Numerical Algorithms
and
Digital Representation

Knut Mørken

October 19, 2008

Preface

These lecture notes form part of the syllabus for the first-semester course MAT-INF1100 at the University of Oslo. The topics roughly cover two main areas: *Numerical algorithms*, and what can be termed *digital understanding*. Together with a thorough understanding of calculus and programming, this is knowledge that students in the mathematical sciences should gain as early as possible in their university career. As subjects such as physics, meteorology and statistics, as well as many parts of mathematics, become increasingly dependent on computer calculations, this training is essential.

Our aim is to train students who should not only be able *to use* a computer for mathematical calculations; they should also have a basic understanding of *how* the computational methods work. Such understanding is essential both in order to judge the quality of computational results, and in order to develop new computational methods when the need arises.

In these notes we cover the basic numerical algorithms such as interpolation, numerical root finding, differentiation and integration, as well as numerical solution of ordinary differential equations. In the area of digital understanding we discuss digital representation of numbers, text, sound and images. In particular, the basics of lossless compression algorithms with Huffman coding and arithmetic coding is included.

A basic assumption throughout the notes is that the reader either has attended a basic calculus course in advance or is attending such a course while studying this material. Basic familiarity with programming is also assumed. However, I have tried to quote theorems and other results on which the presentation rests. Provided you have an interest and curiosity in mathematics, it should therefore not be difficult to read most of the material with a good mathematics background from secondary school.

MAT-INF1100 is a central course in the project *Computers in Science Education* (CSE) at the University of Oslo. The aim of this project is to make sure that students in the mathematical sciences get a unified introduction to computational methods as part of their undergraduate studies. The basic founda-

tion is laid in the first semester with the calculus course, MAT1100, and the programming course INF1100, together with MAT-INF1100. The maths courses that follow continue in the same direction and discuss a number of numerical algorithms in linear algebra and related areas, as well as applications such as image compression and ranking of web pages.

Some fear that a thorough introduction of computational techniques in the mathematics curriculum will reduce the students' basic mathematical abilities. This could easily be true if the use of computations only amounted to running code written by others. However, deriving the central algorithms, programming them, and studying their convergence properties, should lead to a level of mathematical understanding that should certainly match that of a more traditional approach.

Many people have helped develop these notes which have matured over a period of nearly ten years. Solveig Bruvoll, Marit Sandstad and Øyvind Ryan have helped directly with the present version, while Pål Hermunn Johansen provided extensive programming help with an earlier version. Geir Pedersen was my co-lecturer for four years. He was an extremely good discussion partner on all the facets of this material, and influenced the list of contents in several ways. I work at the Centre of Mathematics for Applications (CMA) at the University of Oslo, and I am grateful to the director, Ragnar Winther, for his enthusiastic support of the CSE project and my extensive undertakings in teaching. Over many years, my closest colleagues Geir Dahl, Michael Floater, and Tom Lyche have shaped my understanding of numerical analysis and allowed me to spend considerably more time than usual on elementary teaching. Another colleague, Sverre Holm, has been my source of information on signal processing. To all of you: thank you!

My academic home, the Department of Informatics and its chairman Morten Dæhlen, has been very supportive of this work by giving me the freedom to extend the Department's teaching duties, and by extensive support of the CSE-project. It has been a pleasure to work with the Department of Mathematics over the past eight years, and I have many times been amazed by how much confidence they seem to have in me. I have learnt a lot, and have thoroughly enjoyed teaching at the cross-section between mathematics and computing which is my scientific home. I can only say thank you, and I feel at home in both departments.

A course like MAT-INF1100 is completely dependent on support from other courses. Tom Lindstrøm has done a tremendous job with the parallel calculus course MAT1100, and its sequel MAT1110 on multivariate analysis and linear algebra. Hans Petter Langtangen has done an equally impressive job with INF1100, the introductory programming course with a mathematical and sci-

entific flavour, and I have benefited from many hours of discussions with both of them. Morten Hjorth-Jensen, Arnt-Inge Vistnes and Anders Malthe Sørensen have introduced a computational perspective in a number of physics courses, and discussions with them have convinced me of the importance of introducing computations for all students in the mathematical sciences. Thank you to all of you.

The CSE project is run by a group of five people: Annik Myhre (Dean of Education at the MN-faculty[1]), Hanne Sølna (Head of the teaching section at the MN-faculty[1]), Helge Galdal (Administrative Leader of the CMA), Morten Hjorth-Jensen and myself. This group of people has been the main source of inspiration for this work, and without you, there would still only be uncoordinated attempts at including computations in our elementary courses. Thank you for all the fun we have had.

The CSE project has become much more than I could ever imagine, and the reason is that there seems to be a genuine collegial atmosphere at the University of Oslo in the mathematical sciences. This means that it has been possible to build momentum in a common direction not only within a research group, but across several departments, which seems to be quite unusual in the academic world. Everybody involved in the CSE project is responsible for this, and I can only thank you all.

Finally, as in all teaching endeavours, the main source of inspiration is the students, without whom there would be no teaching. Many students become frustrated when their understanding of the nature of mathematics is challenged, but the joy of seeing the excitement in their eyes when they understand something new is a constant source of satisfaction.

Tønsberg, August 2008

Knut Mørken

---

[1] The faculty of Mathematics and Natural Sciences.

Contents

CHAPTER 1

Introduction

Why are computational methods in mathematics important? What can we do with these methods? What is the difference between computation by hand and by computer? What do I need to know to perform computations on computers?

These are natural questions for a student to ask before starting a course on computational methods. And therefore it is also appropriate to try and provide some short answers already in this introduction. By the time you reach the end of the notes you will hopefully have more substantial answers to these as well as many other questions.

1.1   A bit of history

A major impetus for the development of mathematics has been the need for solving everyday computational problems. Originally, the problems to be solved were quite simple, like adding and multiplying numbers. These became routine tasks with the introduction of the decimal numeral system. Another ancient problem is how to determine the area of a field. This was typically done by dividing the field into small squares, rectangles or triangles with known areas and then adding up. Although the method was time-consuming and could only provide an approximate answer, it was the best that could be done. Then in the 18th century Newton and Leibniz developed the differential calculus. This made it possible to compute areas and similar quantities via quite simple symbolic computations, namely integration and differentiation.

In the absence of good computational devices, this has proved to be a powerful way to approach many computational problems: Look for deeper structures in the problem and exploit these to develop alternative, often non-numerical, ways to compute the desired quantities. At the beginning of the 21st century

this has developed mathematics into an extensive collection of theories, many of them highly advanced and requiring extensive training to master. And mathematics has become much more than a tool to solve practical computational problems. It has long ago developed into a subject of its own, that can be valued for its beautiful structures and theories. At the same time mathematics is the language in which the laws of nature are formulated and that engineers use to build and analyse a vast diversity of man-made objects, that range from aircrafts and economic models to digital music players and special effects in movies.

An outsider might think that the intricate mathematical theories that have been developed have quenched the need for old fashioned computations. Nothing could be further from the truth. In fact, a large number of the developments in science and engineering over the past fifty years would have been impossible without huge calculations on a scale that would have been impossible a hundred years ago. The new device that has made such calculations possible is of course the digital computer.

The birth of the digital computer is usually dated to the early 1940s. From the beginning it was primarily used for mathematical computations, and today it is an indispensable tool in almost all scientific research. But as we all know, the usefulness of computers goes far beyond the scientific laboratories. Computers are now essential tools in virtually all offices and homes in our society, and small computers control a wide range of machines.

The one feature of modern computers that has made such an impact on science and society is undoubtedly the speed with which a computer operates. We mentioned above that the area of a field can be computed by dividing the field into smaller parts like triangles and rectangles whose areas are very simple to compute. This has been known for hundreds of years, but the method was only of limited interest as complicated shapes had to be split into a large number of smaller parts to get sufficient accuracy. The symbolic computation methods that were developed worked well, but only for certain special problems. Symbolic integration, for example, is only possible for a small class of integrals; the vast majority of integrals cannot be computed by symbolic methods. The development of fast computers means that the old methods for computing areas, based on dividing the shape into simple parts, are highly relevant as we can very quickly sum up the areas of a large number of triangles or rectangles.

With all the spectacular accomplishments of computers one may think that formal methods and proofs are not of interest any more. This is certainly not the case. A truth in mathematics is only accepted when it can be proved through strict logical reasoning, and once it has been proved to be true, it will always be true. A mathematical theory may lose popularity because new and better theories are developed, but the old theory remains true as long as those who

discovered it did not make any mistakes. Later, new discoveries are made which may bring the old, and often forgotten, theory back into fashion again. The simple computational methods is one example of this. In the 20th century mathematics went through a general process of more abstraction and many of the old computational techniques were ignored or even forgotten. When the computer became available, there was an obvious need for computing methods and the old techniques were rediscovered and applied in new contexts. All properties of the old methods that had been established with proper mathematical proofs were of course still valid and could be utilised straightaway, even if the methods were several hundred years old and had been discovered at a time when digital computers were not even dreamt of.

This kind of renaissance of old computational methods happens in most areas when computers are introduced as a tool. However, there is usually a continuation of the story that is worth mentioning. After some years, when the classical computational methods have been adapted to work well on modern computers, completely new methods often appear. The classical methods were usually intended to be performed by hand, using pencil and paper. Three characteristics of this computing environment (human with pencil and paper) is that it is quite slow, is error prone, and has a preference for computations with simple numbers. On the other hand, an electronic computer is fast (billions of operations pr. second), is virtually free of errors and has no preference for particular numbers. A computer can of course execute the classical methods designed for humans very well. However, it seems reasonable to expect that even better methods should be obtainable if one starts from scratch and develops new methods that exploit the characteristics of the electronic computer. This has indeed proved to be the case in many fields where the classical methods have been succeeded by better methods that are completely unsuitable for human operation.

1.2   Computers and different types of information

The computer has become a universal tool that is used for all kinds of different purposes and tasks. To understand how this has become possible we must know a little bit about how a computer operates. A computer can really only work with numbers, and in fact, the numbers even have to be expressed in terms of 0s and 1s. It turns out that any number can be represented in terms of 0s and 1s so that is no restriction. But how can computers work with text, sound, images and many other forms of information when it can really only handle numbers?

3

1.2.1 Text

Let us first consider text. In the English alphabet there are 26 letters. If we include upper case and lower case letters plus comma, question mark, space, and other common characters we end up with a total of about 100 different characters. How can a computer handle these characters when it only knows about numbers? The solution is simple; we just assign a numerical code to each character. Suppose we use two decimal digits for the code and that 'a' has the code 01, 'b' the code 02 and so on. Then we can refer to the different letters via these codes, and words can be referred to by sequences of codes. The word 'and' for example, can be referred to by the sequence 011404 (remember that each code consists of two digits). Multi-word texts can be handled in the same way as long as we have codes for all the characters. For this to work, the computer must always know how to interpret numbers presented to it, either as numbers or characters or something else.

1.2.2 Sound

Computers work with numbers, so a sound must be converted to numbers before it can be handled by a computer. What we perceive as sound corresponds to small variations in air pressure. Sound is therefore converted to numbers by measuring the air pressure at regular intervals and storing the measurements as numbers. On a CD for example, measurements are taken 44 100 times a second, so three minutes of sound becomes 7 938 000 measurements of air pressure. Sound on a computer is therefore just a long sequence of numbers. The process of converting a given sound to regular numerical measurements of the air pressure is referred to as digitising the sound, and the result is referred to as digital sound.

1.2.3 Images

Images are handled by computers in much the same way as sound. Digital cameras have an image sensor that records the amount of light hitting its rectangular array of points called pixels. The amount of light at a given pixel corresponds to a number, and the complete image can therefore be stored by storing all the pixel values. In this way an image is reduced to a large collection of numbers, a digital image, which is perfect for processing by a computer.

1.2.4 Film

A film is just a sequence of images shown in quick succession (25-30 images pr. second), and if each image is represented digitally, we have a film represented

by a large number of numerical values, a digital film. A digital film can be manipulated by altering the pixel values in the individual images.

1.2.5 Geometric form

Geometric shapes surround us everywhere in the form of natural objects like rocks, flowers and animals as well as man-made objects like buildings, aircrafts and other machines. A specific shape can be converted to numbers by splitting it into small pieces that each can be represented as a simple mathematical function like for instance a cubic polynomial. A cubic polynomial is represented in terms of its coefficients, which are numbers, and the complete shape can be represented by a collection of cubic pieces, joined smoothly together, i.e., by a set of numbers. In this way a mathematical model of a shape can be built inside a computer.

Graphical images of characters, or fonts, is one particular type of geometric form that can be represented in this way. Therefore, when you read the letters on this page, whether on paper or a computer screen, the computer figured out exactly how to draw each character by computing its shape from a collection of mathematical formulas.

1.2.6 Laws of nature

The laws of nature, especially the laws of physics, can often be expressed in terms of mathematical equations. These equations can be represented in terms of their coefficients and solved by performing computations based on these coefficients. In this way we can simulate physical phenomena by solving the equations that govern the phenomena.

1.2.7 Virtual worlds

We have seen how we can represent and manipulate sound, film, geometry and physical laws by computers. By combining objects of this kind, we can create artificial or virtual worlds inside a computer, built completely from numbers. This is exactly what is done in computer games. A complete world is built with geometric shapes, creatures that can move (with movements governed by mathematical equations), and physical laws, also represented by mathematical equations. An important part of creating such virtual worlds is to deduce methods for how the objects can be drawn on the computer screen — this is the essence of the field of computer graphics.

A very similar kind of virtual world is used in machines like flight simulators and machines used for training purposes. In many cases it is both cheaper and safer to give professionals their initial training by using a computer simulator rather than letting them try 'the real thing'. This applies to pilots as well as

heart surgeons and requires that an adequate virtual world is built in terms of mathematical equations, with a realistic user interface.

In many machines this is taken a step further. A modern passenger jet has a number of computers that can even land the airplane. To do this the computer must have a mathematical model of itself as well as the equations that govern the plane. In addition the plane must be fitted with sensors that measure quantities like speed, height, and direction. These data are measured at regular time intervals and fed into the mathematical model. Instead of just producing a film of the landing on a computer screen, the computer can actually land the aircraft, based on the mathematical model and the data provided by the sensors.

In the same way surgeons may make use of medical imaging techniques to obtain different kinds of information about the interior of the patient. This information can then be combined to produce an image of the area undergoing surgery, which is much more informative to the surgeon than the information that is available during traditional surgery.

Similar virtual worlds can also be used to perform virtual scientific experiments. In fact a large part of scientific experiments are now performed by using a computer to solve the mathematical equations that govern an experiment rather than performing the experiment itself.

1.2.8   Summary

Via measuring devices (sensors), a wide range of information can be converted to digital form, i.e., to numbers. These numbers can be read by computers and processed according to mathematical equations or other logical rules. In this way both real and non-real phenomena can be investigated by computation. A computer can therefore be used to analyse an industrial object before it is built. For example, by making a detailed mathematical model of a car it is possible to compute its fuel consumption and other characteristics by simulation in a computer, without building a single car.

A computer can also be used to guide or run machines. Again the computer must have detailed information about the operation of the machine in the form of mathematical equations or a strict logical model from which it can compute how the machine should behave. The result of the computations must then be transferred to the devices that control the machine.

To build these kinds of models requires specialist knowledge about the phenomenon which is to be modelled as well as a good understanding of the basic tools used to solve the problems, namely mathematics, computing and computers.

1.3 Computation by hand and by computer

As a student of mathematics, it is reasonable to expect that you have at least a vague impression of what classical mathematics is. What I have in mind is the insistence on a strict logical foundation of all concepts like for instance differentiation and integration, logical derivation of properties of the concepts defined, and the development of symbolic computational techniques like symbolic integration and differentiation. This is all extremely important and should be well-known to any serious student of mathematics and the mathematical sciences. Not least is it important to be fluent in algebra and symbolic computations.

When computations are central to classical mathematics, what then is the *new* computational approach? To understand this we first need to reflect a bit on how we do our pencil-and-paper computations. Suppose you are to solve a system of three linear equations in three unknowns, like

$$2x + 4y - 2z = 2,$$
$$3x - 6z = 3,$$
$$4x - 2y + 4z = 2.$$

There are many different ways to solve this, but one approach is as follows. We observe that the middle equation does not contain y, so we can easily solve for one of x or z in terms of the other. If we solve for x we can avoid fractions so this seems like the best choice. From the second equation we then obtain $x = 1 + 2z$. Inserting this in the first and last equations gives

$$2 + 4z + 4y - 2z = 2,$$
$$4 + 8z - 2y + 4z = 2,$$

or

$$4y + 2z = 0,$$
$$-2y + 12z = -2.$$

Using either of these equations we can express y or z in terms of one another. In the first equation, however, the right-hand side is 0 and we know that this will lead to simpler arithmetic. And if we express z in terms of y we avoid fractions. From the first equation we then obtain $z = -2y$. When this is inserted in the last equation we end up with $-2y + 12(-2y) = -2$ or $-26y = -2$ from which we see that $y = 1/13$. We then find $z = -2y = -2/13$ and $x = 1 + 2z = 9/13$. This illustrates how an experienced equation solver typically works, always looking for shortcuts and simple numbers that simplify the calculations.

This is quite different from how a computer operates. A computer works according to a very detailed procedure which states exactly how the calculations are to be done. The procedure can tell the computer to look for simple numbers and shortcuts, but this is usually a waste of time since most computers handle fractions just as well as integers.

Another, more complicated example, is computation of symbolic integrals. For most of us this is a bag of isolated techniques and tricks. In fact the Norwegian mathematician Viggo Brun once said that *differentiation is a craft; integration is an art.* If you have some experience with differentiation you will understand what Brun meant by it being a craft; you arrive at the answer by following fairly simple rules. Many integrals can also be solved by definite rules, but the more complicated ones require both intuition and experience. And in fact most indefinite integrals cannot be solved at all. It may therefore come as a surprise to many that computers can be programmed to perform symbolic integration. In fact, Brun was wrong. There is a precise procedure which will always give the answer to the integral if it exists, or say that it does not exist if this is the case. For a human the problem is of course that the procedure requires so much work that for most integrals it is useless, and integration therefore appears to be an art. For computers, which work so must faster, this is less of a problem, see Figure 1.1. Still there are plenty of integrals (most!) that require so many calculations that even the most powerful computers are not fast enough. Not least would the result require so much space to print that it would be incomprehensible to humans!

These simple examples illustrate that when (experienced) humans do computations they try to find shortcuts, look for patterns and do whatever they can to simplify the work; in short they tend to improvise. In contrast, computations on a computer must follow a strict, predetermined algorithm. A computer may appear to improvise, but such improvisation must necessarily be planned in advance and built into the procedure that governs the calculations.

1.4 Algorithms

In the previous section we repeatedly talked about the 'procedure' that governs a calculation. This procedure is simply a sequence of detailed instructions for how the quantity in question can be computed; such procedures are usually referred to as *algorithms*. Algorithms have always been important in mathematics as they specify how calculations should be done. In the past, algorithms were usually intended to be performed manually by humans, but today many algorithms are designed to work well on digital computers.

If we want an algorithm to be performed by a computer, it must be expressed

$$\int \frac{\sin(x)}{\cos(6x)}\,dx = \left(\frac{1}{6}+\frac{i}{6}\right)(-1)^{1/4}\,\mathrm{ArcTan}\left[\left(\frac{1}{2}+\frac{i}{2}\right)(-1)^{1/4}\,\mathrm{Sec}\left[\frac{x}{2}\right]\left(\mathrm{Cos}\left[\frac{x}{2}\right]+\mathrm{Sin}\left[\frac{x}{2}\right]\right)\right]-$$

$$\left(\frac{1}{6}+\frac{i}{6}\right)(-1)^{3/4}\,\mathrm{ArcTanh}\left[\left(\frac{1}{2}+\frac{i}{2}\right)(-1)^{3/4}\,\mathrm{Sec}\left[\frac{x}{2}\right]\left(\mathrm{Cos}\left[\frac{x}{2}\right]-\mathrm{Sin}\left[\frac{x}{2}\right]\right)\right]+$$

$$\frac{1}{12\left(2+\sqrt{2}\right)}\left(1+\sqrt{2}\right)\left(x+2\sqrt{3}\;\mathrm{ArcTanh}\left[\frac{2+\left(2+\sqrt{2}\right)\mathrm{Tan}\left[\frac{x}{2}\right]}{\sqrt{6}}\right]-\mathrm{Log}\left[\mathrm{Sec}\left[\frac{x}{2}\right]^2\right]+$$

$$\mathrm{Log}\left[\mathrm{Sec}\left[\frac{x}{2}\right]^2\left(\sqrt{2}-2\,\mathrm{Cos}[x]+2\,\mathrm{Sin}[x]\right)\right]\right)+\left(\left(2\left(\sqrt{2}+\sqrt{3}\right)\mathrm{ArcTanh}\left[\frac{2+\left(2+\sqrt{6}\right)\mathrm{Tan}\left[\frac{x}{2}\right]}{\sqrt{2}}\right]+\right.\right.$$

$$\left.\left(3+\sqrt{6}\right)\left(x-\mathrm{Log}\left[\mathrm{Sec}\left[\frac{x}{2}\right]^2\right]+\mathrm{Log}\left[\mathrm{Sec}\left[\frac{x}{2}\right]^2\left(\sqrt{6}-2\,\mathrm{Cos}[x]+2\,\mathrm{Sin}[x]\right)\right]\right)\right)$$

$$\left(1+\sqrt{6}\,\mathrm{Sin}[x]\right)\left(3+\sqrt{6}-\left(2+\sqrt{6}\right)\mathrm{Cos}[x]+\left(2+\sqrt{6}\right)\mathrm{Sin}[x]\right)\right)\Bigg/$$

$$\left(12\left(\left(12+5\sqrt{6}\right)\mathrm{Cos}[2x]+2\,\mathrm{Cos}[x]\left(5+2\sqrt{6}+5\sqrt{6}\,\mathrm{Sin}[x]\right)-\right.\right.$$

$$\left.\left.2\left(12+5\sqrt{6}+4\left(5+2\sqrt{6}\right)\mathrm{Sin}[x]-6\,\mathrm{Sin}[2x]\right)\right)\right)+$$

$$\left(\left(x-2\sqrt{3}\;\mathrm{ArcTanh}\left[\frac{\sqrt{2}+\left(-1+\sqrt{2}\right)\mathrm{Tan}\left[\frac{x}{2}\right]}{\sqrt{3}}\right]-\mathrm{Log}\left[\mathrm{Sec}\left[\frac{x}{2}\right]^2\right]+\right.\right.$$

$$\left.\mathrm{Log}\left[-\mathrm{Sec}\left[\frac{x}{2}\right]^2\left(1+\sqrt{2}\,\mathrm{Cos}[x]-\sqrt{2}\,\mathrm{Sin}[x]\right)\right]\right)$$

$$\left(\sqrt{2}+2\,\mathrm{Sin}[x]\right)\left(-1+\sqrt{2}-\left(-2+\sqrt{2}\right)\mathrm{Cos}[x]+\left(-2+\sqrt{2}\right)\mathrm{Sin}[x]\right)\right)\Bigg/$$

$$\left(24\left(\left(-2+\sqrt{2}\right)\mathrm{Cos}[x]-\left(-1+\sqrt{2}\right)(\mathrm{Cos}[2x]+\mathrm{Sin}[2x])\right)\right)+$$

$$\left(\left(-2\left(-2+\sqrt{6}\right)\mathrm{ArcTanh}\left[\sqrt{2}+\left(\sqrt{2}-\sqrt{3}\right)\mathrm{Tan}\left[\frac{x}{2}\right]\right]+\right.\right.$$

$$\left.\left(3\sqrt{2}-2\sqrt{3}\right)\left(x-\mathrm{Log}\left[\mathrm{Sec}\left[\frac{x}{2}\right]^2\right]+\mathrm{Log}\left[-\mathrm{Sec}\left[\frac{x}{2}\right]^2\left(\sqrt{3}+\sqrt{2}\,\mathrm{Cos}[x]-\sqrt{2}\,\mathrm{Sin}[x]\right)\right]\right)\right)$$

$$\left(\sqrt{2}-2\sqrt{3}\,\mathrm{Sin}[x]\right)\left(-3+\sqrt{6}-\left(-2+\sqrt{6}\right)\mathrm{Cos}[x]+\left(-2+\sqrt{6}\right)\mathrm{Sin}[x]\right)\right)\Bigg/$$

$$\left(24\left(\left(-12+5\sqrt{6}\right)\mathrm{Cos}[2x]+2\,\mathrm{Cos}[x]\left(-5+2\sqrt{6}+5\sqrt{6}\,\mathrm{Sin}[x]\right)-\right.\right.$$

$$\left.\left.2\left(-12+5\sqrt{6}+4\left(-5+2\sqrt{6}\right)\mathrm{Sin}[x]+6\,\mathrm{Sin}[2x]\right)\right)\right)$$

Figure 1.1. An integral and its solution as computed by the computer program Mathematica. The function $\sec(x)$ is given by $\sec(x) = 1/\cos(x)$.

9

in a form that the computer understands. Various languages, such as C++, Java, Python, Matlab etc., have been developed for this purpose, and a computer program is nothing but an algorithm translated into such a language. Programming therefore requires both an understanding of the relevant algorithms and knowledge of the programming language to be used.

We will express the algorithms we encounter in a language close to standard mathematics which should be quite easy to understand. This means that if you want to test an algorithm on a computer, it must be translated to your preferred programming language. For the simple algorithms we encounter, this process should be straightforward, provided you know your programming language well enough.

1.4.1 Statements

The building blocks of algorithms are *statements*, and statements are simple operations that form the basis for more complex computations.

Definition 1.1. *An algorithm is a finite sequence of statements. In these notes there are only five different kinds of statements:*

1. Assignments

2. For-loops

3. If-tests

4. While-loops

5. Print statement

Statements may involve expressions, *which are combinations of mathematical operations, just like in general mathematics.*

The first four types of statements are the important ones as they cause calculations to be done and control how the calculations are done. As the name indicates, the print statement is just a tool for communicating to the user the results of the computations.

Below, we are going to be more precise about what we mean by the five kinds of statements, but let us also ensure that we agree what expressions are. The most common expressions will be formulas like $a + bc$, $\sin(a + b)$, or $e^{x/y}$. But an expression could also be a bit less formal, like *"the list of numbers $\boldsymbol{x}$ sorted in increasing order"*. Usually expressions only involve the basic operations in the

mathematical area we are currently studying and which the algorithm at hand relates to.

1.4.2 Variables and assignment

Mathematics is in general known for being precise, but its notation sometimes borders on being ambiguous. An example is the use of the equals sign, '='. When we are solving equations, like $x + 2 = 3$, the equals sign basically tests equality of the two sides, and the equation is either true or false, depending on the value of x. On the other hand, in an expression like $f(x) = x^2$, the equals sign acts like a kind of definition or *assignment* in that we assign the value x^2 to $f(x)$. In most situations the interpretation can be deduced by the context, but there are situations where confusion may arise as we will see in section 2.3.1.

Computers are not very good at judging this kind of context, and therefore most programming languages differentiate between the two different uses of '='. For this reason it is also convenient to make the same kind of distinction when we describe algorithms. We do this by introducing the operator := for assignment and retaining = for comparison.

When we do computations, we may need to store the results and intermediate values for later use, and for this we use variables. Based on the discussion above, to store the number 2 in a variable a, we will use the notation $a := 2$; we say that the variable a is *assigned* the value 2. Similarly, to store the sum of the numbers b and c in a, we write $a := b + c$. One important feature of assignments is that we can write something like $s := s + 2$. This means: Take the value of s, add 2, and store the result back in s. This does of course mean that the original value of s is lost.

Definition 1.2 (Assignment)**.** *The formulation*

$$var := expression;$$

means that the expression on the right is to be calculated, and the result stored in the variable var. For clarity the expression is often terminated by a semicolon.

Note that the assignment $a := b + c$ is different from the mathematical equation $a = b + c$. The latter basically tests equality: It is true if a and $b + c$ denote the same quantities, and false otherwise. The assignment is more like a command: Calculate the the right-hand side and store the result in the variable on the right.

1.4.3 For-loops

Very often in algorithms it is necessary to repeat essentially the same thing many times. A common example is calculation of a sum. An expression like

$$s = \sum_{i=1}^{100} i$$

in mathematics means that the first 100 integers should be added together. In an algorithm we may need to be a bit more precise since a computer can really only add two numbers at a time. One way to do this is

$$s := 0;$$
for $i := 1, 2, \ldots, 100$
$\qquad s := s + i;$

The sum will be accumulated in the variable s, and before we start the computations we make sure s has the value 0. The for-statement means that the variable i will take on all the values from 1 to 100, and each time we add i to s and store the result in s. After the for-loop is finished, the total sum will then be stored in s.

Definition 1.3 (For-loop). *The notation*

    for *var* := *list of values*
       sequence of statements;

means that the variable var *will take on the values given by* list of values. *For each such value, the indicated* sequence of statements *will be performed. These may include expressions that involve the loop-variable* var.

A slightly more complicated example than the one above is

$s := 0;$
for $i := 1, 2, \ldots, 100$
$\qquad x := \sin(i);$
$\qquad s := s + x;$
$s := 2s;$

which calculates the sum $s = 2\sum_{i=1}^{100} \sin i$. Note that the two indented statements are both performed for each iteration of the for-loop, while the non-indented statement is performed after the for-loop has finished.

1.4.4   If-tests

The third kind of statement lets us choose what to do based on whether or not a condition is true. The general form is as follows.

> **Definition 1.4** (If-statement). *Consider the statement*
>
>   **if** *condition*
>   *sequence of statements;*
>   **else**
>   *sequence of statements;*
>
> *where* condition *denotes an expression that is either true or false. The meaning of this is that the first group of statements will be performed if condition is true, and the second group of statements if condition is false.*

As an example, suppose we have two numbers a and b, and we want to find the largest and store this in c. This can be done with the if-statement

   if $a < b$
       $c := b$;
   else
       $c := a$;

The condition in the if-test can be any expression that evaluates to true or false. In particular it could be something like $a = b$ which tests whether a and b are equal. This should not be confused with the assignment $a := b$ which causes the value of b to be stored in a.

Our next example combines all the three different kinds of statements we have discussed so far. Many other examples can be found in later chapters.

Example 1.5. Suppose we have a sequence of real numbers $(a_k)_{k=1}^n$, and we want to compute the sum of the negative and the positive numbers in the sequence separately. For this we need to compute two sums which we will store in the variables $s1$ and $s2$: In $s1$ we will store the sum of the positive numbers, and in $s2$ the sum of the negative numbers. To determine these sums, we step through the whole sequence, and check whether an element a_k is positive or negative. If it is positive we add it to $s1$ otherwise we add it to $s2$. The following algorithm accomplishes this.

   $s1 := 0$; $s2 := 0$;
   for $k := 1, 2, \ldots, n$
       if $a_k > 0$
           $s1 := s1 + a_k$;

    else
$$s2 := s2 + a_k;$$

After these statements have been performed, the two variables $s1$ and $s2$ should contain the sums of the positive and negative elements of the sequence, respectively. ∎

1.4.5 While-loops

The final type of statement that we need is the while-loop, which is a combination of a for-loop and an if-test.

Definition 1.6 (While-statement). *Consider the statement*

    while *condition*
       sequence of statements;

This will repeat the sequence of statements as long as condition *is true.*

Note that unless the logical condition depends on the computations in the sequence of statements this loop will either not run at all or run forever. Note also that a for-loop can always be replaced by a while-loop.

Consider once again the example of adding the first 100 integers. With a while-loop this can be expressed as

$$s := 0;\ i := 1;$$
    while $i \leq 100$
$$s := s + 1;$$
$$i := i + i;$$

This example is expressed better with a for-loop, but it illustrates the idea behind the while-loop. A typical situation where a while-loop is convenient is when we compute successive approximations to some quantity. In such situations we typically want to continue the computations until some measure of the error has become smaller than a given tolerance, and this is expressed best with a while-loop.

1.4.6 Print statement

Occasionally we may want our toy computer to print something. For this we use a print statement. As an example, we could print all the integers from 1 to 100 by writing

```
for i = 1, 2, ..., 100
    print i;
```

Sometimes we may want to print more elaborate texts; the syntax for this will be introduced when it is needed.

1.5 Doing computations on a computer

So far, we have argued that computations are important in mathematics, and computers are good at doing computations. We have also seen that humans and computers do calculations in quite different ways. A natural question is then how you can make use of computers in your calculations. And once you know this, the next question is how you can learn to use computers in this way.

1.5.1 How can computers be used for calculations?

There are at least two essentially distinct ways in which you can use a computer to do calculations:

1. You can use software written by others; in other words you may use the computer as an advanced calculator.

2. You can develop your own algorithms and implement these in your own programs.

Anybody who uses a computer has to depend on software written by others, so if your are going to do mathematics by computer, you will certainly do so in the 'calculator style' sometimes. The simplest example is the use of a calculator for doing arithmetic. A calculator is nothing but a small computer, and we all know that calculators can be very useful. There are many programs available which you can use as advanced calculators for doing common computations like plotting, integration, algebra and a wide range of other mathematical routine tasks.

The calculator style of computing can be very useful and may help you solve a variety of problems. The goal of these notes however, is to help you learn to develop your own algorithms which you can then implement in your own computer programs. This will enable you to deduce new computer methods and solve problems which are beyond the reach of existing algorithms.

When you develop new algorithms, you usually want to implement the algorithms in a computer program and run the program. To do this you need to know a programming language, i.e., an environment in which you can express your algorithm in such a way that a computer can execute the algorithm. It is

therefore assumed that you are familiar with a suitable programming language already, or that you are learning one while you are working with these notes. Virtually any programming language like Java, C++, Python, Matlab, Mathematica, …, will do. The algorithms in these notes will be written in a form that should make it quite simple to translate them to your choice of programming language. Note however that it will usually not work to just type the text literally into C++ or Python; you need to know the syntax (grammar) of the language you are using and translate the algorithm accordingly.

1.5.2   What do you need to know?

There are a number of things you need to learn in order to become able to deduce efficient algorithms and computer programs:

- You must learn to recognise when a computer calculation is appropriate, and when formal methods or calculations by hand are more suitable

- You must learn to translate your informal mathematical ideas into detailed algorithms that are suitable for computers

- You must understand the characteristics of the computing environment defined by your computer and the programming language you use

Let us consider each of these points in turn. Even if the power of a computer is available to you, you should not forget your other mathematical skills. Sometimes your intuition, computation by hand or logical reasoning will serve you best. On the other hand, with good algorithmic skills you can often use the computer to answer questions that would otherwise be impossible even to consider. You should therefore aim to gain an intuitive understanding for when a mathematical problem is suitable for computer calculation. It is difficult to say exactly when this is the case; a good learning strategy is to read these notes and see how algorithms are developed in different situations. As you do this you should gradually develop an algorithmic thinking yourself.

