1 Background

1.1 Why Machines should learn

1.1.1 AI and machine learning

The ability of the human mind has marvelled scientists and philosophers alike for many centuries and has continuously inspired our quest for intelligent machines. The famous Dartmouth conference in 1950 brought together leading minds in this area, and this is where the label **Artificial Intelligence**(AI) was coined for this emerging scientific area. AI has become to be used as a synonym for more advanced – and sometimes even human-like – information processing with a large array of unique methods and diverse approaches. AI is closely related to cognitive science, the science of thinking systems, which has itself strong links to psychology on one side and computer science or informatics on the other end of the spectrum.

A major focus of research considers symbolic systems which includes such areas as knowledge representation, expert systems and semantic networks. This area employs formal logic for inference, where inference mean here the act of using observations to argue about situations and to draw conclusions that guide further behaviour. For example, we might be concerned with the navigation of vehicles in traffic. The vehicle 'car' could thereby perform an action such as 'move' or 'stop' depending on a condition such as a 'traffic light'. The states of the logic variable 'traffic light' can have several states such as 'green' or 'red'. Describing systems with logical expressions is central in such symbolic systems.

Symbolic systems can be contrasted with explicit systems that work on raw data. Such explicit systems are concerned with explicit sensory information data like a visual system and how to translate them to higher level knowledge that are typically the starting point of symbolic reasoning systems. We are mainly concerned here with such explicit systems. Of course, the boundaries of these systems is sometimes blurry, and ultimately we think that both approaches will need to merge.

A central theme in most of these areas is the ability to learn from data and experiences. Indeed, the ability of humans to find solutions to new problems from experience, and the robustness of such systems in uncertain environments, are increasingly considered to be a key ingredient to advance these research areas and to enable new application. We focus here on how machines can learn from data to improve their performance or learn how to solve problems. This should enable advanced machine applications, and considering the theory of learning in general can help us to shed light on how human learn. Ultimately this area, which is now generally called **Machine Learning**, should be able to bridge gap between the explicit and the symbolic approach in AI by learning to meaning from data. A prime example in this area is object recognition. Machine Learning is now a widely respected scientific area with a growing number of applications, and algorithms cmd out of this research have enabled

new approaches to information processing that made it possible to create new consumer electronics and lead to recent breakthroughs in autonomous robotics. Machine Leaning is now a major enabler driving many business solutions.

While the field of Machine Learning has matured considerably in the last decade, its roots and early breakthroughs are firmly grounded in the 1950s. Indeed, the history of AI is tightly interwoven with the history of machine learning. For example, Arthur Samuel's checkers program from the 1950s, which has been celebrated as an early AI hallmark, was able to learn from experience and thereby was able to outperform its creator. Basic Neural Networks, such as Bernhard Widrow's ADALINE (1959), Karl Steinbuch's Lernmatrix (around 1960), and Frank Rosenblatt's Perceptron (late 1960s), have sparked the imaginations of researcher about brain-like information processing systems. And Richard Bellman's Dynamic Programming (1953) has created a lot of excitement during the 1950s and 60s, and is now considered the foundation of reinforcement learning. We will detail some of these approaches in this book.

Biological systems have often been an inspiration for understanding learning systems and visa versa. Donald Hebb's book *The Organization of Behavior* (1943) has been very influential not only in the cognitive science community, but has been marvelled in the early computer science community. While Hebb speculated how self-organizing mechanisms can enable human behavior, it was Eduardo Caianiello's influential paper *Outline of a theory of thought-processes and thinking machines* (1961) who quantified these ideas into two important equations, the neuronic equation, describing how the functional elements of the networks behave, and the memnonic equation that describe how these systems learn. This opened the doors to more theoretical investigations. But such investigations came to a sudden halt after Marvin Minsky and Seymore Papert published their book *Perceptrons* in 1969 with a proof that simple perceptrons can not learn all problems. At this time, likely somewhat triggered by the vacuum in AI research by the departure of learning networks, mainstream AI shifted towards symbolic systems for much of the 1970s and early 1980s.

Explicit learning systems the form of Neural Networks became again popular in the mid 1980 after the backpropagation algorithms was popularized by David Rumelhart, Geoffrey Hinton and Ronald Williams (1986). There was a brief period of extreme hype with claims that Neural Networks could soon forecast the stock-market and how these learning systems would quickly become little brains. The hype backslashed somewhat. Neural Networks predictive power in scientific explanations became questioned as they seemed to conveniently fit any data and since claims of immanent progress did not substantiate. It turned out that there are two major problems with the basic networks studied in the late 1980s which are **generalizability** and **scalability**. Generalizability refers to the ability to predict previously unseen data in contrast to mainly memorizing training examples. Scalability refers to the ability of such systems to be used on large real world problems in contrast to simple examples often used to illustrate their behaviour.

However, there has been major progress in the understanding of learning systems since the late 1980s through several factors. A major contributed was a better grounding of learning methods in statistical learning theory with more rigorous mathematical insights. A good example is the work by Vladimir Vapnik as published in his book *The Nature of Statistical Learning Theory* in 1995. And more general Bayesian methods

Fig. 1.1 Some pioneers in AI and machine learning.From top left to bottom right: Alan Turing has thought much about AI in general and has foreseen much of its development. Arthur Lee Samuel was an influential early AI researcher and created the first checker program that could learn and was able to beat its creator. Such reinforcement learning was formalized in the 50s by Richard Bellman. Many universal learning machines are inspired by brain functions. Frank Rosenblatt invented the perceptron, and Geoffrey Hinton has made many important contributions, specifically to probabilistic neural networks. Finally, Judea Pearl invented graphical models to describe probabilistic causal systems that are today at the heart of many learning and inference systems.

and graphical models (Judea Pearl, 1985) have clarified and transformed the field. Unsupervised learning, in particular sparse structural learning, is an exciting new area in machine learning. And even more recently it has been shown that deep learning systems, which are basically the good old Neural Networks with a few more tricks and insight applied to larger problems on fast GPUs are now starting to outperform many other systems. It is an exciting time for Machine Learning with major breakthoughts and increasing applications.

1.1.2 Supervised, unsupervised and reinforcement learning

Traditional AI provides many useful approaches for solving specific problems. For example, search algorithms can be used to navigate mazes or to find scheduling solutions, and expert systems can manage large databases of expert knowledge and use this data to argue about (infer) specific solutions. A drawback of such system is that they often require a well defined and structured environment. Learning systems, in contrast, are thought to be a possible approach to situations for which closed solutions are not known. A good example are situations where systems change over time or when systems encounter situations for which they were not designed such as robotics

systems that are often helpless when employed outside their common environment. There is thus an increasing desire to include methods in computing systems that are able to adapt to changing situations and generalizations to unseen environments or unforeseen circumstances. While many AI systems have adaptive components, we are specifically focusing on the theory of learning machines. We will distinguish thereby various learning circumstances such as having a detailed teacher or only receiving feedback after some time. Much of the theoretical investigation is concerned with what would be a good or even optimal solution, at least to give us some perspective of what is possible.

Learning machines are supposed to learn from the environment, either through instructions, by reward or punishment, or just by exploring the environment and learning about typical objects and relations in space and time. For the systematic discussion of learning systems it is useful to distinguish three types of learning circumstances, namely *supervised learning*, *reinforcement learning*, and *unsupervised learning* as briefly outlined next.

