the robot temporarily must suspend its navigation (for example, in order to avoid a moving obstacle), it can easily resume the navigation from wherever it happens to be located when the *Obstacle avoidance* behavior is deactivated.

In the discussion above, only stationary obstacles were considered. Of course, moving obstacles can be included as well. In fact, the potential field method is commonly used in conjunction with, say, a grid-based navigation method, such that the latter generates the nominal path of the robot, whereas the potential field method is used for adjusting the path to avoid moving obstacles. However, methods for reliably *detecting* moving obstacles are beyond the scope of this text.

Using the potential field method

As mentioned above, the potential field only provides the current desired *direction* of motion. In order to specify a potential field navigation behavior completely, one must also provide a method for setting the speed of the robot. This can be done as follows: Given the robot's estimated (from odometry) angle of heading φ_{est} and the desired (reference) direction φ_{ref} (obtained from the potential field), one can form the quantity $\Delta\varphi$ as

$$\Delta\varphi = \varphi_{\mathrm{ref}} - \varphi_{\mathrm{est}}. \tag{6.4}$$

The desired speed differential ΔV (the difference between the right and left wheel speeds) can then be set according to

$$\Delta V = K_{\mathrm{p}} V_{\mathrm{nav}} \Delta\varphi, \tag{6.5}$$

where K_{p} is a regulatory constant (P-regulation is used) and V_{nav} is the (desired) speed of the robot during normal navigation. Once ΔV has been computed, reference speeds are sent to the (velocity-regulated) motors according to

$$v_{\mathrm{L}} = V_{\mathrm{nav}} - \frac{\Delta V}{2}, \tag{6.6}$$

[8]See, however, the paper by Savage *et al.* mentioned in Footnote 6.

$$v_{\mathrm{R}} = V_{\mathrm{nav}} + \frac{\Delta V}{2}, \tag{6.7}$$

where v_{R} and v_{L} are the reference speeds of the left and right wheels, respectively. Note that one can of course only set the *desired* (reference) speed values; the actual speed values obtained depend on the detailed dynamics of the robot and its motors.

If the reference angle differs strongly from the estimated heading (which can happen, for example, in situations where the robot comes sufficiently close to an obstacle whose potential generates a steep hill), the robot may have to suspend its normal navigation and instead carry out a pure rotation, setting $v_{\mathrm{L}} = -V_{\mathrm{rot}}, v_{\mathrm{R}} = V_{\mathrm{rot}}$ for a left (counterclockwise) rotation, where V_{rot} is the rotation velocity, defined along with V_{nav} (and the other constants) during setup. In case the robot should carry out a clockwise rotation, the signs are reversed. The direction of rotation is, of course, determined by the sign of the difference between the reference angle and the (estimated) heading. In this case, the robot should turn until the difference between the reference angle and the estimated heading drops below a user-specified threshold, at which point normal navigation can resume.

6.3 Localization

In Sects. 6.1 and 6.2, it was (unrealistically) assumed that the robot's odometry would provide perfect estimates of the pose. In reality, this will never be the case, and therefore the problem of recalibrating the odometric readings, from time to time, is a fundamental problem in robotics. Doing so requires a method for localization independent from odometry, and such methods usually involve LRFs (even though cameras are also sometimes used), sensors that are difficult to simulate in ARSim (because of the large number of rays which would slow down the simulation considerably). Therefore, in this section, localization will be described as it is implemented in the simulator GPRSim and in GPRBS, where LRFs are used.

Robot localization requires two brain processes: The cognitive *Odometry* process and an independent process for odometric recalibration, which both in GPRSim and in GPRBS goes under the name *Laser localization*, since the behavior for odometric recalibration uses the readings of an LRF, together with a map, to infer its current location using scan matching, as described below.

In fact, the problem of localization can be approached in many different ways. For outdoor applications, a robot may be equipped with GPS, which in many cases will give sufficiently accurate position estimates. However, in indoor applications (standard) GPS cannot be used, since the signal is too weak to penetrate the walls of a building. Of course, it is possible to set up a local GPS system, for example by projecting IR beacons on the ceiling, using which

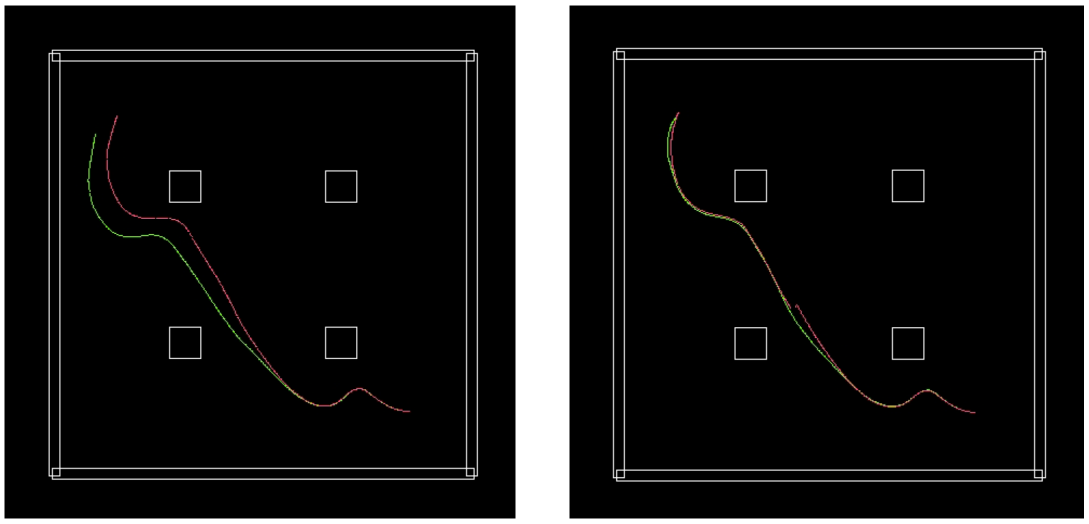

Figure 6.14: *An illustration of the need for localization in mobile robot navigation. In the left panel, the robot navigates using odometry only. As a result, the odometric trajectory (red) deviates quite significantly from the actual (green) trajectory. In the right panel,* Laser localization *was activated periodically, leading to much improved odometric estimates.*

the robot can deduce its position by means of triangulation[9]. However, such a system requires that the arena should be adapted to the robot, something that might not always be desirable or even possible.

The localization method (combining odometry and laser scan matching) that will be described here is normally used together with some navigation behavior. Thus, the robotic brain will consist of at least two motor behaviors, in which case decision-making also becomes important. This topic will be studied in a later chapter: For now, the *Laser localization* behavior will be considered separately.

6.3.1 Laser localization

The behavior is intended for localization in arenas for which a map has been provided to the robot (in the form of a sequence of lines). The map can either be obtained using a robot (executing a *Mapping* behavior) or, for example, from the floor plan of a building. The behavior relies on scans of the arena using a two-dimensional LRF and, like many methods for localization in autonomous robots, it assumes that all scans are carried out in a horizontal plane, thus limiting the behavior to planar (i.e. mostly indoor) environments. In fact, the name *Laser localization* is something of a misnomer: The behavior does not actually carry out (continuous) localization. Instead, when activated, the behavior takes as input the current pose estimate and tries to improve it. If

[9]This is the method used in the **Northstar**® system, developed by Evolution Robotics, inc.

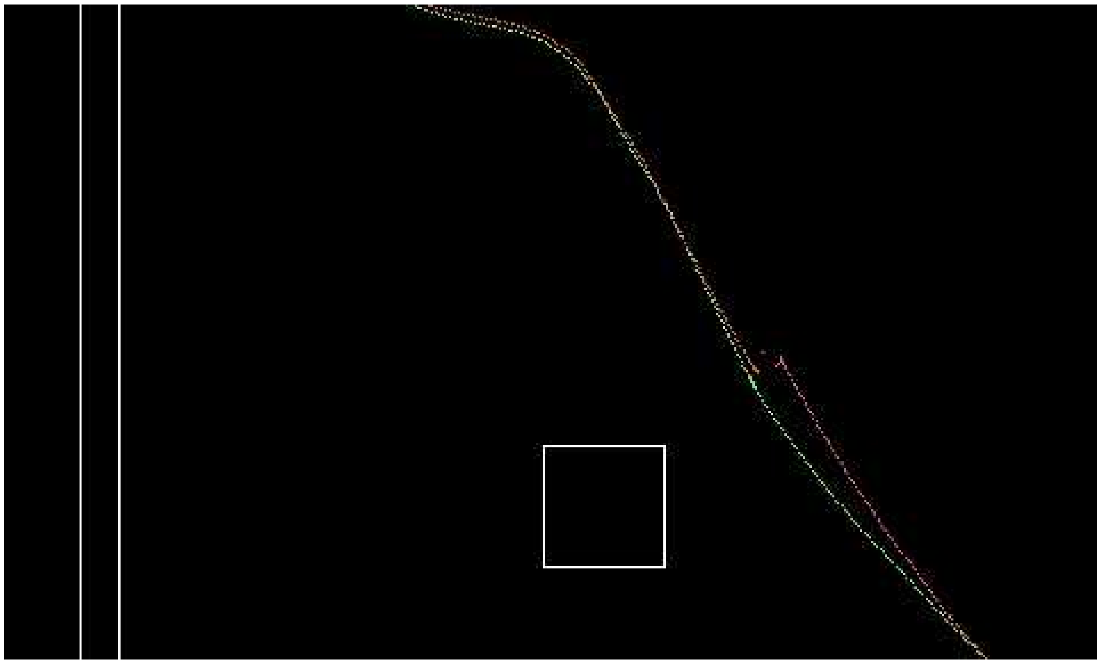

Figure 6.15: *An enlargement of the most significant correction in odometric readings (in the right panel of Fig. 6.14) resulting from the* Laser localization *behavior.*

successful, the odometric pose is reset to the position suggested by the *Laser localization* behavior.