Once you have decided that some computation is suitable for computer implementation you need to formulate the calculation as a precise algorithm that only uses operations available in your computing environment. This is also best learnt through practical experience, and you will see many examples of this process in these notes.

An important prerequisite for both of the first points is to have a good understanding of the characteristics of the computing environment where you intend to do your computations. At the most basic level, you need to understand the general principles of how computers work. This may sound a bit overwhelming,

but at a high level, these principles are not so difficult, and we will consider most of them in later chapters.

1.5.3 Different computing environments

One interesting fact is that as your programming skills increase, you will begin to operate in a number of different computing environments. We will not consider this in any detail here, but a few examples may illustrate this point.

Sequential computing As you begin using a computer for calculations it is natural to make the assumption that the computer works sequentially and does one operation at a time, just like we tend to do when we perform computations manually.

Suppose for example that you are to compute the sum

$$s = \sum_{i=1}^{100} a_i$$

where each a_i is a real number. Most of us would then first add a_1 and a_2, remember the result, then add a_3 to this result and remember the new result, then add a_4 to this and so on until all numbers have been added. The good news is that you can do exactly the same on a computer! This is called sequential computing and is definitely the most common computing environment.

Parallel computing If a group of people work together it is possible to add numbers faster than a single person can. The key observation is that the numbers can be summed in many different ways. We may for example sum the numbers in the order indicated by

$$s = \sum_{i=1}^{100} a_i = \underbrace{a_1 + a_2}_{a_1^1} + \underbrace{a_3 + a_4}_{a_2^1} + \underbrace{a_5 + a_6}_{a_3^1} + \cdots + \underbrace{a_{97} + a_{98}}_{a_{49}^1} + \underbrace{a_{99} + a_{100}}_{a_{50}^1}.$$

The key is that these sums are independent of each other. In other words we may hire 50 people, give them two numbers each and tell them to compute the sum. And when everybody is finished, we can of course repeat this and ask 25 people to add the 50 results,

$$s = \sum_{i=1}^{50} a_i^1 = \underbrace{a_1^1 + a_2^1}_{a_1^2} + \underbrace{a_3^1 + a_4^1}_{a_2^2} + \underbrace{a_5^1 + a_6^1}_{a_3^2} + \cdots + \underbrace{a_{47}^1 + a_{48}^1}_{a_{24}^2} + \underbrace{a_{49}^1 + a_{50}^1}_{a_{25}^2}.$$

17

At the next step we ask 13 people to the compute the 13 sums

$$s = \sum_{i=1}^{25} a_i^2 = \underbrace{a_1^2 + a_2^2}_{a_1^3} + \underbrace{a_3^2 + a_4^2}_{a_2^3} + \underbrace{a_5^2 + a_6^2}_{a_3^3} + \cdots + \underbrace{a_{21}^2 + a_{22}^2}_{a_{11}^2} + \underbrace{a_{23}^2 + a_{24}^2}_{a_{12}^3} + \underbrace{a_{25}^2}_{a_{13}^3}.$$

Note that the last person has an easy job; since the total number of terms in this sum is an odd number she just needs to remember the result.

The structure should now be clear. At the next step we ask 7 people to compute pairs in the sum $s = \sum_{i=1}^{13} a_i^3$ in a similar way. The result is the 7 numbers $a_1^4, a_2^4, \ldots, a_7^4$. We then ask 4 people to compute the pairs in the sum $s = \sum_{i=1}^{7} a_i^4$ which results in the 4 numbers a_1^5, a_2^5, a_3^5 and a_4^5. Two people can then add pairs in the sum $s = \sum_{i=1}^{4} a_i^5$ and obtain the two numbers a_1^6 and a_2^6. Finally one person computes the final sum as $s = a_1^6 + a_2^6$.

Note that at each step, everybody can work independently. At the first step we therefore compute 25 sums in the time that it takes one person to compute one sum. The same is true at each step and the whole sum is computed in 6 steps. If one step takes 10 seconds, we have computed the sum of 100 numbers in one minute, while a single person would have taken 990 seconds or 16 minutes and 30 seconds.

Our simple example illustrates the concept of parallel computing. Instead of making use of just one computing unit, we may attack a problem with several units. Supercomputers, which are specifically designed for number crunching, work on this principle. Today's (July 2007) most powerful computer has 65 536 computing units, each of which again has two processors for a total of 131 072 computing units.

An alternative to expensive supercomputers is to let standard PCs work in parallel. They can either be connected in a specialised network or can communicate via the Internet. [1] In fact, modern PCs themselves are so-called multi-core computers which consist of several computing units or CPUs, although at present, the norm is at most 4 or 8 cores.

One challenge with parallel computing that we have overlooked here is the need for communication between the different computing units. Once the 25 persons have completed their sums, they must communicate their results to the 12 people who are going to do the next sums. This time is significant and supercomputers have very sophisticated communication channels. At the other end of the scale, the Internet is in most respects a relatively slow communication channel for parallel computing.

---

[1] There is a project aimed at computing large prime numbers that make use of the internet in this way, see `www.mersenne.org`.

Other computing environments Computing environments are characterised by many other features than whether or not calculations can be done in parallel. Other characteristics are the number of digits used in numerical computations, how numbers are represented internally in the machine, whether symbolic calculations are possible, and so on. It is not necessary to know the details of how these issues are handled on your computer, but if you want to use the computer efficiently, you need to understand the basic principles. After having studied these notes you should have a decent knowledge of the most common computing environments.

Exercises

1.1 The algorithm in example 1.5 calculates the sums of the positve and negative numbers in a sequence $(a_k)_{k=1}^n$. In this exercise you are going to adjust this algorithm.

    a) Change the algorithm so that it computes the sum of the positive numbers and the absolute value of the sum of the negative numbers.

    b) Change the algorithm so that it also determines the number of positive and negative elements in the sequence.

1.2 Formulate an algorithm for adding two three-digit numbers. You may assume that it is known how to sum one-digit numbers.

1.3 Formulate an algorithm which describes how you multiply together two three-digit numbers. You may assume that it is known how to add numbers .

Part I

Numbers

CHAPTER 2

0 and 1

'0 and 1' may seem like an uninteresting title for this first proper chapter, but most readers probably know that at the most fundamental level computers always deal with 0s and 1s. Here we will first learn about some of the advantages of this, and then consider some of the mathematics of 0 and 1.

2.1   Robust communication

Suppose you are standing at one side of a river and a friend is standing at the other side, 500 meters away; how can you best communicate with your friend in this kind of situation, assuming you have no aids at your disposal? One possibility would be to try and draw the letters of the alphabet in the air, but at this distance it would be impossible to differentiate between the different letters as long as you only draw with your hands. What is needed is a more robust way to communicate where you are not dependent on being able to decipher so many different symbols. As far as robustness is concerned, the best would be to only use two symbols, say 'horizontal arm' and 'vertical arm' or 'h' and 'v' for short. You can then represent the different letters in terms of these symbols. We could for example use the coding shown in table 2.1 which is built up in a way that will become evident in chapter 3. You would obviously have to agree on your coding system in advance.

The advantage of using only two symbols is of course that there is little danger of misunderstanding as long as you remember the coding. You only have to differentiate between two arm positions, so you have generous error margins for how you actually hold your arm. The disadvantage is that some of the letters require quite long codes. In fact, the letter 's' which is among the most frequently used in English, requires a code with five arm symbols, while the two letters 'a' and 'b' which are less common both require one symbol each. If you were to make heavy use of this coding system it would therefore make sense to reorder the letters such that the most frequent ones (in your language) have the shortest codes.

2.2   Why 0 and 1 in computers?

The above example of human communication across a river illustrates why it is not such a bad idea to let computers operate with only two distinct symbols which we may call '0' and '1' just as well as 'h' and 'v'. A computer is built to

a	h	j	vhhv	s	vhhvh
b	v	**k**	vhvh	**t**	vhhvv
c	vh	**l**	vhvv	**u**	vhvhh
d	vv	**m**	vvhh	**v**	vhvhv
e	vhh	**n**	vvhv	**w**	vhvvh
f	vhv	**o**	vvvh	**x**	vhvvv
g	vvh	**p**	vvvv	**y**	vvhhh
h	vvv	**q**	vhhhh	**z**	vvhhv
i	vhhh	**r**	vhhhv		

Table 2.1. Representation of letters in terms of 'horizontal arm' ('h') and 'vertical arm' ('v').

manipulate various kinds of information and this information must be moved between the different parts of the computer during the processing. By representing the information in terms of 0s and 1s, we have the same advantages as in communication across the river, namely robustness and the simplicity of having just two symbols.

In a computer, the 0s and 1s are represented by voltages, magnetic charges, light or other physical quantities. For example 0 may be represented by a voltage in the interval 1.5 V to 3 V and 1 by a voltage in the interval 5 V to 6.5 V. The robustness is reflected in the fact that there is no need to measure the voltage accurately, we just need to be able to decide whether it lies in one of the two intervals. This is also a big advantage when information is stored on an external medium like a DVD or hard disk, since we only need to be able to store 0s and 1s. A '0' may be stored as 'no reflection' and a '1' as 'reflection', and when light is shone on the appropriate area we just need to detect whether the light is reflected or not.

A disadvantage of representing information in terms of 0s and 1s is that we may need a large amount of such symbols to encode the information we are interested in. If we go back to table 2.1, we see that the 'word' hello requires 18 symbols ('h's and 'v's), and in addition we have to also keep track of the boundaries between the different letters. The cost of using just a few symbols is therefore that we must be prepared to process large numbers of them.

Although representation of information in terms of 0s and 1s is very robust, it is not foolproof. Small errors in for example a voltage that represents a 0 or 1 do not matter, but as the voltage becomes more and more polluted by noise, its value will eventually go outside the permitted interval. It will then be impossible to tell which symbol the value was meant to represent. This means that increasing noise levels will not be noticed at first, but eventually the noise will break the

threshold which will make it impossible to recognise the symbol and therefore the information represented by the symbol.

When we think of the wide variety of information that can be handled by computers, it may seem quite unbelievable that it is all comprised of 0s and 1s. In chapter 1 we saw that information commonly processed by computers can be represented by numbers, and in the next chapter we shall see that all numbers may be expressed in terms of 0s and 1s. The conclusion is therefore that a wide variety of information can be represented in terms of 0s and 1s.

Observation 2.1 (0 and 1 in computers)**.** *In a computer, all information is usually represented in terms of two symbols, '0' and '1'. This has the advantage that the representation is robust with respect to noise, and the electronics necessary to process one symbol is simple. The disadvantage is that the code needed to represent a piece of information becomes longer than what would be the case if more symbols were used.*

Whether we call the two possible values 0 and 1, 'v' and 'h' or 'yes' and 'no' does not matter. What is important is that there are only two symbols, and what these symbols are called usually depends on the context. An important area of mathematics that depends on only two values is logic.

2.3   True and False

In everyday speech we make all kinds of statements and some of them are objective and precise enough that we can check whether or not they are true. Most people would for example agree that the statements 'Oslo is the capital of Norway' and 'Red is a colour' are true, while there is less agreement about the statement 'Norway is a beautiful country'. In normal speech we also routinely link such logical statements together with words like 'and' and 'or', and we negate a statement with 'not'.

Mathematics is built by strict logical statements that are either true or false. Certain statements which are called axioms, are just taken for granted and form the foundation of mathematics (something cannot be created from nothing). Mathematical proofs use logical operations like 'and', 'or', and 'not' to combine existing statements and obtain new ones that are again either true or false. For example we can combine the two true statements 'π is greater than 3' and 'π is smaller than 4' with 'and' and obtain the statement 'π is greater than 3 and π is smaller than 4' which we would usually state as 'π lies between 3 and 4'. Likewise the statement 'π is greater than 3' can be negated to the opposite statement 'π is less than or equal to 3' which is false.

Even though this description is true, doing mathematics is much more fun than it sounds. Not all new statements are interesting even though they may be true. To arrive at interesting new truths we use intuition, computation and any other aids at our disposal. When we feel quite certain that we have arrived at an interesting statement comes the job of constructing the formal proof, i.e., combining known truths in such a way that we arrive at the new statement. If this sounds vague you should get a good understanding of this process as you work your way through any university maths course.

2.3.1 Logical variables and logical operators

When we introduced the syntax for algorithms in section 1.4, we noted the possibility of confusion between assignment and test for equality. To avoid this we introduced the notation := for the assignment operator. We are now going to discuss logical expressions which may involve tests for equality, but will also make definitions by using assignments, so for this chapter we adopt the algorithmic notation of using := for assignment.

In this section we are going to introduce the standard logical operators in more detail, and to do this, logical variables will be useful. From elementary mathematics we are familiar with using x and y as symbols that typically denote real numbers. Logical variables are similar except that they may only take the values 'true' or 'false' which we now denote by T and F. So if p is a logical variable, it may denote any logical statement. As an example, we may set

$$p := (4 > 3)$$

which is the same as setting $p :=$ T. More interestingly, if a is any real number we may set

$$p := (a > 3).$$

The value of p will then be either T or F, depending on the value of a so we may think of $p := p(a)$ as a function of a. We then clearly have $p(2) =$ F and $p(4) =$ T. All the usual relational operators like $<$, $>$, $\leq$ and $\geq$ can be used in this way.

The function

$$p(a) := (a = 2)$$

has the value T if a is 2 and the value F otherwise. Without the special notation for assignment this would become $p(a) = (a = b)$ which certainly looks rather confusing.

> **Definition 2.2.** *In the context of logic, the values true and false are denoted* T *and* F*, and assignment is denoted by the operator :=. A logical statement is an expression that is either* T *or* F *and a logical function p(a) is a function that is either* T *or* F*, depending on the value of a.*

Suppose now that we have two logical variables p and q. We have already mentioned that these can be combined with the operators 'and', 'or' and 'not' for which we now introduce the notation $\land$, $\lor$ and $\lnot$. Let us consider each of these in turn.

The expression $\lnot p$ is the opposite of p, i.e., it is T if p is F and F if p is T, see column three in table 2.2. The only way for $p \land q$ to be T, is for both p and q to be T; in all other cases it is F, see column four in the table. Logical or is the opposite: The expression $p \lor q$ is only F if both p and q are F; otherwise it is T; see column five in table 2.2.

p	q	$\lnot p$	$p \land q$	$p \lor q$	$p \oplus q$
F	F	T	F	F	F
T	F	F	F	T	T
F	T	T	F	T	T
T	T	F	T	T	F

Table 2.2. Behaviour of the logical operators $\lnot$ (not), $\land$ (and), $\lor$ (or), and $\oplus$ (exclusive or).

This use of 'not' and 'and' is just like in everyday language. The definition of 'or', however, does not always agree with how it is used in speech. Suppose someone says 'The apple was red or green', is it then possible that the apple was both red and green? Many would probably say no, but to be more explicit we would often say 'The apple was either red or green (but not both)'.

This example shows that there are in fact two kinds of 'or', an inclusive or ($\lor$) which is T when p and q are both T, and an exclusive or ($\oplus$) which is F when both p and q are T, see columns five and six of Table 2.2.

> **Definition 2.3.** *The logical operators 'not', 'and', 'or', and 'exclusive or' are denoted by the symbols $\lnot$, $\land$, $\lor$, and $\oplus$, respectively and are defined in table 2.2.*

So far we have only considered expressions that involve two logical variables. If p, q, r and s are all logical variables, it is quite natural to consider longer ex-

pressions like

$$(p \wedge q) \wedge r, \tag{2.1}$$

$$(p \vee q) \vee (r \vee s), \tag{2.2}$$

(we will consider mixed expressions later). The brackets have been inserted to indicate the order in which the expressions are to be evaluated since we only know how to combine two logical variables at a time. However, it is quite easy to see that both expressions remain the same regardless of how we insert brackets. The expression in (2.1) is T only when all of p, q and r are T, while the expression in (2.2) is always true except in the case when all the variables are F. This means that we can in fact remove the brackets and simply write

$$p \wedge q \wedge r,$$
$$p \vee q \vee r \vee s,$$

without any risk of misunderstanding since it does not matter in which order we evaluate the expressions.

Many other mathematical operations, like for example addition and multiplication of numbers, also have this property, and it therefore has its own name; we say that the operators $\wedge$ and $\vee$ are *associative*. The associativity also holds when we have longer expressions: If the operators are either all $\wedge$ or all $\vee$, the result is independent of the order in which we apply the operators.

What about the third operator, $\oplus$ (exculsive or), is this also associative? If we consider the two expressions

$$(p \oplus q) \oplus r, \qquad p \oplus (q \oplus r),$$

the question is whether they are always equal. If we check all the combinations and write down a truth table similar to Table 2.2, we do find that the two expressions are the same so the $\oplus$ operator is also associative. A general description of such expressions is a bit more complicated than for $\wedge$ and $\vee$. It turns out that if we have a long sequence of logical variables linked together with $\oplus$, then the result is true if the number of variables that is T is an odd number and F otherwise.

The logical operator $\wedge$ has the important property that $p \wedge q = q \wedge p$ and the same is true for $\vee$ and $\oplus$. This is also a property of addition and multiplication of numbers and is usually referred to as commutativity.

For easy reference we sum all of this up in a theorem.

Proposition 2.4. *The logical operators $\wedge$, $\vee$ and $\oplus$ defined in Table 2.2 are all commutative and associative, i.e.,*

$$p \wedge q = q \wedge p, \quad (p \wedge q) \wedge r = p \wedge (q \wedge r),$$
$$p \vee q = q \vee p, \quad (p \vee q) \vee r = p \vee (q \vee r),$$
$$p \oplus q = q \oplus p, \quad (p \oplus q) \oplus r = p \oplus (q \oplus r).$$

where p, q and r are logical variables.

2.3.2 Combinations of logical operators

The logical expressions we have considered so far only involve one logical operator at a time, but in many situations we need to evaluate more complex expressions that involve several logical operators. Two commonly occurring expressions are $\neg(p \wedge q)$ and $\neg(p \vee q)$. These can be expanded by *De Morgan's laws* which are easily proved by considering truth tables for the two sides of the equations.

Lemma 2.5 (De Morgan's laws)**.** *Let p and q be logical variables. Then*

$$\neg(p \wedge q) = (\neg p) \vee (\neg q),$$
$$\neg(p \vee q) = (\neg p) \wedge (\neg q).$$

De Morgan's laws can be generalised to expressions with more than two operators, for example

$$\neg(p \wedge q \wedge r \wedge s) = (\neg p) \vee (\neg q) \vee (\neg r) \vee (\neg s),$$

see exercise 3.

We are going to consider two more laws of logic, namely the two distributive laws.

Theorem 2.6 (Distributive laws)**.** *If p, q and r are logical variables, then*

$$p \wedge (q \vee r) = (p \wedge q) \vee (p \wedge r),$$
$$p \vee (q \wedge r) = (p \vee q) \wedge (p \vee r).$$

As usual, these rules can be proved setting up truth tables for the two sides.

Exercises

2.1 Use a truth table to prove that the exclusive or operator $\oplus$ is associative, i.e., show that if p, q and r are logical operators then $(p \oplus q) \oplus q = p \oplus (q \oplus r)$.

2.2 Prove de Morgan's laws.

2.3 Generalise De Morgan's laws to expressions with any finite number of $\wedge$- or $\vee$-operators, i.e., expressions on the form

$$\neg(p_1 \wedge p_2 \wedge \cdots \wedge p_n) \quad \text{and} \quad \neg(p_1 \vee p_2 \vee \cdots \vee p_n).$$

Hint: Use Lemma 2.5.

2.4 Use truth tables to check that

    a) $(p \wedge q) \vee r = p \wedge (q \vee r)$.

    b) $(p \vee q) \wedge (q \vee r) = (p \wedge r) \vee q$.

CHAPTER 3

Numbers and Numeral Systems

Numbers play an important role in almost all areas of mathematics, not least in calculus. Virtually all calculus books contain a thorough description of the natural, rational, real and complex numbers, so we will not repeat this here. An important concern for us, however, is to understand the basic principles behind how a computer handles numbers and performs arithmetic, and for this we need to consider some facts about numbers that are usually not found in traditional calculus texts.

More specifically, we are going to review the basics of the decimal numeral system, where the base is 10, and see how numbers may be represented equally well in other numeral systems where the base is not 10. We will study representation of real numbers as well as arithmetic in different bases. Throughout the chapter we will pay special attention to the binary numeral system (base 2) as this is what is used in most computers. This will be studied in more detail in the next chapter.

3.1 Terminology and Notation

We will usually introduce terminology as it is needed, but certain terms need to be agreed upon straightaway. In your calculus book you will have learnt about natural, rational and real numbers. The natural numbers $\mathbb{N}_0 = \{0, 1, 2, 3, 4, \ldots\}$[1] are the most basic numbers in that both rational and real numbers can be constructed from them. Any positive natural number n has an opposite number

---

[1] In most books the natural numbers start with 1, but for our purposes it is convenient to include 0 as a natural number as well. To avoid confusion we have therefore added 0 as a subscript.

$-n$, and we denote by $\mathbb{Z}$ the set of natural numbers augmented with all these negative numbers,

$$\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}.$$

We will refer to $\mathbb{Z}$ as the set of integer numbers or just the integers.

Intuitively it is convenient to think of a real number x as a decimal number with (possibly) infinitely many digits to the right of the decimal point. We then refer to the number obtained by setting all the digits to the right of the decimal point to 0 as the *integer part* of x. If we replace the integer part by 0 we obtain the *fractional part* of x. If for example $x = 3.14$, its integer part is 3 and its fractional part is 0.14. A number that has no integer part will often be referred to as a fractional number.

Definition 3.1. *Let $x = d_k d_{k-1} \cdots d_2 d_1 d_0 . d_{-1} d_{-2} \cdots$ be a decimal number. Then the number $d_k d_{k-1} \cdots d_1 d_0$ is called the integer part of x while the number $0 . d_{-1} d_{-2} \cdots$ is called the fractional part of x.*

For rational numbers there are standard operations we can perform to find the integer and fractional parts. When two positive natural numbers a and b are divided, the result will usually not be an integer, or equivalently, there will be a remainder. Let us agree on some notation for these operations.

Notation 3.2 (Integer division and remainder)**.** *If a and b are two integers, the number $a \mathbin{/\!/} b$ is the result obtained by dividing a by b and discarding the remainder. The number $a \% b$ is the remainder when a is divided by b.*

For example $3 \mathbin{/\!/} 2 = 1$, $9 \mathbin{/\!/} 4 = 2$ and $24 \mathbin{/\!/} 6 = 4$, while $3 \% 2 = 1$, $23 \% 5 = 3$, and $24 \% 4 = 0$.

We will use standard notation for intervals of real numbers. Two real numbers a and b with $a < b$ define four intervals that only differ in whether the end points a and b are included or not. The closed interval $[a, b]$ contains all real numbers between a and b, including the end points. Formally we can express this by $[a, b] = \{x \in \mathbb{R} \mid a \le x \le b\}$. The other intervals can be defined similarly,

Definition 3.3 (Intervals)**.** *Two real numbers a and b defined the four intervals*

$(a, b) = \{x \in \mathbb{R} \mid a < x < b\}$ *(open)*; $(a, b] = \{x \in \mathbb{R} \mid a < x \le b\}$ *(half open)*;

$[a, b] = \{x \in \mathbb{R} \mid a \le x \le b\}$ *(closed)*; $[a, b) = \{x \in \mathbb{R} \mid a \le x < b\}$ *(half open)*.

With this notation we can say that a fractional number is a real number in the interval $[0, 1)$.

3.2 Natural Numbers in Different Numeral Systems

We usually represent natural numbers in the decimal numeral system, but in this section we are going to see that this is just one of infinitely many numeral systems. We will also give a simple method for converting a number from its decimal representation to its representation in a different base.

3.2.1 Alternative Numeral Systems

In the decimal system we use a positional convention and express numbers in terms of the ten digits 0, 1, ..., 8, 9, and let the position of a digit determine how much it is worth. For example the string of digits 3761 is interpreted as

$$3761 = 3 \times 10^3 + 7 \times 10^2 + 6 \times 10^1 + 1 \times 10^0.$$

Numbers that have a simple representation in the decimal numeral system are often thought of as special. For example it is common to celebrate a 50th birthday in a special way or mark the centenary anniversary of an important event like a country's independence. However, the numbers 50 and 100 are only special when they are written in the decimal numeral system.

Any natural number can be used as the base for a numeral system. Consider for example the *septenary* numeral system which has 7 as the base and uses the digits 0-6. In this system the numbers 3761, 50 and 100 become

$$3761 = 13652_7 = 1 \times 7^4 + 3 \times 7^3 + 6 \times 7^2 + 5 \times 7^1 + 2 \times 7^0,$$
$$50 = 101_7 = 1 \times 7^2 + 0 \times 7^1 + 1 \times 7^0,$$
$$100 = 202_7 = 2 \times 7^2 + 0 \times 7^1 + 2 \times 7^0,$$

so 50 and 100 are not quite as special as in the decimal system.

These examples make it quite obvious that we can define numeral systems with almost any natural number as a base. The only restriction is that the base must be greater than one. To use 0 as base is quite obviously meaningless, and if we try to use 1 as base we only have the digit 0 at our disposal, which means that we can only represent the number 0. We record the general construction in a formal definition.

Definition 3.4. *Let β be a natural number greater than 1 and let n_0, n_1, ..., $n_{\beta-1}$ be β distinct numerals (also called digits) such that n_i denotes the number $n_i = i$. A natural number representation in base β is an ordered collection of digits $(d_k d_{n-1} \ldots d_1 d_0)_\beta$ which is interpreted as the natural number*

$$d_k \beta^k + d_{k-1} \beta^{k-1} + d_{k-2} \beta^{k-2} + \cdots + d_1 \beta^1 + d_0 \beta^0$$

where each digit d_i is one of the β numerals $\{n_i\}_{i=0}^{\beta-1}$.

Formal definitions in mathematics often appear complicated until one gets under the surface, so let us consider the details of the definition. The base β is not so mysterious. In the decimal system $\beta = 10$ while in the septenary system $\beta = 7$. The beginning of the definition simply states that any natural number greater than 1 can be used as a base.

In the decimal system we use the digits 0–9 to write down numbers, and in any numeral system we need digits that can play a similar role. If the base is 10 or less it is natural to use the obvious subset of the decimal digits as numerals. If the base is 2 we use the two digits $n_0 = 0$ and $n_1 = 1$; if the base is 5 we use the five digits $n_0 = 0$, $n_1 = 1$, $n_2 = 2$, $n_3 = 3$ and $n_4 = 4$. However, if the base is greater than 10 we have a challenge in how to choose numerals for the numbers 10, 11, ..., $\beta - 1$. If the base is less than 40 it is common to use the decimal digits together with the initial characters of the latin alphabet as numerals. In base $\beta = 16$ for example, it is common to use the digits 0–9 augmented with $n_{10} = a$, $n_{11} = b$, $n_{12} = c$, $n_{13} = d$, $n_{14} = e$ and $n_{15} = f$. This is called the *hexadecimal* numeral system and in this system the number 3761 becomes

$$eb1_{16} = e \times 16^2 + b \times 16^1 + 1 \times 16^0 = 14 \times 256 + 11 \times 16 + 1 = 3761.$$

Definition 3.4 defines how a number can be expressed in the numeral system with base β. However, it does not say anything about how to find the digits of a fixed number. And even more importantly, it does not guarantee that a number can be written in the base-β numeral system in only one way. This is settled in our first lemma below.

Lemma 3.5. *Any natural number can be represented uniquely in the base-β numeral system.*

Proof. To keep the argument as transparent as possible, we give the proof for a specific example, namely $a = 3761$ and $\beta = 8$ (the octal numeral system). Since

$8^4 = 4096 > a$, we know that the base-8 representation cannot contain more than 4 digits. Suppose that $3761 = (d_3 d_2 d_1 d_0)_8$; our job is to find the value of the four digits and show that each of them only has one possible value. We start by determining d_0. By definition of base-8 representation of numbers we have the relation

$$3761 = (d_3 d_2 d_1 d_0)_8 = d_3 8^3 + d_2 8^2 + d_1 8 + d_0. \qquad (3.1)$$

We note that only the last term in the sum on the right is not divisible by 8, so the digit d_0 must therefore be the remainder when 3761 is divided by 8. If we perform the division we find that

$$d_0 = 3761 \,\%\, 8 = 1, \qquad 3761 \,/\!/\, 8 = 470.$$

We observe that when the right-hand side of (3.1) is divided by 8 and the remainder discarded, the result is $d_3 8^2 + d_2 8 + d_1$. In other words be must have

$$470 = d_3 8^2 + d_2 8 + d_1.$$

But then we see that d_1 must be the remainder when 470 is divided by 8. If we perform this division we find

$$d_1 = 470 \,\%\, 8 = 6, \qquad 470 \,/\!/\, 8 = 58.$$

Using the same argument as before we see that the relation

$$58 = d_3 8 + d_2 \qquad (3.2)$$

must hold. In other words d_2 is the remainder when 58 is divided by 8,

$$d_2 = 58 \,\%\, 8 = 2, \qquad 58 \,/\!/\, 8 = 7.$$

If we divide both sides of (3.2) by 8 and drop the remainder we are left with $7 = d_3$. The net result is that $3761 = (d_3 d_2 d_1 d_0)_8 = 7261_8$.

We note that during the computations we never had any choice in how to determine the four-digits, they were determined uniquely. We therefore conclude that the only possible way to represent the decimal number 3761 in the base-8 numeral system is as 7261_8. ∎

The proof is clearly not complete since we have only verified Lemma 3.5 in a special case. However, the same argument can be used for any a and β and we leave it to the reader to write down the details in the general case.

Lemma 3.5 says that any natural number can be expressed in a unique way in any numeral system with base greater than 1. We can therefore use any such

numeral system to represent numbers. Although we may feel that we always use the decimal system, we all use a second system every day, the base-60 system. An hour is split into 60 minutes and a minute into 60 seconds. The great advantage of using 60 as a base is that it is divisible by 2, 3, 4, 5, 6, 10, 12, 15, 20 and 30 which means that an hour can easily be divided into many smaller parts without resorting to fractions of minutes. Most of us also use other numeral systems without knowing. Virtually all electronic computers use the base-2 (binary) system and we will see how this is done in the next chapter.

We are really only considering natural numbers in this section, but let us add a comment about how to represent negative numbers in the base-β numeral system. This is not so difficult. There is nothing particularly decimal about the minus sign, so the number $-a$ may be represented like a, but preceded with $-$. Therefore, we represent for example the decimal number -3761 as -7261_8 in the octal numeral system.

3.2.2 Conversion to the Base-β Numeral System

The method used in the proof of Lemma 3.5 for converting a number to base β is important, so we record it as an algorithm.

Algorithm 3.6. *Let a be a natural number that in base β has the $k+1$ digits $(d_k d_{k-1} \cdots d_0)_\beta$. These digits may be computed by performing the operations:*

$\quad a_0 := a;$
$\quad$**for** $i := 0, 1, \ldots, k$
$\quad\quad\quad d_i := a_i \,\% \,\beta;$
$\quad\quad\quad a_{i+1} := a_i \,/\!/\, \beta;$

Let us add a little explanation since this is our first algorithm apart from the examples in section 1.4. We start by setting the variable a_0 equal to a, the number whose digits we want to determine. We then let i take on the values 0, 1, 2, $\ldots$, k. For each value of i we perform the operations that are indented, i.e., we compute the numbers $a_i \% \beta$ and $a_i /\!/ \beta$ and store the results in the variables d_i and a_{i+1}.

Algorithm 3.6 demands that the number of digits in the representation to be computed is known in advance. If we look back on the proof of Lemma 3.5, we note that we do not first check how many digits we are going to compute, since when we are finished the number that we divide (the number a_i in Algorithm 3.6) has become 0. We can therefore just repeat the two indented statements in the algorithm until the result of the division becomes 0. The following version of the algorithm incorporates this. We also note that we do not need to

keep the results of the divisions; we can omit the subscript and store the result of the division $a \mathbin{/\!/} \beta$ back in a.

Algorithm 3.7. *Let a be a natural number that in base β has the $k + 1$ digits $(d_k d_{k-1} \cdots d_0)_\beta$. These digits may be computed by performing the operations:*

$\quad$ $i := 0;$
$\quad$ **while** $a > 0$
$\qquad$ $d_i := a \% \beta;$
$\qquad$ $a := a \mathbin{/\!/} \beta;$
$\qquad$ $i := i + 1;$

Recall that the statement 'while $a > 0$' means that all the indented statements will be repeated until a becomes 0.

It is important to realise that the order of the indented statements is not arbitrary. When we do not keep all the results of the divisions, it is essential that d_i (or d) is computed before a is updated with its new value. And when i is initialised with 0, we must update i at the end, since otherwise the subscript in d_i will be wrong.

The variable i is used here so that we can number the digits correctly, starting with d_0, then d_1 and so on. If this is not important, we could omit the first and the last statements, and replace d_i by d. The algorithm then becomes

$\quad$ **while** $a > 0$
$\qquad$ $d := a \% \beta;$
$\qquad$ $a := a \mathbin{/\!/} \beta;$
$\qquad$ **print** $d;$

Here we have also added a print-statement so the digits of a will be printed (in reverse order).

The results produced by Algorithm 3.7 can be conveniently organised in a table. The example in the proof of Lemma 3.5 can be displayed as

3761	1
470	6
58	2
7	7

The left column shows the successive integer parts resulting from repeated division by 8, whereas the right column shows the remainder in these divisions. Let us consider one more example.

37

Example 3.8. Instead of converting 3761 to base 8 let us convert it to base 16. We find that 3761 // 16 = 235 with remainder 1. In the next step we find 235 // 16 = 14 with remainder 11. Finally we have 14 // 16 = 0 with remainder 14. Displayed in a table this becomes

3761	1
235	11
14	14

Recall that in the hexadecimal system the letters a–f usually denote the values 10–15. We have therefore found that the number 3761 is written $eb1_{16}$ in the hexadecimal numeral system. ■

Since we are particularly interested in how computers manipulate numbers, let us also consider an example of conversion to the binary numeral system, as this is the numeral system used in most computers. Instead of dividing by 16 we are now going to repeatedly divide by 2 and record the remainder. A nice thing about the binary numeral system is that the only possible remainders are 0 and 1: it is 0 if the number we divide is an even integer and 1 if the number is an odd integer.