Supervised learning is characterized by using explicit examples with labels that the system should use to learn to predict labels of previously unseen data. The training labels can be seen as being supplied by a teacher who tells the system exactly the desired answer in response to a specific situation as specified by a particular input to the learning system. For example, an important requirement of natural or artificial agents is its ability to decide on an appropriate course of action given a specific situation. The specific circumstances are communicated to the agent by sensors that specify values of certain features. Let's represent these **feature values** as vector $\mathbf{x}$. The goal of the agent is then to determine an appropriate response (label y) based on this input,

$$y = f(\mathbf{x}). \tag{1.1}$$

This corresponds to the functional form of a controller that we will discuss further in Chapter 2. In Chapter 3 we argued that a probabilistic framework is more appropriate to address uncertainties. The corresponding statement of the deterministic function approximation of equation (1) is then to find a probability density function

$$p(y|\mathbf{x}). \tag{1.2}$$

A common example is object recognition where the feature values might be RGB values of pixels in a digital image and the desired response might be the identification of a person in this image. A learning machines for such a task is a model that is given examples with specific feature vectors $\mathbf{x}$ and corresponding desired **labels** y. Learning in this circumstance is mainly about adjusting the model's parameters from the given examples. Since this type of learning is based on specific training examples with give labels, this type of learning is called **supervised**. Adjusting the parameters should thereby be guided by the best performance of generalize, that of predicting appropriate labels of previously unseen feature vectors.

Reinforcement learning is somewhat similar in that the system receives some feedback from the environment, but this feedback is typically delay in time and does not specify which of the previously taken actions contributed to a success or failure. A main challenge of reinforcement learning is hence to solve the **credit assignment**

problem, and the goal of reinforcement learning is to discover the sequence of actions which maximizes reward over time.

The third form of learning that we will distinguish from the former two is **unsupervised learning**. The aim of this learning is to find useful representations of data based on regularities of data without labels. This includes clustering methods and more sophisticated data-transformation methods to find appropriate new data representations. Finding an appropriate representation of data using unsupervised learning methods is often a key in solving supervised learning problems. While it is useful to distinguish such classical learning systems, they can also augment each other. Examples are semi-supervised learning methods or the combination of supervised and reinforcement learning methods.

Learning systems can help to solve problems for which more direct solutions are not known. This is particularly true for systems that are **unreliable** or applied in **uncertain** environments. For example, a program might read data in a specific format, but some user might supply corrupted files. Software used for public release is often lengthy mostly to consider all kind of situations that could occur. However, it is also increasingly realized that considering all possible situations is often impossible. The situation of unreliable inputs is very apparent in robotics where sensors have often severe limitations. We will see that estimating the state of the system is a major challenge in robotics due to limited data, the inability to process data sufficiently in time, or due to limited resources.

Major progress in many AI areas, in particular in robotics and machine learning, has been made by using concepts of (Bayesian) probability theory. This language of probability theory is appropriate since it acknowledges the fundamental limitations we have in real world applications (such as limited computational researches or inaccurate sensors). The language of probability theory has certainly helped to unify much of related areas and improved communication between researchers. Furthermore, we will see that the representation of uncertain states as a probabilistic map will be very useful. Also, probability theory will provide us with an elegant and powerful solution for a basic computational need common in learning systems, that of combining prior knowledge with new evidence.

1.1.3 Regression and classification

It is useful to outline at this point more specifically the essence of supervised learning with regards to probabilistic regression and classification. As pointed out above, the goal of supervised learning is to make predictions of labels y from feature vectors $\mathbf{x}$ from example (training set) that include feature vectors with their corresponding labels. In a first approximation, not taking probabilistic relationships into account, this relation can be seen as a function that we want to find

$$y = f(\mathbf{x}). \tag{1.3}$$

Unfortunately, finding this function is a really hard problem. However, we can usually guess some form by looking at data. For example, in Fig. 1.2a we plotted some example points of a one dimensional feature space. From the examples we might guess that there is a linear relation between the y-values and the x-values. The functional form of this relations is described by the formula

$$y = ax + b, \tag{1.4}$$

Where a and b are parameters that describe the slope and the offset, respectively. This is an important step in machine learning, making a parameterized hypothesis of the relations between feature values and labels. We often gather the parameters into a parameter factor

$$\theta = \begin{pmatrix} a \\ b \end{pmatrix} \tag{1.5}$$

and indicate the parameterized form of a general hypothesis as

$$y = f(\mathbf{x}, \theta). \tag{1.6}$$

What remains to be done in this supervised learning problem is to determine the specific parameter values for this particular example. The readers might be familiar with some regression methods to achieve this, such as mean square regression, and we will discuss the theory behind this shortly.

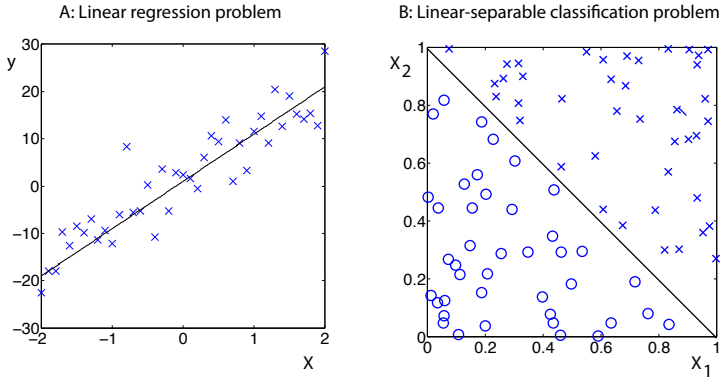

Fig. 1.2 Example of supervised learning in form of (a) linear regression and (b) linear classification.

Another popular subdomain of supervised learning is classification. In classification we have a set of discrete labels y that we call classes. In the following we will consider the simplest case, that of only two class labels. This is called binary classification. An example of a training points from two classes in a two-dimensional feature space, where the labels are indicated with different plotting symbols, $y = o$ and $y = x$, is shown in Fig. 1.2b. In this example we seek to find a decision boundary, a line which separates the two classes. This line can be used to make predictions to which class future unlabelled data belong. If we assume that the data of the two classes are similarly distributed beside its mean, then it is possible to show that the line which maximizes the margin, the space between the line and all the data points, is a optimal solution.

We will later discuss these examples in more detail, but they demonstrate already the gist of the most basic supervised learning. Of course, one of the main challenges of supervised machine learning is to generalize the above ideas to nonlinear cases.

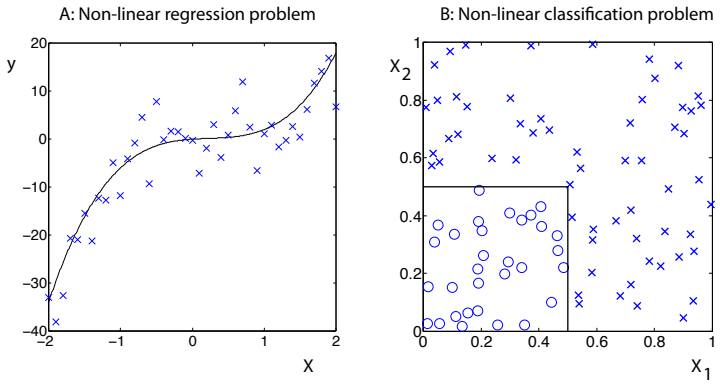

Fig. 1.3 Example of a nonlinear regression and classification problem.