The left panel of Fig. 6.14 illustrates the need for localization: The navigation task shown in Fig. 6.12 was considered again (with the same starting point, but a different initial direction of motion), this time with realistic (i.e. non-zero) levels of noise in the wheel encoders and, therefore, also in the odometry. As can be seen in the figure, the odometric drift causes a rather large discrepancy between the actual trajectory (green) and the odometric estimate (red). In the right panel, the robotic brain contained two behaviors (in addition to the cognitive *Odometry* process), namely *Potential field navigation* and *Laser localization*. The *Laser localization* behavior was activated periodically (thus deactivating the *Potential field navigation* behavior), each time recalibrating (if necessary) the odometric readings. As can be seen in the right panel of Fig. 6.14, with laser localization in place, the discrepancy between the odometric and actual trajectories is reduced significantly. At one point, the *Laser localization* behavior was required to make a rather large correction of the odometric readings. That particular event is shown enlarged in Fig. 6.15. As can be seen, the odometric readings undergo a discrete step at the moment of localization.

When activated, the localization behavior[10] considered here first stops the

[10]See Sandberg, D., Wolff, K., and Wahde, M. *A robot localization method based on laser scan*

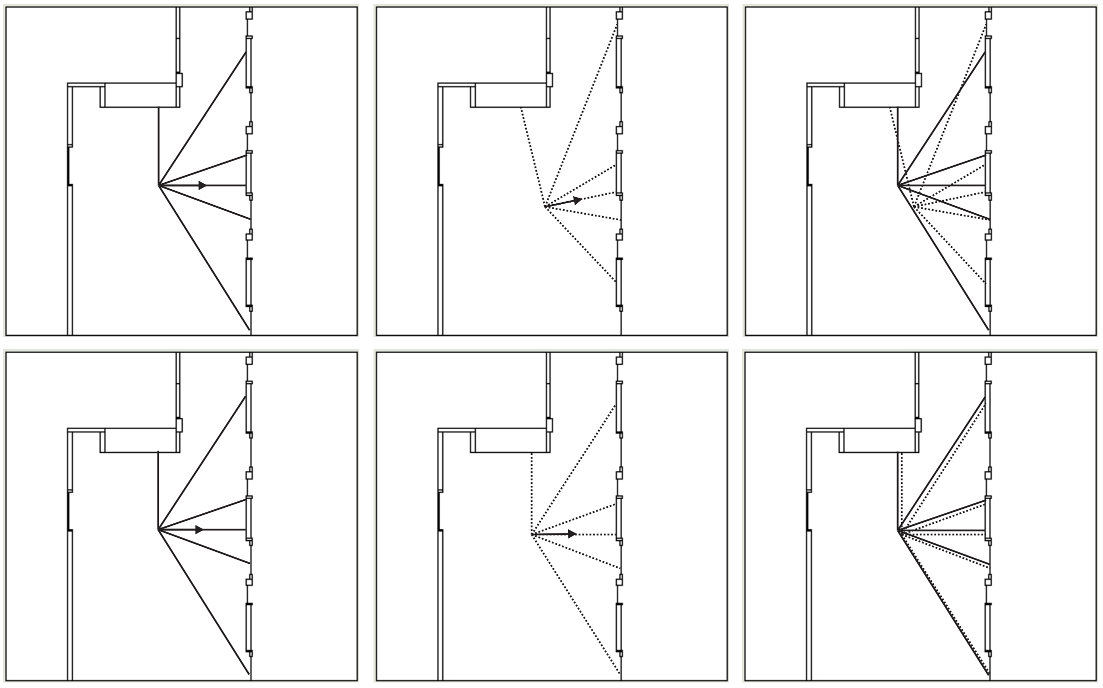

Figure 6.16: *Two examples of scan matching. The leftmost panel in each row shows a few rays (solid lines) from an actual LRF reading (plotted in the map used by the virtual LRF), and the middle panels show the virtual LRF readings (dotted lines) in a case in which the estimated pose differs quite strongly from the correct one (upper row), and one in which the difference is small (bottom row). The direction of heading is illustrated with arrows. The right panel in each row shows both the actual LRF rays and the virtual ones. The figure also illustrates the map, which consists of a sequence of lines.*

robot, and then takes a reading of the LRF. Next, it tries to match this reading to a *virtual* reading taken by placing a virtual LRF (hereafter: vLRF) at various positions *in the map*. Two examples of scan matching are shown in Fig. 6.16. The three panels in the upper row show a situation in which the odometry has drifted significantly. The upper left panel shows the readings (i.e. laser ray distances) from an actual LRF mounted on top of the robot (not shown). Note that, for clarity, the figure only shows a few of the many (typically hundreds) laser ray directions. The upper middle panel shows the readings of the vLRF, placed at the initial position and heading obtained from odometry. As can be seen in the upper right panel, the two scans match rather badly. By contrast, the three panels of the bottom row show a situation in which the pose error is small. The purpose of the search algorithm described below is to be able to correct the odometry, i.e. to reach a situation similar to the one shown in the bottom row of Fig. 6.16. Fig. 6.17 shows another example of a good (left panel) and a bad (right panel) scan match. In the case shown in the left panel, the

matching, Proc. of AMiRE 2009, pp. 171-178, 2009.

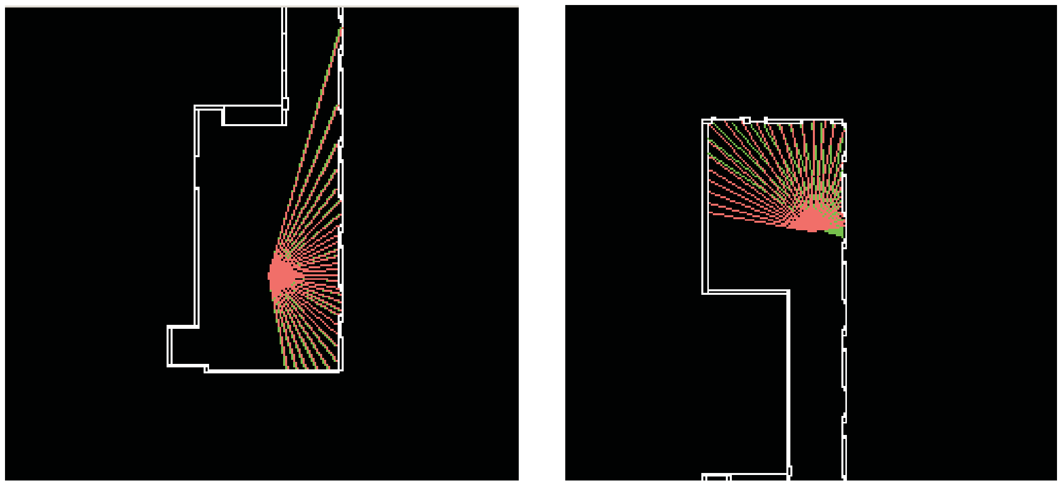

Figure 6.17: *Matching of LRF rays (in a different arena than the one used in the examples above). The readings of the actual LRF are shown in green, and those of the virtual LRF are shown in red. Left panel: An almost exact match. Right panel: In this case, the odometry has drifted enough to cause a large discrepancy between the actual and virtual LRF rays.*

odometric pose estimate is quite good, so that the rays from the actual LRF (green) match those of the vLRF quite well, at the current pose estimate. By constrast, in the situation shown in the right panel, the odometry has drifted significantly.

Scan matching algorithm

Let $\mathbf{p} = (x, y, \varphi)$ denote a pose (in the map) of the vLRF. The distances between the vLRF and an obstacle, along ray i, are obtained using the map[11] and are denoted δ_i. Similarly, the distances obtained for the *real* LRF (at its current pose, which normally differs from $\mathbf{p}$ when the localization behavior is activated) are denoted d_i.

The matching error ϵ between two scans can be defined in various ways. For rays that do not intersect an obstacle, the corresponding reading (d_i or δ_i) is (arbitrarily) set to -1. Such rays should be excluded when computing the error. Thus, the matching error is taken as

$$\epsilon = \sqrt{\frac{1}{\nu}\sum_{i=1}^{n} \chi_i \left(1 - \frac{\delta_i}{d_i}\right)^2}, \tag{6.8}$$

where n is the number of LRF rays used[12]. The parameter χ_i is equal to one

[11]In practice, the ray reading δ_i of the vLRF is obtained by checking for intersection between the lines in the map and a line of length R (the range of the LRF) pointing in the direction of the ray, and then choosing the shortest distance thus obtained (corresponding to the nearest obstacle along the ray). If no intersection is found, the corresponding reading is set to -1.

[12]For example, in the case of a Hokuyo URG-04LX LRF, a maximum of 682 rays are available.

for those indices i for which *both* the real LRF and the vLRF detect an obstacle (i.e. obtain a reading different from -1) whereas χ_i is equal to zero for indices i such that either the real LRF or the vLRF (or both) do not detect any obstacle (out to the range R of the LRF). ν denotes the number of rays actually used in forming the error measure, i.e. the number of rays for which χ_i is equal to one. As can be seen, ϵ is a measure of the (normalized) average relative deviation in detected distances between the real LRF and the vLRF.

Since d_i are given and δ_i depend on the pose of the vLRF, one may write $\epsilon = \epsilon(\mathbf{p})$. Now, if the odometric pose estimate happens to be exact, the virtual and actual LRF scans will be (almost) identical (depending on the accuracy of the map and the noise level in the real LRF), resulting in a very small matching error, in which case the localization behavior can be deactivated and the robot may continue its navigation. However, if the error exceeds a user-defined threshold T, the robot can conclude that its odometric estimates are not sufficiently accurate, and it must therefore try to minimize the matching error by trying various poses in the map, i.e. by carrying out a number of virtual scans, in order to determine the *actual* pose of the robot. The scan matching problem can thus be formulated as the optimization problem of finding the pose $\mathbf{p} = \mathbf{p}_v$ that minimizes $\epsilon = \epsilon(\mathbf{p})$. Once this pose has been found, the new odometric pose $\mathbf{p}^{\text{new}}$ is set equal to $\mathbf{p}_v$.