Example 3.9. Let us continue to use the decimal number 3761 as an example, but now we want to convert it to binary form. If we perform the divisions and record the results as before we find

3761	1
1880	0
940	0
470	0
235	1
117	1
58	0
29	1
14	0
7	1
3	1
1	1

In other words we have $3761 = 111010110001_2$. This example illustrates an important property of the binary numeral system: Computations are simple, but long and tedious. This means that this numeral system is not so good for humans as we tend to get bored and make sloppy mistakes. For computers, how-

ever, this is perfect as computers do not make mistakes and work extremely fast. ∎

3.2.3 Conversion between base-2 and base-16

Computers generally use the binary numeral system internally, and in chapter 4 we are going to study this in some detail. A major disadvantage of the binary system is that even quite small numbers require considerably more digits than in the decimal system. There is therefore a need for a more compact representation of binary numbers. It turns out that the hexadecimal numeral system is convenient for this purpose.

Suppose we have the one-digit hexadecimal number $x = a_{16}$. In binary it is easy to see that this is $x = 1010_2$. A general four-digit binary number $(d_3 d_2 d_1 d_0)_2$ has the value

$$d_0 2^0 + d_1 2^1 + d_2 2^2 + d_3 2^3,$$

and must be in the range 0–15, which corresponds exactly to a one-digit hexadecimal number.

> **Observation 3.10.** *A four-digit binary number can always be converted to a one-digit hexadecimal number, and vice versa.*

This simple observation is the basis for converting general numbers between binary and hexadecimal representation. Suppose for example that we have the eight digit binary number $x = 1100\ 1101_2$. This corresponds to the number

$$1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3 + 0 \times 2^4 + 0 \times 2^5 + 1 \times 2^6 + 1 \times 2^7$$
$$= (1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3) + (0 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3) 2^4.$$

The two numbers in brackets are both in the range 0–15 and can therefore be represented as one-digit hexadecimal numbers. In fact we have

$$1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3 = 13_{10} = d_{16},$$
$$0 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3 = 12_{10} = c_{16}.$$

But then we have

$$x = (1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3) + (0 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3) 2^4$$
$$= d_{16} \times 16^0 + 16^1 \times c_{16} = cd_{16}.$$

The short version of this detailed derivation is that the eight digit binary number $x = 1100\ 1101_2$ can be converted to hexadecimal by converting the two groups of

Hex	Bin	Hex	Bin	Hex	Bin	Hex	Bin
0	0	4	100	8	1000	c	1100
1	1	5	101	9	1001	d	1101
2	10	6	110	a	1010	e	1110
3	11	7	111	b	1011	f	1111

Table 3.1. Conversion between hexadecimal and binary representation.

four binary digits separately. This results in two one-digit hexadecimal numbers, and these are the hexadecimal digits of x,

$$1100_2 = c_{16}, \quad 1101_2 = d_{16}, \quad 1100\ 1101_2 = cd_{16}.$$

This works in general.

> **Observation 3.11.** *A hexadecimal natural number can be converted to binary by converting each digit separately. Conversely, a binary number can be converted to hexadecimal by converting each group of successive binary digits into hexadecimal, starting with the least significant digits.*

Example 3.12. Let us convert the hexadecimal number $3c5_{16}$ to binary. We have

$$5_{16} = 0101_2,$$
$$c_{16} = 1100_2,$$
$$3_{16} = 0011_2,$$

which means that $3c5_{16} = 11\ 1100\ 0101_2$ where we have omitted the two leading zeros. ∎

Observation 3.11 means that to convert between binary and hexadecimal representation we only need to know how to convert numbers in the range 0–15 (decimal). Most will perhaps do this by going via decimal representation, but all the conversions can be found in table 3.1.

3.3 Representation of Fractional Numbers

We have seen how integers can be represented in numeral systems other than decimal, but what about fractions and irrational numbers? In the decimal system such numbers are characterised by the fact that they have two parts, one to the left of the decimal point, and one to the right, like the number 21.828. The

part to the left of the decimal point — the integer part — can be represented in base-β as outlined above. If we can represent the part to the right of the decimal point — the fractional part — in base-β as well, we can follow the convention from the decimal system and use a point to separate the two parts. Negative rational and irrational numbers are as easy to handle as negative integers, so we focus on how to represent positive numbers without an integer part, in other words numbers in the open interval $(0, 1)$.

3.3.1   Rational and Irrational Numbers in Base-β

Let a be a real number in the interval $(0, 1)$. In the decimal system we can write such a number as 0, followed by a point, followed by a finite or infinite number of decimal digits, as in

$$0.45928\ldots$$

This is interpreted as the number

$$4 \times 10^{-1} + 5 \times 10^{-2} + 9 \times 10^{-3} + 2 \times 10^{-4} + 8 \times 10^{-5} + \cdots.$$

From this it is not so difficult to see what a base-β representation must look like.

> **Definition 3.13.** *Let β be a natural number greater than 1 and let n_0, n_1, $\ldots$, $n_{\beta-1}$ be β distinct numerals (also called digits) such that n_i denotes the number $n_i = i$. A fractional representation in base β is a finite or infinite, ordered collection of digits $(0.d_{-1}d_{-2}d_{-3}\ldots)_\beta$ which is interpreted as the real number*
>
> $$d_{-1}\beta^{-1} + d_{-2}\,\beta^{-2} + d_{-3}\,\beta^{-3} + \cdots \tag{3.3}$$
>
> *where each digit d_i is one of the β numerals $\{n_i\}_{i=0}^{\beta-1}$.*

   Definition 3.13 is considerably more complicated than definition 3.4 since we may have an infinite number of digits. This becomes apparent if we try to check the size of numbers on the form given by (3.3). Since none of the terms in the sum are negative, the smallest number is the one where all the digits are 0, i.e., where $d_i = 0$ for $i = -1, -2, \ldots$. But this can be nothing but the number 0.
   The largest possible number occurs when all the digits are as large as possible, i.e. when $d_i = \beta - 1$ for all i. If we call this number x, we find

$$\begin{aligned}
x &= (\beta - 1)\beta^{-1} + (\beta - 1)\beta^{-2} + (\beta - 1)\beta^{-3} + \cdots \\
&= (\beta - 1)\beta^{-1}(1 + \beta^{-1} + \beta^{-2} + \cdots \\
&= \frac{\beta - 1}{\beta} \sum_{i=0}^{\infty} (\beta^{-1})^i.
\end{aligned}$$

In other words x is given by a sum of an infinite geometric series with factor $\beta^{-1} = 1/\beta < 1$. This series converges to $1/(1-\beta^{-1})$ so x has the value

$$x = \frac{\beta-1}{\beta} \frac{1}{1-\beta^{-1}} = \frac{\beta-1}{\beta} \frac{\beta}{\beta-1} = 1.$$

Let us record our findings so far.

Lemma 3.14. *Any number on the form* (3.3) *lies in the interval* $[0,1]$.

The fact that the base-β fractional number with all digits equal to $\beta-1$ is the number 1 is a bit disturbing since it means that real numbers cannot be represented uniquely in base β. In the decimal system this corresponds to the fact that $0.99999999999999\ldots$ (infinitely many 9s) is in fact the number 1. And this is not the only number that has two representations. Any number that ends with an infinite number of digits equal to $\beta-1$ has a simpler representation. Consider for example the decimal number $0.12999999999999\ldots$. Using the same technique as above we find that this number is 0.13. However, it turns out that these are the only numbers that have a double representation, see theorem 3.15 below.

Let us now see how we can determine the digits of a fractional number in a numeral system other than the decimal one. As for natural numbers, it is easiest to understand the procedure through an example, so we try to determine the digits of $1/5$ in the octal (base 8) system. According to definition 3.13 we seek digits $d_{-1}d_{-2}d_{-3}\ldots$ (possibly infinitely many) such that the relation

$$\frac{1}{5} = d_{-1}8^{-1} + d_{-2}8^{-2} + d_{-3}8^{-3} + \cdots \tag{3.4}$$

becomes true. If we multiply both sides by 8 we obtain

$$\frac{8}{5} = d_{-1} + d_{-2}8^{-1} + d_{-3}8^{-2} + \cdots . \tag{3.5}$$

The number $8/5$ lies between 1 and 2 and we know from Lemma 3.14 that the sum $d_{-2}8^{-1} + d_{-3}8^{-2} + \cdots$ can be at most 1. Therefore we must have $d_{-1} = 1$. Since d_{-1} has been determined we can subtract this from both sides of (3.5)

$$\frac{3}{5} = d_{-2}8^{-1} + d_{-3}8^{-2} + d_{-4}8^{-3} + \cdots . \tag{3.6}$$

This equation has the same form as (3.4) and can be used to determine d_{-2}. We multiply both sides of (3.6) by 8,

$$\frac{24}{5} = d_{-2} + d_{-3}8^{-1} + d_{-4}8^{-3} + \cdots . \tag{3.7}$$

The fraction 24/5 lies in the interval $(4,5)$ and since the terms on the right that involve negative powers of 8 must be a number in the interval $[0,1]$, we must have $d_{-2} = 4$. We subtract this from both sides of (3.7) and obtain

$$\frac{4}{5} = d_{-3}8^{-1} + d_{-4}8^{-2} + d_{-5}8^{-3} + \cdots . \tag{3.8}$$

Multiplication by 8 now gives

$$\frac{32}{5} = d_{-3} + d_{-4}8^{-1} + d_{-5}8^{-2} + \cdots .$$

from which we conclude that $d_{-3} = 6$. Subtracting 6 and multiplying by 8 we obtain

$$\frac{16}{5} = d_{-4} + d_{-5}8^{-1} + d_{-6}8^{-2} + \cdots .$$

from which we conclude that $d_{-4} = 3$. If we subtract 3 from both sides we find

$$\frac{1}{5} = d_{-5}8^{-1} + d_{-6}8^{-2} + d_{-7}8^{-3} + \cdots .$$

But this relation is essentially the same as (3.4), so if we continue we must generate the same digits again. In other words, the sequence $d_{-5}d_{-6}d_{-7}d_{-8}$ must be the same as $d_{-1}d_{-2}d_{-3}d_{-4} = 1463$. But once d_{-8} has been determined we must again come back to a relation with 1/5 on the left, so the same digits must also repeat in $d_{-9}d_{-10}d_{-11}d_{-12}$ and so on. The result is that

$$\frac{1}{5} = 0.1463146314631463\cdots_8 .$$

Based on this procedure we can prove an important theorem.

> **Theorem 3.15.** *Any real number in the interval* $(0,1)$ *can be represented in a unique way as a fractional base-β number provided representations with infinitely many trailing digits equal to $\beta - 1$ are prohibited.*

Proof. We have already seen how the digits of 1/5 in the octal system can be determined, and it is easy to generalise the procedure. However, there are two additional questions that must be settled before the claims in the theorem are completely settled.

We first prove that the representation is unique. If we look back on the conversion procedure in the example we considered, we had no freedom in the choice of any of the digits. The digit d_{-2} was for example determined by equation 3.7, where the left-hand side is 4.8 in the decimal system. Then our only

hope of satisfying the equation is to choose $d_{-2} = 4$ since the remaining terms can only sum up to a number in the interval $[0, 1]$.

How can the procedure fail to determine the digits uniquely? In our example, any digit is determined by an equation on the form (3.7), and as long as the left-hand side is not an integer, the corresponding digit is uniquely determined. If the left-hand side should happen to be an integer, as in

$$5 = d_{-2} + d_{-3}8^{-1} + d_{-4}8^{-3} + \cdots,$$

the natural solution is to choose $d_{-2} = 5$ and choose all the remaining digits as 0. However, since we know that 1 may be represented as a fractional number with all digits equal to 7, we could also choose $d_{-2} = 4$ and $d_i = 7$ for all $i < -2$. The natural solution is to choose $d_{-2} = 5$ and prohibit the second solution. This is exactly what we have done in the statement of the theorem, so this secures the uniqueness of the representation.

The second point that needs to be settled is more subtle; do we really compute the correct digits? It may seem strange to think that we may not compute the right digits since the digits are forced upon us by the equations. But if we look carefully, the equations are not quite standard since the sums on the right may contain infinitely many terms. In general it is therefore impossible to achieve equality in the equations, all we can hope for is that we can make the sum on the right in (3.4) come as close to $1/5$ as we wish by including sufficiently many terms.

Set $a = 1/5$. Then equation (3.6) can be written

$$8(a - d_{-1}8^{-1}) = d_{-2}8^{-1} + d_{-3}8^{-2} + d_{-4}8^{-3} + \cdots$$

while (3.8) can be written

$$8^2(a - d_{-1}8^{-1} - d_{-2}8^{-2}) = d_{-3}8^{-1} + d_{-4}8^{-2} + d_{-5}8^{-3} + \cdots.$$

After i steps the equation becomes

$$8^i(a - d_{-1}8^{-1} - d_{-2}8^{-2} - \cdots - d_{-i}8^{-i}) =$$
$$d_{-i-1}8^{-1} + d_{-i-2}8^{-2} + d_{-i-3}8^{-3} + \cdots.$$

The expression in the bracket on the left we recognise as the error e_i in approximating a by the first i numerals in the octal representation. We can rewrite this slightly and obtain

$$e_i = 8^{-i}(d_{-i-1}8^{-1} + d_{-i-2}8^{-2} + d_{-i-3}8^{-3} + \cdots).$$

From Lemma 3.14 we know that the number in the bracket on the right lies in the interval $[0, 1]$ so we have $0 \le e_i \le 8^{-i}$. But this means that by including sufficiently many digits (choosing i sufficiently big), we can get e_i to be as small as we wish. In other words, by including sufficiently many digits, we can get the octal representation of $a = 1/5$ to be as close to a as we wish. Therefore our method for computing numerals does indeed generate the digits of a. ∎

3.3.2 An Algorithm for Converting Fractional Numbers

The basis for the proof of Theorem 3.15 is the procedure for computing the digits of a fractional number in base-β. We only considered the case $\beta = 8$, but it is simple to generalise the algorithm to arbitrary β.

> **Algorithm 3.16.** *Let a be a fractional number whose first k digits in base β are $(0.d_{-1}d_{-2}\cdots d_{-k})_\beta$. These digits may be computed by performing the operations:*
>
>   **for** $i := -1, -2, \ldots, -k$
> $d_i := \lfloor a * \beta \rfloor$;
> $a := a * \beta - d_i$;

Compared with the description on pages 42 to 43 there should be nothing mysterious in this algorithm except for perhaps the notation $\lfloor x \rfloor$. This is a fairly standard way of writing the *floor function* which is equal to the largest integer that is less than or equal to x. We have for example $\lfloor 3.4 \rfloor = 3$ and $\lfloor 5 \rfloor = 5$.

When converting natural numbers to base-β representation there is no need to know or compute the number of digits beforehand, as is evident in algorithm 3.7. For fractional numbers we do need to know how many digits to compute as there may often be infinitely many. A for-loop is therefore a natural construction in algorithm 3.16.

It is convenient to have a standard way of writing down the computations involved in converting a fractional number to base-β, and it turns out that we can use the same format as for converting natural numbers. Let us take as an example the computations in the proof of theorem 3.15 where the fraction $1/5$ was converted to base-8. We start by writing the number to be converted to the left of the vertical line. We then multiply the number by β (which is 8 in this case) and write the integer part of the result, which is the first digit, to the right of the line. The result itself we write in brackets to the right. We then start with the fractional part of the result one line down and continue until the result becomes 0 or we have all the digits we want,

1/5	1	(8/5)
3/5	4	(24/5)
4/5	6	(32/5)
2/5	3	(16/5)
1/5	1	(8/5)

Here we are back at the starting point, so the same digits will just repeat again.

3.3.3   Conversion between binary and hexadecimal

It turns out that hexadecimal representation is handy short-hand for the binary representation of fractional numbers, just like it was for natural numbers. To see why this is, we consider the number $x = 0.11010111_2$. In all detail this is

$$
\begin{aligned}
x &= 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} + 0 \times 2^{-5} + 1 \times 2^{-6} + 1 \times 2^{-7} + 1 \times 2^{-8} \\
&= 2^{-4}(1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0) + 2^{-8}(0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0) \\
&= 16^{-1}(1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0) + 16^{-2}(0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0).
\end{aligned}
$$

From table 3.1 we see that the two four-digit binary numbers in the brackets correspond to the hexadecimal numbers $1101_2 = d_{16}$ and $111_2 = 7_{16}$. We therefore have

$$
x = 16^{-1}13 + 16^{-2}7 = 0.d7_{16}.
$$

As for natural numbers, this works in general.

> **Observation 3.17.** *A hexadecimal fractional number can be converted to binary by converting each digit separately. Conversely, a binary fractional number can be converted to hexadecimal by converting each group of four successive binary digits to hexadecimal, starting with the most significant digits.*

A couple of examples will illustrate how this works in general.

Example 3.18. Let us convert the number $x = 0.3a8_{16}$ to binary. From table 3.1 we find

$$
3_{16} = 0011_2, \quad a_{16} = 1010_2, \quad 8_{16} = 1000_2,
$$

which means that

$$
0.3a8_{16} = 0.0011\ 1010\ 1000_2 = 0.0011\ 1010\ 1_2. \quad \blacksquare
$$

Example 3.19. To convert the binary number $0.1100\ 1001\ 0110\ 1_2$ to hexadecimal form we note from table 3.1 that

$$
1100_2 = c_{16}, \quad 1001_2 = 9_{16}, \quad 0110_2 = 6_{16}, \quad 1000_2 = 8_{16}.
$$

Note that the last group of binary digits was not complete so we added three zeros. From this we conclude that

$$0.1100\ 1001\ 0110\ 1_2 = 0.c968_{16}. \quad \blacksquare$$

3.3.4 Properties of Fractional Numbers in Base-β

Real numbers in the interval $(0, 1)$ have some interesting properties related to their representation. In the decimal numeral system we know that fractions with a denominator that only contains the factors 2 and 5 can be written as a decimal number with a finite number of digits. In general, the decimal representation of a rational number will contain a finite sequence of digits that are repeated infinitely many times, while for an irrational number there will be no such structure. In this section we shall see that similar properties are valid when fractional numbers are represented in any numeral system.

For rational numbers algorithm 3.16 can be expressed in a different form which makes it easier to deduce properties of the digits. So let us consider what happens when a rational number is converted to base-β representation. A rational number in the interval $(0, 1)$ has the form $a = b/c$ where b and c are nonzero natural numbers with $b < c$. If we look at the computations in algorithm 3.16, we note that d_i is the integer part of $(b*\beta)/c$ which can be computed as $(b*\beta) \mathbin{/\!/} c$. The right-hand side of the second statement is $a*\beta - d_1$, i.e., the fractional part of $a*\beta$. But if $a = b/c$, the fractional part of $a*\beta$ is given by the remainder in the division $(b*\beta)/c$, divided by c, so the new value of a is given by

$$a = \frac{(b*\beta)\,\%\,c}{c}.$$

This is a new fraction with the same denominator c as before. But since the denominator does not change, it is sufficient to keep track of the numerator. This can be done by the statement

$$b := (b*\beta)\,\%\,c. \tag{3.9}$$

The result is a new version of algorithm 3.16 for rational numbers.

Algorithm 3.20. *Let $a = b/c$ be a rational number in $(0, 1)$ whose first k digits in base β are $(0.d_{-1}d_{-2}\cdots d_{-k})_\beta$. These digits may be computed by performing the operations:*

> **for** $i := -1, -2, \ldots, -k$
> $\quad d_i := (b*\beta) \mathbin{/\!/} c;$
> $\quad b := (b*\beta)\,\%\,c;$

47

This version of the conversion algorithm is more convenient for deducing properties of the numerals of a rational number. The clue is to consider more carefully the different values of b that are computed by the algorithm. Since b is the remainder when integers are divided by c, the only possible values of b are $0, 1, 2, \ldots, c-1$. Sooner or later, the value of b must therefore become equal to an earlier value. But once b returns to an earlier value, it must cycle through exactly the same values again until it returns to the same value a third time. And then the same values must repeat again, and again, and again, $\ldots$. Since the numerals d_i are computed from b, they must repeat with the same frequency. Note however that may be some initial digits that do not repeat. This proves part of the following lemma.

Lemma 3.21. *Let a be a fractional number. Then the digits of a written in base β will eventually repeat, i.e.,*

$$a = (0.d_{-1} \cdots d_{-i} d_{-(i+1)} \cdots d_{-(i+m)} d_{-(i+1)} \cdots d_{-(i+m)} \cdots)_\beta$$

for some integer $m \geq 1$ if and only if a is a rational number.

As an example, consider the fraction $1/7$ written in different numeral systems. If we run algorithm 3.20 we find

$$1/7 = 0.00100100100100100\cdots_2,$$
$$1/7 = 0.01021201021201021\cdots_3,$$
$$1/7 = 0.1_7.$$

In the binary numeral system, there is no initial sequence of digits; the sequence 001 repeats from the start. In the trinary system, there is no intial sequence either and the repeating sequence is 010212, whereas in the septenary system the initial seqeunce is 1 and the repeating sequence 0 (which we do not write according to the conventions of the decimal system).

An example with an initial sequence is the fraction $87/98$ which in base 7 becomes $0.6133333\cdots_7$. Another example is $503/1100$ which is $0.457272727272\cdots$ in the decimal system.

The argument preceding lemma 3.21 proves the fact that if a is a rational number, then the digits must eventually repeat. But this statement leaves the possibility open that there may be nonrational (i.e., irrational) numbers that may also have digits that eventually repeat. However, this is not possible and this is the reason for the 'only if'-part of the lemma. In less formal language the complete statement is: *The digits of a will eventually repeat if a is a rational*

number, and only if a is a rational number. This means that there are two state-ments to prove: (i) The digits repeat if *a* is a rational number and (ii) if the digits do repeat then *a* must be a rational number. The proof of this latter statement is left to excercise 12.

Although all rational numbers have repeating digits, for some numbers the repeating sequence is '0', like 1/7 in base 7, see above. Or equivalently, some fractional numbers can in some numeral systems be represented exactly by a finite number of digits. It is possible to characterise exactly which numbers have this property.

Lemma 3.22. *The representation of a fractional number a in base-β consists of a finite number of digits,*

$$a = (0.d_{-1}d_{-2}\cdots d_{-k})_\beta,$$

if and only if a is a rational number b/c with the property that all the prime factors of c divide β.

Proof. Since the statement is of the 'if and only if' type, there are two claims to be proved. The fact that a fractional number in base-β with a finite number of digits is a rational number is quite straightforward, see exercise 13.

What remains is to prove that if $a = b/c$ and all the prime factors of c divide β, then the representation of a in base-β will have a finite number of digits. We give the proof in a special case and leave it to the reader to write down the proof in general. Let us consider the representation of the number $a = 8/9$ in base-6. The idea of the proof is to rewrite a as a fraction with a power of 6 in the denominator. The simplest way to do this is to observe that $8/9 = 32/36$. We next express 32 in base 6. For this we can use algorithm 3.7, but in this simple situation we see directly that

$$32 = 5 \times 6 + 2 = 52_6.$$

We therefore have

$$\frac{8}{9} = \frac{32}{36} = \frac{5 \times 6 + 2}{6^2} = 5 \times 6^{-1} + 2 \times 6^{-2} = 0.52_6. \quad \blacksquare$$

In the decimal system, fractions with a denominator that only has 2 and 5 as prime factors have finitely many digits, for example $3/8 = 0.375$, $4/25 = 0.16$ and $7/50 = 0.14$. These numbers will *not* have finitely many digits in most other

numeral systems. In base-3, the only fractions with finitely many digits are the ones that can be written as fractions with powers of 3 in the denominator,

$$\frac{8}{9} = 0.22_3,$$

$$\frac{7}{27} = 0.021_3,$$

$$\frac{1}{2} = 0.111111111111\cdots_3,$$

$$\frac{3}{10} = 0.02200220022\cdots_3.$$

In base-2, the only fractions that have an exact representation are the ones with denominators that are powers of 2,

$$\frac{1}{2} = 0.5 = 0.1_2,$$

$$\frac{3}{16} = 0.1875 = 0.0011_2,$$

$$\frac{1}{10} = 0.1 = 0.00011001100110011\cdots_2.$$

These are therefore the only fractional numbers that can be represented exactly on most computers unless special software is utilised.

3.4   Arithmetic in Base β

The methods we learn in school for performing arithemetic are closely tied to properties of the decimal numeral system, but the methods can easily be generalised to any numeral system. We are not going to do this in detail, but some examples will illustrate the general ideas. All the methods should be familiar from school, but if you never quite understood the arithmetic methods, you may have to think twice to understand why it all works. Although the methods themselves are the same across the world, it should be remembered that there are many variations in how the methods are expressed on paper. You may therefore find the description given here unfamiliar at first.

3.4.1   Addition

Addition of two one-digit numbers is just like in the decimal system as long as the result has only one digit. For example, we have $4_8 + 3_8 = 4 + 3 = 7 = 7_8$. If the result requires two digits, we must remember that the carry is β in base-β, and not 10. So if the result becomes β or greater, the result will have two digits,

where the left-most digit is 1 and the second has the value of the sum, reduced by β. This means that

$$5_8 + 6_8 = 5 + 6 = 11 = 8 + 11 - 8 = 8 + 3 = 13_8.$$

This can be written exactly the same way as you would write a sum in the decimal numeral system, you must just remember that the value of the carry is β.

Let us now try the larger sum $457_8 + 325_8$. This requires successive one-digit additions, just like in the decimal system. One way to write this is

$$
\begin{array}{r}
{}^{1\,1} \\
457_8 \\
+325_8 \\
\hline
= 1004_8
\end{array}
$$

This corresponds to the decimal sum $303 + 213 = 516$.

3.4.2   Subtraction

One-digit subtractions are simple, for example $7_8 - 3_8 = 4_8$. A subtraction like $14_8 - 7_8$ is a bit more difficult, but we can 'borrow' from the '1' in 14 just like in the decimal system. The only difference is that in base-8, the '1' represents 8 and not 10, so we borrow 8. We then see that we must perform the subtraction $12 - 7$ so the answer is 5 (both in decimal and base 8). Subtraction of larger numbers can be done by repeating this. Consider for example $321_8 - 177_8$. This can be written

$$
\begin{array}{r}
{}^{8\,8} \\
\not3\not21_8 \\
-177_8 \\
\hline
= 122_8
\end{array}
$$

By converting everything to decimal, it is easy to see that this is correct.

3.4.3   Multiplication

Let us just consider one example of a multiplication, namely $312_4 \times 12_4$. As in the decimal system, the basis for performing multiplication of numbers with multiple digits is the multiplication table for one-digit numbers. In base 4 the multiplication table is

	1	2	3
1	1	2	3
2	2	10	12
3	3	12	21

We can then perform the multiplication as we are used to in the decimal system

$$312_4 \times 12_4$$
$$1230_4$$
$$312_4$$
$$11010_4$$

The number 1230_4 in the second line is the result of the multiplication $312_4 \times 2_4$, i.e., the first factor 312_4 multiplied by the second digit of the right-most factor 12_4. The number on the line below, 312_4, is the first factor multiplied by the first digit of the second factor. This second product is shifted one place to the left since multiplying with the first digit in 12_4 corresponds multiplication by 1×4. The number on the last line is the sum of the two numbers above, with a zero added at the right end of 312_4, i.e., the sum is $1230_4 + 3120_4$. This sum is calculated as indicated in section 3.4.1 above.

Exercises

3.1 Convert the following natural numbers:

    a) 40 to base-4

    b) 17 to base-5

    c) 17 to base-2

    d) 123456 to base-7

    e) 22875 to base-7

    f) 126 to base 16

3.2 Convert the following rational numbers:

    a) 1/4 to base-2

    b) 3/7 to base-3

    c) 1/9 to base-3

    d) 1/18 to base-3

    e) 7/8 to base-8

    f) 7/8 to base-7

    g) 7/8 to base-16

    h) 5/16 to base-8

    i) 5/8 to base-6

3.3 Convert π to base-9.

3.4 Convert to base-8:

    a) 1011001_2

    b) 110111_2

c) 10101010_2

3.5 Convert to base-2:

    a) 44_8

    b) 100_8

    c) 327_8

3.6 Convert to base-16:

    a) 1001101_2

    b) 1100_2

    c) 10100111100100_2

    d) 0.0101100101_2

    e) 0.000001010001_2

    f) 0.1111111111_2

3.7 Convert to base-2:

    a) $3c_{16}$

    b) 100_{16}

    c) $e51_{16}$

    d) $0.0aa_{16}$

    e) 0.001_{16}

    f) $0.f01_{16}$

3.8   **a)** Convert 7 to base-7, 37 to base-37, and 4 to base-4 and formulate a generalisation of what you observe.

    b) Determine β such that $13 = 10_\beta$. Also determine β such that $100 = 10_\beta$ For which numbers $a \in \mathbb{N}$ is there a β such that $a = 10_\beta$?

3.9   **a)** Convert 400 to base-20, 4 to base-2, 64 to base-8, 289 to base-17 and formulate a generalisation of what you observe.

    b) Determine β such that $25 = 100_\beta$. Also determine β such that $841 = 100_\beta$. For which numbers $a \in \mathbb{N}$ is there a number β such that $a = 100_\beta$?

    c) For which numbers $a \in \mathbb{N}$ is there a number β such that $a = 1000_\beta$?

3.10   **a)** For which value of β is $a = b/c = 0.b_\beta$? Does such a β exist for all $a < 1$? And for $a \geq 1$?

    b) For which rational number $a = b/c$ does there exist a β such that $a = b/c = 0.01_\beta$?

    c) For which rational number $a = b/c$ is there a β such that $a = b/c = 0.0b_\beta$? If β exists, what will it be?

3.11 If $a = b/c$, what is the maximum length of the repeating sequence?

3.12 Show that if the digits of the fractional number a eventually repeat, then a must be a rational number.

3.13 Show that a fractional numbers in base-β with a finite number of digits is a rational number.

3.14 Perform the following additions:

     a) $3_7 + 1_7$

     b) $5_6 + 4_6$

     c) $110_2 + 1011_2$

     d) $122_3 + 201_3$

     e) $43_5 + 10_5$

     f) $3_5 + 1_7$

3.15 Perform the following subtractions:

     a) $5_8 - 2_8$

     b) $100_2 - 1_2$

     c) $527_8 - 333_8$

     d) $210_3 - 21_3$

     e) $43_5 - 14_5$

     f) $3_7 - 11_7$

3.16 Perform the following multiplications:

     a) $110_2 \cdot 10_2$

     b) $110_2 \cdot 11_2$

     c) $110_3 \cdot 11_3$

     d) $43_5 \cdot 2_5$

     e) $720_8 \cdot 15_8$

     f) $210_3 \cdot 12_3$

     g) $101_2 \cdot 11_2$

CHAPTER 4

Computers, Numbers and Text

In this chapter we are going to study how numbers are represented in a computer. We already know that at the most basic level, computers just handle sequences of 0s and 1s. We also know that numbers can be represented in different numeral systems, in particular the binary (base-2) numeral system which is perfectly suited for computers. We first consider representation of integers which is quite straightforward, and then representation of fractional numbers which is a bit more challenging.

4.1 Representation of Integers

If computers are to perform calculations with integers, we must obviously have a way to represent the numbers in terms of the computers' electronic circuitry. This presents us with one major challenge and a few minor ones. The big challenge is that integer numbers can become arbitrarily large in magnitude. This means that there is no limit to the number of digits that may be needed to write down integer numbers. On the other hand, the resources in terms of storage capacity and available computing time is always finite, so there will always be an upper limit on the magnitude of numbers that can be handled by a given computer. There are two standard ways to handle this problem.

The most common solution is to restrict the number of digits. If for simplicity we assume that we can work in the decimal numeral system, we could restrict the number of digits to 6. This means that the biggest number we can handle would be 999999. The advantage of this limitation is that we could put a lot of effort into making the computer's operation on 6 digit decimal numbers

Traditional		SI prefixes			
Symbol	Value	Symbol	Value	Alternative	Value
kB (kilobyte)	2^{10}	KB	10^3	kibibyte	2^{10}
MB (megabyte)	2^{20}	MB	10^6	mibibyte	2^{20}
GB (gigabyte)	2^{30}	GB	10^9	gibibyte	2^{30}
TB (terabyte)	2^{40}	TB	10^{12}	tibibyte	2^{40}
PB (petabyte)	2^{50}	PB	10^{15}	pibibyte	2^{50}
EB (exabyte)	2^{60}	EB	10^{18}	exbibyte	2^{60}
ZB (zettabyte)	2^{70}	ZB	10^{21}	zebibyte	2^{70}
YB (yottabyte)	2^{80}	YB	10^{24}	yobibyte	2^{80}

Table 4.1. The Si-prefixes for large collections of bits and bytes.

extremely efficient. On the other hand the computer could not do much other than report an error message and give up if the result should become larger than 999999.

The other solution would be to not impose a specific limit on the size of the numbers, but rather attempt to handle as large numbers as possible. For any given computer there is bound to be an upper limit, and if this is exceeded the only response would be an error message. We will discuss both of these approaches to the challenge of big numbers below.

4.1.1 Bits, bytes and numbers

At the most basic level, the circuitry in a computer (usually referred to as the *hardware*) can really only differentiate between two different states, namely '0' and '1' (or 'false' and 'true'). This means that numbers must be represented in terms of 0 and 1, in other words in the binary numeral system. From what we learnt in the previous chapter, this is not a difficult task, but for reasons of efficiency the electronics have been designed to handle groups of binary digits. The smallest such group consists of 8 binary digits (bits) and is called a byte. Larger groups of bits are usually groups of bytes. For manipulation of numbers, groups of 4 and 8 bytes are usually used, and computers have special computational units to handle groups of bits of these sizes.

Fact 4.1. *A binary digit is called a bit and a group of 8 bits is called a byte. Numbers are usually represented in terms of 4 bytes (32 bits) or 8 bytes (64 bits).*

The standard SI prefixes are used when large amounts of bits and bytes are referred to, see table 4.1. Note that traditionally the factor between each prefix has been $1024 = 2^{10}$ in the computer world, but use of the SI-units is now encouraged. However, memory size is always reported using the traditional binary units and most operating systems also use these units to report hard disk sizes and file sizes. So a file containing 3 913 880 bytes will typically be reported as being 3.7 MB.