Table 1.1 Using libsvm for classification

```
%example of nonlinear classification
clear; clf; hold on;

%generating training data and plotting them
x=rand(2,100); y=zeros(1,100);
y(x(1,:)<0.5&x(2,:)<0.5)=1;
plot(x(1,y>0.5),x(2,y>0.5),'o')
plot(x(1,y<0.5),x(2,y<0.5),'x')

%train Support Vector Machine
svm=svmtrain(x,y,'kernel_function','rbf');

%generating test data and predicting labels
x=rand(2,100);
ypred=svmclassify(svm,x');
plot(x(1,ypred>0.5),x(2,ypred>0.5),'ro')
plot(x(1,ypred<0.5),x(2,ypred<0.5),'rx')

%plot correct decision boundary
plot([0,0.5],[0.5,0.5],'k');
plot([0.5,0.5],[0.0,0.5],'k');
```

Such a nonlinear regression and classification with nonlinear decision boundaries are shown in Fig. 1.3. The example of a binary class problem shown in Fig. 1.3b consists of uniformly distributed data in four quadrants of which the data in the lower-left quadrant are set to a different class than the other three quadrants. Table 1.1 shows how to use two functions of the Matlab statistics toolbox that includes an implementation of a Support Vector Machine (SVM). Run this function to explore how the program predicts labels of new data for which we did not supply the theoretical labels. If you don't know how to run this program yet, please wait until Section 1.4 where we will introduce how to use Matlab.

This example is meant to show how easy it is in practice to use sophisticated machine learning methods within a few lines of Matlab code thanks to the large

number of machine learning algorithms published in Matlab. You might even be able to apply this to binary classification problems you encounter or you can design your own examples. SVMs are now widely used in practice and often work well. However, there are also many reports where they fail. It is important to learn more about the underlying algorithms as a appropriate application can depend on this. These discussions will also lead us to a deeper understanding of learning and its limits, challenges and new opportunities.

1.2 Classical Robotics

In the following journey we will demonstrate and explore algorithms and associated problems in machine learning not only with the help of computer simulations, but also with the help of robots. Computer simulations are a great way to experiment with many of the ideas, but robotics implementations in the physical world have the advantage to show more clearly the challenges in real applications. Our emphasis in this book is to use general machine learning techniques to solve robotic tasks even though more direct engineering solutions might be possible. While this is not always the way robots are controlled today, machine learning methods are becoming increasingly important in robotics.

The dream of having machines that can act more autonomously for human benefits is quite old. The word 'robot' is credited to the Czech writer Karel Čapek (1921) and to Isaac Asimov (1941), and the first industrial robot is considered to be the **Unimate** (1961). Robots are now invaluable in manufacturing. However, robotics is also still an increasingly active research area. In particular, there is also much research to make robots more autonomous and robust to be able to work in hostile and increasingly uncertain environments. Robotics has many subdisciplines, including mechanical and electrical engineering, computer or machine vision. Even behavioral studies have become prominent in this field. A good definition of robotics as given by Thrun, Burgard, and Fox[1] is:

> "Robotics is the science of perceiving and manipulating the physical world through computer-controlled devices"

That is, we use the word **robot** or **agent** to describe a system which interacts actively with the environment. An agent can be implemented in software or hardware, and the 'brain' of an agent is a **controller** in a classical engineering context. The essential part in the above definition of a robot is that it must act in the physical world. For this it needs **sensors** to sense the state of the world and **actuators** that can be used to change the state of the environment. **Mobile robots** are a good example of this and we will be using two prototype robots for most of our examples in this course, a simple robot arm with a web cam, and a wheeled vehicle with an ultrasonic distance sensor. In contrast to these robotic systems, a vision system alone that uses a digital camera to recognize objects is not a robot in our definition as it is not able to manipulate the environment. While one could say that the physical world is manipulated by displaying the image on a screen, we really want to consider the case where a robot arm moves objects or at least points to them, or where a mobile robot is navigated through a maze.

[1] Sebastian Thrun, Wolfram Burgard, and Dieter Fox, **Probabilistic Robotics** ,MIT press 2006.

Software agents such as crawlers surfing the net are on the borderline. They need sensors, such as the ability to read content of web pages, and act in the physical world as they have to visit web pages physically located at different servers. If they gather information and use this to actively influence the physical world then this could also be considered a robot in the strict sense used here.

Robots are intended to make useful actions to achieve some goals, so robotics is all about finding and safely executing those appropriate actions. This area is generally the subject of **control theory**. A functioning robotics system typically needs controllers on many different levels, controlling the low level functions such as the proper rotation of motors, as well as ensuring that high level tasks are accomplished. A main question in designing a robotics system is often how to combine the different levels of control. A very successful approach has been the **subsumption architecture**, which is illustrated in Figure 1.4. Such systems are typically build bottom-up in that more complex functions use lower-level functions to achieve more complex tasks. The higher level modules can chose to use the lower level function or can inhibit them if necessary.

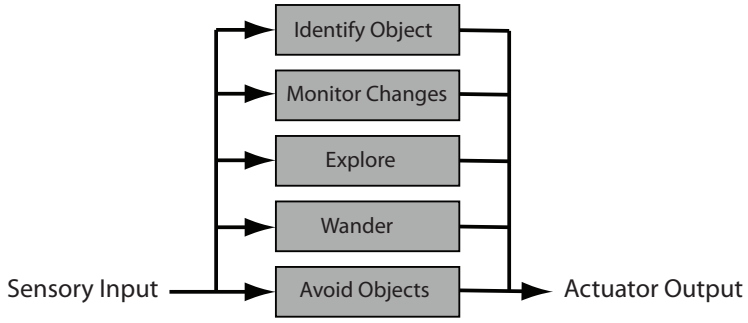

Fig. 1.4 An example of an subsumption architecture. From Maja J. Matarić, *The Robotics Primer, MIT Press 2007*

More generally, it is useful to think of two opposing approaches to robotic control, the **deliberative approach** and the **reactive approach**, though a combination is commonly useful in practice. In the deliberative approach we gather all available information and plan action carefully based on all available actions. Such a **planning process** usually takes time and is based on searching through all available alternatives. The advantage of such systems is that they usually provide superior actions, but the search for such actions can be time consuming and might thus not be applicable to all applications. Also, deliberative systems require a large amount of knowledge about the environment that might not be available in certain applications. In contrast, reactive systems have a more direct approach of translating sensory information in actions. For example, a typical obstacle-avoiding mobile robot turns when a proximity sensor senses some objects in its path. To generate more complex behavior, such reactive systems need to combining lower level control systems with higher level functions that implement specific strategies.