Note that it is assumed that the robot is standing still during localization. This restriction (which, in principle, can be removed) is introduced in order to (i) avoid having to correct for the motion occuring during the laser sweep, which typically lasts between 0.01 and 0.1 s and (ii) avoid having to correct for the motion that would otherwise take place during scan matching procedure, which normally takes a (non-negligible) fraction of a second. Thus, only *one* scan needs to be carried out using the real LRF mounted on the robot. The remaining work consists of generating virtual scans in the map, at a sequence of poses, and to match these to the actual LRF readings. Unlike some other scan matching methods, the method used here does not attempt to fit lines to the LRF readings. Instead, the LRF rays (actual and virtual, as described above) are used directly during scan matching. The sequence of poses for the vLRF is generated as follows: First the actual LRF scan is carried out, generating the distances d_i. Next, a virtual scan is carried out (in the map) at the current estimated position $\mathbf{p}_0$. If the error $\epsilon_0 = \epsilon(\mathbf{p}_0)$ is below the threshold T, localization is complete. If not, the algorithm picks a random pose $\mathbf{p}_j$ (where $j = 1$ in the first iteration) in a rectangular box of size $L_x \times L_y \times L_\varphi$, centered on $\mathbf{p}_0$ in pose space, and computes the matching error $\epsilon_j = \epsilon(\mathbf{p}_j)$. The constants L_x and L_y are typically set to around 0.1 m and the constant L_φ is set to around 0.1 radians.

The process is repeated until, for some $j = j_1$, an error $\epsilon_{j_1} < \epsilon_0$ is found. At this point, the rectangular box is re-centered to $\mathbf{p}_{j_1}$, and the search continues, now picking a random pose in the rectangular box centered on $\mathbf{p}_{j_1}$. Once a po-

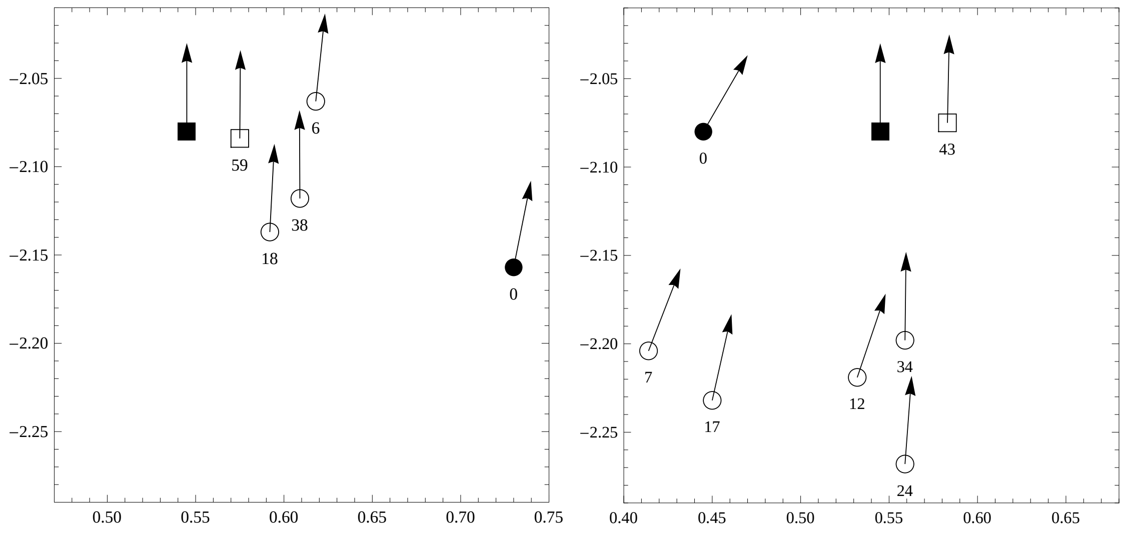

Figure 6.18: *An illustration of the sequence of poses generated during two executions of the search algorithm (with arrows indicating the direction of heading). In each panel, the actual position (measured in meters) of the robot is indicated with a filled square. The initial estimated position (i.e. from odometry, before correction) is shown as a filled disc, and the final estimated position is visualized as an open square. The intermediate points generated during the search are represented as open discs and are shown together with the corresponding iteration number. Note that, for clarity, only some of the intermediate points are shown.*

sition $\mathbf{p}_{j_2}$ is found for which $\epsilon_{j_2} < \epsilon_{j_1}$, the rectangular box is again re-centered etc. The procedure is repeated for a given number (N) of iterations[13].

Even though the algorithm is designed to improve both the position and the heading simultaneously, in practice, the result of running the algorithm is usually to correct the heading first (which is easiest, since an error in heading typically has a larger effect on the scan match than a position error), as can be seen clearly in the right panel of Fig. 6.18. At this stage, the estimated pose can make a rather large excursion in position space. However, once a fairly correct heading has been found, the estimated position normally converges quite rapidly to the correct position.

[13]Note that this algorithm resembles an evolutionary algorithm with a population size of 1.

Chapter 7

Utility and rational decision-making

The ability to make appropriate decisions in any given situation is clearly a necessary prerequisite for the survival of any animal and, indeed, even simple animals are generally able to take the correct action even in difficult situations, provided that they are operating in their natural environment. While the concept of **rational decision-making** has been studied in ethology (and, more recently, in robotics), the theory of rational decision-making was formalized within the framework of economics, particularly in the important work by von Neumann and Morgenstern[1]. Their work also remains one of the cornerstones of **game theory**.

The choices facing a decision-maker within the framework of economics can often be illustrated by means of **lotteries**, at least in cases where the number of **consequences** (or **outcomes**) is finite. Intuitively, one may think that the expected payoff, i.e. the amount of money that is likely to be gained by participating in the lottery, may determine a person's inclination to do so. However, things are a bit more complicated than that. As a first example, consider a lottery in which, with probability $p_1 = 0.5$ one would gain \$3 (i.e. with the **consequence** $c_1 = +\$3$), say, and, with probability $p_2 = 1 - p_1$, one would have to pay \$2. (outcome c_2 =-\$2). Thus, the expected payoff from this bet would be

$$P = p_1 c_1 + p_2 c_2 = 0.5 \times 3 - 0.5 \times 2 = 0.5. \tag{7.1}$$

Thus, it is likely that most people would accept this bet since the expected payoff is larger than zero. However, consider now a lottery with the same probabilities p_1 and p_2, but with the consequences c_1 =\$300,000 and c_2 =-\$200,000. In this case, the expected payoff would be \$50,000, a considerable amount of money, yet most people would be disinclined to accept the bet, given the risk of losing \$200,000.

[1]See von Neumann, J. and Morgenstern, O., *Theory of games and economic behavior*, Princeton University Press, 1944.

As a second example, consider a situation where a person must carry out a potentially lethal task in order to gain \$10,000. If the person's total wealth is \$0 it is possible that he would accept the task, regardless of the risk. On the other hand, if the person's total wealth is \$1,000,000, it would hardly be worth taking any risk for a measly additional \$10,000. Thus, clearly, the *amount* that can be gained is not the sole determinant, or even the most important one, when contemplating what action to take in the situations just described.

7.1 Utility

In order to underline further the fact that the expected payoff alone is not what determines one's inclination to accept a bet, consider a repeated lottery in which a fair coin (equal probability for heads and tails) is tossed repeatedly, and where the player receives 2^k dollars if the first head, say, occurs after k tosses of the coin. The probability p_k of this event occurring equals $(1/2)^k$. Thus, the expected payoff from playing this lottery (taking into account the cost r of entering the lottery) would be

$$P = -r + \sum_{k=1}^{\infty} p_k c_k = -r + \sum \left(\frac{1}{2}\right)^k 2^k = -r + \sum_{k=1}^{\infty} 1 \tag{7.2}$$

which is infinite! Thus, *if* the expected payoff P was all that mattered, a player should be willing to pay any finite sum of money, however large, in order to participate in this lottery, since the expected payoff would be larger. This is, of course, absurd; few people would be willing to bet their entire savings on a lottery. The situation just described is called the *St. Petersburg paradox*, and it was formulated by Bernoulli, who proposed a way of resolving the paradox, by postulating that, rather than the expected payoff itself, it is a player's *perception* of the amount of money gained that determines her actions. Bernoulli postulated that the subjective value of N currency units (e.g. dollars) varies essentially as the logarithm of N. Let W denote a person's wealth before participating in the lottery, and r the cost of entering the lottery. Using Bernoulli's postulate, the subjective value P_s of the expected payoff can then be written as the change in wealth for different outcomes, multiplied by the probability of the outcome in question, i.e.

$$P_s = \sum_{k=1}^{\infty} (\ln(W - r + 2^k) - \ln W) 2^{-k}, \tag{7.3}$$

which is finite. A person should, at most, be willing to pay an amount r that will make P_s positive, in order to participate in the lottery. For example, with $W = 1,000$, the maximum value of r would be around 10.95 currency units.

The subjective value of a certain amount of money, set arbitrarily to the logarithm of the amount by Bernoulli, is a special case of the concept of **utility**, which can be used for weighing different situations against each other and, thus, to decide which action to take.

In fact, it has been *proven* (rigorously) by von Neumann and Morgenstern[2] that, given certain assumptions that will be listed below, there exists a **utility function** which maps members c_i of the set of outcomes to a numerical value $u(c_i)$, called the utility of c_i, which has the following properties:

1. $u(c_1) > u(c_2)$ if and only if the person prefers[3] c_1 to c_2,

2. u is affine, i.e.

$$u\left(pc_1 + (1-p)c_2\right) = pu(c_1) + (1-p)u(c_2), \tag{7.4}$$

   for any value of $p_1 \in [0,1]$.

Furthermore, as shown by von Neumann and Morgenstern, u is unique up to a positive linear transformation, i.e. if a function v also describes a person's preferences, then $v = \alpha_1 u + \alpha_2$, where $\alpha_1 > 0$.