To illustrate the size of the numbers in table 4.1 it is believed that the world's total storage in 2006 was 160 exabytes, and the projection is that this will grow to nearly one zettabyte by 2010.

4.1.2   Fixed size integers

Since the hardware can handle groups of 4 or 8 bytes efficiently, the representation of integers is usually adapted to this format. If we use 4 bytes we have 32 binary digits at our disposal, but how should we use these bits? We would certainly like to be able to handle both negative and positive numbers, so we use one bit to signify whether the number is positive or negative. We then have 31 bits left to represent the binary digits of the integer. This means that the largest 32-bit integer that can be handled is the number where all 31 digits are 1, i.e.,

$$1 \cdot 2^{30} + 1 \cdots 2^{29} + \cdots + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 2^{31} - 1.$$

Based on this it may come as a little surprise that the most negative number that can be represented is -2^{31} and not $-2^{31} + 1$. The reason is that with 32 bits at our disposal we can represent a total of 2^{32} numbers. Since we need 2^{31} bit combinations for the positive numbers and 0, we have $2^{32} - 2^{31} = 2^{31}$ combinations of digits left for the negative numbers. Similar limits can be derived for 64-bit integers.

Fact 4.2. *The smallest and largest numbers that can be represented by 32-bit integers are*

$$I_{\min 32} = -2^{31} = -2\ 147\ 483\ 648, \quad I_{\max 32} = 2^{31} - 1 = 2\ 147\ 483\ 647.$$

With 64-bit integers the corresponding numbers are

$$I_{\min 64} = -2^{63} = -9\ 223\ 372\ 036\ 854\ 775\ 808,$$
$$I_{\max 64} = 2^{63} - 1 = 9\ 223\ 372\ 036\ 854\ 775\ 807.$$

Decimal	Two's compl.	Decimal	Two's compl.
7	0111	-1	1111
6	0110	-2	1110
5	0101	-3	1101
4	0100	-4	1100
3	0011	-5	1011
2	0010	-6	1010
1	0001	-7	1001
0	0000	-8	1000

Table 4.2. Two's complement representation of four bit integers.

What we have discussed so far is the typical hardware support for integer numbers. When we program a computer we have to use a suitable programming language, and different languages may provide different interfaces to the hardware. There are a myriad of computer languages, and particularly the handling of integers may differ quite a bit. We will briefly review integer handling in two languages, Java and Python, as representatives of two different approaches.

4.1.3 Two's complement

The informal description of how integers are represented left out most details. Suppose for example that we use 4 bits, then we indicated that the number 0 would be represented as 0000, 1 as 0001, and the largest possible positive integer would be 0111 (7 in decimal). The negative numbers would have '1' as the leftmost bit, so -1 would be 1001. This causes a slight problem since $-1 + 1$ in the computer would become

$$1001 + 0001 = 1000.$$

This result would have to treated as 0, so both 1000 and 0000 would represent zero. This little example shows that the naive representation of integers leads to complications.

The actual representation of integers on most computers makes use of a technique called *two's complement*. In this system the positive integers are represented as above, but the negative integers are represented differently. For 4 bit integers, the representation is shown in table 4.2. We observe immediately that there is only one representation of 0. But more importantly, addition has become much simpler: We just add the two numbers, and if we get overflow we discard the extra bit. Some examples will illustrate the details.

The addition $-3 + 1$ corresponds to $1101 + 0001$ in two's complement. By using ordinary binary addition the result is 1110, which we see from the table is the correct -2 in decimal.

The addition $-1 + 1$ was problematic before. In two's complement it becomes $1111 + 0001$. The result of this is the five bit number 10000, but we only have four bits available so the fifth bit is discarded. The result is 0000 which represents 0, the correct answer. It therefore appears that two's complement has overcome both the problems we had with the naive representation above.

Two's complement representation can be summarised quite concisely with a simple formula.

Fact 4.3 (Two's complement). *With two's complement representation a non-negative number is represented by its binary digits, while a negative number x is represented by the binary digits of the positive number*

$$x^- = 2^n - |x|, \tag{4.1}$$

where n is the total number of bits in the representation.

Example 4.4. Let us find the representation of $x = -6$ in two's complement from (4.1) when $n = 4$. In this case $|x| = 6$ so $x^- = 2^4 - 6 = 10$. The binary representation of the decimal number 10 is 1010 and this is the representation of -6 in two's complement. ∎

4.1.4 Integers in Java

Java is a typed language which means that the type of all variables has to be stated explicitly. If we wish to store 32-bit integers in the variable n, we use the declaration int n and we say that n is an int variable. If we wish to use n as a 64-bit variable, we use the declaration long n and say that n is a long variable. Integers appearing to the right of an assignment are considered to be of type int, but you may specify that an integer is to be interpreted as a long integer by appending an L. In other words, an expression like 2+3 will be computed as an int whereas the expression 2L+3L will be computed as a long, using 64 bits.

Since Java has integer types of fixed size, something magic must happen when the result of an integer computation becomes too large for the type. Suppose for example that we run the code segment

```
int a;
a = 2147483647;
a = a + 1;
```

The staring value for `a` is the largest possible 32-bit integer, and when we add 1 we obtain a number that is too big for an `int`. This is referred to by saying that an *overflow* occurs. So what happens when an integer overflows in Java? The statements above will lead to `a` receiving the value -2147483648, and Java gives no warning about this strange behaviour! If you look carefully, the result is -2^{31}, i.e., the smallest possible `int`. Basically Java (and similar languages) consider the 32-bit integers to lie in a ring where the integer succeeding $2^{31} - 1$ is -2^{31} (overflow in `long` integers are handled similarly). Sometimes this may be what you want, but most of the time this kind of behaviour is probably going to give you a headache unless you remember this paragraph!

Note that Java also has 8 bit integers (`byte`) and 16 bit integers (`short`). These behave completely analogously to `int` and `long` variables.

It is possible to work with integers that require more than 64 bits in Java, but then you have to resort to an auxiliary class called `BigInteger`. In this class integers are only limited by the total resources available on your computer, but the cost of resorting to `BigInteger` is a big penalty in terms of computing time.

4.1.5 Integers in Python

Python handles integers quite differently from Java. First of all you do not need to declare the type of variables in Python. So if you write something like `a=2+3` then Python will look at the right-hand side of the assignment, conclude that this is an integer expression and store the result in an integer variable. An integer variable in Python is called an `int` and on most computers this will be a 32-bit integer. The good news is that Python handles overflow much more gracefully than Java. If Python encounters an integer expression that is greater than $2^{31} - 1$ it will be converted to what is called a `long` integer variable in Python. Such variables are only bounded by the available resources on the computer, just like `BigInteger` in Java. You can force an integer expression that fits into an `int` to be treated as a `long` integer by using the function `long`. For example, the expression `long(13)` will give the result 13L, i.e., the number 13 represented as a `long` integer. Similarly, the expression `int(13L)` will convert back to an `int`.

This means that overflow is very seldom a problem in Python, as virtually all computers today should have sufficient resources to avoid overflow in ordinary computations. But it may of course happen that you make a mistake that result in a computation producing very large integers. You will notice this in that your program takes a very long time and may seem to be stuck. This is because your computation is consuming all resources in the computer so that everything else comes to a standstill. You could wait until you get an error message, but this may take a long time so it is usually better to just abort the computation.

Since `long` integers in Python can become very large, it may be tempting

to use them all the time and ignore the `int` integers. The problem with this is that the `long` integers are implemented in extra program code (usually referred to as *software*), just like the `BigInteger` type in Java, and this is comparatively slow. In contrast, operations with `int` integers are explicitly supported by the hardware and is very fast.

4.1.6   Division by zero

Other than overflow, the only potential problem with integer computation is division by zero. This is mathematically illegal and results in an error message and the computations being halted (or an exception is raised) in most programming languages.

4.2   Computers and real numbers

Computations with integers is not sufficient for many parts of mathematics; we must also be able to compute with real numbers. And just like for integers, we want fast computations so we can solve large and challenging problems. This inevitably means that there will be limitations on the class of real numbers that can be handled efficiently by computers.

To illustrate the challenge, consider the two real numbers

$$\pi = 3.14159265358979323846264338327950288419\ldots,$$
$$10^6\pi = 3.14159265358979323846264338327950288419\ldots \times 10^6.$$

Both of these numbers are irrational and require infinitely many digits in any numeral system with an integer base. With a fixed number of digits at our disposal we can only store the most significant (the left-most) digits, which means that we have to ignore infinitely many digits. But this is not enough to distinguish between the two numbers π and $10^6\pi$, we also have to store information about the size of the numbers.

The fact that many real numbers have infinitely many digits and we can only store a finite number of these means that there is bound to be an error when real numbers are represented on a computer. This is in marked contrast to integer numbers where there is no error, just a limit on the size of numbers. The errors are usually referred to as *rounding errors* or *round-off errors*. These errors are also present on calculators and a simple situation where round-off error can be observed is by computing $\sqrt{2}$, squaring the result and subtracting 2. On one calculator the result is approximately 4.4×10^{-16}, a clear manifestation of round-off error.

Usually the round-off error is small and remains small throughout a computation. In some cases however, the error grows throughout a computation and

61

may become significant. In fact, there are situations where the round-off error in a result is so large that all the displayed digits are wrong! Computations which lead to large round-off errors are said to be *badly conditioned* while computations with small errors are said to be *well conditioned*.

Since some computations may lead to large errors it is clearly important to know in advance if a computation may be problematic. Suppose for example you are working on the development of a new aircraft and you are responsible for simulations of the forces acting on the wings during flight. Before the first flight of the aircraft you had better be certain that the round-off errors (and other errors) are under control. Such error analysis is part of the field called *Numerical Analysis*.

4.2.1 Representation of real numbers

To understand round-off errors and other characteristics of how computers handle real numbers, we must understand how real numbers are represented. We are going to do this by first pretending that computers work in the decimal numeral system. Afterwards we will translate our observations to the binary representation that is used in practice.

Any real number can be expressed in the decimal system, but infinitely many digits may be needed. To represent such numbers with finite resources we must limit the number of digits. Suppose for example that we use four decimal digits to represent real numbers. Then the best representations of the numbers π, 1/700 and 100003/17 would be

$$\pi \approx 3.142,$$

$$\frac{1}{700} \approx 0.001429,$$

$$\frac{100003}{17} \approx 5883.$$

If we consider the number $100000000/23 \approx 4347826$ we see that it is not representable with just four digits. However, if we write the number as 0.4348×10^7 we can represent the number if we also store the exponent 7. This is the background for the following simple observation.

Observation 4.5 (Normal form of real number)**.** *Let a be a real number different from zero. Then a can be written uniquely as*

$$a = b \times 10^n \tag{4.2}$$

Note that the digits of a and b are the same; to arrive at the normal form in (4.2) we simply multiply a by the power of 10 that brings b into the range given by (4.3).

The normal form of π, 1/7, 100003/17 and 10000000/23 are

$$\pi \approx 0.3142 \times 10^1,$$
$$\frac{1}{7} \approx 0.1429 \times 10^0,$$
$$\frac{100003}{17} \approx 0.5883 \times 10^4,$$
$$\frac{10000000}{23} \approx 0.4348 \times 10^7.$$

From this we see that if we reserve four digits for the significand and one digit for the exponent, plus a sign for both, then we have a format that can accommodate all these numbers. If we keep the significand fixed and vary the exponent, the decimal point moves among the digits. For this reason this kind of format is called *floating point*, and numbers represented in this way are called *floating point numbers*.

It is always useful to be aware of the smallest and largest numbers that can be represented in a format. With four digits for the significand and one digit for the exponent plus signs, these numbers are

$$-0.9999 \times 10^9, \quad -0.1000 \times 10^{-9},$$
$$0.1000 \times 10^{-9}, \quad 0.9999 \times 10^9.$$

In practice, a computer uses a binary representation. Before we consider details of how many bits to use etc., we must define a normal form for binary numbers. This is a straightforward generalisation from the decimal case.

Observation 4.6 (Binary normal form of real number). *Let a be a real number different from zero. Then a can be written uniquely as*

$$a = b \times 2^n$$

> *where b is bounded by*
> $$\frac{1}{2} \le |b| < 1$$
> *and n is an integer. This is called the binary normal form of a, and the number b is called the significand while n is called the exponent of a. The normal form of 0 is $0 = 0 \times 2^0$.*

This is completely analogous to the decimal version in Observation 4.5 in that all occurrences of 10 have been replaced by 2. Most of today's computers use 32 or 64 bits to represent real numbers. The 32-bit format is useful for applications that do not demand high accuracy, but 64 bits has become a standard for most scientific applications. Occasionally higher accuracy is required in which case there are some formats with more bits or even a format with no limitation other than the resources available in the computer.

To describe a floating point format, it is not sufficient to state how many bits are used in total, we also have to know how many bits are used for the significand and how many for the exponent. There are several possible ways to do this, but there is an international standard for floating point computations that is used by most computer manufacturers. This standard is referred to as the IEEE[1] 754 standard, and the main details of the 32-bit version is given below.

> **Fact 4.7** (IEEE 32-bit floating point format). *With 32-bit floating point numbers 23 bits are allocated for the significand and 9 bits for the exponent, both including signs. This means that numbers have about 6–9 significant decimal digits. The smallest and largest negative numbers in this format are*
>
> $$F_{\min 32}^{-} \approx -3.4 \times 10^{38}, \quad F_{\max 32}^{-} \approx -1.4 \times 10^{-45}.$$
>
> *The smallest and largest positive numbers are*
>
> $$F_{\min 32}^{+} \approx 1.4 \times 10^{-45}, \quad F_{\max 32}^{+} \approx 3.4 \times 10^{38}.$$

This is just a summary of the most important characteristics of the 32-bit IEEE-standard; there are a number of details that we do not want to delve into here. However, it is worth pointing out that when any nonzero number a is expressed in binary normal form, the first bit of the significand will always be 1

---

[1] IEEE is an abbreviation for Institute of Electrical and Electronics Engineers which is a professional technological association.

(remember that we simply shift the binary point until the first bit is 1). Since this bit is always 1, it does not need to be stored. This means that in reality we have 24 bits (including sign) available for the significand. The only exception to this rule is when the exponent has its smallest possible value. Then the first bit is assumed to be 0 (these correspond to so-called denormalised numbers) and this makes it possible to represent slightly smaller numbers than would otherwise be possible. In fact the smallest positive 32-bit number with 1 as first bit is approximately 1.2×10^{-38}.

Not all bit combinations in the IEEE standard are used for ordinary numbers. Three of the extra 'numbers' are `-Infinity`, `Infinity` and `NaN`. The infinities typically occur during overflow. For example, if you use 32-bit floating point and perform the multiplication $10^{30} * 10^{30}$, the result will be `Infinity`. The negative infinity behaves in a similar way. The `NaN` is short for 'Not a Number' and is the result if you try to perform an illegal operation. A typical example is if you try to compute $\sqrt{-1}$ without using complex numbers, this will give `NaN` as the result. And once you have obtained a `NaN` result it will pollute anything that it touches; `NaN` combined with anything else will result in `NaN`.

With 64-bit numbers we have 32 extra bits at our disposal and the question is how these should be used. The creators of the IEEE standard believed improved accuracy to be more important than support for very large or very small numbers. They therefore increased the number of bits in the significand by 30 and the number of bits in the exponent by 2.

Fact 4.8 (IEEE 64-bit floating point format)**.** *With 64-bit floating point numbers 53 bits are allocated for the significand and 11 bits for the exponent, both including signs. This means that numbers have about 15–17 significant decimal digits. The smallest and largest negative number in this format are*

$$F_{\min 64}^{-} \approx -1.8 \times 10^{308}, \quad F_{\max 64}^{-} \approx -5 \times 10^{-324}.$$

The smallest and largest positive numbers are

$$F_{\min 64}^{+} \approx 5 \times 10^{-324}, \quad F_{\max 64}^{+} \approx 1.8 \times 10^{308}.$$

Other than the extra bits available, the 64-bit format behaves just like its 32-bit little brother, with the leading 1 not being stored, the use of denormalised numbers, `-Infinity`, `Infinity` and `NaN`.

4.2.2 Floating point numbers in Java

Java has two floating point types, `float` and `double`, which are direct implementations of the 32-bit and 64-bit IEEE formats described above. In Java the result of `1.0/0.0` will be `Infinity` without a warning.

4.2.3 Floating point numbers in Python

In Python floating point numbers come into action as soon as you enter a number with a decimal point. Such numbers are represented in the 64-bit format described above and most of the time the computations adhere to the IEEE standard. However, there are some exceptions. For example, the division `1.0/0.0` will give an error message and the symbol for 'Infinity' is `Inf`.

In standard Python, there is no support for 32-bit floating point numbers. However, you gain access to this if you import the `NumPy` library.

4.3 Representation of letters and other characters

At the lowest level, computers can just handle 0s and 1s, and since any number can be expressed uniquely in the binary number system it can also be represented in a computer (except for the fact that we may have to limit both the size of the numbers and their number of digits). We all know that computers can also handle text and in this section we are going to see the basic principles of how this is done.

A text is just a sequence of individual characters like 'a', 'B', '3', '.', '?', i.e., upper- and lowercase letters, the digits 0–9 and various other symbols used for punctuation and other purposes. So the basic challenge in handling text is how to represent the individual characters. With numbers at our disposal, this is a simple challenge to overcome. Internally in the computer a character is just represented by a number and the correspondence between numbers and characters is stored in a table. The letter 'a' for example, usually has code 97. So when the computer is told to print the character with code 97, it will call a program that draws an 'a'[2]. Similarly, when the user presses the 'a' on the keyboard, it is immediately converted to code 97.

> **Fact 4.9** (Representation of characters). *In computers, characters are represented in terms of integer codes and a table that maps the integer codes to the different characters. During input each character is converted to the corresponding integer code, and during output the code is used to determine which character to draw.*

---

[2]The shape of the different characters are usually defined as mathematical curves.

Although the two concepts are slightly different, we will use the terms 'character sets' and 'character mappings' as synonyms.

From fact 4.9 we see that the character mapping is crucial in how text is handled. Initially, the mappings were simple and computers could only handle the most common characters used in English. Today there are extensive mappings available that make the characters of most of the world's languages, including the ancient ones, accessible. Below we will briefly describe some of the most common character sets.

4.3.1 The ASCII table

In the infancy of the digital computer there was no standard for mapping characters to numbers. This made it difficult to transfer information from one computer to another, and the need for a standard soon became apparent. The first version of ASCII (American Standard Code for Information Interchange) was published in 1963 and it was last updated in 1986. ASCII defines codes for 128 characters that are commonly used in English plus some more technical characters. The fact that there are $128 = 2^7$ characters in the ASCII table means that 7 bits are needed to represent the codes. Today's computers usually handle one byte (eight bits) at a time so the ASCII character set is now normally just part of a larger character set, see below.

Table 4.3 (towards the end of this chapter) shows the ASCII characters with codes 32–127. We notice the upper case letters with codes 65–90, the lower case letters with codes 97–122 and the digits 0–9 with codes 48–57. Otherwise there are a number of punctuation characters and brackets as well as various other characters that are used more or less often. Observe that there are no characters from the many national alphabets that are used around the world. ASCII was developed in the US and was primarily intended to be used for giving a textual representation of computer programs which mainly use vocabulary from English. Since then computers have become universal tools that process all kinds of information, including text in many different languages. As a result new character sets have been developed, but almost all of them contain ASCII as a subset.

Character codes are used for arranging words in alphabetical order. To compare the two words 'high' and 'all' we just check the character codes. We see that 'h' has code 104 while 'a' has code 97. So by ordering the letters according to their character codes we obtain the normal alphabetical order. Note that the codes of upper case letters are smaller than the codes of lower case letters. This means that capitalised words and words in upper case precede words in lower case in the standard ordering.

Table 4.4 shows the first 32 ASCII characters. These are quite different from most of the others (with the exception of characters 32 and 127) and are called

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
32	20	SP	64	40	@	96	60	'
33	21	!	65	41	A	97	61	a
34	22	"	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	'	71	47	G	103	67	g
40	28	(	72	48	H	104	68	h
41	29	)	73	49	I	105	69	i
42	2a	*	74	4a	J	106	6a	j
43	2b	+	75	4b	K	107	6b	k
44	2c	,	76	4c	L	108	6c	l
45	2d	-	77	4d	M	109	6d	m
46	2e	.	78	4e	N	110	6e	n
47	2f	/	79	4f	O	111	6f	o
48	30	0	80	50	P	112	70	p
49	31	1	81	51	Q	113	71	q
50	32	2	82	52	R	114	72	r
51	33	3	83	53	S	115	73	s
52	34	4	84	54	T	116	74	t
53	35	5	85	55	U	117	75	u
54	36	6	86	56	V	118	76	v
55	37	7	87	57	W	119	77	w
56	38	8	88	58	X	120	78	x
57	39	9	89	59	Y	121	79	y
58	3a	:	90	5a	Z	122	7a	z
59	3b	;	91	5b	[	123	7b	{
60	3c	<	92	5c	\	124	7c		
61	3d	=	93	5d	]	125	7d	}
62	3e	>	94	5e	^	126	7e	~
63	3f	?	95	5f	_	127	7f	BCD

Table 4.3. The ASCII characters with codes 32–127. The character with decimal code 32 is white space, and the one with code 127 is 'delete'.

control characters. They are not intended to be printed in ink on paper, but rather indicate some kind of operation to be performed by the printing equipment or a signal to be communicated to a sender or receiver of the text. Some of the characters are hardly used any more, but others have retained their significance. Character 4 (^D) has the description 'End of Transmission' and is often

Dec	Hex	Abbr	CS	Description
0	00	NUL	^@	Null character
1	01	SOH	^A	Start of Header
2	02	STX	^B	Start of Text
3	03	ETX	^C	End of Text
4	04	EOT	^D	End of Transmission
5	05	ENQ	^E	Enquiry
6	06	ACK	^F	Acknowledgment
7	07	BEL	^G	Bell
8	08	BS	^H	Backspace
9	09	HT	^I	Horizontal Tab
10	0a	LF	^J	Line feed
11	0b	VT	^K	Vertical Tab
12	0c	FF	^L	Form feed
13	0d	CR	^M	Carriage return
14	0e	SO	^N	Shift Out
15	0f	SI	^O	Shift In
16	10	DLE	^P	Data Link Escape
17	11	DC1	^Q	XON
18	12	DC2	^R	Device Control 2
19	13	DC3	^S	XOFF
20	14	DC4	^T	Device Control 4
21	15	NAK	^U	Negative Acknowledgement
22	16	SYN	^V	Synchronous Idle
23	17	ETB	^W	End of Trans. Block
24	18	CAN	^X	Cancel
25	19	EM	^Y	End of Medium
26	1a	SUB	^Z	Substitute
27	1b	ESC	^[	Escape
28	1c	FS	^\	File Separator
29	1d	GS	^]	Group Separator
30	1e	RS	^^	Record Separator
31	1f	US	^_	Unit Separator

Table 4.4. The first 32 characters of the ASCII table. The first two columns show the code number in decimal and octal, the third column gives a standard abbreviation for the character and the fourth column gives a printable representation of the character. The last column gives a more verbose description of the character.

used to signify the end of a file, at least under Unix-like operating systems. Because of this, many programs that operate on files, like for example text-editors, will quit if you type ^D (hold down the control-key while you press 'd'). Various combinations of characters 10, 12 and 13 are used in different operating systems

for indicating a new line within a file. The meaning of character 13 ('Carriage Return') was originally to move back to the beginning of the current line and character 10 ('Line Feed') meant forward one line.

4.3.2   ISO latin character sets

As text processing by computer became a generally available in the 1980s, extensions of the ASCII character set that included various national characters used in European languages were needed. The International Standards Organisation (ISO) developed a number of such character sets, like ISO Latin 1 ('Western'), ISO Latin 2 ('Central European') and ISO Latin 5 ('Turkish'), and so did several computer manufacturers. Virtually all of these character sets retained ASCII in the first 128 positions, but increased the code from seven to eight bits to accomodate another 128 characters. This meant that different parts of the Western world had local character sets which could encode their national characters, but if a file was interpreted with the wrong character set, some of the characters beyond position 127 would come out wrong.

Table 4.5 shows characters 192–255 in the ISO Latin 1 character set. These include most latin letters with diacritics used in the Western European languages. Positions 128–191 in the character set are occupied by some control characters similar to those at the beginning of the ASCII table but also a number of other useful characters.

4.3.3   Unicode

By the early 1990s there was a critical need for character sets that could handle multilingual characters, like those from English and Chinese, in the same document. A number of computer companies therefore set up an organisation called Unicode. Unicode has since then organised the characters of most of the world's languages in a large table called the Unicode table, and new characters are still being added. There are a total of about 100 000 characters in the table which means that at least three bytes are needed for their representation. The codes range from 0 to 1114111 (hexadecimal 10ffff_{16}) which means that only about 10 % of the table is filled. The characters are grouped together according to language family or application area, and the empty spaces make it easy to add new characters that may come into use. The first 256 characters of Unicode is identical to the ISO Latin 1 character set, and in particular the first 128 characters correspond to the ASCII table. You can find all the Unicode characters at `http://www.unicode.org/charts/`.

One could use the same strategy with Unicode as with ASCII and ISO Latin 1 and represent the characters via their integer codes (usually referred to as *code*

Dec	Hex	Char	Dec	Hex	Char
192	c0	À	224	e0	à
193	c1	Á	225	e1	á
194	c2	Â	226	e2	â
195	c3	Ã	227	e3	ã
196	c4	Ä	228	e4	ä
197	c5	Å	229	e5	å
198	c6	Æ	230	e6	æ
199	c7	Ç	231	e7	ç
200	c8	È	232	e8	è
201	c9	É	233	e9	é
202	ca	Ê	234	ea	ê
203	cb	Ë	235	eb	ë
204	cc	Ì	236	ec	ì
205	cd	Í	237	ed	í
206	ce	Î	238	ee	î
207	cf	Ï	239	ef	ï
208	d0	Ð	240	f0	ð
209	d1	Ñ	241	f1	ñ
210	d2	Ò	242	f2	ò
211	d3	Ó	243	f3	ó
212	d4	Ô	244	f4	ô
213	d5	Õ	245	f5	õ
214	d6	Ö	246	f6	ö
215	d7	×	247	f7	÷
216	d8	Ø	248	f8	ø
217	d9	Ù	249	f9	ù
218	da	Ú	250	fa	ú
219	db	Û	251	fb	û
220	dc	Ü	252	fc	ü
221	dd	Ý	253	fd	ý
222	de	Þ	254	fe	þ
223	df	ß	255	ff	ÿ

Table 4.5. The last 64 characters of the ISO Latin1 character set.

points) in the Unicode table. This would mean that each character would require three bytes of storage. The main disadvantage of this is that a program for reading Unicode text would give completely wrong results if by mistake it was used for reading 'old fashioned' eight bit text, even if it just contained ASCII characters. Unicode has therefore developed variable length encoding schemes

for encoding the characters.

4.3.4 UTF-8

A popular encoding of Unicode is UTF-8[3]. UTF-8 has the advantage that ASCII characters are encoded in one byte so there is complete backwards compatibility with ASCII. All other characters require from two to four bytes.

To see how the code points are actually encoded in UTF-8, recall that the ASCII characters have code points in the range 0–127 (decimal) which is 00_{16}–$7f_{16}$ in hexadecimal or 00000000_2–01111111_2 in binary. These characters are just encoded in one byte in the obvious way and are characterised by the fact that the most significant (the left-most) bit is 0. All other characters require more than one byte, but the encoding is done in such a way that none of these bytes start with 0. This is done by adding some set fixed bit combinations at the beginning of each byte. Such codes are called prefix codes. The details are given in a fact box.

Fact 4.10 (UTF-8 encoding of Unicode). *A Unicode character with code point c is encoded in UTF-8 according to the following four rules:*

1. *If $c = (d_6 d_5 d_4 d_3 d_2 d_1 d_0)_2$ is in the decimal range 0–127 (hexadecimal 00_{16}–$7f_{16}$), it is encoded in one byte as*

$$0d_6 d_5 d_4 d_3 d_2 d_1 d_0. \tag{4.4}$$

2. *If $c = (d_{10} d_9 d_8 d_7 d_6 d_5 d_4 d_3 d_2 d_1 d_0)_2$ is in the decimal range 128–2047 (hexadecimal 80_{16}–$7ff_{16}$) it is encoded as the two-byte binary number*

$$110d_{10} d_9 d_8 d_7 d_6 \quad 10d_5 d_4 d_3 d_2 d_1 d_0. \tag{4.5}$$

3. *If $c = (d_{15} d_{14} d_{13} d_{12} d_{11} d_{10} d_9 d_8 d_7 d_6 d_5 d_4 d_3 d_2 d_1 d_0)_2$ is in the decimal range 2048–65535 (hexadecimal 800_{16}–$ffff_{16}$) it is encoded as the three-byte binary number*

$$1110d_{15} d_{14} d_{13} d_{12} \quad 10d_{11} d_{10} d_9 d_8 d_7 d_6 \quad 10d_5 d_4 d_3 d_2 d_1 d_0. \tag{4.6}$$

4. *If $c = (d_{20} d_{19} d_{18} d_{17} d_{16} d_{15} d_{14} d_{13} d_{12} d_{11} d_{10} d_9 d_8 d_7 d_6 d_5 d_4 d_3 d_2 d_1 d_0)_2$ is in the decimal range 65536–1114111 (hexadecimal 10000_{16}–$10ffff_{16}$) it is encoded as the four-byte binary number*

$$11110d_{20} d_{19} d_{18} \quad 10d_{17} d_{16} d_{15} d_{14} d_{13} d_{12}$$
$$10d_{11} d_{10} d_9 d_8 d_7 d_6 \quad 10d_5 d_4 d_3 d_2 d_1 d_0. \tag{4.7}$$

---

[3]UTF is an abbreviation of Unicode Transformation Format.

This may seem complicated at first sight, but is in fact quite simple and elegant. Note any given byte in a UTF-8 encoded text must start with the binary digits 0, 10, 110, 1110 or 11110. If the first bit in a byte is 0, the remaining bits represent a seven bit ASCII character. If the first two bits are 10, the byte is the second, third or fourth byte of a multi-byte code point, and we can find the first byte by going back in the byte stream until we find a byte that does not start with 10. If the byte starts with 110 we know that it is the first byte of a two-byte code point; if it starts with 1110 it is the first byte of a three-byte code point; and if it starts with 11110 it is the first of a four-byte code point.

Observation 4.11. *It is always possible to tell if a given byte within a text encoded in UTF-8 is the first, second, third or fourth byte in the encoding of a code point.*

The UTF-8 encoding is particularly popular in the Western world since all the common characters of English can be represented by one byte, and almost all the national European characters can be represented with two bytes.

Example 4.12. Let us consider a concrete example of how the UTF-8 code of a code point is determined. The ASCII characters are not so interesting since for these characters the UTF-8 code agrees with the code point. The Norwegian character 'Å' is more challenging. If we check the Unicode charts,[4] we find that this character has the code point $c5_{16} = 197$. This is in the range 128–2047 which is covered by rule 2 in fact 4.10. To determine the UTF-8 encoding we must find the binary representation of the code point. This is easy to deduce from the hexadecimal representation. The least significant numeral (5 in our case) determines the four least significant bits and the most significant numeral (c) determines the four most significant bits. Since $5 = 0101_2$ and $c_{16} = 1100_2$, the code point in binary is

$$000\overbrace{1100}^{c}\overbrace{0101}^{5}{}_2,$$

where we have added three 0s to the left to get the eleven bits referred to by rule 2. We then distribute the eleven bits as in (4.5) and obtain the two bytes

$$11000011, \quad 10000101.$$

In hexadecimal this corresponds to the two values $c3$ and 85 so the UTF-8 encoding of 'Å' is the two-byte number $c385_{16}$. ∎

---

[4]The Latin 1 supplement can be found at `www.unicode.org/charts/PDF/U0080.pdf`.

4.3.5 UTF-16

Another common encoding is UTF-16. In this encoding most Unicode characters with two-byte code points are encoded directly by their code points. Since the characters of major Asian languages like Chinese, Japanese and Korean are encoded in this part of Unicode, UTF-16 is popular in this part of the world. UTF-16 is also the native format for representation of text in the recent versions of Microsoft Windows and Apple's Mac OS X as well as in programming environments like Java, .Net and Qt.

UTF-16 uses a variable width encoding scheme similar to UTF-8, but the basic unit is two bytes rather than one. This means that all code points are encoded in two or four bytes. In order to recognize whether two consecutive bytes in an UTF-16 encoded text correspond to a two-byte code point or a four-byte code point, the initial bit patterns of each pair of a four byte code has to be illegal in a two-byte code. This is possible since there are big gaps in the Unicode table. In fact certain Unicode code points are reserved for the specific purpose of signifying the start of pairs of four-byte codes (so-called surrogate pairs).

Fact 4.13 (UTF-16 encoding of Unicode). *A Unicode character with code point c is encoded in UTF-16 according to two rules:*

1. *If the number*

$$c = (d_{15}d_{14}d_{13}d_{12}d_{11}d_{10}d_9d_8d_7d_6d_5d_4d_3d_2d_1d_0)_2$$

   is a code point in the range 0–65535 (hexadecimal 0000_{16}–ffff_{16}) it is encoded as the two bytes

$$d_{15}d_{14}d_{13}d_{12}d_{11}d_{10}d_9d_8 \quad d_7d_6d_5d_4d_3d_2d_1d_0.$$

2. *If the number*

$$c = (d_{20}d_{19}d_{18}d_{17}d_{16}d_{15}d_{14}d_{13}d_{12}d_{11}d_{10}d_9d_8d_7d_6d_5d_4d_3d_2d_1d_0)_2$$

   is a code point in the range 65536–1114111 (hexadecimal 10000_{16}–10ffff_{16}), compute the number $c' = c - 65536$ (subtract 10000_{16}). This number can be represented by 20 bits,

$$c' = (d'_{19}d'_{18}d'_{17}d'_{16}d'_{15}d'_{14}d'_{13}d'_{12}d'_{11}d'_{10}d'_9d'_8d'_7d'_6d'_5d'_4d'_3d'_2d'_1d'_0)_2.$$

   The encoding of c is then given by the four bytes

$$110110d'_{19}d'_{18} \quad d'_{17}d'_{16}d'_{15}d'_{14}d'_{13}d'_{12}d'_{11}d'_{10}$$
$$110111d'_9d'_8 \quad d'_7d'_6d'_5d'_4d'_3d'_2d'_1d'_0.$$

Superficially it may seem like UTF-16 does not have the prefix property, i.e., it may seem that a pair of bytes produced by rule 2 may occur as a pair generated by rule 1 and vice versa. However, the existence of gaps in the Unicode table means that this problem does not occur.