Robots act in the physical world through actuators based on sensory information. Actuators are mainly motors to move the robot around (locomotion) or to move limbs to

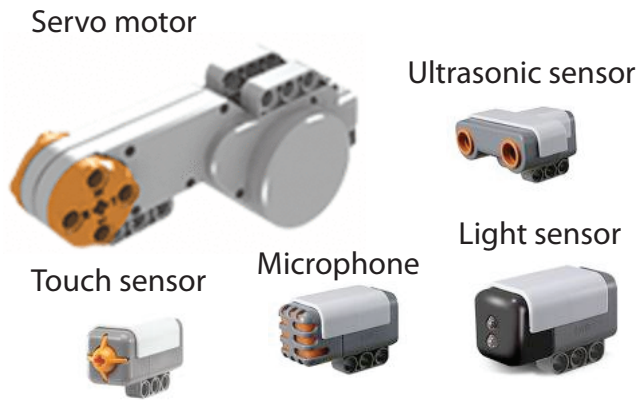

Fig. 1.5 The actuator and sensors of the basic Lego Mindstorm NXT robotics set.

grasp objects (manipulation). Motors for continuous rotations are typically DC (direct current) motors, but in robotics we often need to move a limb to a specific position (location and orientation). Motors that can turn to a specific position are called **servo motors**. Such motors are based on gears and position sensors together with some electronics to control the desired rotation angle. The motor of the LEGO NXT robotics kit is shown in Figure 1.5. This actuator is a stepping motor that can be told to run for a specific duration, a specific number or angle of rotation, and with various powers that influence the rotation speed.

Sensors come in many varieties. Table 1.2 gives some examples of what kind of sensing technology is often used to sense (measure) certain physical properties. The basic Lego actuator and sensors used in this course are shown in Figure 1.5. The motor itself can be used as a rotation sensor as it provides feedback when it is externally rotated. The ultrasonic sensor sends out an high-frequency tone whose reflection is then sensed by its integrated microphone to estimate distances to surfaces. The light sensor can detect light with different wavelengths and can hence be used to detect colours and to some extend also short distances. The basic toolkit also includes a touch sensor and a microphone. We will also use a web camera for basic vision as outlined in the chapter on computer vision. Finally, for special projects we can use additional sensors such as the Kinect sensors from Microsoft, or special Lego sensors such as a GPS, an accelerometer, a giro, or a compass sensor (see Figure 1.6).

1.3 Vector and matrix notations

The remainder of the introductory chapter is dedicated to remind ourselves about some useful mathematical notations and to introduce our programming environment. Since vector and matrix notations are so useful for our discussions, we will start by review their basic concept. Basically, matrices are a shorthand notation to simplify the representation of linear equation systems. We would have indeed lengthy looking formulas without this shorthand notation, and formulas written in this notation can

Table 1.2 Some sensors and the information they measure. From From Maja J. Matari c, *The Robotocs Primer, MIT Press 2007*

Physical Property	**Sensing Technology**
Contact	bump, switch
Distance	ultrasound, radar, infra red
Light level	photocells, cameras
Sound levels	microphones
Strain	strain gauges
Rotation	encoders, potentiometers
Acceleration	accelerometers, gyroscopes
Magnetism	compass
Smell	chemical sensors
Temperature	thermal, infra red
Inclination	inclinometers, gyroscopes
Pressure	pressure gauge
Altitude	altimeter

Fig. 1.6 Some additional sensors that we could use such as the Microsft Kinect and a GPS for the Lego NXT.

easily be entered into Matlab as shown below. In terms of computer science, they represent our basic data types. We consider three basic data types:

1. Scalar:

$$a \text{ for example } 41 \tag{1.7}$$

2. Vector:

$$\mathbf{a} \text{ or component-wise } \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \text{ for example } \begin{pmatrix} 41 \\ 7 \\ 13 \end{pmatrix} \tag{1.8}$$

3. Matrix:

$$\mathbf{a} \text{ or component-wise } \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix} \text{ for example } \begin{pmatrix} 41 & 12 \\ 7 & 45 \\ 13 & 9 \end{pmatrix} \tag{1.9}$$

We used bold-typed characters to indicate both a vector and a matrix because the difference is usually apparent from the circumstances. A matrix is just a collection of scalars or vectors. We talk about an $n \times m$ matrix where n is the number of rows and m is the number of columns. A scalar is thus a 1×1 matrix, and a vector of length n can be considered an $n \times 1$ matrix. A similar collection of data is called **array** in computer science. However, a matrix is different because we also define operations on these data collections. The rules of calculating with matrices can be applied to scalars and vectors.

We define how to add and multiply two matrices so that we can use them in algebraic equations. The **sum of two matrices** is defined as the sum of the individual components

$$(\mathbf{a} + \mathbf{b})_{ij} = \mathbf{a}_{ij} + \mathbf{b}_{ij}. \tag{1.10}$$

For example, $\mathbf{a}$ and $\mathbf{b}$ are 3×2 matrices, then

$$\mathbf{a} + \mathbf{b} = \begin{pmatrix} a_{11} + b_{11} & a_{12} + b_{12} \\ a_{21} + b_{21} & a_{22} + b_{22} \\ a_{31} + b_{31} & a_{32} + b_{32} \end{pmatrix} \tag{1.11}$$

Matrix multiplication is defined as

$$(\mathbf{a} * \mathbf{b})_{ij} = \sum_k \mathbf{a}_{ik} \mathbf{b}_{kj}. \tag{1.12}$$

The matrix multiplication is hence only defined as multiplication matrices $\mathbf{a}$ and $\mathbf{b}$ where the number of columns of the matrix $\mathbf{a}$ is equal to the number of rows of matrix $\mathbf{b}$. For example, for two square matrices with two rows and two columns, their product is given by

$$\mathbf{a} * \mathbf{b} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix} \tag{1.13}$$

A handy rule for matrix multiplications is illustrated in Fig. 1.7. Each component in the resulting matrix is calculated from the sum of two multiplicative terms. The rule for multiplying two matrices is tedious but straightforward and can easily be implemented in a computer.

Another useful definition is the **transpose** of a matrix. This operation is indicated usually by a superscript t or a prime ($'$). Taking the transpose of a matrix means that the matrix is rotated 90 degrees; the first row becomes the first column, the second row becomes the second column, etc. For example, the transpose of the example in 1.9 is

$$\mathbf{a}' = \begin{pmatrix} 41 & 7 & 13 \\ 12 & 45 & 9 \end{pmatrix} \tag{1.14}$$

The transpose of a vector transforms a column vector into a row vector and vice versa.

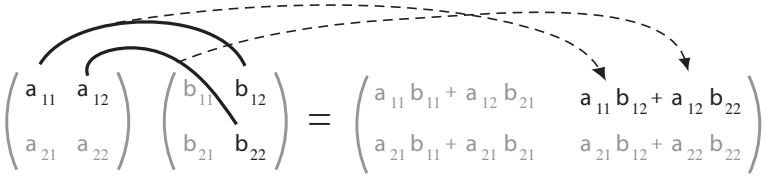

Fig. 1.7 Illustration of a matrix multiplication. Each element in the resulting matrix consists of terms that are taken from the corresponding row of the first matrix and column of the second matrix. Thus in the example we calculate the highlighted element from the components of the first row of the first matrix and the second column of the the second matrix. From these rows and columns we add all the terms that consist of the element-wise multiplication of the terms.