Clearly, there is no unique set of preferences, valid for all persons: one person may prefer a consequence c_1 to another consequence c_2, whereas another person's preferences may be exactly the opposite. Thus, utility tells us nothing about a person's preferences. However, it *does* tell us that, given that the preferences can be stated in a consistent way (see below), there exists a function u which can serve as a **common currency** in decision-making, i.e. when weighing different options against each other. As previously mentioned, the existence of a utility function with the properties listed above depends on certain axioms, namely

Axiom 1 (Ordering) Given two outcomes c_1 and c_2 an individual can decide, and remain consistent, concerning his preferences, i.e. whether he prefers c_1 to c_2 (denoted $c_1 > c_2$), c_2 to c_1, or is indifferent (denoted $c_1 \sim c_2$).

Axiom 2 (Transitivity) If $c_1 \geq c_2$ and $c_2 \geq c_3$ then $c_1 \geq c_3$.

Axiom 3 (The Archimedean axiom) If $c_1 > c_2 > c_3$, there exists a $p \in [0,1]$ such that $pc_1 + (1-p)c_3 > c_2$ and a $q \in [0,1]$ such that $c_2 > qc_1 + (1-q)c_3$.

[2]See von Neumann, J. and Morgenstern, O., *Theory of games and economic behavior*, Princeton University Press, 1944.

[3]If (and only if) the person is indifferent between c_1 and c_2, then $u(c_1) = u(c_2)$.

Axiom 4 (Independence) For all outcomes c_1, c_2, and c_3, $c_1 \geq c_2$ if and only if $pc_1 + (1-p)c_3 \geq pc_2 + (1-p)c_3$ for all $p \in [0,1]$.

If these four axioms hold[4] it is possible to prove the existence of a utility function with the properties listed above (the proof will not be given here). However, most decision-making situations involve uncertainty, i.e. many different outcomes are possible. Such situations can also be handled, since it follows (not completely trivially, though, at least not for the most general case) from the utility function properties listed above that the **expected utility** $U(c)$ of a mixed consequence $c = p_1c_1 + p_2c_2 + \ldots p_nc_n$, where p_k is the probability for consequence c_k, is given by

$$U(c) = \sum p_k u(c_k), \tag{7.5}$$

so that a consequence $c^{\mathrm{I}} = \sum p_k c_k$ is preferred to another consequence $c^{\mathrm{II}} = \sum q_k c_k$, if and only if $U(c^{\mathrm{I}}) > U(c^{\mathrm{II}})$.

Returning to the axioms above, it should be noted that none of them is trivial, and they have all been challenged in the literature. For example, the ordering and transitivity axioms may be violated in cases where a person has very vague preferences among a certain set of outcomes, or (more importantly) where the outcomes differ considerably in their implications, making it hard to express clear preferences.

The Archimedean axiom also leads to problems when some of the outcomes are extremely negative. For example if c_1 consists of winning \$2, c_2 consists of winning \$1, and c_3 means that the person is killed, it is quite evident that $c_1 > c_2 > c_3$. Thus, by axiom 3, there must be a value of p such that $pc_1 + (1-p)c_3 > c_2$, or, in other words, that the individual faced with the consequences c_1, c_2, and c_3 would prefer the possibility (however small) of death to the prospect of winning \$1 with certainty, if only the probability of winning \$2 is sufficiently close to 1. Given the small amounts of money involved it seems, perhaps, unlikely that one would accept even the slightest risk of being killed in order to gain a measly \$2. On the other hand, people *do* accept risking their lives on a daily basis, for example by driving a car, in order to gain time (or money).

If the four axioms are accepted, however, the resulting utility function can be used as a powerful tool in decision-making. In order to do so, one must first be able to construct the utility function, a procedure that will now be described by means of an example. Consider a case in which a person (A) sees a close friend (B) on the other side of a street, and contemplates whether or not to rush across the street to catch the friend. He envisions two possible consequences: Either c_1, in which he manages to cross the street safely, or c_2 in which he is

[4]Note that von Neumann and Morgenstern used slightly different axioms, which, however, amounted to essentially the same assumptions as those underlying axioms 1-4 listed here.

hit by a car and ends up in hospital. Clearly, c_1 is preferable to c_2, and let us suppose that A has assigned utilities[5] such that $u(c_1) = 5$ and $u(c_2) = -10$. By the affinity property of the utility function, the expected utility of the **mixed consequence** c_m where, say, the probability p of c_1 is 0.2 and the probability $q = 1 - p$ of c_2 is 0.8 (it is assumed that no other potential consequences exist, once the decision to cross the street has been made) equals

$$U(c_m) = U(pc_1 + qc_2) = pu(c_1) + qu(c_2) = 0.2 \times 5 + 0.8 \times (-10) = -7. \tag{7.6}$$

Using the same procedure, utility values can be derived for any mixed consequence (i.e. for any value of $p \in [0, 1]$).

Next, consider the consequence c of not trying to cross the street at all, and therefore not meeting B (who, let us say, owes A a considerable amount of money, which A expects to be able to collect upon meeting B). In that case, c_1 is preferred to c and (unless A is absolutely desperate to collect the debt) c is preferred to c_2. Thus

$$u(c_1) > U(c) > u(c_2), \tag{7.7}$$

but how should $U(c)$ be determined? Given some time to think, and by considering the two outcomes (c_1 and c_2) with known utility values, A will come up with a value of p at which he is indifferent between the mixed consequence $pc_1 + (1 - p)c_2$ and c. Because of the affinity property of the utility function, one then obtains

$$U(c) = U(pu(c_1) + (1 - p)u(c_2)) = pu(c_1) + (1 - p)u(c_2). \tag{7.8}$$

Let us say that A has a rather high tolerance for risk, so that the point of indifference occurs at $p = 0.1$. Then

$$U(c) = 0.1 \times 5 + 0.9 \times (-10) = -8.5. \tag{7.9}$$

Thus, the expected utility for the consequence c has now been determined, and the expected utility of any other consequence preferred to c_2 but not to c_1 can be computed in a similar way. Note that there is no right answer – the exact shape of the utility function depends on A's preferences. For example, if he were more cautious, he would perhaps conclude that the point of indifference occurs at $p = 0.5$ rather than $p = 0.1$, in which case $U(c) = -2.5$.

As a more formal example, consider a lottery in which a person (C) is indifferent between the consequence c_{30} of a certain gain of \$30 and a the mixed consequence of gaining \$100 with probability $p = 0.5$ and gaining nothing with probability $p = 0.5$. Assume further that C has assigned utility $u(c_0) = 0$ and $u(c_{100}) = 1$ to the consequences of gaining \$0 and \$100, respectively. Simplifying the notation somewhat by writing $u(x)$ for $u(c_x)$ one then finds

$$U(30) = 0.5 \times u(0) + 0.5 \times u(100) = 0.5. \tag{7.10}$$

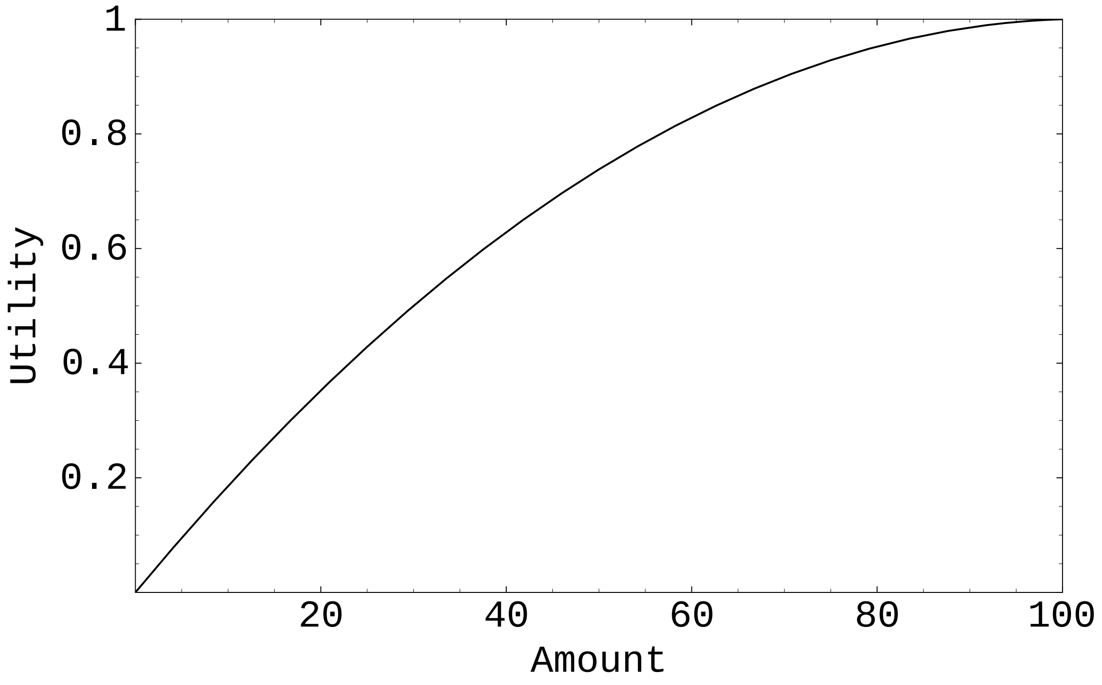

Figure 7.1: *A typical utility function, showing diminishing (sub-linear) utility values for larger amounts.*

In other words, the **certainty equivalent** for the mixed consequence in Eq. (7.10) is equal to \$30. Proceeding in a similar way, using certainty equivalents, the expected utility for any amount \$x in the range $[0, 100]$ can be determined, and the utility function can be plotted. A common shape of the resulting curve is shown in Fig. 7.1. The curve shows the utility function for a person who is **risk-averse**, i.e. who would prefer a sure payoff of x \$ to a lottery resulting in the same *expected* payoff. If a person is **risk-neutral** the utility function will be be a straight line through the origin and, similarly, for a **risk-prone** person, the utility function would bend upwards. Put differently, risk-aversion implies that the second derivative $U''(x)$ of the utility function is negative.