> **Observation 4.14.** *None of the pairs of bytes produced by rule 2 in fact 4.13 will ever match a pair of bytes produced by the first rule as there are no two-byte code points that start with the bit sequences 110110 or 110111. It is therefore always possible to determine whether a given pair of consecutive bytes in an UTF-16 encoded text corresponds directly to a code point (rule 1), or is the first or second pair of a four byte encoding.*

The UTF-16 encoding has the advantage that all two-byte code points are encoded directly by their code points. Since the characters that require more than two-byte code points are very rare, this means that virtually all characters are encoded directly in two bytes.

UTF-16 has one technical complication. Different computer architectures code pairs of bytes in different ways: Some will insist on sending the eight most significant bits first, some will send the eight least significant bits first; this is usually referred to as *little endian* and *big endian*. To account for this there are in fact three different UTF-16 encoding schemes, UTF-16, UTF-16BE and UTF-16LE. UTF-16BE uses strict big endian encoding while UTF-16LE uses strict little endian encoding. UTF-16 does not use a specific endian convention. Instead any file encoded with UTF-16 should indicate the endian by having as its first two bytes what is called a *Byte Order Mark* (BOM). This should be the hexadecimal sequence feff_{16} for big-endian and fffe_{16} for little-endian. This character, which has code point feff, is chosen because it should never legitimately appear at the beginning of a text.

4.3.6 UTF-32

UTF-32 encode Unicode characters by encoding the code point directly in four bytes or 32 bits. In other words it is a fixed length encoding. The disadvantage is that this encoding is rather extravagant since many frequently occurring characters in Western languages can be encoded with only one byte, and almost all characters can be encoded with two bytes. For this reason UTF-32 is little used in practice.

4.3.7 Text in Java

Characters in Java are represented with the `char` data type. The representation is based on the UTF-16 encoding of Unicode so all the Unicode characters are

available. The fact that some characters require three bytes to represent their code points is a complication, but this is handled nicely in the libraries for processing text.

4.3.8 Text in Python

Python also has support for Unicode. You can use Unicode text in your source file by including a line which indicates the specific encoding, for example as in

```
# coding=utf-8/
```

You can then use Unicode in your string constants which in this case will be encoded in UTF-8. All the standard string functions also work for Unicode strings, but note that the default encoding is ASCII.

4.4 Representation of general information

So far we have seen how numbers and characters can be represented in terms of bits and bytes. This is the basis for representing all kinds of different information. Let us start with general text files.

4.4.1 Text

A text is simply a sequence of characters. We know that a character is represented by an integer code so a text file is a sequence of integer codes. If we use the ISO Latin 1 encoding, a file with the text

Knut
Mørken

is represented by the hexadecimal codes (recall that each code is a byte)

$$4b\ 6e\ 75\ 74\ 0a\ 4d\ f8\ 72\ 6b\ 65\ 6e$$

The first four bytes you will find in table 4.3 as the codes for 'K', 'n', 'u' and 't' (remember that the codes of latin characters in ISO Latin 1 are the same as in ASCII). The fifth character has decimal code 10 which you find in table 4.4. This is the Line feed character which causes a new line on my computer. The remaining codes can all be found in table 4.3 except for the seventh which has decimal code 248. This is located in the upper 128 ISO Latin 1 characters and corresponds to the Norwegian letter 'ø' as can be seen in table 4.5.

If instead the text is represented in UTF-8, we obtain the bytes

$$4b\ 6e\ 75\ 74\ 0a\ 4d\ c3\ b8\ 72\ 6b\ 65\ 6e$$

We see that these are the same as for ISO Latin 1 except that 'f8' has become two bytes 'c3 b8' which is the two-byte code for 'ø' in UTF-8.

In UTF-16 the text is represented by the codes

ff fe 4b 00 6e 00 75 00 74 00 0a 00 4d 00 f8 00 72 00 6b 00 65 00 6e 00

All the characters can be represented by two bytes and the leading byte is '00' since we only have ISO Latin 1 characters. It may seem a bit strange that the zero byte comes after the nonzero byte, but this is because the computer uses little endian representation. A program reading this file would detect this from the first two bytes which is the byte-order mark referred to above.

4.4.2 Numbers

A number can be stored in a file by finding its binary representation and storing the bits in the appropriate number of bytes. The number $13 = 1101_2$ for example could be stored as a 32 bit integer by storing the the bytes 00 00 00 0d (in hexadecimal).[5] But here there is a possibly confusing point: Why can we not just store the number as a text? This is certainly possible and if we use UTF-8 we can store 13 as the two bytes 31 33 (in hexadecimal). This even takes up less space than the four bytes required by the true integer format. For bigger numbers however the situation is the opposite: Even the largest 32-bit integer can be represented by four bytes if we use integer format, but since it is a ten-digit number we would need ten bytes to store it as a text.

In general it is advisable to store numbers in the appropriate number format (integer or floating point) when we have a large collection of them. This will usually require less space and we will not need to first read a text and then extract the numbers from the text. The advantage of storing numbers as text is that the file can then be viewed in a normal text editor which for example may be useful for debugging.

4.4.3 General information

Many computer programs process information that consists of both numbers and text. Consider for example digital music. For a given song we may store its name, artist, lyrics and all the sound data. The first three items of information are conveniently stored as text. As we shall see later, the sound data is just a very long list of numbers. If the music is in CD-format, the numbers are 16-bit integers, i.e., integers in the interval$[-2^{15}, 2^{15} - 1]$ or $[-32768, 32767]$, and there are 5.3 million numbers for each minute of music. These numbers can be saved in text format which would require five bytes for most numbers. We can reduce

---

[5]For technical reasons integers are in fact usually stored in so-called two's complement.

the storage requirement considerably by saving them in standard binary integer format. This format only requires two bytes for each number so the size of the file would be reduced by a factor of almost 2.5.

4.4.4 Computer programs

Since computers only can interpret sequences of 0s and 1s, computer programs must also be represented in this form at the lowest level. All computers come with an *assembly* or *machine language* which is the level just above the 0s and 1s. Programs written in higher level languages must be translated (compiled) into assembly language before they can be executed. To do regular programming in assembly language is rather tedious and prone to error as many details that happen behind the scenes in high level languages must be programmed in detail. Each command in the assembly language is represented by an appropriate number of bytes, usually four or eight and therefore corresponds to a specific sequence of 0s and 1s.

4.5 A fundamental principle of computing

In this chapter we have seen that by combining bits into bytes, both numbers, text and more general information can be represented, manipulated and stored in a computer. It is important to remember though, that however complex the information, if it is to be processed by a computer it must be encoded into a sequence of 0s and 1s. When we want the computer to do anything with the information it must interpret and assemble the bits in the correct way before it can perform the desired operation. Suppose for example that as part of a programming project you need to temporary store some information in a file, for example a sound file in the simple format outlined in Subsection 4.4.3. When you read the information back from the file it is your responsibility to interpret the information in the correct way. In the sound example this means that you must be able to extract the name of the song, the artist, the lyrics and the sound data from the file. One way to do this is to use a special character, that is not otherwise in use, to indicate the end of one field and the beginning of the next. In the first three fields we can allow text of any length while in the last field only 16 bit integers are allowed. This is a simple example of a *file format*, i.e., a precise description of how information is stored. If your program is going to read information from a file, you need to know the file format to read the information correctly.

In many situations well established conventions will tell you the file format. One type of convention is that filenames often end with a dot and three or more characters that identify the file format. Some examples are .doc (Microsoft

Word), `.html` (web-documents), `.mp3`(mp3-music files), `.jpg` (photos in jpeg-format). If you are going to write a program that will process one of these file formats you have at least two options: You can find a description of the format and write your own functions that read the information from the file, or you can find a software library written by somebody else that has functions for reading the appropriate file format and converting the information to text and numbers that is returned to your program.

Program code is a different type of file format. A programming language has a precise syntax, and specialised programs called *compilers* or *interpreters* translate programs written according to this syntax into low level commands that the computer can execute.

This discussion of file formats illustrates a fundamental principle in computing: A computer must always be told exactly what to do, or equivalently, must know how to interpret the 0s and 1s it encounters.

Fact 4.15 (A principle of computing). *For a computer to function properly it must always be known how it should interpret the 0s and 1s it encounters.*

This principle is absolute, but there are of course many ways to instruct a computer how information should be interpreted. A lot of the interpretation is programmed into the computer via the operating system, programs that are installed on the computer contain code for encoding and decoding information specific to each program, sometimes the user has to tell a given program how to interpret information (for example tell a program the format of a file), sometimes a program can determine the format by looking for special bit sequences (like the endian convention used in a UTF-16 encoded file). And if you write programs yourself you must of course make sure that your program can process the information from a user in an adequate way.

Exercises

4.1 Write the largest and smallest 32 bit integers, represented in two's complement, as hexadecimal numbers.

4.2 This exercise investigates some properties of the two's representation of integers with $n = 4$ bits. Table 4.2 will be useful.

    a) Perform the addition $-3 + 3$ with two's complement.

    b) What is the result of the addition $7 + 1$ in two's complement?

    c) What is the result of the subtraction $-8 - 1$ in two's complement?

4.3 Suppose you have a computer which works in the ternary (base-3) numeral system. Can you devise a three's complement representation with 4 digits, similar to two's complement?

4.4 It is said that when the game of chess was invented, the emperor was so delighted by the game, that he promised the inventor to grant him a wish. The inventor said that he wanted a grain of rice in the first square of the chessboard, two grains in the second square of the chessboard, four grains in the third square, eight in the fourth and so on. The emperor considered this a very modest request, but was it?

How many grains of rice would the inventor get? Translate this into a problem of converting numbers between different bases, and solve it. (Hint: A chessboard is divided into 64 squares).

4.5 Real numbers may be written as decimal numbers, usually with a fractional part, and floating point numbers result when the number of digits is required to be finite. But real numbers can also be viewed as limits of rational numbers which means that any real number can be approximated arbitrarily well be a rational number. An alternative to representing real numbers with floating point numbers is therefore a representation in terms of rational numbers. Discuss advantages and disadvantages with this kind of representation (how do the limitations of finite resources appear, will there be 'rounding errors' etc.).

4.6 Write the following numbers (or approximations of them) in normal form, using both 4 and 8 digits

  a) 4752735

  b) $602214179 * 10^{15}$

  c) 0.00008617343

  d) 9.81

  e) 385252

  f) $e^{10\pi}$

4.7 Redo exercise 6d, but write the number in binary normal form.

4.8 Determine the UTF-8 encodings of the Unicode characters with the following code points:

  a) $5a_{16}$.

  b) $f5_{16}$.

  c) $3f8_{16}$.

  d) $8f37_{16}$.

4.9 Determine the UTF-16 encodings of the Unicode characters in exercise 8.

4.10 In this exercise you may need to use the Unicode table which can be found on the web page `www.unicode.org/charts/`.

  a) Suppose you save the characters 'æ', 'ø' and 'å' in a file with UTF-8 encoding. How will these characters be displayed if you open the file in an editor using the ISO Latin 1 encoding?

  b) What will you see if you do the opposite?

c) Repeat (a) and (b), but use UTF-16 instead of UTF-8.

d) Repeat (a) and (b), but use UTF-16 instead of ISO Latin 1.

4.11 **a)** Write down the hexadecimal representation of your first name in ISO Latin 1, UTF-8, and UTF-16 with BE (If your first name is only three letters or less, do your last name as well, include a blank space between the names).

b) Consider the first 3 bytes in the codes for your name (not counting the specific BE-code for UTF-16) and view each of them as an integer. What decimal numbers do they correspond to? Can these numbers be represented as 32-bit integers?

c) Repeat *b*) for the first 4 bytes of your name (not counting the specific BE-code).

CHAPTER 5

Round–off errors

In the two previous chapters we have seen how numbers can be represented in the binary numeral system and how this is the basis for representing numbers in computers. Since any given integer only has a finite number of digits, we can represent all integers below a certain limit *exactly*. Non-integer numbers are considerably more cumbersome since infinitely many digits are needed to represent most of them, regardless of which numeral system we employ. This means that most non-integer numbers cannot be represented in a computer without committing an error which is usually referred to as *round-off error* or *rounding error*.

The standard computer representation of non-integer numbers is as floating point numbers with a fixed number of bits. Not only is an error usually committed when a number is represented as a floating point number; most calculations with floating point numbers will induce further round-off errors. In most situations these errors will be small, but in a long chain of calculations there is a risk that the errors may accumulate and seriously pollute the final result. It is therefore important to be able to recognise when a given computation is going to be troublesome or not so that we may know whether the result can be trusted.

In this chapter we will start our study of round-off errors. We will see how round-off errors arise and study how it affects evaluation of mathematical functions. For this we need to to introduce a way to measure the error, and we need to understand how arithmetic is performed.

5.1   Measuring the error

Suppose we have a number a and an approximation $\tilde{a}$. We are going to consider two ways to measure the error in such an approximation.

5.1.1 Absolute error

The first error measure is the most obvious, it is essentially the difference between a and $\tilde{a}$.

Definition 5.1 (Absolute error). *Suppose $\tilde{a}$ is an approximation to the number a. The absolute error in this approximation is given by $|a - \tilde{a}|$.*

If $a = 100$ and $\tilde{a} = 100.1$ the absolute error is 0.1, whereas if $a = 1$ and $\tilde{a} = 1.1$ the absolute error is still 0.1. Note that if a is an approximation to $\tilde{a}$, then $\tilde{a}$ is an equally good approximation to a with respect to the absolute error.

5.1.2 Relative error

For some purposes the absolute error may be what we want, but in many cases it is reasonable to say that the error is smaller in the first example above than in the second, since the numbers involved are bigger. The *relative error* takes this into account.

Definition 5.2 (Relative error). *Suppose that $\tilde{a}$ is an approximation to the nonzero number a. The relative error in this approximation is given by*

$$\frac{|a - \tilde{a}|}{|a|}.$$

We note that the relative error is obtained by scaling the absolute error with the size of the number that is approximated. If we compute the relative errors in the approximations above we find that it is 0.001 when $a = 100$ and $\tilde{a} = 100.1$, while when $a = 1$ and $\tilde{a} = 1.1$ the relative error is 0.1. In contrast to the absolute error, we see that the relative error tells us that the approximation in the first case is much better than in the latter case. In fact we will see in a moment that the relative error gives a good measure of the number of digits that a and $\tilde{a}$ have in common.

We use concepts which are closely related to absolute and relative errors in many everyday situations. One example is profit from investments, like bank accounts. Suppose you have invested some money and after one year, your profit is 100 (in your local currency). If your initial investment was 100, this is a very good profit in one year, but if your initial investment was 10 000 it is not so impressive. If we let a denote the initial investment and $\tilde{a}$ the investment after one year, we see that, apart from the sign, the profit of 100 corresponds to the 'absolute error'in the two cases. On the other hand, the relative error corresponds to

Figure 5.1. In the figure to the left the red parts of the line indicate the numbers which have two digits in common with a, while the numbers in the blue part have three digits in common with a. The figure on the right is just a magnification of the figure on the left.

profit measured in %, again apart from the sign. This says much more about how good the investment is since the profit is compared to the initial investment. In the two cases here, we find that the profit in % is $1 = 100\%$ in the first case and $0.01 = 1\%$ in the second.

Another situation where we use relative measures is when we give the concentration of an ingredient within a substance. If for example the fat content in milk is 3.5 % we know that a grams of milk will contain $0.035a$ grams of fat. In this case $\tilde{a}$ denotes how many grams that are not fat. Then the difference $a - \tilde{a}$ is the amount of fat and the equation $a - \tilde{a} = 0.035a$ can be written

$$\frac{a - \tilde{a}}{a} = 0.035$$

which shows the similarity with relative error.

5.1.3 Properties of the relative error

The examples above show that we may think of the relative error as the 'concentration' of the error $|a - \tilde{a}|$ in the total 'volume' $|a|$. However, this interpretation can be carried further and linked to the number of common digits in a and $\tilde{a}$: If the size of the relative error is approximately 10^{-m}, then a and $\tilde{a}$ have roughly m digits in common. If the relative error is r, this corresponds to $-\log_{10} r$ being approximately m.

Observation 5.3 (Relative error and significant digits). *Let a be a nonzero number and suppose that $\tilde{a}$ is an approximation to a with relative error*

$$r = \frac{|a - \tilde{a}|}{|a|} \approx 10^{-m} \tag{5.1}$$

where m is an integer. Then roughly the m most significant decimal digits of a and $\tilde{a}$ agree.

Figure 5.1 shows how numbers with two and three digits in common with a given number are located along the real line.

85

a	$\tilde{a}$	r
12.3	12.1	1.6×10^{-2}
12.8	13.1	2.3×10^{-2}
0.53241	0.53234	1.3×10^{-4}
8.96723	8.96704	2.1×10^{-5}

Table 5.1. Some numbers a with approximations $\tilde{a}$ and corresponding relative errors r.

Sketch of 'proof'. Since a is nonzero, it can be written as $a = \alpha \times 10^n$ where α is a number in the range $1 \le \alpha < 10$ that has the same decimal digits as a, and n is an integer. The approximation $\tilde{a}$ be be written similarly as $\tilde{a} = \tilde{\alpha} \times 10^n$ and the digits of $\tilde{\alpha}$ are exactly those of $\tilde{a}$. Our job is to prove that roughly the first m digits of a and $\tilde{a}$ agree.

Let us insert the alternative expressions for a and $\tilde{a}$ in (5.1). If we cancel the common factor 10^n and multiply by α, we obtain

$$|\alpha - \tilde{\alpha}| \approx \alpha \times 10^{-m}. \tag{5.2}$$

Since α is a number in the interval $[1, 10)$, the right-hand side is roughly a number in the interval $[10^{-m}, 10^{-m+1})$. This means that by subtracting $\tilde{\alpha}$ from α, we cancel out the digit to the left of the decimal point, and $m - 1$ digits to the right of the decimal point. But this means that the m most significant digits of α and $\tilde{\alpha}$ agree. ∎

Some examples of numbers a and $\tilde{a}$ with corresponding relative errors are shown in Table 5.1. In the first case everything works as expected. In the second case the rule only works after a and $\tilde{a}$ have been rounded to two digits. In the third example there are only three common digits even though the relative error is roughly 10^{-4}, although the fourth digit is only off by one unit. Similarly, in the last example, the relative error is approximately 10^{-5}, but a and $\tilde{a}$ only have four digits in common, with the fifth digits differing by two units.

The last two examples illustrate that the link between relative error and significant digits is just a rule of thumb. If we go back to the 'proof' of observation 5.3, we note that as the left-most digit in a (and α) becomes larger, the right-hand side of (5.2) also becomes larger. This means that the number of common digits may become smaller, especially when the relative error approaches 10^{-m+1} as well. In spite of this, observation 5.3 is a convenient rule of thumb.

Observation 5.3 is rather vague when it just assumes that $r \approx 10^{-m}$. The basis for making this more precise is the fact that m is equal to $-\log_{10} r$, rounded to

the nearest integer. This means that m is characterised by the inequalities

$$m - 0.5 < -\log_{10} r \leq m + 0.5.$$

and this is in turn equivalent to

$$r = \rho \times 10^{-m}, \quad \text{where } \frac{1}{\sqrt{10}} < \rho < \sqrt{10}.$$

The absolute error has the nice property that if $\tilde{a}$ is a good approximation to a, then a is an equally good approximation to $\tilde{a}$. The relative error has a similar, but more complicated property.

Lemma 5.4 (Reciprocal relative error). *Suppose that $\tilde{a}$ is an approximation of a with relative error $r < 1$. Then a is an approximation of $\tilde{a}$ with relative error*

$$\frac{|a - \tilde{a}|}{|\tilde{a}|} \leq \frac{r}{1 - r}. \tag{5.3}$$

Proof. We first observe that since

$$1 > r = \frac{|a - \tilde{a}|}{|a|} = \left| 1 - \frac{\tilde{a}}{a} \right|,$$

both a and $\tilde{a}$ must have the same sign, for otherwise the right-hand side in this inequality would be greater than 1. On the other hand we also have from the triangle inequality that

$$1 = \left| 1 - \frac{\tilde{a}}{a} + \frac{\tilde{a}}{a} \right| \leq \left| 1 - \frac{\tilde{a}}{a} \right| + \left| \frac{\tilde{a}}{a} \right| \tag{5.4}$$

or $|1 - \tilde{a}/a| \geq 1 - |\tilde{a}/a|$. We therefore have

$$\frac{1}{1 - r} = \frac{1}{1 - |1 - \tilde{a}/a|} \geq \frac{1}{|\tilde{a}/a|} = \frac{|a|}{|\tilde{a}|}.$$

From this we conclude that

$$\frac{r}{1 - r} \geq \left| 1 - \frac{\tilde{a}}{a} \right| \left| \frac{a}{\tilde{a}} \right| = \left| \frac{a}{\tilde{a}} - 1 \right| = \frac{|a - \tilde{a}|}{|\tilde{a}|}. \quad \blacksquare$$

The condition $r < 1$ is essential in Lemma 5.4. An example with $r = 1$ is the situation where $\tilde{a} = 0$ is an approximation to some arbitrary, nonzero real number a. Since $r = 1$ corresponds to such an abnormal case, the condition $r < 1$ cannot be very restrictive.

It is worth pointing out that the inequality in (5.3) often is equality. In fact, if $\tilde{a} \leq a$, then there is equality in (5.4) and therefore also in (5.3). But even if $a < \tilde{a}$, the inequality in (5.4) is very close to equality for small r. Therefore, as long as $\tilde{a}$ is a reaonable approximation of a, the inequality in (5.3) can be considered an equality. In this situation we also have $1 - r \approx 1$ so the right-hand side of (5.3) is close to r. In other words, if $\tilde{a}$ is a good approximation to a, it does not matter much whether we consider $\tilde{a}$ to be an approximation to a or vice versa, even when we use a relative error measure.

5.2 Errors in integer arithmetic

Integers and integer arithmetic on a computer is simple both to understand and analyse. All integers up to a certain size are represented exactly and arithmetic with integers is exact. The only thing that can go wrong is that the result becomes larger than what is representable by the type of integer used. The computer hardware, which usually represents integers with two's complement (see section 4.1.3), will not protest and just wrap around from the largest positive integer to the smallest negative integer or vice versa.

As was mentioned in chapter 4, different programming languages handle overflow in different ways. Most languages leave everything to the hardware which means that overflow is not reported, even though it leads to completely wrong results. This kind of behaviour is not really problematic since it is usually easy to detect (the results are completely wrong), but it may be difficult to understand what went wrong unless you are familiar with the basics of two's complement. Other languages, like Python, gracefully switch to higher precision. In this case, integer overflow is not serious, but may reduce the computing speed.

Other error situations, like division by zero, or attempts to extract the square root of negative numbers, lead to error messages and are therefore not serious.

The conclusion is that errors in integer arithmetic are not serious, at least not if you know a little bit about how integer arithmetic is performed.

5.3 Errors in floating point arithmetic

Errors in floating point arithmetic are much more subtle than errors in integer arithmetic. In contrast to integers, floating point numbers can be just a little bit wrong. A result that appears to be reasonable may therefore contain errors, and it may be difficult to judge how large the error is.

In this section we will discuss the potential for errors in the four elementary operations, addition, subtraction , multiplication, and division with floating point numbers. Perhaps a bit surprisingly, the conclusion will be that the most

critical operation is addition/subtraction, which in certain situations may lead to dramatic errors.

A word of warning is necessary before we continue. In this chapter, and in particular in this section, where we focus on the shortcomings of computer arithmetic, some may conclude that round-off errors are so dramatic that we had better not use a computer for serious calculations. This is a misunderstanding. The computer is an extremely powerful tool for numerical calculations that you should use whenever you think it may help you, and most of the time it will give you answers that are much more accurate than you need. However, you should be alert and aware of the fact that in certain cases errors may cause considerable problems.

The essential features of round-off errors are present if we use the same model of floating point numbers as in section 4.2. Recall that any real number a may be written as a normalised decimal number

$$\alpha \times 10^n,$$

where the number α is in the range $[0.1, 1)$ and is called the *significand*, while the integer n is called the *exponent*.

Fact 5.5 (Model of floating point numbers). *Most of the shortcomings of floating point arithmetic become visible in a computer model that uses 4 digits for the significand, 1 digit for the exponent plus an optional sign for both the significand and the exponent.*

Two typical normalised (decimal) numbers in this model are 0.4521×10^1 and -0.9×10^{-1}. The examples in this section will use this model, but the results can be generalised to a floating point model in base β, see exercise 3.

On most computers, floating point numbers are represented, and arithmetic performed according to, the IEEE[1] standard. This is a carefully designed suggestion for how floating point numbers should behave, and is aimed at providing good control of round-off errors and prescribing the behaviour when numbers reach the limits of the floating point model. Regardless of the particular details of how the floating point arithmetic is performed, however, the use of floating point numbers inevitably leads to problems, and in this section we will consider some of those.

---

[1] IEEE is an abbreviation for *Institute of Electrical and Electronic Engineers* which is a large professional society for engineers and scientists in the USA. The floating point standard is described in a document called *IEEE standard reference 754*.

One should be aware of the fact the IEEE standard is extensive and complicated, and far from all computers and programming languages support the whole standard. So if you are going to do serious floating point calculations you should check how much of the IEEE standard that is supported in your environment. In particular, you should realise that if you move your calculations from one environment to another, the results may change slightly because of different implementations of the IEEE standard.

5.3.1   Errors in floating point representation

Recall from chapter 3 that most real numbers cannot be represented exactly with a finite number of decimals. In fact, there are only a finite number of floating point numbers in total. This means that very few real numbers can be represented exactly by floating point numbers, and it is useful to have a bound on how much the floating point representation of a real number deviates from the number itself.

Lemma 5.6. *Suppose that a is a nonzero real number within the range of base-β normalised floating point numbers with an m digit significand, and suppose that a is represented by the nearest floating point number $\tilde{a}$. Then the relative error in this approximation is at most*

$$\frac{|a - \tilde{a}|}{|a|} \leq 5\beta^{-m}. \tag{5.5}$$

Proof. For simplicity we first do the proof in the decimal numeral system. We write a as a normalised decimal number,

$$a = \alpha \times 10^n,$$

where α is a number in the range $[0.1, 1)$. The floating point approximation $\tilde{a}$ can also be written as

$$\tilde{a} = \tilde{\alpha} \times 10^n,$$

where $\tilde{\alpha}$ is α rounded to m digits. Suppose for example that $m = 4$ and $\tilde{\alpha} = 0.3218$. Then the absolute value of the significand $|\alpha|$ of the original number must lie in the range $[0.32175, 0.32185)$, so in this case the largest possible distance between α and $\tilde{\alpha}$ is 5 units in the fifth decimal, i.e., the error is bounded by $5 \times 10^{-5} = 0.5 \times 10^{-4}$. A similar argument shows that in the general decimal case we have

$$|\alpha - \tilde{\alpha}| \leq 0.5 \times 10^{-m}.$$

The relative error is therefore bounded by

$$\frac{|\alpha - \tilde{\alpha}|}{|a|} \leq \frac{0.5 \times 10^{-m} \times 10^n}{|\alpha| \times 10^n} = \frac{0.5 \times 10^{-m}}{|\alpha|} \leq 5 \times 10^{-m}$$

where the last inequality follows since $|\alpha| \geq 0.1$.

Although we only considered normalised numbers in bases 2 and 10 in chapter 4, normalised numbers may be generalised to any base. In base β the distance between α and $\tilde{\alpha}$ is at most $0.5\beta\beta^{-m-1} = 0.5\beta^{-m}$, see exercise 3. Since we also have $|\alpha| \geq \beta^{-1}$, an argument completely analogous to the one in the decimal case proves (5.5). ■

Note that lemma 5.6 states what we already know, namely that a and $\tilde{a}$ have roughly $m - 1$ or m digits in common.

Floating point numbers on most computers use binary representation, and we know that in the binary numeral system all fractions a/b where $b = 2^k$ for some positive integer k can be represented exactly, see lemma 3.22. This means that numbers like 1/2, 1/4 and 5/16 are represented exactly. On the other hand, it is easy to forget that numbers like 0.1 and 0.43 are *not* represented exactly.

5.3.2   Floating point errors in addition/subtraction

We are now ready to see how round-off errors affect the arithmetic operations. Since $a - b = a + (-b)$, it is sufficient to consider addition of numbers. The basic procedure for adding floating point numbers is simple (in reality it is more involved than what is stated here).

Algorithm 5.7. *To add two real numbers on a computer, the following steps are performed:*

1. *The number with largest absolute value is written in normalised form*

$$a = \alpha \times 10^n,$$

   and the other number b is written as

$$b = \beta \times 10^n$$

   with the same exponent as a and the same number of digits for the significand β.

2. *The significands α and β are added,*

$$\gamma = \alpha + \beta$$

3. *The result $c = \gamma \times 10^n$ is converted to normalised form.*

This apparently simple algorithm contains a serious pitfall which is most easily seen from some examples. Let us first consider a situation where everything works out nicely.

Example 5.8. Suppose that $a = 5.645$ and $b = 7.821$. We convert the numbers to normal form and obtain

$$a = 0.5645 \times 10^1, \quad b = 0.7821 \times 10^1.$$

We add the two significands $0.5645 + 0.7821 = 1.3466$ so the correct answer is 1.3466×10^1. The last step is to convert this to normal form. In exact arithmetic this would yield the result 0.13466×10^2. However, this is not in normal form since the significand has five digits. We therefore perform rounding $0.13466 \approx 0.1347$ and get the final result

$$0.1347 \times 10^2.$$

The relative error in this result is roughly 3×10^{-4}. ∎

Example 5.8 shows that we easily get an error when normalised numbers are added and the result converted to normalised form with a fixed number of digits for the significand. In this first example all the digits of the result are correct, so the error is far from serious.

Example 5.9. If $a = 42.34$ and $b = 0.0033$ we convert the largest number to normal form

$$42.34 = 0.4234 \times 10^2.$$

The smaller number b is then written in the same form (same exponent)

$$0.0033 = 0.000033 \times 10^2.$$

The significand in this second number must be rounded to four digits, and the result of this is 0.0000. The addition therefore becomes

$$0.4234 \times 10^2 + 0.0000 \times 10^2 = 0.4234 \times 10^2.$$

Here the relative error is approximately 8×10^{-5}. ∎

The error in example 5.9 may seem serious, but once again the result is correct to four decimal digits, which is the best we can hope for when we only have this number of digits available.

Although there have been errors in the two previous examples, it has not been dramatic. This is reflected in the relative errors which are close to 10^{-4}. This confirms that there should be about 4 correct digits in the results according to observation 5.3.

Example 5.10. Consider next a case where $a = 10.34$ and $b = -10.27$ have opposite signs. We first rewrite the numbers in normal form

$$a = 0.1034 \times 10^2, \quad b = -0.1027 \times 10^2.$$

We then add the significands, which really amounts to a subtraction,

$$0.1034 - 0.1027 = 0.0007.$$

Finally, we write the number in normal form and obtain

$$a + b = 0.0007 = 0.7000 \times 10^{-4}. \tag{5.6}$$

In this case the relative error is 0. ∎

Example 5.10 may seem innocent since the result is exact, but in fact it contains the seed for serious problems. A similar example will reveal what may easily happen.

Example 5.11. Suppose that $a = 10/7$ and $b = -1.42$. Conversion to normal form yields

$$\frac{10}{7} \approx a = 0.1429 \times 10^1, \quad b = -0.142 \times 10^1.$$

Adding the significands yield

$$0.1429 - 0.142 = 0.0009.$$

When this is converted to normal form, the result is

$$0.9000 \times 10^{-3}$$

while the true result rounded to four correct digits is

$$0.8571 \times 10^{-3}. \tag{5.7}$$

The relative error is roughly 10^{-1}. ∎

In example 5.11 there is a serious problem: We started with two numbers with full four digit accuracy, but the computed result had only one correct digit. In other words, we lost almost all accuracy when the subtraction was performed. The potential for this loss of accuracy was present in example 5.10 where we also had to add digits in the last step (5.6), but there we were lucky in that the added digits were correct. In example 5.11 however, there would be no way for a computer to know that the additional digits to be added in (5.7) should be

taken from the decimal expansion of 10/7. Note how the bad loss of accuracy in example 5.11 is reflected in the relative error.

One could hope that what happened in example 5.11 is exceptional; after all example 5.10 worked out very well. This is not the case. It is example 5.10 that is exceptional since we happened to add the correct digits to the result in (5.6). This was because the numbers involved could be represented exactly in our decimal model. In example 5.11 one of the numbers was 10/7 which cannot be represented exactly, and this leads to problems.

Our observations can be summed up as follows.

Observation 5.12. *Suppose the k most significant digits in the two numbers a and b are the same. Then k digits may be lost when the subtraction $a - b$ is performed with algorithm 5.7.*

This observation is very simple: If we start out with m correct digits in both a and b, and the k most significant of those are equal, they will cancel in the subtraction. When the result is normalised, the missing k digits will be filled in from the right. These new digits are almost certainly wrong and hence the computed result will only have $m - k$ correct digits. This effect is called *cancellation* and is a very common source of major round-off problems. The moral is: *Be on guard when two almost equal numbers are subtracted.*

In practice, a computer works in the binary numeral system. However, the cancellation described in observation 5.12 is just as problematic when the numbers are represented as normalised binary numbers.

Example 5.9 illustrates another effect that may cause problems in certain situations. If we add a very small number ϵ to a nonzero number a the result may become exactly a. This means that a test like

$$\textbf{if } a = a + \epsilon$$

may in fact become true.

Observation 5.13. *Suppose that the floating point model in base β uses m digits for the significand and that a is a nonzero floating point number. The addition $a + \epsilon$ will be computed as a if*

$$|\epsilon| < 0.5\beta^{-m}|a|. \tag{5.8}$$

The exact factor to be used on the right in (5.8) depends on the details of the arithmetic. The factor $0.5\beta^{-m}$ used here corresponds to rounding to the nearest m digits.