As already mentioned, matrices were invented to simplify the notations for systems of coupled algebraic equations. Consider, for example, the system of three equations

$$41x_1 + 12x_2 = 17 \tag{1.15}$$

$$7x_1 + 45x_2 = -83 \tag{1.16}$$

$$13x_1 + 9x_2 = -5. \tag{1.17}$$

This can be written as

$$\mathbf{a}\mathbf{x} = \mathbf{b} \tag{1.18}$$

with the matrix $\mathbf{a}$ as in the example of 1.9, the vector $\mathbf{x} = (x_1\ x_2)'$, and the vector $\mathbf{b} = (17\ -83\ -5)'$.

The solution of this linear equation system is equivalent to finding the inverse of matrix $\mathbf{a}$ which we write as $\mathbf{a}^{-1}$. The inverse of the matrix is defined by

$$\mathbf{a}^{-1}\mathbf{a} = \mathbb{1}, \tag{1.19}$$

where the matrix $\mathbb{1}$ is the unit matrix that has element of one on the diagonal and zeros otherwise. Multiplying equation 1.18 from left with $\mathbf{a}^{-1}$ is hence

$$\mathbf{x} = \mathbf{a}^{-1}\mathbf{b} \tag{1.20}$$

There are many programs that solve linear equations systems. The programming environment that we will discuss next, called Matlab, has its roots in linear algebra and is in particular powerful in representing matrices. Matlab actually stands for Matrix Laboratory, and this environment is also the dominating tool in the machine learning community.

1.4 Scientific computing with Matlab

While some of the material presented here is theoretical in nature, it is fun and educational to get it to work in either a simulated environment or with the help of physical robots. The brain of our devices will be little programs that control a Lego robot or analyses data given to them. We will use the Matlab[2] programming

[2]Matlab is a registered trademark of The MathWorks, Inc.

environment for this work. This high level language has a lot of support for the kind of things we want to do, such as basic signal processing, computer vision support, Lego NXT communication, and implementation of machine learning algorithms. There are other high level scientific programming languages with similar support such as Python or R. Some of the experiments can certainly be replicated in such systems and the Matlab scripts in this tutorial can be seen as high level description language. There are some limitations that made us decide against these programming environments. In particular, Python has limited support for new machine learning algorithms since many machine learning researchers publish their algorithms in Matlab. In particular, we could not find sufficient support for Bayesian networks. R is also less common in the machine learning community. While Matlab is an expensive commercial product, many academic institutions have research licenses, student versions are quite affordable, and there is a free interpreter that can run Matlab programs called Octave. We will be using a MS Windows environment mainly because the tools which support the Matlab-Lego NXT communications, in particular the USB communication, has still restrictions in a Mac environment. Linux environments should mostly work but has not been considered in detail.

In this section we briefly introduce programming in Matlab with a focus on the basic elements to get started. We assume thereby little programming experience, although experienced programmers in other programming languages might want to scan through this chapter to see some differences, particularly with matrix notations that are central in Matlab. Matlab is an interactive programming environment for scientific computing. This environment is very convenient for us for several reasons, including its interpreted execution mode, which allows fast and interactive program development, advanced graphic routines, which allow easy and versatile visualization of results, a large collection of predefined routines and algorithms, which makes it unnecessary to implement known algorithms from scratch, and the use of matrix notations, which allows a compact and efficient implementation of mathematical equations and machine learning algorithms. Matlab stands for matrix laboratory, which emphasizes the fact that most operations are array or matrix oriented. Similar programming environments are provided by the open source systems called **Scilab** and **Octave**. The Octave system seems to emphasize syntactic compatibility with Matlab, while Scilab is a fully fledged alternative to Matlab with similar interactive tools but some different syntax and function names. The distribution includes a converter for Matlab programs, but we are uncertain how the integration with the Lego system would work.

1.4.1 The Matlab programming environment

Matlab is a programming environment and collection of tools to write programs, execute them, and visualize results. Matlab has to be installed on your computer to run the programs mentioned in the manuscript. It is commercially available for many computer systems, including Windows, Mac, and UNIX systems. The Matlab web page includes a set of excellent tutorial videos, also accessible from the **demos** link on the Matlab desktop, which are highly recommended to learn Matlab.

As already mentioned, there are several reasons why Matlab is easy to use and appropriate for our programming need. Matlab is an interpreted language, which means that commands can be executed directly by an interpreter program. This makes

the time-consuming compilation steps of other programming languages redundant and allows a more interactive working mode. A disadvantage of this operational mode is that the programs could be less efficient compared to compiled programs. However, there are two possible solution to this problem in case efficiency become a concern. The first is that the implementations of many Matlab functions is very efficient and are themselves pre-compiled. Matlab functions, specifically when used on whole matrices, can therefore outperform less well-designed compiled code. For example, it is strongly recommend to use matrix notations wherever possible instead of explicit component-wise operations. A second possible solutions to increase the performance is to use the Matlab compiler to either produce compiled Matlab code in `.mex` files or to translate Matlab programs into compilable language such as C.

The ability of Matlab to support matrices is very valuable for our purpose as it makes the code very compact and comparable to the mathematical notations used in the manuscript. Furthermore, Matlab has very powerful visualization routines, and the new versions of Matlab include tools for documentation and publishing of codes and results. In addition, Matlab includes implementations of many mathematical and scientific methods on which we can base our programs. For example, Matlab includes many functions and algorithms for linear algebra and to solve systems of differential equations. Specialized collections of functions and algorithms, called a 'toolbox' in Matlab, can be purchased in addition to the basic Matlab package or imported from third parties, including many freely available programs and tools published by researchers. For example, the Matlab Neural Network Toolbox incorporates functions for building and analyzing standard neural networks. This toolbox covers many algorithms particularly suitable for connectionist modelling and neural network applications. A similar toolbox, called NETLAB, is freely available and contains many advanced machine learning methods. We will use some toolboxes later in this course, including the LIBSVM toolbox and the Matlab NXT toolbox to program the Lego robots.

1.4.1.1 Starting a Matlab session

Starting Matlab opens the Matlab desktop as shown in Fig. 1.8 for Matlab version 8. The Matlab desktop is comprised of several windows which can be customized or undocked to move them into a separate window. A list of these tools are available under the **desktop menu**, and includes tools such as the **command window**, **editor**, **workspace**, etc. We will use some of these tools later, but for now we only need the **Matlab command window**. We can thus close the other windows if they are open, such as the **launch pad** or the **current directory window**; we can always get them back from the **desktop** menu. Alternatively, we can undock the command window by clicking the arrow button on the command window bar. An undocked command window is illustrated on the left in Fig. 1.9. Older versions of Matlab start directly with a command window. The command window, or the interpreter behind it, interprets and executes command. It is therefore the platform for interactive programming and to access other Matlab functionalities with line commands.