7.2 Rational decision-making

Once the utility functions for a given individual have been determined, the expected utility values can be used to guide behavior as follows: Given two possible actions a_1 and a_2, leading to the consequences c_1 and c_2, respecively, a **rational agent** will choose the action for which the corresponding expected utility is larger. Thus, if $U(c_1) > U(c_2)$, the agent would choose a_1, otherwise a_2 would be chosen. This behavior, in fact, *defines* a rational agent.

[5]The exact numerical values do not matter, as long as $u(c_1) > u(c_2)$.

At this point, it should be noted that, even if all the axioms necessary for the existence of a utility function holds true, it is not always so that utilities can be computed in a simple way. In the examples considered above, the **state of nature** was known, i.e. the various consequences and their probabilities could be listed, and all that remained was the uncertainty due to randomness. However, in many situations, the state of nature is not known, and it is therefore much more difficult to assign appropriate utility values to guide behavior. The general theory of decision-making under uncertainty is, however, beyond the scope of this text.

7.2.1 Decision-making in animals

Animals (including humans) generally face the problem of scarce resources, making the ability to choose between different activities completely central to survival. For example, a common problem is to allocate time to various essential activities in such a way as to keep physiological variables (e.g. hunger, thirst, temperature) within acceptable limits.

Clearly, even simple animals are capable of rational behavior. It should be noted however, that rational behavior does *not* require rational thought. There are many examples (one of which will be described below) of rational behavior in animals that simply lack the brain power to contemplate their sensory input in any detail (let alone maintain complex internal states). In such cases, rational decision-making occurs as a result of (evolutionary) design of the animal. Note that rational behavior does not automatically imply **intelligent behavior**. An agent is rational if it strives to maximize utility, but intelligent behavior will follow only if the animal's utility functions have been shaped (by evolution or as a result of learning) in an appropriate way.

As was discussed above, rational behavior amounts to the maximization of a quantity called utility that functions as a common currency for the comparison of different behaviors in any given situation. Put in a different way, a rational animal should switch between activities when (and only when) the switch leads to an increase in (marginal) utility. In ethology, it is customary to work with **cost** rather than utility. Thus, in ethology, the goals of the animal are commonly expressed in terms of the minimization of cost (e.g. energy expenditure). However, the term **benefit** (negative cost) which is also used in ethology, corresponds to utility in economics.

The physiological state of an animal can be represented by a point in a multi-dimensional physiological space, exemplified in the left panel of Fig. 7.2, in which limits for each variable (e.g. the level of some nutrient in the body) can be introduced. Similarly, the motivational state of the animal, which is generated by the combination of the physiological and perceptual states, can be represented as a point in a multi-dimensional space. In this space, **isoclines**

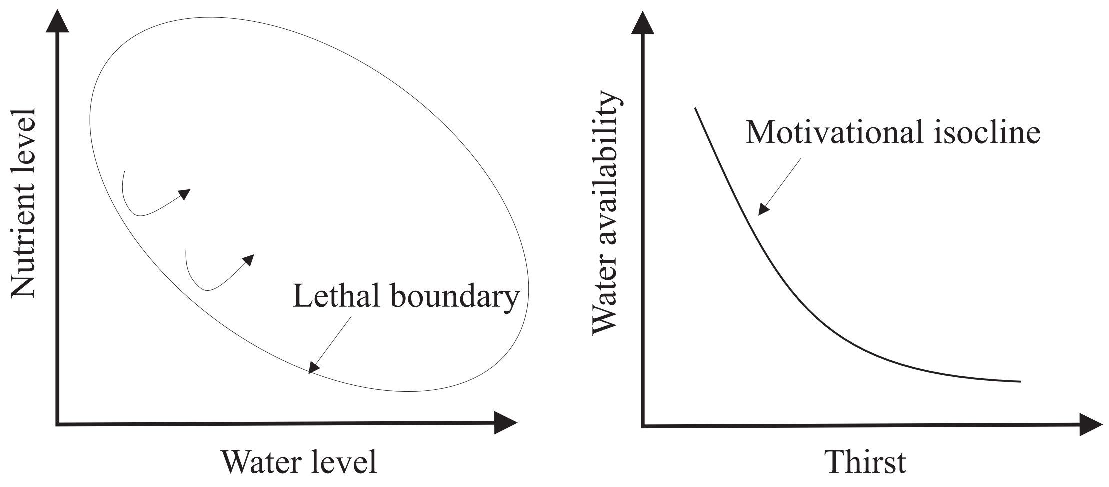

Figure 7.2: *Left panel: A physiological space. The curved arrows show typical trajectories followed as the animal comes sufficiently close to the lethal boundary. Right panel: A motivational space. The isocline joins points at which the tendency to perform the behavior in question (drinking, in this case) takes a given constant value.*

determining a given strength of the tendency to display a given behavior, can be introduced. A simplified, two-dimensional case, involving a single behavior, is shown in the right panel of Fig. 7.2. As the animal displays a given behavior, its location in the motivational space will change, as a result of, for example, a change in the perceived stimuli or its physiological state (e.g. satiation as a result of eating).

At the points in the (multi-dimensional) motivational space where two such isoclines cross each other, a switch in behavior (e.g. from eating to drinking) can be observed. An animal's motivations will generally be such as to keep it away from the lethal boundaries of the physiological space.

The problem of decision-making is made more complicated by the fact that the consequence of a given choice may be hard to assess (cognitively, or by design) accurately in an unstructured environment. Furthermore, the switch between two activities may involve a **cost of changing**, further complicating the decision-making process.

Decision-making in Stentor

Stentor is a simple, single-celled animal (see Fig. 7.3), that attaches itself to e.g. a rock, and uses its hair-like **cilia** to sweep nutrients into its trumpet-shaped mouth. Obviously, this animal has no nervous system, but is nevertheless capable of quite complicated self-preserving behaviors: Besides the feeding behavior (B_1), Stentor is equipped with four different avoidance behaviors (in response, for example, to the presence of a noxious substance). In increasing order of energy expenditure, the avoidance behaviors are (a) *turn-*

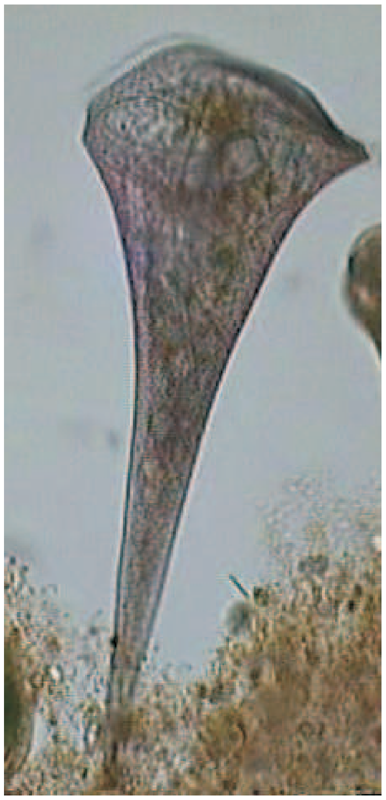

Figure 7.3: *A Stentor. Reproduced with kind permission of Dr. Ralf Wagner.*

ing away (B_2), (b) *reversing the cilia*, which interrupts the feeding activity (B_3), (c) *contraction, followed by waiting* (B_4), and (d) *detachment*, that is, breaking away from the object to which the animal is attached (B_5). Despite its simple architecture, Stentor is able to execute the various avoidance behaviors in a rational sequence, i.e. starting with B_2 and, if this behavor is insufficient to escape the noxious substance, proceeding with B_3 etc. However, the sequence of activation of the different avoidance behaviors is not fixed: Sometimes, B_2 is followed by B_4 instead of B_3 etc. How can such a simple animal be capable of such complex behavior, given its utter inability to reason about which activity to perform? It turns out, as described in detail by Staddon[6] that Stentor's behavior can be accounted for by a very simple model, involving several leaky integrator elements. A two-parameter leaky integrator is described by the equation

$$\frac{\mathrm{d}U}{\mathrm{d}t} + aU(t) = bX(t), \tag{7.11}$$

where a and b are constants and X is the external stimulus. Now, let U_i denote the utility associated with executing B_i, and set $U_1 = C =$ constant. For the

[6]See Staddon, J.E.R. *Adaptive dynamics*, MIT Press, 2001.

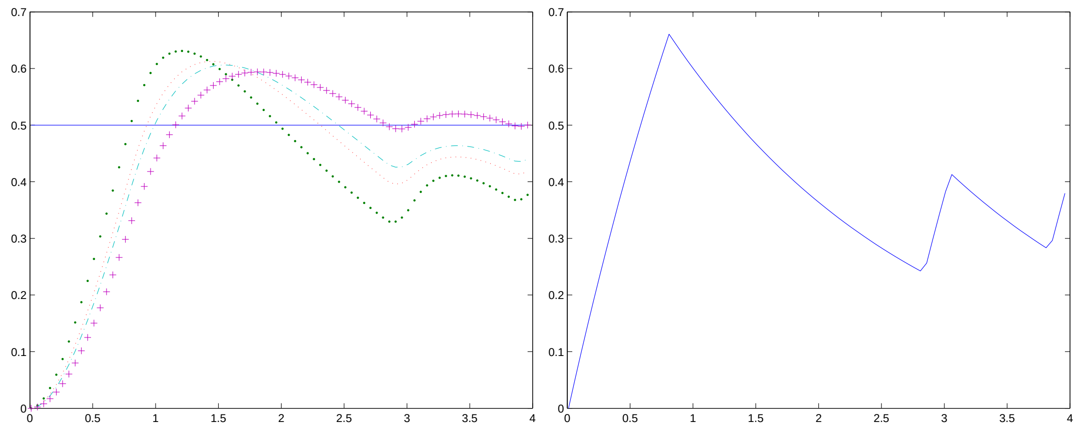

Figure 7.4: *Left panel: The variation of the utility functions for the 5 behaviors. Solid curve: B_1, large-dotted curve: B_2, small-dotted curve: B_3, dot-dashed curve: B_4, plus-sign curve: B_5. Right panel: The variation $X(t)$ of the concentration of the noxious substance.*

avoidance behaviors, let the utility functions be given by

$$\frac{\mathrm{d}U_i}{\mathrm{d}t} + a_i U_i(t) = b_i X(t), \quad i = 2, 3, 4, 5 \tag{7.12}$$