A general observation to made from the examples is simply that there are almost always small errors in floating point computations. This means that if you have computed two different floating point numbers a and $\tilde{a}$ that from a strict mathematical point of view should be equal, it is very risky to use a test like

$$\textbf{if } a = \tilde{a}$$

since it is rather unlikely that this will be true. One situation where this problem may occur is in a for loop like

$$\textbf{for } x = 0.0, 0.1, \ldots, 1.0$$

What happens behind the scenes is that 0.1 is added to x each time we pass through the loop, and the loop stops when x becomes larger than 1.0. The last time the result may become $1 + \epsilon$ rather than 1, where ϵ is some small positive number, and hence the last time through the loop with $x = 1$ may never happen.

In fact, for many floating point numbers a and b, even the two computations $a + b$ and $b + a$ give different results!

5.3.3 Floating point errors in multiplication and division

We usually think of multiplication and division as being more complicated than addition, so one may suspect that these operations may also lead to larger errors for floating point numbers. This is not the case.

Example 5.14. Consider the two numbers $a = 23.57$ and $b = -6.759$ which in normalised form are

$$a = 0.2357 \times 10^2, \quad b = -0.6759 \times 10^1.$$

To multiply the numbers, we multiply the significands and add the exponents, before we normalise the result at the end, if necessary. In our example we obtain

$$a \times b = -0.15930963 \times 10^3.$$

The significand in the result must be rounded to four digits and yields the floating point number

$$-0.1593 \times 10^3,$$

i.e., the number -159.3. ∎

Let us also consider an example of division.

Example 5.15. We use the same numbers as in example 5.14, but now we perform the division a/b. We have

$$\frac{a}{b} = \frac{0.2357 \times 10^2}{-0.6759 \times 10^1} = \frac{0.2357}{-0.6759} \times 10^1,$$

i.e., we divide the significands and subtract the exponents. The division yields $0.2357/-0.6759 \approx -0.3487202$. We round this to four digits and obtain the result

$$\frac{a}{b} \approx -0.3487 \times 10^1 = -3.487. \quad \blacksquare$$

The most serious problem with floating point arithmetic is loss of correct digits. In multiplication and division this cannot happen, so these operations are quite safe.

Observation 5.16. *Floating point multiplication and division do not lead to loss of correct digits as long as the the numbers are within the range of the floating point model. In the worst case, the last digit in the result may be one digit wrong.*

The essence of observation 5.16 is that the above examples are representative of what happens. However, there are some other potential problems to be aware of.

First of all observation 5.16 only says something about *one* multiplication or division. When many such operations are stringed together with the output of one being fed to the next, the errors may accumulate and become large even when they are very small at each step. We will see examples of this in later chapters.

In observation 5.16 there is one assumption, namely that the numbers are within the range of the floating point model. This applies to each of the operands (the two numbers to be multiplied or divided) and the result. When the numbers approach the limits of our floating point model, things will inevitably go wrong.

Underflow. *Underflow* occurs when a positive result becomes smaller than the smallest representable, positive number; the result is then set to 0. This also happens with negative numbers with small magnitude. In most environments you will get a warning, but otherwise the calculations will continue. This will usually not be a problem, but you should be aware of what happens.

Overflow. When the magnitude of a result becomes too large for the floating point standard, *overflow* occurs. This is indicated by the result becoming `infinity` or possibly `positive infinity` or `negative infinity`. There is a special combination of bits in the IEEE standard for these infinity values. When you see `infinity` appearing in your calculations, you should be aware; the chances are high that the reason is some programming error, or even worse, an error in your algorithm. An operation like $a/0.0$ will yield `infinity` when a is a nonzero number.

Undefined operations. The division $0.0/0.0$ is undefined in mathematics, and in the IEEE standard this will give the result `NaN` (not a number). Other operations that may produce `NaN` are square roots of negative numbers, inverse sine or cosine of a number larger than 1, the logarithm of a negative number, etc. (unless you are working with complex numbers). `NaN` is infectious in the sense that any arithmetic operation that combines `NaN` with a normal number always yields `NaN`. For example, the result of the operation $1 + \mathtt{NaN}$ will be `NaN`.

5.4   Rewriting formulas to avoid rounding errors

Certain formulas lead to large rounding errors when they are evaluated with floating point numbers. In some cases though, the number can be computed with an alternative formula that is not problematic. In this section we will consider some examples of this.

Example 5.17. Suppose we are going to evaluate the expression

$$\frac{1}{\sqrt{x^2+1}-x} \tag{5.9}$$

for a large number like $x = 10^8$. The problem here is the fact that x and $\sqrt{x^2+1}$ are almost equal when x is large,

$$x = 10^8 = 100000000,$$
$$\sqrt{x^2+1} \approx 100000000.000000005.$$

Even with 64-bit floating point numbers the square root will therefore be computed as 10^8, so the denominator in (5.9) will be computed as 0, and we get division by 0. This is a consequence of floating point arithmetic since the two terms in the denominator are not really equal. A simple trick helps us out of this problem. Note that

$$\frac{1}{\sqrt{x^2+1}-x} = \frac{(\sqrt{x^2+1}+x)}{(\sqrt{x^2+1}-x)(\sqrt{x^2+1}+x)} = \frac{\sqrt{x^2+1}+x}{x^2+1-x^2} = \sqrt{x^2+1}+x.$$

This alternative expression can be evaluated for large values of x without any problems with cancellation. The result for $x = 10^8$ is

$$\sqrt{x^2 + 1} + x \approx 200000000$$

where all the digits are correct. ∎

The fact that we were able to find an alternative formula in example 5.17 may seem very special, but it it is not unusual. Here is another example.

Example 5.18. For most values of x, the expression

$$\frac{1}{\cos^2 x - \sin^2 x} \qquad (5.10)$$

can be computed without problems. However, for $x = \pi/4$ the denominator is 0, so we get division by 0 since $\cos x$ and $\sin x$ are equal for this value of x. This means that when x is close to $\pi/4$ we will get cancellation. This can be avoided by noting that $\cos^2 x - \sin^2 x = \cos 2x$, so the expression (5.10) is equivalent to

$$\frac{1}{\cos 2x}.$$

This can be evaluated without problems for all values of x for which the denominator is nonzero, as long as we do not get overflow. ∎

5.5 Summary

The main content of this chapter is a discussion of basic errors in computer arithmetic. For integers, the errors are dramatic when they occur and therefore quite easy to detect. For floating point numbers, the error is much more subtle since it is usually not dramatic. This means that it is important to understand and recognise situations and computations that may lead to significant round-off errors. The most dangerous operation is subtraction of two almost equal numbers since this leads to cancellation of digits. We may also get large round-off errors from a long sequence of operations with small round-off errors in step; we will see examples of this in later chapters.

Exercises

5.1 Suppose that $\tilde{a}$ is an approximation to a in the problems below, calculate the absolute and relative errors in each case, and check that the relative error estimates the number of correct digits as predicted by observation 5.3.

    a) $a = 1$, $\tilde{a} = 0.9994$.

b) $a = 24$, $\tilde{a} = 23.56$.

c) $a = -1267$, $\tilde{a} = -1267.345$.

d) $a = 124$, $\tilde{a} = 7$.

5.2 Use the floating point model defined in this chapter with 4 digits for the significand and 1 digit for the exponent, and use algorithm 5.7 to do the calculations below in this model.

    a) $12.24 + 4.23$.

    b) $9.834 + 2.45$.

    c) $2.314 - 2.273$.

    d) $23.45 - 23.39$.

    e) $1 + x - e^x$ for $x = 0.1$.

5.3   **a)** Formulate a model for floating point numbers in base β.

    b) Generalise the rule for rounding of fractional numbers in the decimal system to the octal system and the hexadecimal system.

    c) Formulate the rounding rule for fractional numbers in the base-β numeral system.

5.4 From the text it is clear that on a computer, the addition $1.0 + \epsilon$ will give the result 1.0 if ϵ is sufficiently small. Write a computer program which can help you to determine the smallest integer n such that $1.0 + 2^{-n}$ gives the result 1.0.

5.5 Consider the simple algorithm

$x := 0.0$;
while $x \leq 2.0$
    print x
    $x := x + 0.1$;

What values of x will be printed?

Implement the algorithm in a program and check that the correct values are printed. If this is not the case, explain what happened.

5.6   **a)** A fundamental property of real numbers is given by the distributive law

$$(x + y)z = xz + yz. \tag{5.11}$$

In this problem you are going to check whether floating point numbers obey this law. To do this you are going to write a program that runs through a loop 10 000 times and each time draws three random numbers in the interval $(0, 1)$ and then checks whether the law holds (whether the two sides of (5.11) are equal) for these numbers. Count how many times the law fails, and at the end, print the percentage of times that it failed. Print also a set of three numbers for which the law failed.

    b) Repeat (a), but test the associative law $(x + y) + z = x + (y + z)$ instead.

    c) Repeat (a), but test the commutative law $x + y = y + x$ instead.

    d) Repeat (a) and (b), but test the associative and commutative laws for multiplication instead.

5.7 The binomial coefficient $\binom{n}{i}$ is defined as

$$\binom{n}{i} = \frac{n!}{i!\,(n-i)!} \tag{5.12}$$

where $n \geq 0$ is an integer and i is an integer in the interval $0 \leq i \leq n$. The binomial coefficients turn up in a number of different contexts and must often be calculated on a computer. Since all binomial coefficients are integers (this means that the division in (5.12) can never give a remainder), it is reasonable to use integer variables in such computations. For small values of n and i this works well, but for larger values we may run into problems because the numerator and denominator in (5.12) may become larger than the largest permissible integer, even if the binomial coefficient itself may be relatively small. In many languages this will cause overflow, but even in a language like Python, which on overflow converts to a format in which the size is only limited by the resources available, the performance is reduced considerably. By using floating point numbers we may be able to handle larger numbers, but again we may encounter too big numbers during the computations even if the final result is not so big.

An unfortunate feature of the formula (5.12) is that even if the binomial coefficient is small, the numerator and denominator may both be large. In general, this is bad for numerical computations and should be avoided, if possible. If we consider the formula (5.12) in some more detail, we notice that many of the numbers cancel out,

$$\binom{n}{i} = \frac{1 \cdot 2 \cdots i \cdot (i+1) \cdots n}{1 \cdot 2 \cdots i \cdot 1 \cdot 2 \cdots (n-i)} = \frac{i+1}{1} \cdot \frac{i+2}{2} \cdots \frac{n}{n-i}.$$

Employing the product notation we can therefore write $\binom{n}{i}$ as

$$\binom{n}{i} = \prod_{j=1}^{n-i} \frac{i+j}{j}.$$

a) Write a program for computing binomial coefficients based on this formula, and test your method on the coefficients

$$\binom{9998}{4} = 416083629102505,$$

$$\binom{100000}{70} = 8.14900007813826 \cdot 10^{249},$$

$$\binom{1000}{500} = 2.702882409454366 \cdot 10^{299}.$$

Why do you have to use floating point numbers and what results do you get?

b) Is it possible to encounter too large numbers during those computations if the binomial coefficient to be computed is smaller than the largest floating point number that can be represented by your computer?

c) In our derivation we cancelled $i!$ against $n!$ in (5.12), and thereby obtained the alternative expression for $\binom{n}{i}$. Another method can be derived by cancelling $(n-i)!$ against $n!$ instead. Derive this alternative method in the same way as above, and discuss when the two methods should be used (you do not need to program the other method; argue mathematically).

100

5.8 Identify values of x for which the formulas below may lead to large round-off errors, and suggest alternative formulas which do not have these problems.

    a) $\sqrt{x+1} - \sqrt{x}$.

    b) $\ln x^2 - \ln(x^2 + x)$.

    c) $\cos^2 x - \sin^2 x$.

5.9 Suppose you are going to write a program for computing the real roots of the quadratic equation $ax^2 + bx + c = 0$, where the coefficients a, b and c are real numbers. The traditional formulas for the two solutions are

$$x_1 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}, \quad x_2 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}.$$

Identify situations (values of the coefficients) where the formulas do not make sense or lead to large round-off errors, and suggest alternative formulas for these cases which avoid the problems.

Part II

Sequences of Numbers

CHAPTER 6

Difference Equations
and Round-off Errors

An important ingredient in school mathematics is solution of algebraic equations like $x + 3 = 4$. The challenge is to determine a numerical value for x such that the equation holds. In this chapter we are going to give a brief review of *difference equations* or *recurrence relations*. In contrast to traditional equations, the unknown in a difference equation is not a single number, but a sequence of numbers.

For some simple difference equations, an explicit formula for the solution can be found with pencil-and-paper methods, and we will review some of these methods in section 6.4. For most difference equations, there are no explicit solutions. However, a large group of equations can be solved numerically on a computer, and in section 6.3 we will see how this can be done.

In chapter 5 we saw real numbers are approximated by floating-point numbers, and how the limitations inherent in floating-point numbers sometimes may cause dramatic errors. In section 6.5 we will see how round-off errors affect the numerical solutions of difference equations.

6.1 Why equations?

The reason equations are so useful is that they allow us to characterise unknown quantites in terms of natural principles that may be formulated as equations. Once an equation has been written down, we can apply standard techniques for solving the equation and determining the unknown. To illustrate, let us consider a simple example.

A common type of puzzle goes like this: *Suppose a man has a son that is half*

his age, and the son will be 16 years younger than his father in 5 years time. How
old are they?

With equations we do not worry about the ages, but rather write down what we know. If the age of the father is x and the age of the son is y, the first piece of information can be expressed as $y = x/2$, and the second as $y = x - 16$. This has given us two equations in the two unknowns x and y,

$$y = x/2,$$
$$y = x - 16.$$

Once we have the equations we use standard techniques to solve them. In this case, we find that $x = 32$ and $y = 16$. This means that the father is 32 years old, and the son 16.

Difference equations work in the same way, except that they allow us to model phenomena where the unknown is a sequence of values, like the annual growth of money in a bank account, or the size of a population of animals over a period of time. The difference equation is obtained from known principles, and then the equation is solved by a mathematical or numerical method.

6.2   Difference equations defined

A simple difference equation arises if we try to model the growth of money in a bank account. Suppose that the amount of money in the account after n years is x_n, and the interest rate is 5 % per year. If interest is added once a year, the amount of money after $n + 1$ years is given by the difference equation

$$x_{n+1} = x_n + 0.05x_n = 1.05x_n. \tag{6.1}$$

This equation characterises the growth of all bank accounts with a 5 % interest rate — in order to characterise a specific account we need to know how much money there was in the account to start with. If the initial deposit was 100000 (in your favourite currency) at time $n = 0$, we have an *initial condition* $x_0 = 100000$. This gives the complete model

$$x_{n+1} = 1.05x_n, \quad x_0 = 100000. \tag{6.2}$$

This is an example of a *first-order* difference equation with an initial condition. Suppose that we withdraw 1000 from the account every year. If we include this in our model we obtain the equation

$$x_{n+1} = 1.05x_n - 1000. \tag{6.3}$$
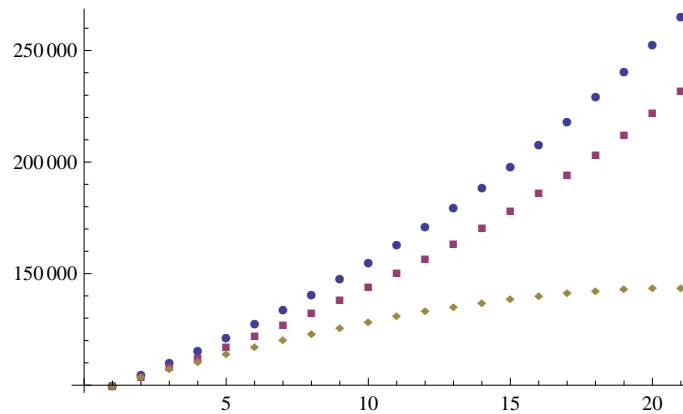
Figure 6.1. The growth of capital according to the models (6.1) (largest growth), (6.3) (middle growth), and (6.4) (smallest growth).

As the capital accumulates, it is reasonable that the owner increases the withdrawals. If for example the amount withdrawn increases by 300 each year, we get the model

$$x_{n+1} = 1.05x_n - (1000 + 300n). \tag{6.4}$$

A plot of the development of the capital in the three different cases is shown in figure 6.1. Note that in the case of (6.4) it appears that the growth of the capital levels out. In fact, it can be shown that after about 45 years, all the capital will be lost.

After these simple examples, let us define difference equations in general.

> **Definition 6.1** (Difference equation)**.** *A difference equation or recurrence relation is an equation that involves the terms of an unknown sequence (x_n). The equation is said to be of order k if a term in the sequence depends on k previous terms, as in*
>
> $$x_{n+k} = f(n, x_n, x_{n+1}, \ldots, x_{n+k-1}), \tag{6.5}$$
>
> *where f is a function of $k + 1$ variables. The actual values of n for which* (6.5) *should hold may vary, but would typically be all nonzero integers.*

It is instructive to see how the three examples above fit into the general setting of definition 6.1. In all three cases we have $k = 1$; in the case (6.1) we have $f(t, x) = 0.5x$, in (6.3) we see that $f(t, x) = 0.5x - 1000$, and in (6.4) we have $f(t, x) = 1.05x - (1000 + 300t)$.

107

The examples above all led to a simple first-order difference equation. Here is an example where we end up with an equation of higher order.

Example 6.2. An illness spreads by direct contact between individuals. Each day a person with the illness infects one new person, and the infected person becomes ill after three days. This means that on day n, the total number of ill persons are the people who were ill yesterday, plus the number of people who were infected three days ago. But this latter number equals the number of people that were ill three days ago. If x_n denotes the number of ill people on day n, we therefore have

$$x_n = x_{n-1} + x_{n-3}, \quad n = 3, 4, \ldots,$$

or

$$x_{n+3} = x_{n+2} + x_n, \quad n = 0, 1, \ldots$$

We obtain a difference equation of order k if we assume that the incubation time is k days. By reasoning in the same way we then find

$$x_{n+k} = x_{n+k-1} + x_n, \quad n = 0, 1, \ldots \tag{6.6}$$

Note that in the case $k = 2$ we get the famous Fibonacci model. ∎

Difference equations are particularly nice from a computational point of view since we have an explicit formula for a term in the sequence in terms of previous terms. In the bank example above, next year's balance is given explicitly in terms of this year's balance in formulas (6.1), (6.3), and (6.4). For general equations, we can compute x_{n+k} from the k previous terms in the sequence, as in (6.5). In order for this to work, we must be able to start somewhere, i.e., we need to know k consecutive terms in the sequence. It is common to assume that these terms are $x_0, \ldots, x_{k-1}$, but they could really be any k consecutive terms.

Observation 6.3 (Initial conditions)**.** *For a difference equation of order k, the solution is uniquely determined if k consecutive values of the solution is specified. These* initial conditions *are usually given as*

$$x_0 = a_0, \quad x_1 = a_1, \quad \ldots \quad x_k = a_k,$$

where $a_0, \ldots, a_k$ are given numbers.

Note that the number of initial conditions required equals the order of the equation. The model for population growth (6.6) therefore requires k initial conditions. A natural way to choose the initial conditions in this model is to set

$$x_0 = \cdots = x_k = 1. \tag{6.7}$$
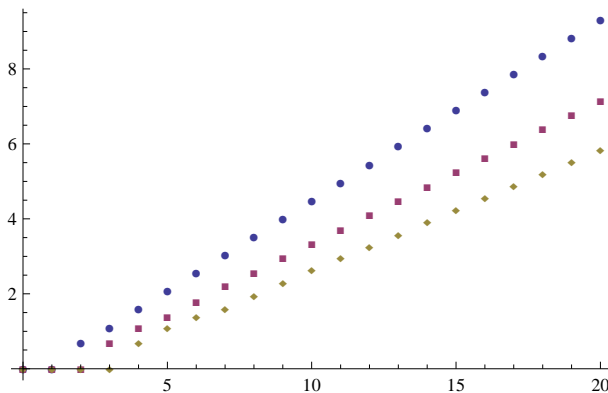
Figure 6.2. Plots of the natural logarithm of the solution of the difference equation (6.6) for $k = 2$ (fastest growth), 3 (medium growth), and 4 (slowest growth).

This corresponds to starting with a population of one new-born pair which remains the only one until this pair gives birth to another pair after k months. Plots of the logarithm of the solutions for $k = 2$, 3 and 4 are shown in figure 6.2. Since the logarithm is a straight line,

$$\ln x_n = a + bn,$$

where a and b are constants that can be found from the graphs, we have

$$x_n = e^{a+bn} = Ce^{bn},$$

with $C = e^a$. In other words, the solutions of (6.6), with initial values given by (6.7), grow exponentially.

We end this section by introducing some more terminology.

Definition 6.4. *A kth-order difference equation is said to be* linear *and* inhomogenous *if it has the form*

$$x_{n+k} = g(n) + f_0(n)x_n + f_1(n)x_{n+1} + \cdots + f_{k-1}(n)x_{n+k-1},$$

where g and f_0, ..., f_{k-1} are functions of n. It is said to have constant coefficients *if the functions f_0 ..., f_{k-1} do not depend on n. It is said to be* homogenous *if $g(n) = 0$ for all n.*

From this definition we see that the all the difference equations we have encountered so far have been linear, with constant coefficients. The equations (6.3) and (6.4) are inhomogenous, the others are homogenous.

6.3 Simulating difference equations

We mentioned above that difference equations are well suited for computations. More specifically, it is easy to calculate the solution numerically since we have an explicit formula for the next term in the sequence. Let us start by doing this for second-order linear equations. This is an equation in the form

$$x_{n+2} = g(n) + f_0(n)x_n + f_1(n)x_{n+1}, \quad x_0 = a_0, \, x_1 = a_1 \tag{6.8}$$

where g, f_0 and f_1 are given functions of n, and a_0 and a_1 are given real numbers. The following is an algorithm for computing the first terms of the solution $\{x_n\}$.

Algorithm 6.5. *Suppose the second-order equation* (6.8) *is given, i.e., the functions* g, f_0, *and* f_1 *are given together with the initial values* a_0 *and* a_1. *The following algorithm will compute the first* $N + 1$ *terms* x_0, x_1, ..., x_N *of the solution:*

    $x_0 := a_0;$
    $x_1 := a_1;$
    for $i := 2, 3, \dots, N$
        $x_i := g(i-2) + f_0(i-2)x_{i-2} + f_1(i-2)x_{i-1};$

This algorithm computes all the $N + 1$ terms and saves them in the array $\boldsymbol{x} = [x_0, \dots, x_N]$. Sometimes we are only interested in the last term x_N, or we just want to print out the terms as they are computed — then there is no need to store all the terms.

Algorithm 6.6. *The following algorithm computes the solution of* (6.8), *just like algorithm 6.5, but prints each term instead of storing them:*

    $x_{pp} := a_0;$
    $x_p := a_1;$
    for $i := 2, 3, \dots, N$
        $x := g(i-2) + f_0(i-2)x_{pp} + f_1(i-2)x_p;$
        print $x;$
        $x_{pp} := x_p;$
        $x_p := x;$

The algorithm is based on the simple fact that in order to compute the next term, we only need to know the two previous terms, since the equation is of second order. At time i, the previous term x_{i-1} is stored in x_p and the term x_{i-2}

is stored in x_{pp}. Once x_i has been computed, we must prepare for the next step and make sure that x_p is shifted down to x_{pp}, which is not needed anymore, and x is stored in x_p. Note that it is important that these assignments are performed in the right order. At the beginning, the values of x_p and x_{pp} are given by the initial values.

In both of these algorithms it is assumed that the coefficients given by the functions g, f_0 and f_1, as well as the initial values a_0 and a_1, are known. In practice, the coefficient functions would usually be entered as functions (or methods) in the programming language you are using, while the initial values could be read from the terminal or via a graphical user interface.

Algorithms (6.5) and (6.6) can easily be generalised to 3rd or 4th order, or equations of any fixed order. The most convenient is to have an algorithm which takes the order of the equation as input.

Algorithm 6.7. *The following algorithm computes and prints the first $N + 1$ terms of the solution of the kth-order difference equation*

$$x_{n+k} = f(n, x_n, x_{n+1}, \ldots, x_{n+k-1}), \quad n = 0, 1, \ldots, N - k \qquad (6.9)$$

with initial values $x_0 = a_0$, $x_1 = a_1$, $\ldots$, $x_{k-1} = a_{k-1}$. Here f is a given function of $k + 1$ variables, and a_0, $\ldots$, a_{k-1} are given real numbers:

> **for** $i := 0, 1, \ldots, k - 1$
> > $z_i := a_i$;
> > **print** z_i;
>
> **for** $i := k, k + 1, \ldots, N$
> > $x := f(i - k, z_0, \ldots, z_{k-1})$;
> > **print** x;
> > **for** $j := 0, \ldots, k - 2$
> > > $z_i := z_{i+1}$;
> >
> > $z_{k-1} := x$;

Algorithm 6.7 is similar to algorithm 6.6 in that it does not store all the terms of the solution sequence. To compensate it keeps track of the k previous terms in the array $\mathbf{z} = [z_0, \ldots, z_{k-1}]$. The values x_k, x_{k+1}, $\ldots$, x_N are computed in the second **for**-loop. By comparison with (6.9) we observe that $i = n + k$; this explains $i - k = n$ as the first argument to f. The initial values are clearly correct the first time through the loop, and at the end of the loop they are shifted along so that the value in z_0 is lost and the new value x is stored in z_{k-1}.

Difference equations have the nice feature that a term in the unknown se-

quence is defined explicitly as a function of previous values. This is what makes it so simple to generate the values by algorithms like the ones sketched here. Provided the algorithms are correct and all operations are performed without errors, the exact solution will be computed. When the algorithms are implemented on a computer, this is the case if all the initial values are integers and all computations can be performed without introducing fractional numbers. One example is the Fibonacci equation

$$x_{n+2} = x_n + x_{n+1}, \quad x_0 = 1, \, x_1 = 1.$$

However, if floating-point numbers are needed for the computations, round-off errors are bound to occur and it is important to understand how this affects the computed solution. This is quite difficult to analyse in general, so we will restrict our attention to linear equations with constant coefficients. First we need to review the basic theory of linear difference equations.

6.4   Review of the theory for linear equations

Linear difference equations with constant coefficients have the form

$$x_{n+k} + b_{k-1} x_{n+k-1} + \cdots + b_1 x_{n+1} + b_0 x_n = g(n)$$

where $b_0, \ldots, b_{k-1}$ are real numbers and $g(n)$ is a function of one variable. Initially we will focus on first-order ($k = 1$) and second-order ($k = 2$) equations for which $g(n) = 0$ for all n (homogenous equations).

One may wonder why we need to analyse these equations in detail when we can simulate them so easily numerically. One reason is that for degree 1 and 2, the analytic solutions are so simple that they have practical importance. For our purposes however, it is equally important that the theory to be presented below will help us to understand how round-off errors influence the results of numerical simulations — this is the main topic in section 6.5.

6.4.1   First-order homogenous equations

The general first-order linear equation with constant coefficients has the form

$$x_{n+1} = bx_n, \tag{6.10}$$

where b is some real number. Often we are interested in x_n for all $n \geq 0$, but any value of $n \in \mathbb{Z}$ makes sense in the following. From (6.10) we find

$$x_{n+1} = bx_n = b^2 x_{n-1} = b^3 x_{n-2} = \cdots = b^{n+1} x_0. \tag{6.11}$$

This is the content of the first lemma.

Lemma 6.8. *The first-order homogenous difference equation*

$$x_{n+1} = bx_n, \quad n \in \mathbb{Z},$$

where b is an arbitrary real number, has the general solution

$$x_n = b^n x_0, \quad n \in \mathbb{Z}.$$

If x_0 is specified, the solution is uniquely determined.

The fact that the solution also works for negative values of n follows just like in (6.11) if we rewrite the equation as $x_n = b^{-1} x_{n+1}$ (assuming $b \neq 0$).

We are primarily interested in the case where $n \geq 0$, and then we have the following simple corollary.

Corollary 6.9. *For $n \geq 0$, the solution of the difference equation $x_{n+1} = bx_n$ will behave according to one of the following three cases:*

$$\lim_{n \to \infty} |x_n| = \begin{cases} 0, & \text{if } |b| < 1; \\ \infty, & \text{if } |b| > 1; \\ |x_0|, & \text{if } |b| = 1. \end{cases}$$

Phrased differently, the solution of the difference equation will either tend to 0 or ∞, except in the case where $|b| = 1$.

6.4.2   Second-order homogenous equations

The general second-order homogenous equation is

$$x_{n+2} + b_1 x_{n+1} + b_0 x_n = 0. \tag{6.12}$$

The basic idea behind solving this equation is to try with a solution $x_n = r^n$ in the same form as the solution of first-order equations, and see if there are any values of r for which this works. If we insert $x_n = r^n$ in (6.12) we obtain

$$0 = x_{n+2} + b_1 x_{n+1} + b_0 x_n = r^{n+2} + b_1 r^{n+1} + b_0 r^n = r^n (r^2 + b_1 r + b_0).$$

In other words, we must either have $r = 0$, which is uninteresting, or r must be a solution of the quadratic equation

$$r^2 + b_1 r + b_0 = 0$$

which is called the *characteristic equation* of the difference equation. If the characteristic equation has the two solutions r_1 and r_2, we know that both $y_n = r_1^n$ and $z_n = r_2^n$ will be solutions of (6.12). And since the equation is linear, it can be shown that any combination

$$x_n = Cr_1^n + Dr_2^n$$

is also a solution of (6.12) for any choice of the numbers C and D. However, in the case that $r_1 = r_2$ this does not give the complete solution, and if the two solutions are complex conjugates of each other, the solution may be expressed in a more adequate form that does not involve complex numbers. In either case, the two free coefficients can be adapted to two initial conditions like $x_0 = a_0$ and $x_1 = a_1$.

Theorem 6.10. *The solution of the homogenous, second-order difference equation*

$$x_{n+2} + b_1 x_{n+1} + b_0 x_n = 0 \tag{6.13}$$

is governed by the solutions r_1 and r_2 of the characteristic equation

$$r^2 + b_1 r + b_0 = 0$$

as follows:

1. *If the two roots r_1 and r_2 are real and distinct, the general solution of (6.13) is given by*
$$x_n = Cr_1^n + Dr_2^n.$$

2. *If the two roots are equal, $r_1 = r_2$, the general solution of (6.13) is given by*
$$x_n = (C + Dn)r_1^n.$$

3. *If the two roots are complex conjugates of each other so that $r_1 = r$ and $r_2 = \bar{r}$, and r can be written in polar form as $r = \rho e^{i\theta}$, then the general solution of (6.13) is given by*

$$x_n = \rho^n (C \cos n\theta + D \sin n\theta).$$

In all three cases the solution can be determined uniquely by two initial conditions $x_0 = a_0$ and $x_1 = a_1$, where a_0 and a_1 are given real numbers, since this determines the two free coefficients C and D uniquely.

The proof of this theorem is not so complicated and can be found in a text on difference equations. The following is a consequence of the theorem which is analogous to corollary 6.9.

Corollary 6.11. *Suppose that one root, say r_1, of the characteristic equation satisfies $|r_1| > 1$, that $C \neq 0$, and that $|r_2| < |r_1|$. Then*

$$\lim_{n \to \infty} |x_n| = \infty.$$

On the other hand, if both $|r_1| < 1$ and $|r_2| < 1$, then

$$\lim_{n \to \infty} x_n = 0.$$

Note that in cases 2 and 3 in theorem 6.10, the two roots have the same absolute value (in case 2 the roots are equal and in case 3 they both have absolute value ρ). In those cases both roots will therefore either be larger than 1 in absolute value or smaller than 1 in absolute value. This means that it is only in the first case that we need to distinguish between the two roots in the conditions in corollary 6.11.

Proof of corollary 6.11. In cases 2 and 3 in theorem 6.10 $|r_1| = |r_2|$, so if $|r_1| > 1$ and $|r_2| < |r_1|$ we must have two real roots. Then we can write the solution as

$$x_n = r_1^n \left(C + D \left(\frac{r_2}{r_1} \right)^n \right)$$

and therefore

$$\lim_{n \to \infty} |x_n| = \lim_{n \to \infty} |r_1|^n \left| C + D \left(\frac{r_2}{r_1} \right)^n \right| = |C| \lim_{n \to \infty} |r_1| = \infty.$$

If both $|r_1| < 1$ and $|r_2| < 1$ and both roots are real, the triangle inequality leads to

$$\lim_{n \to \infty} |x_n| \leq \lim_{n \to \infty} \left(|C||r_1|^n + |D||r_2|^n \right) = 0.$$

If $r_1 = r_2$, and $|r_1| < 1$ (case 2 in theorem 6.10), we have the same conclusion since $n|r_1|^n$ tends to 0 when n tends to ∞. Finally, in the case of complex conjugate roots of absolute value less than 1 we have $\rho < 1$, so the term ρ^n will ensure that $|x_n|$ tends to 0. ∎

A situation that is not covered by corollary 6.11 is the case where both roots are real, but of opposite sign, and larger than 1 in absolute value. In this case the solution will also tend to infinity in most cases, but not always. Consider for example the case where $x_n = 2^n + (-2)^n$. Then $x_{2n+1} = 0$ for all n while $\lim_{n \to \infty} x_{2n} = \infty$.

6.4.3 Linear homogenous equations of general order

Consider now a kth-order, homogenous, and linear difference equation with constant coefficients,

$$x_{n+k} + b_{k-1} x_{n+k-1} + \cdots + b_1 x_{n+1} + b_0 x_n = 0,$$

where all the coefficients $\{b_i\}$ are real numbers. It is quite easy to show that if we have k solutions $\{x_n^i\}_{i=1}^k$, then the combination

$$x_n = C_1 x_n^1 + C_2 x_n^2 + \cdots + C_k x_n^k \tag{6.14}$$

will also be a solution for any choice of the coefficients $\{C_i\}$. As we have already seen, an equation of order k can be adapted to k initial values.

To determine k solutions, we follow the same procedure as for second-order equations and try the solution $x_n = r^n$. We then find that r must solve the characteristic equation

$$r^k + b_{k-1} r^{k-1} + \cdots + b_1 r + b_0 = 0.$$

From the fundamental theorem of algebra we know that this equation has k distinct roots, and complex roots occur in conjugate pairs since the coefficients are real. A theorem similar to theorem 6.10 can therefore be proved.

Observation 6.12. *The general solution of the difference equation*

$$x_{n+k} + b_{k-1} x_{n+k-1} + \cdots + b_1 x_{n+1} + b_0 x_n = 0$$

is a combination of k terms

$$x_n = C_1 x_n^1 + C_2 x_n^2 + \cdots + C_k x_n^k$$

where each term $\{x_k^i\}$ is a solution of the difference equation. The solution $\{x_n^i\}_n$ is essentially of the form $x_n^i = r_i^n$ where r_i is the ith root of the characteristic equation

$$r^k + b_{k-1} r^{k-1} + \cdots + b_1 r + b_0 = 0.$$

Note the word 'essentially' in the last sentence: just like for quadratic equations we have to take special care when there are double roots (or roots of even higher multiplicity) or complex roots.