1.4.1.2 Basic variables in Matlab

The Matlab programming environment is interactive in that all commands can be typed into the command window after the command prompt (see Fig. 1.9). The commands are interpreted directly, and the results are displayed in the command window. For

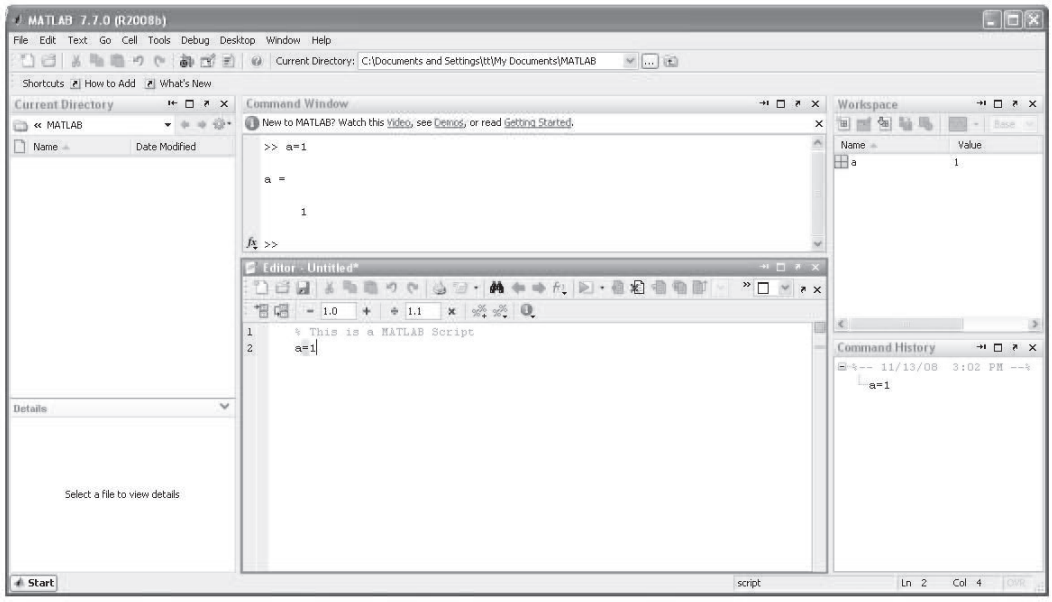

Fig. 1.8 The Matlab *desktop window* of Matlab Version 8.

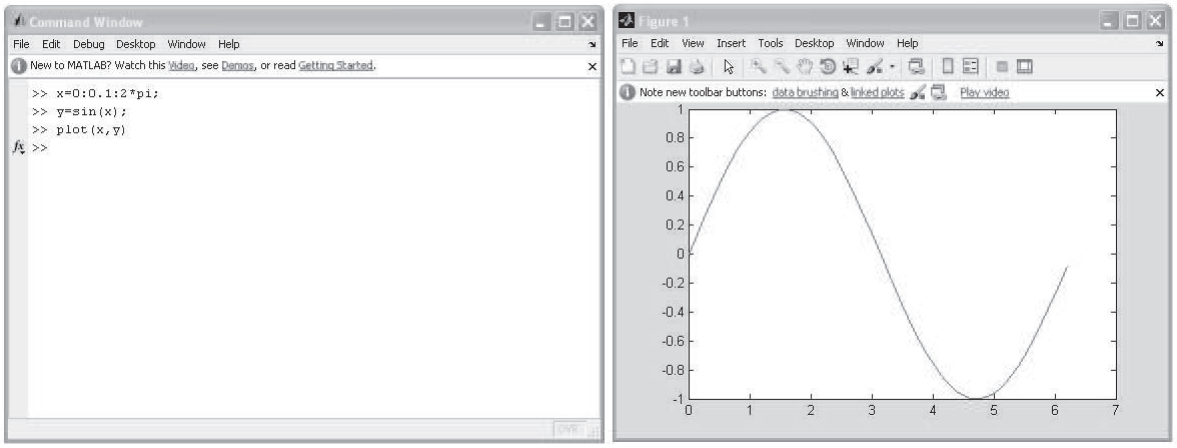

Fig. 1.9 A Matlab *command window* (left) and a Matlab *figure window* (right) displaying the results of the function `plot_sin` developed in the text.

example, a variable is created and assigned a value with the = operator, such as

```
>> a=3

a =
```

```
     3
```

Ending a command with semicolon (`;`) suppresses the printing of the result on screen. It is therefore generally included in our programs unless we want to view some intermediate results. All text after a percentage sign (`%`) is not interpreted and thus treated as comment. For example, the command

```
>> b='Hello World!'; % delete the semicolon to echo Hello World!
```

creates a new string variable `b` with a value `Hello World`, and we included an instruction how to print this value on the command window. This example also demonstrates that the type of a variable, such as being an integer, a real number, or a string, is determined by the values assigned to the elements. This is called **dynamic typing**. Thus, variables do not have to be declared as holding specific values as in some other programming languages. While dynamic typing is convenient, a disadvantage is that a mistyped variable name can not be detected by the compiler. Inspecting the list of created variables is thus a useful step for debugging.

All the variables that are created by a program are kept in a buffer called **workspace**. These variable can be viewed with the command `whos` or displayed in the **workspace** window of the Matlab desktop. For example, after declaring the variables above, the `whos` command results in the responds

```
>> whos
  Name      Size             Bytes  Class     Attributes

  a         1x1                  8  double
  b         1x12                24  char
```

It displays the name, the size, and the type (class) of the variable. The size is often useful to check the orientation of matrices as we will see below. The variables in the workspace can be used as long as Matlab is running and as long as it is not cleared with the command `clear`. The workspace can be saved with the command `save` **filename**, which creates a file **filename**`.mat` with internal Matlab format. The saved workspace can be reloaded into Matlab with the command `load` **filename**. The load command can also be used to import data in ASCII format. The workspace is very convenient as we can run a program within a Matlab session and then work interactively with the results, for example, to plot some of the generated data.

Variables in Matlab are generally matrices or data arrays. While a matrix and an array are both two-dimensional data representations, the difference is that matrices are defined with specific operations that can be applied to them, which we will review below. Matrices include scalars (1×1 matrix) and vectors ($1 \times N$ matrix) as special cases. Values can be assigned to matrix elements in several ways. The most basic one is using square brackets and separating rows by a semicolon within the square brackets, for example (see Fig. 1.9),

```
>> a=[1 2 3; 4 5 6; 7 8 9]

a =

     1     2     3
     4     5     6
```

```
     7     8     9
```

One can access individual matrix elements with indices in round brackets, where the common mathematical convention is followed in that the first index specifies the row from the top, and the second index specifies the column from the left. For example, the element `a(3,2)` has the value 8. A range of indices can be selected with the column `(:)` notation, such as a sub-matrix

```
>> a(1:2,2:3)

ans =

     2     3
     5     6
```

The defaults of the start and end indices are the first and last index, so that omitting them is also possible. Thus, the 3rd column vector of the matrix above can be generated by

```
>> a(:,3)

ans =

     3
     6
     9
```

A vector of elements with consecutive values can be assigned by column operators like `start:step:end`, for example

```
>> v=0:1.5:7

v =

         0    1.5000    3.0000    4.5000    6.0000
```

There are several other ways to create matrices in Matlab such as an **array editor**, and data in ASCII files can be assigned to matrices when loaded into Matlab. Also, Matlab functions often return matrices which can be used to assign values to matrix elements. For example, a uniformly distributed random 3×3 matrix can be generated with the command

```
>> b=rand(3)

b =

    0.9501    0.4860    0.4565
    0.2311    0.8913    0.0185
    0.6068    0.7621    0.8214
```

The multiplication of two matrices, following the matrix multiplication rules reviewed further below in section 1.3, can be done in Matlab by typing

```
>> c=a*b
```

```
c =

    3.2329    4.5549    2.9577
    8.5973   10.9730    6.8468
   13.9616   17.3911   10.7360
```

This is equivalent to

```
c=zeros(3);
for i=1:3
   for j=1:3
      for k=1:3
         c(i,j)=c(i,j)+a(i,k)*b(k,j);
      end
   end
end
```

which is the common way of writing matrix multiplications in other programming languages. Formulating operations on whole matrices, rather than on the individual components separately, is not only more convenient and clear, but can enhance program performance considerably. Whenever possible, operations on whole matrices should be used. This is likely to be the major change in your programming style when converting from another programming language to Matlab. The performance disadvantage of an interpreted language is often negligible when using operations on whole matrices.