Now, given initial values of the utilities U_2, U_3, U_4, and U_5 (here all set, arbitrarily, to zero), and the variation of X with time, the utility for each behavior can be computed at all times. Using a utility-maximizing (rational!) procedure for behavior selection (i.e decision-making), where the behavior $B_{i_{\text{sel}}}$ corresponding to maximum current utility is selected, i.e.

$$i_{\text{sel}} = \operatorname{argmax}(U_i), \tag{7.13}$$

the behavior of Stentor, including the variable activation sequence of the avoidance behaviors, can be modelled quite accurately, by selecting appropriate values for the constants a_i and b_i. An example is shown in Fig. 7.4. Here, it was assumed that the variation $X(t)$ of the noxious substance can be modelled as

$$\frac{\mathrm{d}X}{\mathrm{d}t} + k_1 X = X_1, \tag{7.14}$$

if B_1 is active, and

$$\frac{\mathrm{d}X}{\mathrm{d}t} + k_2 X = X_2, \tag{7.15}$$

if any other behavior is active. k_1, k_2, X_1, and X_2 are non-negative constants, satisfying $X_1 > X_2$ (i.e. the amount of noxious substance tends towards higher values if no evasive action is taken). The left panel of the figure shows the variation of U_i for $i = 1, \ldots 5$. As can be seen in the figure, when X becomes sufficiently large, B_2 is activated for a short while. Next B_3, B_4, and B_5 are activated, in order. With the parameter values used in the example, the fall in X is quite slow. Thus, in subsequent activations of the avoidance behaviors, $B_2 - B_4$ are skipped, and the Stentor immediately activates B_5.

7.2.2 Decision-making in robots

The decision-making problems faced by autonomous robots are very similar to those faced by animals: an autonomous robot must complete certain tasks (e.g. delivery of objects in a factory), while avoiding collisions and while maintaining a sufficient level of energy in its batteries.

Guided by ethological results, using the principles of utility maximization (or cost minimization), McFarland[7] and McFarland and Bösser[8] have modelled elementary decision-making in robots, using quadratic cost functions. Furthermore, a general-purpose decision-making method based on utility functions has been developed at the Adaptive systems research group at Chalmers[9]. In this method (called the **utility function method**), which forms an integral part of the GPRBS framework, a utility function is assigned to each behavior, and the principle of utility maximization is then used for the activation of (motor) behaviors. This method will be studied in the next chapter.

[7]See McFarland, D. *Animal behaviour*, Addison-Wesley, 1999.

[8]See McFarland, D. and Bösser, T. *Intelligent behavior in animals and robots*, MIT Press, 1993.

[9]See Wahde, M. *A method for behavioural organization for autonomous robots based on evolutionary optimization of utility functions*, J. Systems and Control Engineering, **217**, pp. 249-258, 2003, and Wahde, M. *A general-purpose method for decision-making in autonomous robots*, Lecture notes in Artificial Intelligence, **5579**, pp. 1-10, 2009.

Chapter 8

Decision-making

8.1 Introduction and motivation

In Chapters 5 and 6, the topic of generating individual brain processes was studied. In complex robotic brains, however, there is more than one brain process available: The brains of such robots contain a **brain process repertoire** (also known, in BBR, as the **behavioral repertoire**) from which behaviors are dynamically chosen for activation. In GPRBS, the term **brain process** is used instead of behavior, and brain processes are, in turn, divided into **motor behaviors** (that make use of the robot's motor(s)) and **cognitive processes** (that do not make use of any motors).

A central issue in BBR, and in GPRBS, is **decision-making**[1], i.e. the process of determining which brain process(es) to activate at any given time. In this chapter, a brief introduction to the (vast) topic of decision-making will be given, with emphasis on a particular decision-making method based on utility functions.

8.1.1 Taxonomy for decision-making methods

A researcher aiming to generate a method for decision-making has a considerable amount of freedom and, not surprisingly, a very large number of such methods have appeared in the literature, such as e.g. the pioneering **subsumption method** which was developed by Rodney Brooks[2] and marked the starting point for research in behavior-based robotics. The purpose of any method for decision-making in BBR is to determine when different behaviors should be

[1]In BBR, decision-making is also known as **behavior selection**, **behavioral organization** or **behavior arbitration**. However, because of the distinction between cognitive brain processes and (motor) behaviors in GPRBS, the term *decision-making* will be preferred here.

[2]See, for example, Brooks, R. *A robust layered control system for a mobile robot*, IEEE Journal of Robotics and Automation, **RA-2**, pp. 14-23, 1986.

activated, and it is therefore natural to categorize methods of decision-making based on the procedure they use for selecting behaviors. There are two main categories, namely **arbitration methods** and **cooperative methods**[3]. In arbitration methods, exactly one (motor) behavior is active, and the selection of which behavior to activate is generally a function both of present sensor readings and the internal state of the robot. In cooperative methods, the action taken by the robot is a weighted combination of the actions suggested by several behaviors. The potential field navigation method, described in Chapter 6, can, in fact, be seen as a cooperative decision-making method, in which the direction of motion is obtained as an average of suggestions provided by the various potentials representing obstacles and the navigation goal.

However, here we shall only consider a single decision-making method, namely the utility function (UF) method that, in turn, is an integral part of the GPRBS framework. It should be noted that the choice of method (for this course) to a great extent reflects the author's preferences: There are many other methods for decision-making available, but they will not be considered here. Note also that, in the UF method, exactly *one* (motor) behavior is active, along with any number of cognitive processes. Thus, this method does not fall neatly into any of the two categories (defined above) for decision-making methods in BBR.

8.2 The utility function method

The **utility function method** is a biologically inspired decision-making method, intended mainly for motor tasks (such as, for example, navigation, delivery, map-building etc.) rather than tasks involving higher cognitive functions (e.g. interaction with people using speech synthesis and speech recognition). As its name implies, in this method, the selection of brain processes is based on the concept of utility described in Chapter 7. In the UF method, utility functions are used as a common currency for guiding the activation and de-activation of brain processes.

As an example, consider a floor-sweeping robot, which is given a fitness increment (i.e. an increase in its performance score) for each square meter of floor it sweeps. The robot should try to sweep as much floor as possible, in order to maximize its performance score. However, if the robot runs out of battery energy, it will no longer be able to move. Thus, the utility of a (motor) behavior that temporarily forces the robot to suspend its floor-sweeping activities to charge its batteries should rise as the energy in the battery decreases, allowing the robot to activate (based on the principle of utility maximization) the battery charging behavior before the battery energy drops to zero. Thus, even though

[3]Cooperative methods are sometimes also called **command fusion methods.**

the battery charging behavior does not lead to any *direct* increase in performance, it should nevertheless be activated, so that the robot can continue carrying out its floor-sweeping task when the batteries have been charged. Hence, in order to receive the highest possible performance score over a long period of time, the robot must, in fact, maximize *utility*.

Utility also provides a means of allocating limited resources in an optimal way. The life of any animal (or robot) inevitably involves many trade-offs, where less relevant behaviors must be sacrificed or at least postponed in order to carry out the most relevant behaviors, i.e. those associated with largest utility value. Next, a brief description of the UF method will be given, starting with the important concept of state variables.

8.2.1 State variables

Robots obtain information about the surroundings using their sensors or other input devices such as, for example, touch screens, keyboards, or microphones. Some sensors, for example infrared (IR) sensors, provide a scalar reading that can be used, for instance, in proximity detection. Other sensors generate vector-valued readings (e.g. laser range finders) or matrix-valued readings (e.g. digital cameras). In principle, all readings obtained from a robot's sensors could be used when specifying the state of the robot. For example, the rays (often several hundred) of a laser range finder (LRF) could, in principle, be used as state variables. However setting the parameters of a decision-making system involving hundreds of variables would be extremely difficult and, more importantly, highly undesirable: Even if such a decision-making system could be generated, it would be almost impossible to interpret and understand. Thus, in case the system fails (in certain situations), modifying it would be a very challenging task.

The human brain is capable of filtering out irrelevant information, in such a way that one is only consciously aware of information that is likely to be relevant for assessing the current situation. A similar approach can be used in robotics. Thus, rather than making decisions based on the massive raw data flow from all available sensors, in the UF method, a **sensory preprocessing system** (SPS) is introduced, which maps the raw data obtained through all the sensors (or a subset thereof) to a manageable number of **state variables**. A schematic illustration of an SPS is given in Fig. 8.1. Many different sensory preprocessing methods can be employed. As an example, a state variable[4] z may be defined as a weighted average of the readings obtained from a few IR sensors. Such a state variable can be used by the robot to avoid frontal collisions: A high value of z would indicate an imminent collision. Letting Σ_i

[4]The use of state variables will be described in connection with the description of utility functions below.