Closed formulas for the roots can be found when for quadratic, cubic and quartic equations, but the expressions even for cubic equations can be rather complicated. For higher degree than 2, one therefore has to resort to numerical techniques, like the ones in chapter 10, for finding the roots.

There is also a close of analog of corollary 6.11 which shows that a solution will tend to zero if all roots have absolute value less than 1. And if there is a root with absolute value greater than 1, whose corresponding coefficient in (6.14) is nonzero, then the solution will grow beyond all bounds when n becomes large.

6.4.4 Inhomogenous equations

So far we have only discussed homogenous difference equations. For inhomogenous equations there is an important, but simple lemma, which can be found in the standard text books on difference equations.

Lemma 6.13. *Suppose that $\{x_n^p\}$ is a particular solution of the inhomogenous difference equation*

$$x_{n+k} + b_{k-1} x_{n+k-1} + \cdots + b_1 x_{n+1} + b_0 x_n = g(n). \tag{6.15}$$

Then all other solutions of the inhomogenous equation will have the form

$$x_n = x_n^p + x_n^h$$

where $\{x_n^h\}$ is some solution of the homogenous equation

$$x_{n+k} + b_{k-1} x_{n+k-1} + \cdots + b_1 x_{n+1} + b_0 x_n = 0.$$

More informally, lemma 6.13 means that we can find the general solution of (6.15) by just finding one solution, and then adding the general solution of the homogenous equation. The question is how to find one solution. The following observation is useful.

Observation 6.14. *One of the solutions of the inhomogenous equation*

$$x_{n+k} + b_{k-1} x_{n+k-1} + \cdots + b_1 x_{n+1} + b_0 x_n = g(n)$$

has the same form as $g(n)$.

Some examples will illustrate how this works.

Example 6.15. Consider the equation

$$x_{n+1} - 2x_n = 3. \tag{6.16}$$

Here the right-hand side is constant, so we try with the a particular solution $x_n^h = A$, where A is an unknown constant to be determined. If we insert this in the equation we find

$$A - 2A = 3,$$

so $A = -3$. This means that $x_n^p = -3$ is a solution of (6.16). Since the general solution of the homogenous equation $x_{n+1} - 2x_n = 0$ is $x_n^h = C2^n$, the general solution of (6.16) is

$$x_n = x_n^h + x_n^p = C2^n - 3. \quad \blacksquare$$

In general, when $g(n)$ is a polynomial in n of degree d, we try with a particular solution which is a general polynomial of degree d. When this is inserted in the equation, we obtain a relation between two polynomials that should hold for all values of n, and this requires corresponding coefficients to be equal. In this way we obtain a set of equations for the coefficients.

Example 6.16. In the third-order equation

$$x_{n+3} - 2x_{n+2} + 4x_{n+1} + x_n = n \tag{6.17}$$

the right-hand side is a polynomial of degree 1. We therefore try with a solution $x_n^p = A + Bn$ and insert this in the difference equation,

$$n = A + B(n+3) - 2(A + B(n+2)) + 4(A + B(n+1)) + A + Bn = (4A + 3B) + 4Bn.$$

The only way for the two sides to be equal for all values of n is if the constant terms and first degree terms on the two sides are equal,

$$4A + 3B = 0,$$
$$4B = 1.$$

From these equations we find $B = 1/4$ and $A = -3/16$, so one solution of (6.17) is

$$x_n^p = \frac{n}{4} - \frac{3}{16}. \quad \blacksquare$$

There are situations where the technique above does not work because the trial polynomial solution is also a homogenous solution. In this case the degree

of the polynomial must be increased. For more details we refer to a text book on difference equations.

Other types of right-hand sides can be treated similarly. One other type is given by functions like

$$g(n) = p(n)a^n,$$

where $p(n)$ is a polynomial in n and a is a real number. In this case, one tries with a solution $x_n^p = q(n)a^n$ where $q(n)$ is a general polynomial in n of the same degree as $p(n)$.

Example 6.17. Suppose we have the equation

$$x_{n+1} + 4x_n = n3^n. \tag{6.18}$$

The right hand side is a first-degree polynomial in n multiplied by 3^n, so we try with a particular solution in the form

$$x_n^p = (A + Bn)3^n.$$

When this is inserted in the difference equation we obtain

$$\begin{aligned}
n3^n &= \big(A + B(n+1)\big)3^{n+1} + 4(A + Bn)3^n \\
&= 3^n\Big(3\big(A + B(n+1)\big) + 4A + 4Bn\Big) \\
&= 3^n(7A + B + 5Bn).
\end{aligned}$$

Here we can cancel 3^n, which reduces the equation to an equality between two polynomials. If these are to agree for all values of n, the constant terms and the linear terms must agree,

$$7A + B = 0,$$
$$5B = 1.$$

This system has the solution $B = 1/5$ and $A = -1/35$, so a particular solution of (6.18) is

$$x_n^p = \left(\frac{1}{5}n - \frac{1}{35}\right)3^n.$$

The homogenous equation $x_{n+1} - 4x_n = 0$ has the general solution $x_n^h = C4^n$ so according to lemma 6.13 the general solution of (6.18) is

$$x_n = x_n^h + x_n^p = C4^n + \left(-\frac{1}{35} + \frac{1}{5}n\right)3^n. \quad \blacksquare$$

6.5 Round-off errors and stability for linear equations

In chapter 5, we saw that computations on a computer often lead to errors, at least when we use floating-point numbers. Therefore, when the solution of a difference equation is computed via one of the algorithms in section 6.3, we must be prepared for errors. In this section we are going to study this in some detail. We will restrict our attention to linear difference equations with constant coefficients.

We first recall that integer arithmetic is always correct, except for the possibility of overflow, which is so dramatic that it is usually quite easy to detect. We therefore focus on the case where floating-point numbers must be used. Note that we use 64-bit floating-point numbers in all the examples in this chapter.

The effect of round-off errors become quite visible from a couple of examples.

Example 6.18. Consider the equation

$$x_{n+2} - \frac{2}{3}x_{n+1} - \frac{1}{3}x_n = 0, \quad x_0 = 1, \ x_1 = 0. \tag{6.19}$$

Since the two roots of the characteristic equation $r^2 - 2r/3 - 1/3 = 0$ are $r_1 = 1$ and $r_2 = -1/3$, the general solution of the difference equation is

$$x_n = C + D\left(-\frac{1}{3}\right)^n.$$

The initial conditions yield the equations

$$C + D = 1,$$
$$C - D/3 = 0,$$

which has the solution $C = 1/4$ and $D = 3/4$. The solution of (6.19) is therefore

$$x_n = \frac{1}{4}\left(1 + (-1)^n 3^{1-n}\right).$$

We observe that x_n tends to $1/4$ as n tends to infinity.

Note that if we simulate the equation (6.19) on a computer, the next term is computed by the formula $x_{n+2} = (2x_{n+1} + x_n)/3$. The division by 3 means that floating-point numbers are required to calculate this expression. If we simulate the difference equation, we obtain the four approximate values

$$\tilde{x}_{10} = 0.250012701316,$$
$$\tilde{x}_{15} = 0.249999947731,$$
$$\tilde{x}_{20} = 0.250000000215,$$
$$\tilde{x}_{30} = 0.250000000000,$$

which agree with the exact solution to 12 digits. In other words, numerical simulation in this case works very well, even if floating-point numbers are used in the calculations. ∎

Example 6.19. We consider the difference equation

$$x_{n+2} - \frac{19}{3}x_{n+1} + 2x_n = -10, \quad x_0 = 2, \, x_1 = 8/3. \tag{6.20}$$

The two roots of the characteristic equation are $r_1 = 1/3$ and $r_2 = 6$, so the general solution of the homogenous equation is

$$x_n^h = C3^{-n} + D6^n.$$

To find a particular solution we try a solution $x_n^p = A$ which has the same form as the right-hand side. We insert this in the difference equation and find $A = 3$, so the general solution is

$$x_n = x_n^h + x_n^p = 3 + C3^{-n} + D6^n. \tag{6.21}$$

If we enforce the initial conditions, we end up with the system of equations

$$\begin{aligned} 2 = x_0 = 3 + C + D, \\ 8/3 = x_1 = 3 + C/3 + 6D. \end{aligned} \tag{6.22}$$

This may be rewritten as

$$\begin{aligned} C + D = -1, \\ C + 18D = -1. \end{aligned} \tag{6.23}$$

which has the solution $C = -1$ and $D = 0$. The final solution is therefore

$$x_n = 3 - 3^{-n}, \tag{6.24}$$

which tends to 3 when n tends to infinity.

Let us simulate the equation on the computer. As in the previous example we have to divide by 3 so we have to use floating-point numbers. Some early terms in the computed sequence are

$$\begin{aligned} \tilde{x}_5 &= 2.99588477366, \\ \tilde{x}_{10} &= 2.99998306646, \\ \tilde{x}_{15} &= 3.00001192858, \end{aligned}$$

(throughout this section we will use $\tilde{x}_n$ to denote a computed version of x_n). These values appear to approach 3 as they should. However, some later values are

$$\tilde{x}_{20} = 3.09329859009,$$
$$\tilde{x}_{30} = 5641411.98633, \qquad (6.25)$$
$$\tilde{x}_{40} = 3.41114428655 \times 10^{14},$$

and at least the last two of these are obviously completely wrong. ∎

6.5.1 Explanation of example 6.19

Let us see what went wrong in example 6.19. First of all we note that the initial values are $x_0 = 2$ and $x_1 = 8/3$. The first of these will be represented exactly in a computer whether we use integers or floating-point numbers. The second definitely requires floating-point numbers. Note though that the fraction 8/3 cannot be represented exactly with a finite number of digits, and therefore there will inevitably be round-off error.

> **Observation 6.20.** *The initial value $x_0 = 2$ is represented exactly by floating-point numbers, but the initial value 8/3 at x_1 becomes $\tilde{x}_1 = a_1$, where a_1 is the floating-point number closest to 8/3.*

The fact that the initial value at x_1 is not quite correct means that when the coefficients C and D in (6.21) are determined, the solutions are not exactly $C = 1$ and $D = 0$. If the initial values used in computations are $\tilde{x}_0 = 2$ and $\tilde{x}_1 = 8/3 + \delta$ where δ is a small number (in the range 10^{-18} to 10^{-15} for 64-bit floating-point numbers), we can determine the new values of C and D from equations like (6.22). If we solve these equations we find

$$C = -1 - \frac{3}{17}\delta, \quad D = \frac{3}{17}\delta. \qquad (6.26)$$

This is summarised in the next observation.

> **Observation 6.21.** *Because of round-off errors in the second initial value, the exact values of the coefficients C and D in (6.21) used in calculations are*
>
> $$C = -1 + \epsilon_1, \quad D = \epsilon_2 \qquad (6.27)$$
>
> *where ϵ_1 and ϵ_2 are small numbers. The solution used in the simulations is therefore*
> $$\tilde{x}_n = 3 - (1 - \epsilon_1)3^{-n} + \epsilon_2 6^n.$$

The actual computations are based on the formula

$$x_{n+2} = \frac{19}{3} x_{n+1} - 2x_n - 10$$

for computing the next term in the solution. Here as well there is division by 3, which inevitably brings with it further round-off errors each time a new term is computed. It turns out that these errors may also be roughly accounted for by assuming that the initial values are changed.

Observation 6.22. *The computed solution of the difference equation* (6.20), *including all round-off errors, is essentially the solution of a problem where the initial values have been changed, i.e., the computed values are the solution of the problem*

$$x_{n+2} - \frac{19}{3} x_{n+1} + 2x_n = -10, \quad x_0 = 2 + \delta_1, \, x_1 = 8/3 + \delta_2, \qquad (6.28)$$

where the actual values of δ_1 and δ_2 depend on the details of the computations.

Observation 6.22 means that we should be able to predict the effect of round-off errors by solving a problem with slightly different initial values. When we change the initial values, the coefficients C and D also change. When we only perturbed x_1 we obtained the solutions given by (6.26). With the initial values in (6.28) we find

$$C = -1 + \frac{18\delta_1 - 3\delta_2}{17}, \quad D = \frac{3\delta_2 - \delta_1}{17}.$$

These solutions may be written as in (6.27) if we set

$$\begin{aligned} \epsilon_1 &= (18\delta_1 - 3\delta_2)/17, \\ \epsilon_2 &= (3\delta_2 - \delta_1)/17. \end{aligned} \qquad (6.29)$$

The conclusion is therefore as in observation 6.21.

Observation 6.23. *When the difference equation* (6.20) *is simulated numerically, the computed numbers are essentially given by the formula*

$$\tilde{x}_n = 3 - (1 - \epsilon_1)3^{-n} + \epsilon_2 6^n, \qquad (6.30)$$

where ϵ_1 and ϵ_2 are small numbers.

From observation 6.23 it is easy to explain where the values in (6.25) came from. Because of round-off errors, the computed solution is given by (6.30), where ϵ_2 is a small nonzero number. Even if ϵ_2 is small, the product $\epsilon_2 6^n$ will eventually become large, since 6^n grows beyond all bounds when n becomes large. In fact we can use the values in (6.25) to estimate ϵ_2. For $n = 40$ we have $3^{-n} \approx 8.2 \times 10^{-20}$ and $6^n \approx 1.3 \times 10^{31}$. Since we have used 64-bit floating-point numbers, this means that only the last term in (6.30) is relevant (the other two terms affect the result in about the 30th digit and beyond. This means that we can find ϵ_2 from the relation

$$3.4 \times 10^{14} \approx \tilde{x}_{40} \approx \epsilon_2 6^{40} \approx \epsilon_2 1.3 \times 10^{31}.$$

From this we see that $\epsilon_2 \approx 2.6 \times 10^{-17}$. This is a reasonable value since we know from (6.29) that ϵ_2 is roughly as large as the round-off error in the initial values. With 64-bit floating-point numbers we have about 15–18 decimal digits, so a round-off error of about 10^{-17} is to be expected when the numbers are close to 1 as in this example.

> **Observation 6.24.** *When ϵ_2 is nonzero in* (6.30), *the last term $\epsilon_2 6^n$ will eventually dominate the computed solution of the difference equation completely, and the computations will eventually end in overflow.*

6.5.2 Round-off errors for linear equations of general order

It is important to realise that the problem with the computed solution in (6.25) becoming large is not really because of unfortunately large round-off errors; any round-off error at all would give the same kind of behaviour. The general problem is that we have a family of solutions given by

$$x_n = 3 + C3^{-n} + D6^n, \tag{6.31}$$

and different initial conditions pick out different solutions (different values of C and D) within this family. With floating-point numbers it is basically impossible to get D to be exactly 0, so the last term in (13.4) will always dominate the computed solution for large values of n and completely overwhelm the other two parts of the solution.

The difference equation in example 6.19 is not particularly demanding — we will get the same effect whenever we have a difference equation where at least one of the roots of the characteristic equation is larger than 1 in absolute value, but the exact solution remains bounded, or at least significantly smaller than the part of solutions corresponding to the largest root.

In example 6.19, the solution family has three components: the two solutions 6^n and 3^{-n} from the homogenous equation, and the constant solution 3 from the inhomogenous equation. When the solution we are after just involves 3^{-n} and 3 we get into trouble because we invariably also bring along 6^n because of round-off errors. On the other hand, if the exact initial values lead to a solution that includes 6^n, then we will not get problems with round-off: The coefficient multiplying 6^n will be accurate enough, and the other terms are too small to pollute the 6^n solution.

Example 6.26. We consider the third-order difference equation

$$x_{n+3} - \frac{16}{3}x_{n+2} + \frac{17}{3}x_{n+1} - \frac{4}{3}x_n = 10 \times 2^n, \quad x_0 = -2, \ x_1 = -\frac{17}{3}, \ x_2 = -\frac{107}{9}.$$

The coefficients have been chosen so that the roots of the characteristic equation are $r_1 = 1/3$, $r_2 = 1$ and $r_3 = 4$. To find a particular solution we try with $x_n^p = A2^n$. If this is inserted in the equation we find $A = -3$, so the general solution is

$$x_n = -3 \times 2^n + B3^{-n} + C + D4^n. \tag{6.32}$$

The initial conditions force $B = 0$, $C = 1$ and $D = 0$, so the exact solution is

$$x_n = 1 - 3 \times 2^n. \tag{6.33}$$

The discussion above shows that this is bound to lead to problems. Because of round-off errors, the coefficients B and D will not be exactly 0 when the equation is simulated. Instead we will have

$$\tilde{x}_n = -3 \times 2^n + \epsilon_1 3^{-n} + (1 + \epsilon_2) + \epsilon_3 4^n$$

Even if ϵ_3 is small, the term $\epsilon_3 4^n$ will dominate when n becomes large. This is confirmed if we do the simulations. The computed value $\tilde{x}_{100}$ is approximately 4.5×10^{43}, while the exact value is -3.8×10^{30}. ∎

6.6 Summary

In this chapter we met the effect of round-off errors on realistic computations for the first time. We saw that innocent-looking computations like the simulation of the difference equation in example 6.19 led to serious problems with round-off errors. By making use of the theory behind difference equations we were able to understand why the simulations behave the way they do. From this insight we also realise that for this particular equation and initial values, the blow-up is unavoidable, just like cancellation is unavoidable when we subtract two almost equal numbers. Such problems are usually referred to as being *badly conditioned*. On the other hand, a different choice of initial conditions may lead to calculations with no round-off problems; then the problem is said to be *well conditioned*.

Exercises

6.1 Which equations are linear?

    a) $x_{n+2} + 3x_{n+1} - \sin(n)x_n = n!$.

    b) $x_{n+3} - x_{n+1} + x_n^2 = 0$.

    c) $x_{n+2} + x_{n+1}x_n = 0$.

    d) $nx_{n+2} - x_{n+1}e^n + x_n = n^2$.

6.2 Program algorithm 6.6 and test it on the Fibonacci equation

$$x_{n+2} = x_{n+1} + x_n, \quad x_0 = 0, x_1 = 1.$$

6.3 Generalise algorithm 6.6 to third order equations and test it on the Fibonacci like equation

$$x_{n+3} = x_{n+2} + x_{n+1} + x_n, \quad x_0 = 0, x_1 = 1, x_2 = 1.$$

6.4 In this exercise we are going to study the difference equation

$$x_{n+1} - 3x_n = 5^{-n}, \quad x_0 = -5/14. \tag{6.34}$$

    a) Show that the general solution of (6.34) is

$$x_n = C3^n - \frac{5}{14}5^{-n}$$

    and that the initial condition leads to the solution

$$x_n = -\frac{5}{14}5^{-n}.$$

b) Explain what will happen if you simulate equation 6.34 numerically.

c) Do the simulation and check that your prediction in (b) is correct.

6.5 We consider the Fibonacci equation with nonstandard initial values

$$x_{n+2} - x_{n+1} - x_n = 0, \quad x_0 = 0, \, x_1 = (1 - \sqrt{5})/2. \tag{6.35}$$

a) Show that the general solution of the equation is

$$x_n = C\frac{1 + \sqrt{5}}{2} + D\frac{1 - \sqrt{5}}{2},$$

and that the initial values picks out the solution

$$x_n = \frac{1 - \sqrt{5}}{2}.$$

b) What will happen if you simulate (6.35) on a computer?

c) Do the simulation and check that your predictions are correct.

6.6 We have the difference equation

$$x_{n+2} - \frac{2}{5}x_{n+1} + \frac{1}{45}x_n = 0, \quad x_0 = 1, \, x_1 = 1/15. \tag{6.36}$$

a) Determine the general solution of (6.36) as well as the solution selected by the initial condition.

b) Why must you expect problems when you do a numerical simulation of the equation?

c) Determine approximately the value of n when the numerical solution has lost all significant digits.

d) Perform the numerical simulation and check that your predictions are correct.

6.7 In this exercise we consider the difference equation

$$x_{n+2} - \frac{5}{2}x_{n+1} + x_n = 0, \quad x_0 = 1, \, x_1 = 1/2.$$

a) Determine the general solution, and the solution corresponding to the initial conditions.

b) What kind of behaviour do you expect if you simulate the equation numerically?

c) Do the simulation and explain your results.

CHAPTER 7

Lossless Compression

Computers can handle many different kinds of information like text, equations, games, sound, photos, and film. Some of these information sources require a huge amount of data and may quickly fill up your hard disk or take a long time to transfer across a network. For this reason it is interesting to see if we can somehow rewrite the information in such a way that it takes up less space. This may seem like magic, but does in fact work well for many types of information. There are two general classes of methods, those that do not change the information, so that the original file can be reconstructed exactly, and those that allow small changes in the data. Compression methods in the first class are called *lossless compression methods* while those in the second class are called *lossy compression methods*. Lossy methods may sound risky since they will change the information, but for data like sound and images small alterations do not usually matter. On the other hand, for certain kinds of information like for example text, we cannot tolerate any change so we have to use lossless compression methods.

In this chapter we are going to study lossless methods; lossy methods will be considered in a later chapter. To motivate our study of compression techniques, we will first consider some examples of technology that generate large amounts of information. We will then study two lossless compression methods in detail, namely *Huffman coding* and *arithmetic coding*. Huffman coding is quite simple and gives good compression, while arithmetic coding is more complicated, but gives excellent compression.

In section 7.3.2 we introduce the *information entropy* of a sequence of symbols which essentially tells us how much information there is in the sequence. This is useful for comparing the performance of different compression strategies.

7.1 Introduction

The potential for compression increases with the size of a file. A book typically has about 300 words per page and an average word length of four characters. A book with 500 pages would then have about 600 000 characters. If we write in English, we may use a character encoding like ISO Latin 1 which only requires one byte per character. The file would then be about 700 KB (kilobytes)[1], including 100 KB of formatting information. If we instead use UTF-16 encoding, which requires two bytes per character, we end up with a total file size of about 1300 KB or 1.3 MB. Both files would represent the same book so this illustrates straight away the potential for compression, at least for UTF-16 encoded documents. On the other hand, the capacity of present day hard disks and communication channels are such that a saving of 0.5 MB is usually negligible.

For sound files the situation is different. A music file in CD-quality requires 44 100 two-byte integers to be stored every second for each of the two stereo channels, a total of about 176 KB per second, or about 10 MB per minute of music. A four-minute song therefore corresponds to a file size of 40 MB and a CD with one hour of music contains about 600 MB. If you just have a few CDs this is not a problem when the average size of hard disks is approaching 1 TB (1 000 000 MB or 1 000 GB). But if you have many CDs and want to store the music in a small portable player, it is essential to be able to compress this information. Audio-formats like Mp3 and Aac manage to reduce the files down to about 10 % of the original size without sacrificing much of the quality.

Not surprisingly, video contains even more information than audio so the potential for compression is considerably greater. Reasonable quality video requires at least 25 images per second. The images used in traditional European television contain 576×720 small coloured dots, each of which are represented with 24 bits[2]. One image therefore requires about 1.2 MB and one second of video requires about 31MB. This corresponds to 1.9 GB per minute and 112 GB per hour of video. In addition we also need to store the sound. If you have more than a handful of films in such an uncompressed format, you are quickly going to exhaust the capacity of even quite large hard drives.

These examples should convince you that there is a lot to be gained if we can compress information, especially for video and music, and virtually all video formats, even the high-quality ones, use some kind of compression. With compression we can fit more information onto our hard drive and we can transmit information across a network more quickly.

---

[1] Here we use the SI prefixes, see Table 4.1.

[2] This is a digital description of the analog PAL system.

```
000000000000000000000000000000000000000000
000000000000000001110000000000000000000000
000000000000000001110000000000000000000000
000000000000000001110000000000000000000000
000000000000000001110000000000000000000000
000000000000000001110000000000000000000000
000000000000000001110000000000000000000000
000000000000000001110000000000000000000000
000000000000000001110000000000000000000000
000000000000000001110000000000000000000000
000000000000000001110000000000000000000000
000000000000000001110000000000000000000000
000000000000000001110000000000000000000000
000000000000000001110000000000000000000000
000000000000000001110000000000000000000000
000000000000000001110000000000000000000000
000000000000000001110000000000000000000000
000000000000000001110000000000000000000000
000000000000000000000000000000000000000000
```

Figure 7.1. A representation of a black and white image, for example part of a scanned text document.

7.1.1   Run-length coding

We have already seen that English text stored in UTF-16 encoding can be compressed to at least half the number of bits by encoding in UTF-8 instead. Another simple strategy that sometimes work well is *run-length encoding*. One example is text documents scanned in black and white. This will produce an image of each page represented by a two-dimensional array of a large number of intensity values. At a point which is covered by text the intensity value will be 1, at all other points the value will be 0. Since most pages of text contain much more white than black, there will be long sequences of 0s in the intensity array, see figure 7.1. A simple way to compress such a file is to replace a sequence of n consecutive 0s by a code like n. Programs that read this file must then of course know that this code is to be interpreted as n consecutive 0s. As long as the sequences of 0s are long enough, this can lead to significant compression of the image file.

7.2   Huffman coding

The discussion in section 7.1 illustrates both the potential and the possibility of compression techniques. In this section we are going to approach the subject in more detail and describe a much used technique.

Before we continue, let us agree on some notation and vocabulary.

Definition 7.1 (Jargon used in compression). *A sequence of symbols is called a* text *and is denoted $x = \{x_1, x_2, \ldots, x_m\}$. The symbols are assumed to be taken from an* alphabet *that is denoted $\mathscr{A} = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$, and the number of times that the symbol α_i occurs in x is called its* frequency *and is denoted by $f(\alpha_i)$. For compression each symbol α_i is assigned a binary code $c(\alpha_i)$, and the text x is stored as the bit-sequence z obtained by replacing each character in x by its binary code. The set of all binary codes is called a* dictionary *or* code book.

If we are working with English text, the sequence x will just be a string of letters and other characters like $x = \{$h,e, l, l, o, , a, g, a, i, n, .$\}$ (the character after 'o' is space, and the last character a period). The alphabet $\mathscr{A}$ is then the ordinary Latin alphabet augmented with the space character, punctuation characters and digits, essentially characters 32–127 of the ASCII table, see Table 4.3. In fact, the ASCII codes define a dictionary since it assigns a binary code to each character. However, if we want to represent a text with few bits, this is not a good dictionary because the codes of very frequent characters are no shorter than the codes of the characters that are hardly ever used.

In other contexts, we may consider the information to be a sequence of bits and the alphabet to be $\{0, 1\}$, or we may consider sequences of bytes in which case the alphabet would be the 256 different bit combinations in a byte.

Let us now suppose that we have a text $x = \{x_1, x_2, \ldots, x_m\}$ with symbols taken from an alphabet $\mathscr{A}$. A simple way to represent the text in a computer is to assign an integer code $c(\alpha_i)$ to each symbol and store the sequence of codes $\{c(x_1), c(x_2), \ldots, c(x_m)\}$. The question is just how the codes should be assigned.

Small integers require fewer digits than large ones so a good strategy is to let the symbols that occur most frequently in x have short codes and use long codes for the rare symbols. This leaves us with the problem of knowing the boundary between the codes. Huffman coding uses a clever set of binary codes which makes it impossible to confuse the codes even though they have different lengths.

Fact 7.2 (Huffman coding). *In Huffman coding the most frequent symbols in a text x get the shortest codes, and the codes have the* prefix *property which means that the bit sequence that represents a code is never a prefix of any other code. Once the codes are known the symbols in x are replaced by their codes and the resulting sequence of bits z is the compressed version of x.*

Example 7.3. This may sound a bit vague, so let us consider an example. Suppose we have the four-symbol text $x =$ DBACDBD of length 7. We note that

132

the different symbols occur with frequencies $f(A) = 1$, $f(B) = 2$, $f(C) = 1$ and $f(D) = 3$. We use the codes

$$c(D) = 1, \quad c(B) = 01, \quad c(C) = 001, \quad c(A) = 000. \qquad (7.1)$$

We can then store the text as

$$z = 1010000011011, \qquad (7.2)$$

altogether 13 bits, while a standard encoding with one byte per character would require 56 bits. Note also that we can easily decipher the code since the codes have the prefix property. The first bit is 1 which must correspond to a 'D' since this is the only character with a code that starts with a 1. The next bit is 0 and since this is the start of several codes we read one more bit. The only character with a code that start with 01 is 'B' so this must be the next character. The next bit is 0 which does not uniquely identify a character so we read one more bit. The code 00 does not identify a character either, but with one more bit we obtain the code 000 which corresponds to the character 'A'. We can obviously continue in this way and decipher the complete compressed text. ∎

Compression is not quite as simple as it was presented in example 7.3. A program that reads the compressed code must clearly know the codes (7.1) in order to decipher the code. Therefore we must store the codes as well as the compressed text z. This means that the text must have a certain length before it is worth compressing it.

7.2.1   Binary trees

The description of Huffman coding in fact 7.2 is not at all precise since it does not state how the codes are determined. The actual algorithm is quite simple, but requires a new concept.

Definition 7.4 (Binary tree). *A binary tree T is a finite collection of nodes where one of the nodes is designated as the* root *of the tree, and the remaining nodes are partitioned into two disjoint groups T_0 and T_1 that are also trees. The two trees T_0 and T_1 are called the* subtrees *or* children *of T. Nodes which are not roots of subtrees are called* leaf nodes. *A connection from one node to another is called an* edge *of the tree.*

An example of a binary tree is shown in figure 7.2. The root node which is shown at the top has two subtrees. The subtree to the right also has two subtrees, both of which only contain leaf nodes. The subtree to the left of the root only has one subtree which consists of a single leaf node.
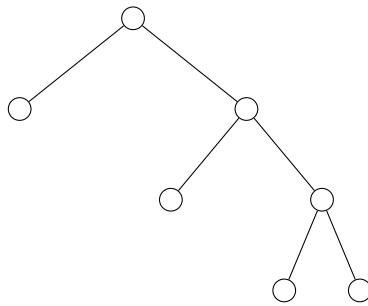
Figure 7.2. An example of a binary tree.

7.2.2 Huffman trees

It turns out that Huffman coding can conveniently be described in terms of a binary tree with some extra information added. These trees are usually referred to as *Huffman trees.*

Definition 7.5. *A Huffman tree is a binary tree that can be associated with an alphabet consisting of symbols $\{\alpha_i\}_{i=1}^n$ with frequencies $f(\alpha_i)$ as follows:*

1. *Each leaf node is associated with exactly one symbol α_i in the alphabet, and all symbols are associated with a leaf node.*

2. *Each node has an associated integer weight:*

   (a) *The weight of a leaf node is the frequency of the symbol.*

   (b) *The weight of a node is the sum of the weights of the roots of the node's subtrees.*

3. *All nodes that are not leaf nodes have exactly two children.*

4. *The Huffman code of a symbol is obtained by following edges from the root to the leaf node associated with the symbol. Each edge adds a bit to the code: a 0 if the edge points to the left and a 1 if it points to the right.*

Example 7.6. In figure 7.3 the tree in figure 7.2 has been turned into a Huffman tree. The tree has been constructed from the text CCDACBDC with the alphabet {A,B,C,D} and frequencies $f(A) = 1$, $f(B) = 1$, $f(C) = 4$ and $f(D) = 2$. It is easy to see that the weights have the properties required for a Huffman tree, and by
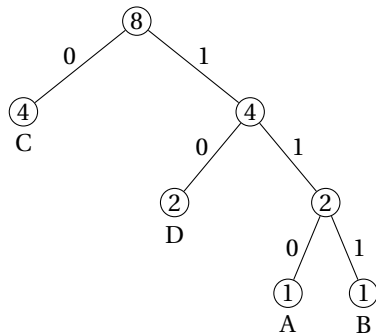
Figure 7.3. A Huffman tree.

following the edges we see that the Huffman codes are given by $c(C) = 0$, $c(D) = 10$, $c(A) = 110$ and $c(B) = 111$. Note in particular that the root of the tree has weight equal to the length of the text. ■

We will usually omit the labels on the edges since they are easy to remember: An edge that points to the left corresponds to a 0, while an edge that points to the right yields a 1.

7.2.3 The Huffman algorithm

In example 7.6 the Huffman tree was just given to us; the essential question is how the tree can be constructed from a given text. There is a simple algorithm that accomplishes this.

Algorithm 7.7 (Huffman algorithm)**.** *Let the text x with symbols $\{\alpha_i\}_{i=1}^n$ be given, and let the frequency of α_i be $f(\alpha_i)$. The Huffman tree is constructed by performing the following steps:*

1. *Construct a one-node Huffman tree from each of the n symbols α_i and its corresponding weight; this leads to a collection of n one-node trees.*

2. *Repeat until the collection consists of only one tree:*

    (a) *Choose two trees T_0 and T_1 with minimal weights and replace them with a new tree which has T_0 as its left subtree and T_1 as its right subtree.*

3. *The tree remaining after the previous step is a Huffman tree for the given text x.*

Most of the work in algorithm 7.7 is in step 2, but note that the number of trees is reduced by one each time, so the loop will run at most n times.

The easiest way to get to grips with the algorithm is to try it on a simple example.