The transpose operation of a matrix changes columns to rows. Thus, a row vector such as `v` can be changed to a column vector with the Matlab transpose operator (`'`),

```
>> v'

ans =

     0
     2
     4
```

which can then be used in a matrix-vector multiplication like

```
>> a*v'

ans =

    16
    34
    52
```

The inconsistent operation `a*v` does produce an error,

```
>> a*v
??? Error using ==> mtimes
Inner matrix dimensions must agree.
```

Table 1.3 Basic programming contracts in Matlab.

Programming construct	Command	Syntax
Assignment	=	`a=b`
Arithmetic operations	add	`a+b`
	multiplication	`a*b` (matrix), `a.*b` (element-wise)
	division	`a/b` (matrix), `a./b` (element-wise)
	power	`a^b` (matrix), `a.^b` (element-wise)
Relational operators	equal	`a==b`
	not equal	`a~=b`
	less than	`a<b`
Logical operators	AND	`a & b`
	OR	`a||b`
Loop	for	`for` **index=start:increment:end** statement `end`
	while	`while` **expression** statement `end`

Conditional command	if statement	`if` **logical expressions**
		statement
		`elseif` **logical expressions**
		statement
		`else`
		statement
		`end`

Function		`function [x,y,...]=`**name**`(a,b,...)`

Component-wise operations in matrix multiplications (`*`), divisions (`/`) and potentiation `^` are indicated with a dot modifier such as

```
>> v.^2

ans =

     0     4    16
```

The most common operators and basic programming constructs in Matlab are similar to those in other programming languages and are listed in Table 1.3.

1.4.1.3 Control flow and conditional operations

Besides the assignments of values to variables, and the availability of basic data structures such as arrays, a programming language needs a few basic operations for building loops and for controlling the flow of a program with conditional statements (see Table 1.3). For example, the **for loop** can be used to create the elements of the vector `v` above, such as

```
>> for i=1:3; v(i)=2*(i-1); end
```

```
>> v

v =

     0     2     4
```

Table 1.3 lists, in addition, the syntax of a while loop. An example of a conditional statement within a loop is

```
>> for i=1:10; if i>4 & i<=7; v2(i)=1; end; end
>> v2

v2 =

     0     0     0     0     1     1     1
```

In this loop, the statement `v2(i)=1` is only executed when the loop index is larger than 4 and less or equal to 7. Thus, when `i=5`, the array `v2` with 5 elements is created, and since only the elements `v2(5)` is set to 1, the previous elements are set to 0 by default. The loop adds then the two element `v2(6)` and `v2(7)`. Such a vector can also be created by assigning the values 1 to a specified range of indices,

```
>> v3(4:7)=1

v3 =

     0     0     0     1     1     1     1
```

A 1×7 array is thereby created with elements set to 0, and only the specified elements are overwritten with the new value 1. Another method to write compact loops in Matlab is to use vectors as index specifiers. For example, another way to create a vector with values such as `v2` or `v3` is

```
>> i=1:10

i =

     1     2     3     4     5     6     7     8     9    10

>> v4(i>4 & i<=7)=1

v4 =

     0     0     0     0     1     1     1
```

1.4.1.4 Creating Matlab programs and functions

If we want to repeat a series of commands, it is convenient to write this list of commands into an ASCII file with extension '`.m`'. Any ASCII editor (for example; WordPad, Emacs, etc.) can be used, but not a word processor like Word since this adds hidden characters for formatting purposes into the text that can not be interpreted by the Matlab compiler. The Matlab package contains an editor that has the advantage of

colouring the content of Matlab programs for better readability and also provides direct links to other Matlab tools. The list of commands in the ASCII file (e.g. **prog1**`.m`) is called a **script** in Matlab and makes up a Matlab program. This program can be executed with a run button in the Matlab editor or by calling the name of the file within the command window (for example, by typing **prog1**). We assumed here that the program file is in the current directory of the Matlab session or in one of the search paths that can be specified in Matlab. The Matlab desktop includes a 'current directory' window (see desktop menu). Some older Matlab versions have instead a 'path browser'. Alternatively, one can specify absolute path when calling a program, or change the current directories with UNIX-style commands such as `cd` in the command window (see Fig. 1.10).

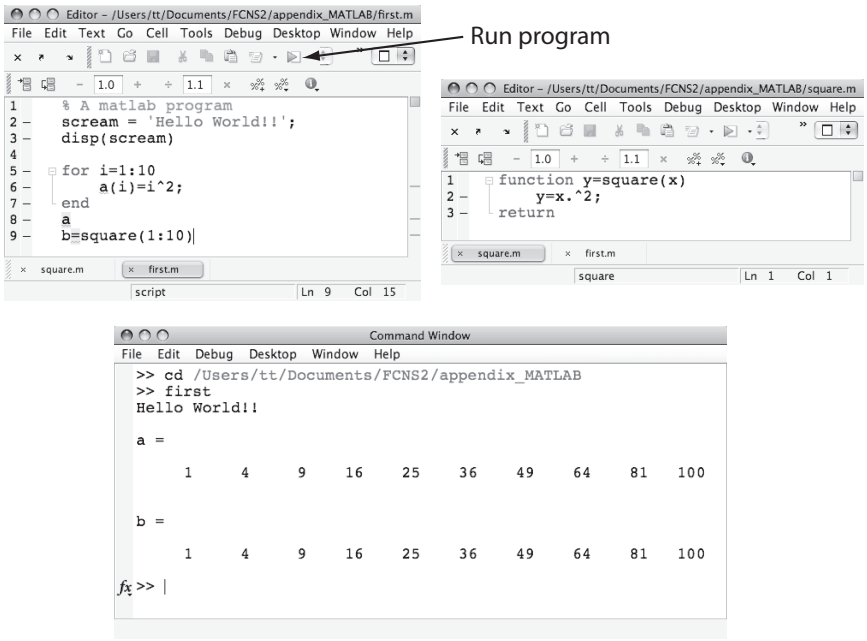

Fig. 1.10 Two editor windows and a command window.