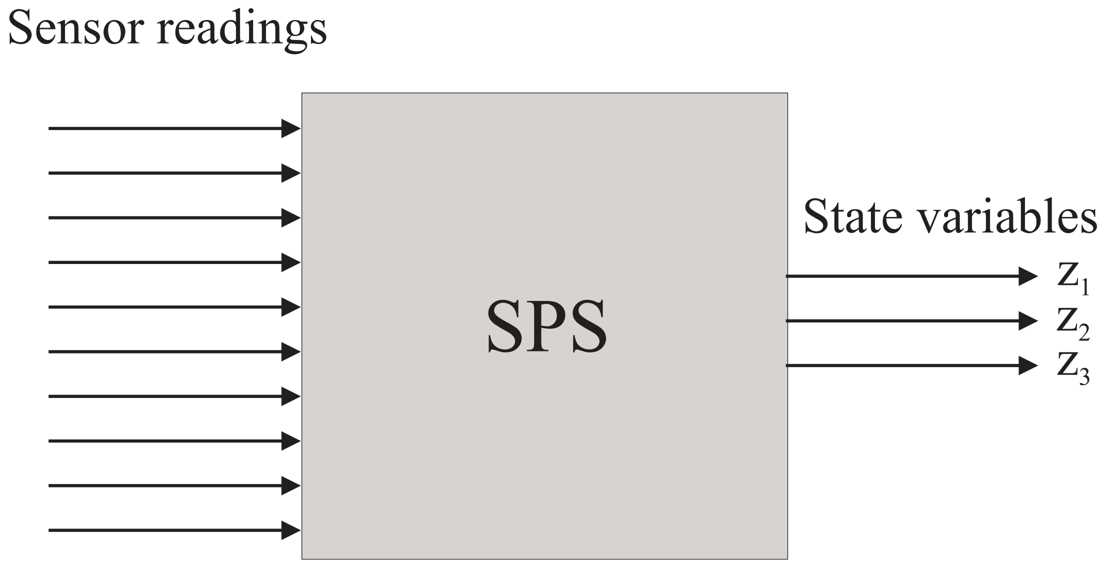

Figure 8.1: *A schematic representation of an SPS. In general, an SPS takes raw sensory readings as input and generates state variables as output.*

denote the reading of IR sensor i, a state variable can thus be formed as

$$z = \sum_{i=1}^{j} \nu_i \Sigma_i, \tag{8.1}$$

where j denotes the number of IR sensors (or a subset thereof) and $\nu_i, i = 1, \ldots, j$ is the weight for sensor i. Note that some weights may be set to zero[5]. For example, one may wish to define a state variable measuring proximity to obstacles on the side of the robot, in which case one should use some sensors on the left and right sides of the robot, while skipping sensors that point roughly in the robot's direction of heading.

Of course, the definition of state variables need not be limited to weighted averages. Another possible mapping is to use the *minimum* reading of a set or sensors

$$z = \min_{i \in [1,j]} \Sigma_i, \tag{8.2}$$

or the maximum

$$z = \max_{i \in [1,j]} \Sigma_i, \tag{8.3}$$

An even more general mapping can be defined by using a neural network, taking sensor readings as input, and generating a single scalar value as output.

An SPS normally contains several mappings, each of which produces a scalar output z_k. The state variables are then collected in a vector (denoted **z**) which is used as input to the utility functions (see below).

[5]Setting weights to zero is, of course, equivalent to not including the corresponding IR sensor at all. This is allowed: The sum in Eq. (8.1) does not have to include all IR sensors.

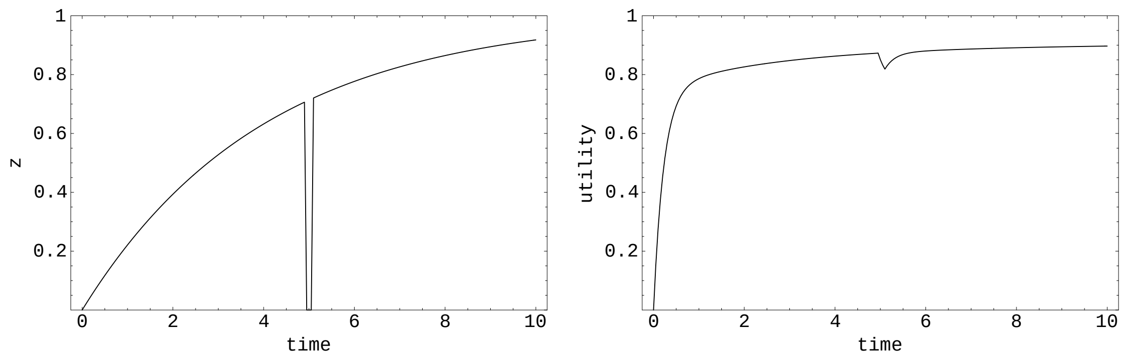

Figure 8.2: *An illustration of the noise tolerance resulting from the differential form of the utility functions. The left panel shows an example of signal loss in a state variable z, in turn caused by signal loss in the sensor(s) whose readings are used when forming the state variable: At around $t = 5$ s, the state variable suddenly dropped to zero for 0.15 s. Right panel: The corresponding change in a utility function, defined by $\tau\dot{u} + u = \sigma(b + wz)$, where $\tau = 0.3$, $b = 1.0$ and $w = 0.5$. The sigmoid parameter c was set to 1.0.*

8.2.2 Utility functions

In the UF method, each brain process is associated with a **utility function** that is intended to determine the relative merit of the brain process in any given situation. Utility values are constrained to the interval $[-1, 1]$. A high utility value u_i (i.e. around 1) of a brain process i indicates that the process is likely to be useful in the current situation, whereas a strongly negative value (i.e. around -1) indicates that the process should not be active. The utility u_i of brain process i is obtained by solving the differential equation

$$\tau_i \dot{u}_i + u_i = \sigma_i \left(\sum_{k=1}^{m} w_{ik} z_k + b_i + \Gamma_i \right) \; i = 1, \ldots, n. \tag{8.4}$$

Here, n denotes the number of brain processes, τ_i is a time constant determining the reaction time of the robot (typically set to around 0.1 s), m is the number of state variables[6], w_{ik} and b_i are tunable parameters, and $\sigma_i(x)$ is taken as

$$\sigma_i(x) = \tanh(c_i x), \tag{8.5}$$

where c_i is a positive constant. The squashing functions σ_i serve to keep utility values in the range $[-1, 1]$ provided, of course, that the values are initialized in this range (which they should be).

[6]In practice, Eq. (8.4) is discretized and integrated using a time step (typically around 0.01 s) much smaller than the smallest time constant. The most recent values of the state variables are used as inputs to the utility functions. Some state variables may change very frequently, whereas others (e.g. those based on LRF readings) are updated less frequently (typically with a frequency of around 10-30 Hz, in the case of LRFs).

Now, from the form of the utility functions given in Eq. (8.4) it is clear that the utility values will depend on the state variables z_k ($k = 1, \ldots, m$). Ideally, the state variables should provide the robot with all the information needed to make an informed decision regarding which brain processes to keep active in any situation encountered. However, in some cases, one may wish to directly activate or de-activate some brain process. For that reason, the Γ_i ($i = 1, \ldots, n$) parameters, henceforth simply referred to as **gamma parameters**, have been introduced (see Eq. (8.4)). The values of the gamma parameters are not obtained from the state variables, but are instead set directly by the brain processes. For example, a brain process j may, under certain circumstances, set the parameter Γ_i of some brain process i either to a large positive value (in order to raise the utility u_i, so as to activate brain process i) or a large negative value (to achieve the opposite effect, i.e. de-activation of brain process i). However, once the intended result has been achieved, Γ_i should return to its default value of zero. This is achieved by letting Γ_i vary (at all times) as

$$\tau_i^{\Gamma} \dot{\Gamma}_i = -\Gamma_i \tag{8.6}$$

where τ_i^{Γ} is a time constant[7], determining the decay rate of Γ_i. Thus, in the normal situation Γ_i is equal to zero, whereas if some brain process abruptly sets Γ_i to a value different from zero, it subsequently falls off exponentially.

The differential form of Eq. (8.4) has the important effect of filtering out the noise which is always present in sensor readings and therefore also in the state variables. Thus, for example, if a given sensor fails to give readings for a brief interval of time (a few ms, say), the corresponding state variable will drop to zero. However, because of the time constant (usually of order 0.1 - 0.3 s) in the utility functions, a brief interruption in the sensor readings will only cause a small fluctuation in the corresponding utility value, as illustrated in Fig. 8.2.

8.2.3 Activation of brain processes

As is evident from Eq. (8.4), the utility values will depend on the values of the parameters w_{ik}, b_i, τ_i and c_i, as well as any non-zero values of the Γ_i parameters (set by the brain processes) and their subsequent decay, as specified by the τ_i^{Γ} parameters. Furthermore, the precise mappings used in the SPS (e.g. the weights ν_i) will affect the state variable values z_k and therefore also the utility functions.

The construction of a decision-making system in the UF method consists of (i) setting the parameters of the utility functions, as well as specifying (ii) the mappings constituting the SPS and (iii) the use (if any) of the Γ_i parameters, so as to reach the desired overall result. In many cases, this is in fact easier

[7]The superscript (which is *not* an exponent!) is introduced in order to distinguish this time constant from τ_i defined in Eq. (8.4).

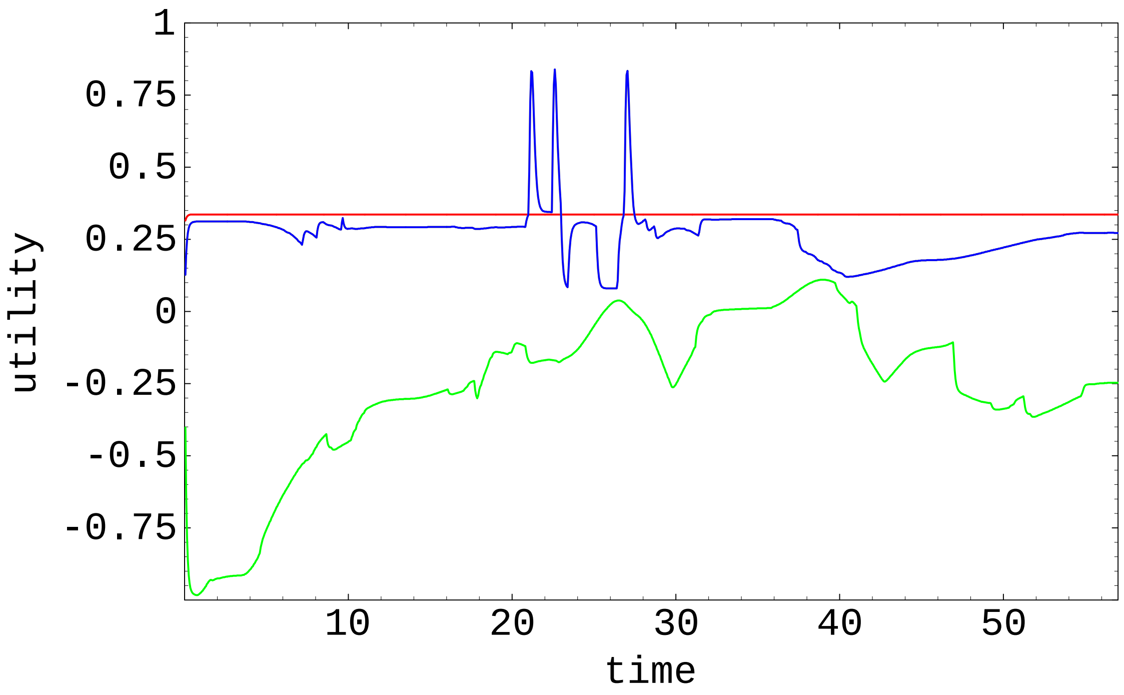

Figure 8.3: *An example of utility function dynamics, showing the variation (with time) of three utility functions u_1 (red), u_2 (green) and u_3 (blue).*

than it may seem. However, in complex applications, it may be difficult to set appropriate parameter values by hand. One may then apply optimization using, for example, stochastic optimization algorithms such as particle swarm optimization (PSO) or genetic algorithms (GAs).

In any case, assuming that one has been able to find appropriate values for the parameters (either by hand or using some optimization method), the form of the state variables and the utility functions is thus determined. The decision-making is then quite straightforward in the UF method: Among the cognitive processes, any such process with positive utility is active. Furthermore, the (motor) behavior with highest utility is active, and all other (motor) behaviors are inactive. Of course, this does not imply that the robot will constantly be moving: A motor behavior may also set the desired motor torques (or set speeds) to zero, allowing the robot to stand still. It should be noted that the utility functions of *all* brain processes are updated continuously, so that an inactive (motor) behavior can be activated, should its utility value exceed the utility of the currently active (motor) behavior. Similarly, an inactive cognitive process can become active if its utility reaches a value above zero.

An illustration of the typical dynamics of utility functions is shown in Fig. 8.3. The figure shows the variation (over time) of three utility functions associated with motor behaviors. At any given time, the behavior with highest utility is active. In the particular case shown in the figure the utility (u_1) for one of the behaviors (B1) was essentially constant. Since it is only the relative utility values that matter, it is common to let one such process have (asymptotically) constant utility values, by setting all the corresponding weights w_{ik} to zero. However, for the other two behaviors, B2 and B3, whose utility functions u_2 and u_3 are shown as green and blue curves, respectively, some weights w_{ik} were non-zero. In the particular case shown here, B1 was active most of the time, the only interruptions occurring around $t = 21$ s and $t = 27$ s, at which points B3 was activated instead, the first time for around 2.5 s and the second time for less than 1 s. In the time interval covered by the figure, B2 was never activated.

As mentioned above the UF method is mainly intended for motor tasks, and it has been successfully applied in several tasks of that kind, both in simulated and real robots[8]

[8] A specific example can be found in: Wahde, M. *A general-purpose method for decision-making in autonomous robots*, LNCS, **5579**, 1-10, 2009.

Appendix A:
Matlab functions in ARSim

The following is an alphabetical list of all the Matlab functions associated with `ARSim`. For each function, the library to which the function belongs is given, along with the interface of the function and a brief description. For further information, see the actual source code for the function in question.

AddMotionResults

Library: ResultFunctions
Interface:
`motionresults = AddMotionResults(oldMotionResults, time, robot)`
Description: This function updates the motion results by adding the current position, velocity, heading, and sensor readings of the robot.

BrainStep

Library: –
Interface: `b = BrainStep(robot, time);`
Description: The `BrainStep` implements the decision-making system (i.e. the brain) of the robot. The detailed form of this function will vary from experiment to experiment.

CalibrateOdometer

Library: RobotFunctions
Interface: `o = CalibrateOdometer(robot)`
Description: In simulations in which an odometer is used, a call to `CalibrateOdometer` is made just before the start of the simulation, in order to set the correct position and heading of the robot.
See also: `CreateOdometer`

CheckForCollisions

Library: RobotFunctions
Interface: `coll = CheckForCollisions(arena, robot);`
Description: This function carries out a collision check, by running through all arena objects (polygons) line by line, and checking for intersections between the current line and the spherical body of the robot.

CreateArena

Library: ArenaFunctions
Interface: `arena = CreateArena(name,size,objectArray)`
Description: This function generates an arena, given an array of arena objects.
See also: `CreateArenaObject`

CreateArenaObject

Library: ArenaFunctions
Interface: `arenaobject = CreateArenaObject(name,vertexArray)`
Description: This function generates an arena object, given an array of coordinates for vertices.

CreateBrain

Library: –
Interface: `b = CreateBrain;`
Description: This function generates the brain of a robot. Its exact form will vary from experiment to experiment.

CreateCompass

Library: RobotFunctions
Interface: `c = CreateCompass(name,sigma);`
Description: This function generates a compass which can be used for estimating the heading of the robot. The parameter `sigma` determines the noise level.

CreateIRSensor

Library: RobotFunctions
Interface:
```
s = CreateIRSensor(name,relativeAngle,size,numberOfRays,
                   openingAngle,range,c1,c2,sigma);
```

Description: `CreateIRSensor` creates an IR sensor that uses the ray tracing procedure described above to obtain its readings. The parameter `sigma` is defined as in Eq. (3.1).

CreateMotor

Library: RobotFunctions
Interface: `m = CreateMotor(name);`
Description: `CreateMotor` generates a DC motor, using settings suitable for a robot with a mass of a few kg.

CreateOdometer

Library: RobotFunctions
Interface: `o = CreateOdometer(name, sigma);`
Description: This function generates an odometer, which, in turn, provides estimates for the position and heading of the robot. The parameter `sigma` determines the noise level.

CreateRobot

Library: RobotFunctions
Interface:
```
robot = CreateRobot(name,mass,momentOfInertia,radius,
                    wheelRadius,rayBasedSensorArray,
                    motorArray,compass,odometer,brain)
```
Description: `CreateRobot` sets up a robot, and computes the dynamical parameters typical of a robot with a mass of a few kg.

GetCompassReading

Library: RobotFunctions
Interface: `c = GetCompassReading(robot, dt);`
Description: This function updates the compass readings of a robot.

GetDistanceToLineAlongRay

Library: RobotFunctions
Interface: `l = GetDistanceToLineAlongRay(beta,p1,p2,x1,y1);`
Description: This function, which is used by the IR sensors, computes the distance from a given point (x_1, y_1) to a line segment.
See also: `GetIRSensorReading`, `GetDistanceToNearestObject.`

GetDistanceToNearestObject

Library: RobotFunctions
Interface: `d = GetDistanceToNearestObject(beta, x, y, arena);`
Description: This function, which is used by the IR sensors, determines the distance between an IR sensor and the nearest object along a given ray.
See also: `GetIRSensorReading`.

GetIRSensorReading

Library: RobotFunctions
Interface: `s = GetIRSensorReading(sensor,arena);`
Description: `GetIRSensorReading` determines the reading of an IR sensor.

GetMinMaxAngle

Library: RobotFunctions
Interface: `[aMin,aMax] = GetMinMaxAngle(v1,v2);`
Description: This function determines the direction angles of the vectors connecting the origin of the coordinate system to the tips of a line segment.
See also: `GetDistanceToNearestObject`.

GetMotorSignalsFromBrain

Library: RobotFunctions
Interface: `s = GetMotorSignalsFromBrain(brain);`
Description: This function extracts the motor signals (one for each motor) from the brain of the robot.
See also: `MoveRobot`.

GetOdometerReading

Library: RobotFunctions
Interface: `o = GetOdometerReading(robot, dt);`
Description: This function updates the odometer readings of a robot.

GetRayBasedSensorReadings

Library: RobotFunctions
Interface: `s = GetRayBasedSensorReadings(robot, arena)`
Description: This function obtains the reading of all (IR) sensors of the robot.
See also: `GetIRSensorReading.`

GetTorque

Library: RobotFunctions
Interface: `m = GetTorque(motor, voltage);`
Description: This function determines the torque delivered by a DC motor, given a value of the applied voltage.

InitializeMotionResults

Library: ResultFunctions
Interface: `motionResults = InitializeMotionResults(robot)`
Description: This function initializes a Matlab structure used for storing the results of the simulation, i.e. the position, velocity, heading, and sensor readings of the robot.

InitPlot

Library: PlotFunctions
Interface: `plotHandle = InitializePlot(robot, arena)`
Description: This function generates the plot of the robot and the arena.
See also: `CreateArena`, `CreateRobot`.

MoveRobot

Library: RobotFunctions
Interface: `r = MoveRobot(robot,dt);`
Description: `MoveRobot` moves the robot according to the equations of motion for a differentially steered two-wheeled robot.

ScaleMotorSignals

Library: RobotFunctions
Interface: `v = ScaleMotorSignals(robot,s);`
Description: This function scales the motor signals (`s`) to the appropriate range, as set by the voltage requirements of the robot's DC motors.

SetPositionAndVelocity

Library: RobotFunctions
Interface:
```
r = SetPosition(robot,position,heading,
                velocity,angularSpeed);
```
Description: This function places the robot at a given location, and also sets is direction of motion, velocity, and angular velocity.

ShowRobot

Library: PlotFunctions
Interface: `ShowRobot(plot,robot)`
Description: `ShowRobot` updates the plot of the robot using Matlab's handle graphics: Each part of the plot of the robot can be accessed and its position can be be updated. `ShowRobot` also supports the plotting of an odometric ghost, i.e. a plot showing the robot at the location determined by its odometer.
See also: `MoveRobot`.

UpdateMotorAxisAngularSpeed

Library: RobotFunctions
Interface: `r = UpdateMotorAxisAngularSpeed(robot)`
Description: This function determines the angular speed of each motor axis, using the wheel speed and wheel radius.

UpdateSensorPositions

Library: RobotFunctions
Interface: `s = UpdateSensorPositions(robot);`
Description: This function updates the positions (and directions) of the sensors as the robot is moved.