Functions are another way to encapsulate code. They have the additional advantage that they can be pre-compiled with the Matlab Compiler^{TM} available from MathWorks, Inc. Functions are kept in files with extension '`.m`' which start with the command line like

```
function y=f(a,b)
```

where the variables `a` and `b` are passed to the function and `y` contains the values returned by the function. The return values can be assigned to a variable in the calling Matlab script and added to the workspace. All other internal variables of the functions are local to the function and will not appear in the workspace of the calling script. As an example, consider a function that is given two numbers and should return the larger of the two values. This can be encapsulated into a function like

```
function larger=returnMax(a,b)
larger = b;
if a>b; larger=a; end
return
```

This function must be saved into a file with name `returnMax`. This function can then be called from another program in the same directory, or from another directory when the path to the function is added to the search path in Matlab. As an example, we could call

```
>> returnMax(4,3)

ans =

     4
```

Matlab has a rich collection of predefined functions implementing many algorithms, mathematical constructs, and advanced graphic handling, as well as general information and help functions. You can always search for some keywords using the useful command `lookfor` followed by the keyword (search term). This command lists all the names of the functions that include the keywords in a short description in the function file within the first **comment lines** after the function declaration in the function file. The command `help`, followed by the function name, displays the first block of comment lines in a function file. This description of functions is usually sufficient to know how to use a function. A list of some frequently used functions is listed in Table 1.4.1.4.

1.4.1.5 Graphics

Matlab is a great tool for producing scientific graphics. We want to illustrate this by writing our first program in Matlab: calculating and plotting the sine function. The program is

```
x=0:0.1:2*pi;
y=sin(x);
plot(x,y)
```

The first line assigns elements to a vector $\mathbf{x}$ starting with $x(1) = 0$ and incrementing the value of each further component by 0.1 until the value 2π is reached (the variable `pi` has the appropriate value in Matlab). The last element is $x(63) = 6.2$. The second line calls the Matlab function `sin` with the vector `x` and assigns the results to a vector `y`. The third line calls a Matlab plotting routine. You can type these lines into an ASCII file that you can name `plot_sin.m`. The code can be executed by typing `plot_sin` as illustrated in the **command window** in Fig. 1.9, provided that the Matlab session points to the folder in which you placed the code. The execution of this program starts a **figure window** with the plot of the sine function as illustrated on the right in Fig. 1.9.

The appearance of a plot can easily be changed by changing its attributes. There are several functions that help in performing this task, for example, the function `axis` that can be used to set the limits of the axis. New versions of Matlab provide window-based interfaces to the attributes of the plot. However, there are also two basic commands, `get` and `set`, that we find useful. The command `get(gca)` returns a list with the axis

Name	Brief description	Name	Brief description
abs	absolute function	mod	modulus function
axis	sets axis limits	num2str	converts number to string
bar	produces bar plot	ode45	ordinary differential equation solver
ceil	round to larger integer	ones	produces matrix with unit elements
colormap	colour matrix for surface plots	plot	plot lines graphs
cos	cosine function	plot3	plot 3-dimensional graphs
diag	diagonal elements of a matrix	prod	product of elements
disp	display in command window	rand	uniformly distributed random variable
errorbar	plot with error bars	randn	normally distributed random variable
exp	exponential function	randperm	random permutations
fft	fast Fourier transform	reshape	reshaping a matrix
find	index of non-zero elements	set	sets values of parameters in structure
floor	round to smaller integer	sign	sign function
hist	produces histogram	sin	sine function
int2str	converts integer to string	sqrt	square root function
isempty	true if array is empty	std	calculates standard deviation
length	length of a vector	subplot	figure with multiple subfigures
log	logarithmic function	sum	sum of elements
lsqcurevfit	least mean square curve	surf	surface plot
	fitting (statistics toolbox)	title	writes title on plot
max	maximum value and index	view	set viewing angle of 3D plot
mix	minimum value and index	xlabel	label on x-axis of a plot
mean	calculates mean	ylabel	label on y-axis of a plot
meshgrid	creates matrix to plot grid	zeros	creates matrix of zero elements

Table 1.4 Matlab functions used in this course. The Matlab command `help cmd`, where `cmd` is any of the functions listed here, provides more detailed explanations.

properties currently in effect. This command is useful for finding out what properties exist. The variable `gca` (get current axis) is the **axis handle**, which is a variable that points to a memory location where all the attribute variables are kept. The attributes of the axis can be changed with the `set` command. For example, if we want to change the size of the labels we can type `set(gca,'fontsize',18)`. There is also a handle for the current figure `gcf` that can be used to get and set other attributes of the figure. Matlab provides many routines to produce various special forms of plots including plots with error-bars, histograms, three-dimensional graphics, and multi-plot figures.

1.4.2 Some programming exercises and examples

The following exercises are intended to practice programming yourself. It is thus important to try to solve these exercises before looking at the solution given below. Note that there are usually many different ways to program a specific task, so don't worry if your answer looks different as long as it captures the main task. Observing others programs is a good way to learn tricks, but you also need to learn how to write programs on your own.

Writing programs is easy for some people and more difficult for others. It is important to realize that a program strictly (!) follows rules. Matlab will respond with

a red error message if it discovers syntactic rules. These error messages are your friend. Read then as they usually tell you quite well what the specific problem is. More difficult are runtime errors in which there might be logical errors in our algorithm. These are much harder to find. In general, to debug a program it is a good idea to follow each line of the program and to predict what your expected outcome of this operation is. Then write out what the program produces and see if it matches. If not then there is a problem.

As you have seen, a programming language has only a handful of commands and maybe a large list of predefined functions, but most of programming is exercising your analytic mind for solving a problem. Key is thereby to divide the problem into small manageable tasks. Another key to learning programming is to understand what caused the bugs in your code. Simply changing random statements until 'it works' is a bad approach. Find out what exactly happened and you can be sure that you really solved the problem and not just masqueraded it.

Random Walk

The first example is a program that should plot a random walk. In a random walk, an agent has an initial position at time $t = 0$ and travels a certain distance in each time step Δt. The distance is thereby given by a Gaussian distributed random number, which can be produced in Matlab by the function `randn()`. Plot the path of the agent over 100 time steps and plot s second figure that shows a histogram of the velocities during this path. Note that you can create a new figure window with the command `figure`, and that you can create a histogram with the Matlab function `hist()`. If you type `help hist` you will get more information on how to use this function.

Inverting String

Write a program that prompts a user to insert a string. After entering the string, the program should then respond by printing the string in reverse order.

Function

Write a program with a function that takes two arrays, a and b, and calculates the matrix $a * b'^2 + b$.

Sorting

Write a program that prompts the user to enter words until the esc key is pressed and then takes this list and prints it in a sorted order.

Possible solutions

Below are some sample solutions for the first two examples. As mentioned before, don't worry if your solutions used a different methods. However, it is useful to make sure that the following solutions make sense to you and that you understand how they work.

The random walk can be plotted with

```
x=10;
for t=1:100
    x(t+1)=x(t)+randn;
```

```
end
plot(x)
```

To plot the histogram of the individual changes we need first to calculate the changes and gather them into a new array (vector). A general trick is to make an empty array, `v=[]` and then make a new array in the loop from the previous version and the element to be added.

```
v=[];
for t=1:100
    v=[v,x(t+1)-x(t)];
end
figure;
hist(v)
```

Inverting the string can be done with specifying the elements of the string array.

```
yourString=input('Please input a string followed by enter ','s')
reverseString=yourString(size(yourString,2):-1:1)
```

This could also be programmed with an explicit loop like

```
reverseString=[];
for i=size(yourString,2):-1:1
    reverseString=[reverseString, yourString(i)];
end
```

However, it is recommended to avoid explicit loops as much as possible since implicit looping is often much more efficient.