Example 7.8. Let us try out algorithm 7.7 on the text 'then the hen began to eat'. This text consists of 32 characters, including the five spaces. We first determine the frequencies of the different characters by counting. We find the collection of one-node trees
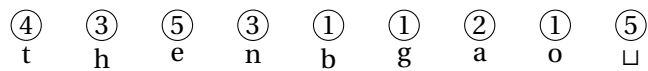
where the last character denotes the space character. Since 'b' and 'g' are two characters with the lowest frequency, we combine them into a tree,
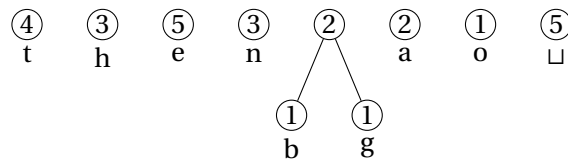
The two trees with the lowest weights are now the character 'o' and the tree we formed in the last step. If we combine these we obtain
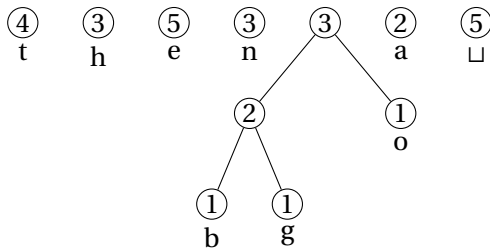
Now we have several choices. We choose to combine 'a' and 'h',
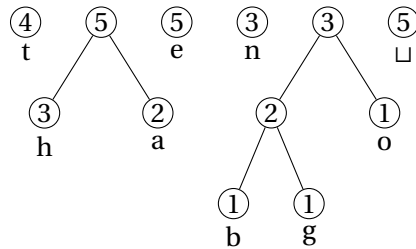
At the next step we combine the two trees with weight 3,
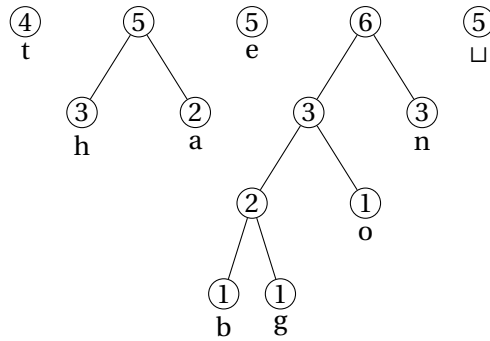
Next we combine the 't' and the 'e',
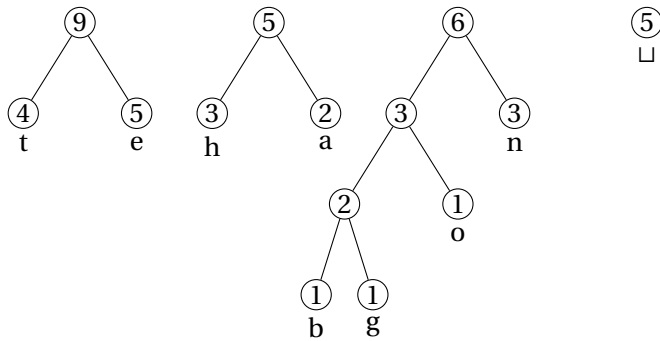
We now have two trees with weight 5 that must be combined
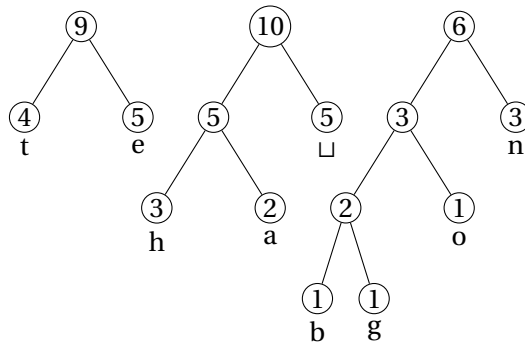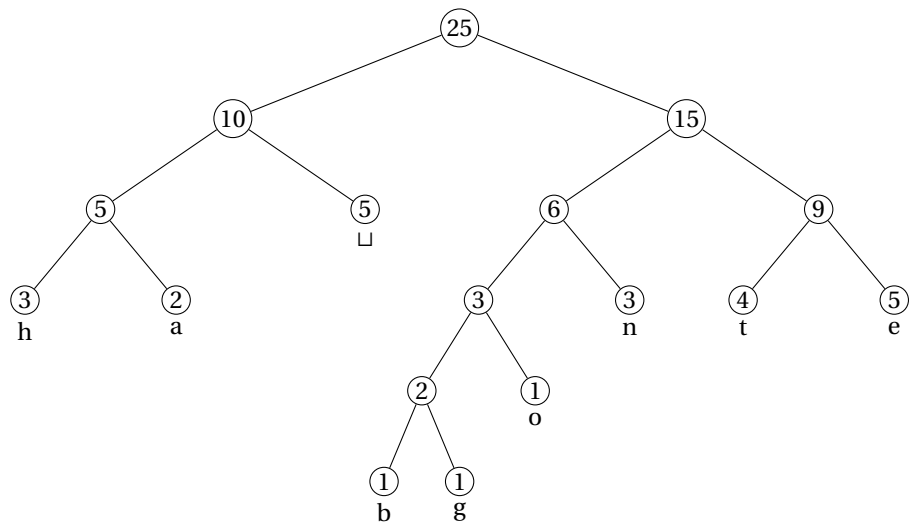
Figure 7.4. The Huffman tree for the text 'then the hen began to eat'.

Again we combine the two trees with the smallest weights,
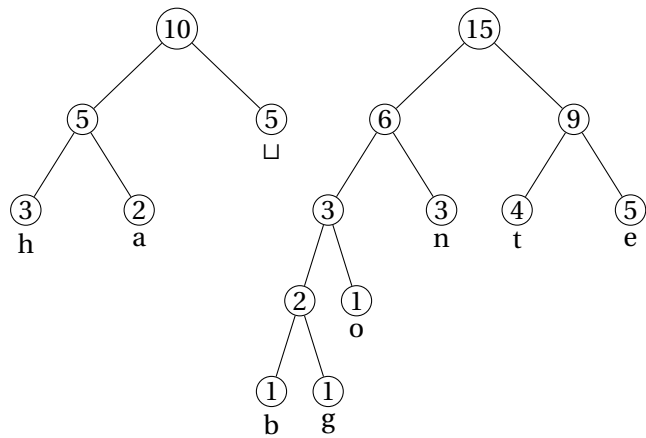
By combining these two trees we obtain the final Huffman tree in figure 7.4. From this we can read off the Huffman codes as

$$c(h) = 000, \qquad c(b) = 10000, \qquad c(n) = 101,$$
$$c(a) = 001, \qquad c(g) = 10001, \qquad c(t) = 110,$$
$$c(\sqcup) = 01, \qquad c(o) = 1001, \qquad c(e) = 111.$$

so we see that the Huffman coding of the text 'then the hen began to eat' is

110 000 111 101 01 110 000 111 01 000 111 101 01 10000

111 10001 001 101 01 110 1001 01 111 001 110

The spaces and the new line have been added to make the code easier to read; on a computer these will not be present.

The original text consists of 25 characters including the spaces. Encoding this with standard eight-bit encodings like ISO Latin or UTF-8 would require 400 bits. Since there are only nine symbols we could use a shorter fixed width encoding for this particular text. This would require five bits per symbol and would reduce the total length to 125 bits. In contrast the Huffman encoding only requires 75 bits. ∎

7.2.4 Properties of Huffman trees

Huffman trees have a number of useful properties, and the first one is the prefix property, see fact 7.2. This is easy to deduce from simple properties of Huffman trees.

Proposition 7.9 (Prefix property)**.** *Huffman coding has the prefix property: No code is a prefix of any other code.*

Proof. Suppose that Huffman coding does not have the prefix property, we will show that this leads to a contradiction. Let the code c_1 be the prefix of another code c_2, and let n_i be the node associated with the symbol with code c_i. Then the node n_1 must be somewhere on the path from the root down to n_2. But then n_2 must be located further from the root than n_1, so n_1 cannot be a leaf node, which contradicts the definition of a Huffman tree (remember that symbols are only associated with leaf nodes). ∎

We emphasise that it is the prefix property that makes it possible to use variable lengths for the codes; without this property we would not be able to decode an encoded text. Just consider the simple case where $c(A) = 01$, $c(B) = 010$ and $c(C) = 1$; which text would the code 0101 correspond to?

In the Huffman algorithm, we start by building trees from the symbols with lowest frequency. These symbols will therefore end up the furthest from the root and end up with the longest codes, as is evident from example 7.8. Likewise, the symbols with the highest frequencies will end up near the root of the tree and therefore receive short codes. This property of Huffman coding can be quantified, but to do this we must introduce a new concept.

Note that any binary tree with the symbols at the leaf nodes gives rise to a coding with the prefix property. A natural question is then which tree gives the coding with the fewest bits?

Theorem 7.10 (Optimality of Huffman coding)**.** *Let $\boldsymbol{x}$ be a given text, let T be any binary tree with the symbols of $\boldsymbol{x}$ as leaf nodes, and let $\ell(T)$ denote the number of bits in the encoding of $\boldsymbol{x}$ in terms of the codes from T. If T^* denotes the Huffman tree corresponding to the text $\boldsymbol{x}$ then*

$$\ell(T^*) \leq \ell(T).$$

Theorem 7.10 says that Huffman coding is optimal, at least among coding schemes based on binary trees. Together with its simplicity, this accounts for the popularity of this compression method.

7.3   Probabilities and information entropy

Huffman coding is the best possible among all coding schemes based on binary trees, but could there be completely different schemes, which do not depend on binary trees, that are better? And if this is the case, what would be the best possible scheme? To answer questions like these, it would be nice to have a way to tell how much information there is in a text.

7.3.1   Probabilities rather than frequencies

Let us first consider more carefully how we should measure the quality of Huffman coding. For a fixed text $\boldsymbol{x}$, our main concern is how many bits we need to encode the text, see the end of example 7.8. If the symbol α_i occurs $f(\alpha_i)$ times and requires $\ell(\alpha_i)$ bits and we have n symbols, the total number of bits is

$$B = \sum_{i=1}^{n} f(\alpha_i)\ell(\alpha_i). \tag{7.3}$$

However, we note that if we multiply all the frequencies by the same constant, the Huffman tree remains the same. It therefore only depends on the relative frequencies of the different symbols, and not the length of the text. In other words, if we consider a new text which is twice as long as the one we used in example 7.8, with each letter occurring twice as many times, the Huffman tree would be the same. This indicates that we should get a good measure of the quality of an encoding if we divide the total number of bits used by the length of

the text. If the length of the text is m this leads to the quantity

$$\bar{B} = \sum_{i=1}^{n} \frac{f(\alpha_i)}{m} \ell(\alpha_i). \tag{7.4}$$

If we consider longer and longer texts of the same type, it is reasonable to believe that the relative frequencies of the symbols would converge to a limit $p(\alpha_i)$ which is usually referred to as the *probability* of the symbol α_i. As always for probabilities we have $\sum_{i=1}^{n} p(\alpha_i) = 1$.

Instead of referring to the frequencies of the different symbols in an alphabet we will from now on refer to the probabilities of the symbols. We can then translate the bits per symbol measure in equation 7.4 to a setting with probabilities.

> **Observation 7.11** (Bits per symbol). *Let $\mathcal{A} = \{\alpha_1, \ldots, \alpha_n\}$ be an alphabet where the symbol α_i has probability $p(\alpha_i)$ and is encoded with $\ell(\alpha_i)$ bits. Then the average number of bits per symbol in a text encoded with this alphabet is*
>
> $$\bar{b} = \sum_{i=1}^{n} p(\alpha_i)\ell(\alpha_i). \tag{7.5}$$

Note that the Huffman algorithm will work just as well if we use the probabilities as weights rather than the frequencies, as this is just a relative scaling. In fact, the most obvious way to obtain the probabilities is to just divide the frequencies with the number of symbols for a given text. However, it is also possible to use a probability distribution that has been determined by some other means. For example, the probabilities of the different characters in English have been determined for typical texts. Using these probabilities and the corresponding codes will save you the trouble of processing your text and computing the probabilities for a particular text. Remember however that such pre-computed probabilities are not likely to be completely correct for a specific text, particularly if the text is short. And this of course means that your compressed text will not be as short as it would be had you computed the correct probabilities.

In practice, it is quite likely that the probabilities of the different symbols change as we work our way through a file. If the file is long, it probably contains different kinds of information, as in a document with both text and images. It would therefore be useful to update the probabilities at regular intervals. In the case of Huffman coding this would of course also require that we update the Huffman tree and therefore the codes assigned to the different symbols. This

141

may sound complicated, but is in fact quite straightforward. The key is that the decoding algorithm must compute probabilities in exactly the same way as the compression algorithm and update the Huffman tree at exactly the same position in the text. As long as this requirement is met, there will be no confusion as the compression end decoding algorithms will always use the same codes.

7.3.2 Information entropy

The quantity $\bar{b}$ in observation 7.11 measures the number of bits used per symbol for a given coding. An interesting question is how small we can make this number by choosing a better coding strategy. This is answered by a famous theorem.

Theorem 7.12 (Shannon's theorem). *Let $\mathcal{A} = \{\alpha_1, \ldots, \alpha_n\}$ be an alphabet where the symbol α_i has probability $p(\alpha_i)$. Then the minimal number of bits per symbol in an encoding using this alphabet is given by*

$$H = H(p_1, \ldots, p_n) = - \sum_{i=1}^{n} p(\alpha_i) \log_2 p(\alpha_i).$$

where $\log_2$ denotes the logarithm to base 2. The quantity H is called the information entropy of the alphabet with the given probabilities.

Example 7.13. Let us return to example 7.8 and compute the entropy in this particular case. From the frequencies we obtain the probabilities

$$c(t) = 4/25, \qquad c(n) = 3/25, \qquad c(a) = 2/25,$$
$$c(h) = 3/25, \qquad c(b) = 1/25, \qquad c(o) = 1/25,$$
$$c(e) = 1/5, \qquad c(g) = 1/25, \qquad c(\sqcup) = 1/5.$$

We can then compute the entropy to be $H \approx 2.93$. If we had a compression algorithm that could compress the text down to this number of bits per symbol, we could represent our 25-symbol text with 74 bits. This is only one bit less than what we obtained in example 7.8, so Huffman coding is very close to the best we can do for this particular text. ∎

Note that the entropy can be written as

$$H = \sum_{i=1}^{n} p(\alpha_i) \log_2 \bigl(1/p(\alpha_i)\bigr).$$

If we compare this expression with equation (7.5) we see that a compression strategy would reach the compression rate promised by the entropy if the length

of the code for the symbol α_i was $\log_2\big(1/p(\alpha_i)\big)$. But we know that this is just the number of bits in the number $1/p(\alpha_i)$. This therefore indicates that an optimal compression scheme would represent α_i by the number $1/p(\alpha_i)$. Huffman coding necessarily uses an integer number of bits for each code, and therefore only has a chance of reaching entropy performance when $1/p(\alpha_i)$ is a power of 2 for all the symbols. In fact Huffman coding does reach entropy performance in this situation, see exercise 3.

7.4   Arithmetic coding

When the probabilities of the symbols are far from being fractions with powers of 2 in their denominators, the performance of Huffman coding does not come close to entropy performance. This typically happens in situations with few symbols as is illustrated by the following example.

Example 7.14. Suppose that we have a two-symbol alphabet $\mathscr{A} = \{0,1\}$ with the probabilities $p(0) = 0.9$ and $p(1) = 0.1$. Huffman coding will then just use the obvious codes $c(0) = 0$ and $c(1) = 1$, so the average number of bits per symbol is 1, i.e., there will be no compression at all. If we compute the entropy we obtain

$$H = -0.9\log_2 0.9 - 0.1\log_2 0.1 \approx 0.47.$$

So while Huffman coding gives no compression, there may be coding methods that will reduce the file size to less than half the original size.   ∎

7.4.1   Arithmetic coding basics

Arithmetic coding is a coding strategy that is capable of compressing files to a size close to the entropy limit. It uses a different strategy than Huffman coding and does not need an integer number of bits per symbol and therefore performs well in situations where Huffman coding struggles. The basic idea of arithmetic coding is quite simple.

> **Idea 7.15** (Basic idea of arithmetic coding). *Arithmetic coding associates sequences of symbols with different subintervals of $[0,1)$. The width of a subinterval is proportional to the probability of the corresponding sequence of symbols, and the* arithmetic code *of a sequence of symbols is a floating-point number in the corresponding interval.*

To illustrate some of the details of arithmetic coding, it is easiest to consider an example.
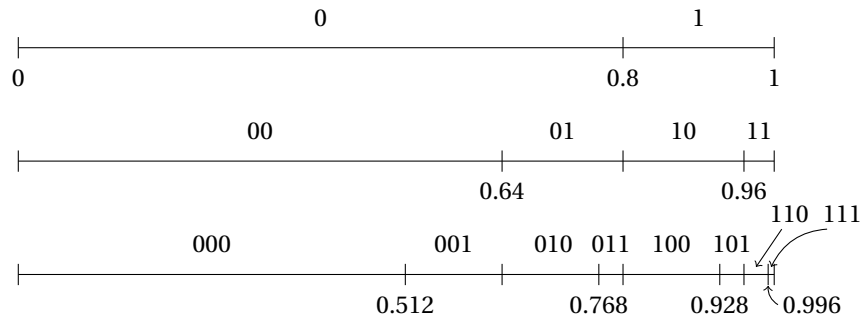
Figure 7.5. The basic principle of arithmetic coding applied to the text in example 7.16.

Example 7.16 (Determining an arithmetic code)**.** We consider the two-symbol text '00100'. As for Huffman coding we first need to determine the probabilities of the two symbols which we find to be $p(0) = 0.8$ and $p(1) = 0.2$. The idea is to allocate different parts of the interval $[0, 1)$ to the different symbols, and let the length of the subinterval be proportional to the probability of the symbol. In our case we allocate the interval $[0, 0.8)$ to '0' and the interval $[0.8, 1)$ to '1'. Since our text starts with '0', we know that the floating-point number which is going to represent our text must lie in the interval $[0, 0.8)$, see the first line in figure 7.5.

We then split the two subintervals according to the two probabilities again. If the final floating point number ends up in the interval $[0, 0.64)$, the text starts with '00', if it lies in $[0.64, 0.8)$, the text starts with '01', if it lies in $[0.8, 0.96)$, the text starts with '10', and if the number ends up in $[0.96, 1)$ the text starts with '11'. This is illustrated in the second line of figure 7.5. Our text starts with '00', so the arithmetic code we are seeking must lie in the interval $[0, 0.64)$.

At the next level we split each of the four sub-intervals in two again, as shown in the third line in figure 7.5. Since the third symbol in our text is '1', the arithmetic code must lie in the interval $[0.512, 0.64)$. We next split this interval in the two subintervals $[0.512, 0.6144)$ and $[0.6144, 0.64)$. Since the fourth symbol is '0', we select the first interval. This interval is then split into $[0.512, 0.59392)$ and $[0.59392, 0.6144)$. The final symbol of our text is '0', so the arithmetic code must lie in the interval $[0.512, 0.59392)$.

We know that the arithmetic code of our text must lie in the half-open interval $[0.512, 0.59392)$, but it does not matter which of the numbers in the interval we use. The code is going to be handled by a computer so it must be represented in the binary numeral system, with a finite number of bits. We know that any number of this kind must be on the form $i/2^k$ where k is a positive integer and i is an integer in the range $0 \leq i < 2^k$. Such numbers are called *dyadic*

144

numbers. We obviously want the code to be as short as possible, so we are looking for the dyadic number with the smallest denominator that lies in the interval $[0.512, 0.59392)$. In our simple example it is easy to see that this number is $9/16 = 0.5625$. In binary this number is 0.1001_2, so the arithmetic code for the text '00100' is 1001. ∎

Example 7.16 shows how an arithmetic code is computed. We have done all the computations in decimal arithmetic, but in a program one would usually use binary arithmetic.

It is not sufficient to be able to encode a text; we must be able to decode as well. This is in fact quite simple. We split the interval $[0, 1]$ into the smaller pieces, just like we did during the encoding. By checking which interval contains our code, we can extract the correct symbol at each stage.

7.4.2   An algorithm for arithmetic coding

Let us now see how the description in example 7.16 can be generalised to a systematic algorithm in a situation with n different symbols. An important tool in the algorithm is a function that maps the interval $[0, 1]$ to a general interval $[a, b]$.

Observation 7.17. *Let $[a, b]$ be a given interval with $a < b$. The function*

$$g(z) = a + z(b - a)$$

will map any number z in $[0, 1]$ to a number in the interval $[a, b]$. In particular the endpoints are mapped to the endpoints and the midpoint to the midpoint,

$$g(0) = a, \quad g(1/2) = \frac{a + b}{2}, \quad g(1) = b.$$

We are now ready to study the details of the arithmetic coding algorithm. As before we have a text $\boldsymbol{x} = \{x_1, \ldots, x_m\}$ with symbols taken from an alphabet $\mathcal{A} = \{\alpha_1, \ldots, \alpha_n\}$, with $p(\alpha_i)$ being the probability of encountering α_i at any given position in $\boldsymbol{x}$. It is much easier to formulate arithmetic coding if we introduce one more concept.

Definition 7.18 (Cumulative probability distribution)**.** *Let $\mathcal{A} = \{\alpha_1, \ldots, \alpha_n\}$ be an alphabet where the probability of α_i is $p(\alpha_i)$. The* cumulative probability distribution *F is defined as*

$$F(\alpha_j) = \sum_{i=1}^{j} p(\alpha_i), \quad \text{for } j = 1, 2, \ldots, n.$$

$$L(\alpha_j) = F(\alpha_j) - p(\alpha_j) = F(\alpha_{j-1}), \quad for\ j = 2, 3, \ldots, n.$$

It is important to remember that the functions F, L and p are defined for the symbols in the alphabet $\mathscr{A}$. This means that $F(x)$ only makes sense if $x = \alpha_i$ for some i in the range $1 \leq i \leq n$.

The basic idea of arithmetic coding is to split the interval $[0, 1)$ into the n subintervals

$$\big[0, F(\alpha_1)\big), \quad \big[F(\alpha_1), F(\alpha_2)\big), \quad \ldots, \quad \big[F(\alpha_{n-2}), F(\alpha_{n-1})\big), \quad \big[F(\alpha_{n-1}), 1\big) \qquad (7.6)$$

so that the width of the subinterval $\big[F(\alpha_{i-1}), F(\alpha_i)\big)$ is $F(\alpha_i) - F(\alpha_{i-1}) = p(\alpha_i)$. If the first symbol is $x_1 = \alpha_i$, the arithmetic code must lie in the interval $[a_1, b_1)$ where

$$a_1 = p(\alpha_1) + p(\alpha_2) + \cdots + p(\alpha_{i-1}) = F(\alpha_{i-1}) = L(\alpha_i) = L(x_1),$$
$$b_1 = a_1 + p(\alpha_i) = F(\alpha_i) = F(x_1).$$

The next symbol in the text is x_2. If this were the first symbol of the text, the desired subinterval would be $\big[L(x_2), F(x_2)\big)$. Since it is the second symbol we must map the whole interval $[0, 1)$ to the interval $[a_1, b_1]$ and pick out the part that corresponds to $\big[L(x_2), F(x_2)\big)$. The mapping from $[0, 1)$ to $[a_1, b_1)$ is given by $g_2(z) = a_1 + z(b_1 - a_1) = a_1 + zp(x_1)$, see observation 7.17, so our new interval is

$$[a_2, b_2) = \Big[g_2\big(L(x_2)\big), g_2\big(F(x_2)\big)\Big] = \big[a_1 + L(x_2)p(x_1), a_1 + F(x_2)p(x_1)\big).$$

The third symbol x_3 would be associated with the interval $\big[L(x_3), F(x_3)\big)$ if it were the first symbol. To find the correct subinterval, we map $[0, 1)$ to $[a_2, b_2)$ with the mapping $g_3(z) = a_2 + z(b_2 - a_2)$ and pick out the correct subinterval as

$$[a_3, b_3) = \Big[g_3\big(L(x_3)\big), g_3\big(F(x_3)\big)\Big].$$

This process is then continued until all the symbols in the text have been processed.

With this background we can formulate a precise algorithm for arithmetic coding of a text of length m with n distinct symbols.

> **Algorithm 7.19** (Arithmetic coding). *Let the text $\mathbf{x} = \{x_1, \ldots, x_m\}$ be given, with the symbols being taken from an alphabet $\mathcal{A} = \{\alpha_1, \ldots, \alpha_n\}$, with probabilities $p(\alpha_i)$ for $i = 1, \ldots, n$. Generate a sequence of m subintervals of $[0, 1)$:*
>
> 1. *Set $[a_0, b_0] = [0, 1)$.*
>
> 2. *For $k = 1, \ldots, m$*
>
>   (a) *Define the linear function $g_k(z) = a_{k-1} + z(b_{k-1} - a_{k-1})$.*
>
>   (b) *Set $[a_k, b_k] = \left[g_k\big(L(x_k)\big), g_k\big(F(x_k)\big) \right)$.*
>
> *The arithmetic code of the text $\mathbf{x}$ is the midpoint $C(\mathbf{x})$ of the interval $[a_m, b_m)$, i.e., the number*
> $$\frac{a_m + b_m}{2},$$
> *truncated to*
> $$\left\lceil -\log_2\big(p(x_1)p(x_2)\cdots p(x_m)\big) \right\rceil + 1$$
> *binary digits. Here $\lceil w \rceil$ denotes the smallest integer that is larger than or equal to w.*

A program for arithmetic coding needs to output a bit more information than just the arithmetic code itself. For the decoding we also need to know exactly which probabilities were used and the ordering of the symbols (this influences the cumulative probability function). In addition we need to know when to stop decoding. A common way to provide this information is to store the length of the text. Alternatively, there must be a unique symbol that terminates the text so when we encounter this symbol during decoding we know that we are finished.

Let us consider another example of arithmetic coding in a situation with a three-symbol alphabet.

Example 7.20. Suppose we have the text $\mathbf{x} = \{ACBBCAABAA\}$ and we want to encode it with arithmetic coding. We first note that the probabilities are given by

$$p(A) = 0.5, \quad p(B) = 0.3, \quad p(C) = 0.2,$$

so the cumulative probabilities are $F(A) = 0.5$, $F(B) = 0.8$ and $F(C) = 1.0$. This means that the interval $[0, 1)$ is split into the three subintervals

$$[0, 0.5), \quad [0.5, 0.8), \quad [0.8, 1).$$

The first symbol is A, so the first subinterval is $[a_1, b_1) = [0, 0.5)$. The second symbol is C so we must find the part of $[a_1, b_1)$ that corresponds to C. The mapping from $[0, 1)$ to $[0, 0.5)$ is given by $g_2(z) = 0.5z$ so $[0.8, 1]$ is mapped to

$$[a_2, b_2) = \big[g_2(0.8), g_2(1)\big) = [0.4, 0.5).$$

The third symbol is B which corresponds to the interval $[0.5, 0.8)$. We map $[0, 1)$ to the interval $[a_2, b_2)$ with the function $g_3(z) = a_2 + z(b_2 - a_2) = 0.4 + 0.1z$ so $[0.5, 0.8)$ is mapped to

$$[a_3, b_3) = \big[g_3(0.5), g_3(0.8)\big) = [0.45, 0.48).$$

Let us now write down the rest of the computations more schematically in a table,

$$g_4(z) = 0.45 + 0.03z, \quad x_4 = B, \quad [a_4, b_4) = \big[g_4(0.5), g_4(0.8)\big) = [0.465, 0.474),$$
$$g_5(z) = 0.465 + 0.009z, \quad x_5 = C, \quad [a_5, b_5) = \big[g_5(0.8), g_5(1)\big) = [0.4722, 0.474),$$
$$g_6(z) = 0.4722 + 0.0018z, \quad x_6 = A, \quad [a_6, b_6) = \big[g_6(0), g_6(0.5)\big) = [0.4722, 0.4731),$$
$$g_7(z) = 0.4722 + 0.0009z, \quad x_7 = A, \quad [a_7, b_7) = \big[g_7(0), g_7(0.5)\big) = [0.4722, 0.47265),$$
$$g_8(z) = 0.4722 + 0.00045z, \quad x_8 = B, \quad [a_8, b_8) = \big[g_8(0.5), g_8(0.8)\big) = [0.472425, 0.47256),$$
$$g_9(z) = 0.472425 + 0.000135z, \quad x_9 = A,$$
$$[a_9, b_9) = \big[g_9(0), g_9(0.5)\big) = [0.472425, 0.4724925),$$
$$g_{10}(z) = 0.472425 + 0.0000675z, \quad x_{10} = A,$$
$$[a_{10}, b_{10}) = \big[g_{10}(0), g_{10}(0.5)\big) = [0.472425, 0.47245875).$$

The midpoint M of this final interval is

$$M = 0.472441875 = 0.0111100011111000111111_2,$$

and the arithmetic code is M rounded to

$$\left\lceil -\log_2\big(p(A)^5 p(B)^3 p(C)^2\big) \right\rceil + 1 = 16$$

bits. The arithmetic code is therefore the number

$$C(\boldsymbol{x}) = 0.0111100011110001_2 = 0.472427,$$

but we just store the 16 bits 0111100011110001. In this example the arithmetic code therefore uses 1.6 bits per symbol. In comparison the entropy is 1.49 bits per symbol. ∎

7.4.3   Properties of arithmetic coding

In example 7.16 we chose the arithmetic code to be the dyadic number with the smallest denominator within the interval $[a_m, b_m)$. In algorithm 7.19 we have chosen a number that is a bit easier to determine, but still we need to prove that the truncated number lies in the interval $[a_m, b_m)$. This is necessary because when we throw away some of the digits in the representation of the midpoint, the result may end up outside the interval $[a_m, b_m]$. We combine this with an important observation on the length of the interval.

Theorem 7.21. *The width of the interval $[a_m, b_m)$ is*

$$b_m - a_m = p(x_1)p(x_2)\cdots p(x_m) \tag{7.7}$$

and the arithmetic code $C(\boldsymbol{x})$ lies inside this interval.

Proof. The proof of equation(7.7) is by induction on m. For $m = 1$, the length is simply $b_1 - a_1 = F(x_1) - L(x_1) = p(x_1)$, which is clear from the last equation in Definition 7.18. Suppose next that

$$b_{k-1} - a_{k-1} = p(x_1)\cdots p(x_{k-1});$$

we need to show that $b_k - a_k = p(x_1)\cdots p(x_k)$. This follows from step 2 of algorithm 7.19,

$$\begin{aligned}
b_k - a_k &= g_k\big(F(x_k)\big) - g_k\big(L(x_k)\big) \\
&= \big(F(x_k) - L(x_k)\big)(b_{k-1} - a_{k-1}) \\
&= p(x_k)p(x_1)\cdots p(x_{k-1}).
\end{aligned}$$

In particular we have $b_m - a_m = p(x_1)\cdots p(x_m)$.

Our next task is to show that the arithmetic code $C(\boldsymbol{x})$ lies in $[a_m, b_m)$. Define the number μ by the relation

$$\frac{1}{2^\mu} = b_m - a_m = p(x_1)\cdots p(x_m) \quad \text{or} \quad \mu = -\log_2\big(p(x_1)\cdots p(x_m)\big).$$

In general μ will not be an integer, so we introduce a new number λ which is the smallest integer that is greater than or equal to μ,

$$\lambda = \lceil\mu\rceil = \left\lceil -\log_2\big(p(x_1)\cdots p(x_m)\big)\right\rceil.$$

This means that $1/2^\lambda$ is smaller than or equal to $b_m - a_m$ since $\lambda \ge \mu$. Consider the collection of dyadic numbers D_λ on the form $j/2^\lambda$ where j is an integer in
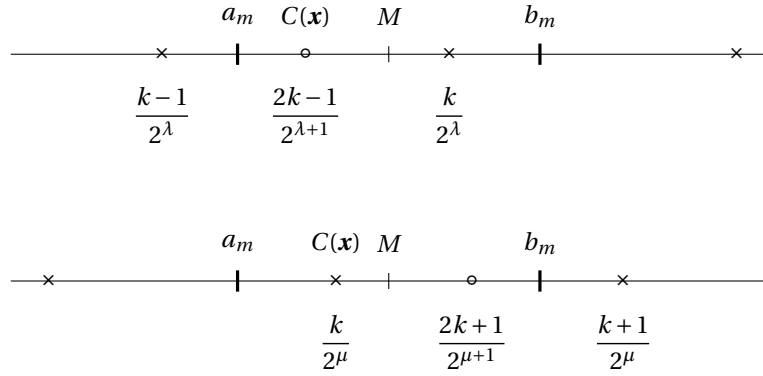
Figure 7.6. The two situations that can occur when determining the number of bits in the arithmetic code.

the range $0 \le j < 2^\lambda$. At least one of them, say $k/2^\lambda$, must lie in the interval $[a_m, b_m)$ since the distance between neighbouring numbers in D_λ is $1/2^\lambda$ which is at most equal to $b_m - a_m$. Denote the midpoint of $[a_m, b_m)$ by M. There are two situations to consider which are illustrated in figure 7.6.

In the first situation shown in the top part of the figure, the number $k/2^\lambda$ is larger than M and there is no number in D_λ in the interval $[a_m, M]$. If we form the approximation $\tilde{M}$ to M by only keeping the first λ binary digits, we obtain the number $(k-1)/2^\lambda$ in D_λ that is immediately to the left of $k/2^\lambda$. This number may be smaller than a_m, as shown in the figure. To make sure that the arithmetic code ends up in $[a_m, b_m)$ we therefore use one more binary digit and set $C(\boldsymbol{x}) = (2k-1)/2^{\lambda+1}$, which corresponds to keeping the first $\lambda+1$ binary digits in M.

In the second situation there is a number from D_λ in $[a_m, M]$ (this was the case in example 7.16). If we now keep the first λ digits in M we would get $C(\boldsymbol{x}) = k/2^\lambda$. In this case algorithm 7.19 therefore gives an arithmetic code with one more bit than necessary. In practice the arithmetic code will usually be at least thousands of bits long, so an extra bit does not matter much. ∎

Now that we know how to compute the arithmetic code, it is interesting to see how the number of bits per symbol compares with the entropy. The number of bits is given by

$$\left\lceil -\log_2\big(p(x_1)p(x_2)\cdots p(x_m)\big)\right\rceil + 1.$$

Recall that each x_i is one of the n symbols α_i from the alphabet so by properties of logarithms we have

$$\log_2\bigl(p(x_1)p(x_2)\cdots p(x_m)\bigr) = \sum_{i=1}^{n} f(\alpha_i)\log_2 p(\alpha_i)$$

where $f(\alpha_i)$ is the number of times that α_i occurs in $\mathbf{x}$. As m becomes large we know that $f(\alpha_i)/m$ approaches $p(\alpha_i)$. For large m we therefore have that the number of bits per symbol approaches

$$\frac{1}{m}\Bigl[-\log_2\bigl(p(x_1)p(x_2)\cdots p(x_m)\bigr)\Bigr] + \frac{1}{m} \leq -\frac{1}{m}\log_2\bigl(p(x_1)p(x_2)\cdots p(x_m)\bigr) + \frac{2}{m}$$

$$= -\frac{1}{m}\sum_{i=1}^{n} f(\alpha_i)\log_2 p(\alpha_i) + \frac{2}{m}$$

$$\approx -\sum_{i=1}^{n} p(\alpha_i)\log_2 p(\alpha_i)$$

$$= H(p_1,\ldots,p_n).$$

In other words, arithmetic coding gives compression rates close to the best possible for long texts.

> **Corollary 7.22.** *For long texts the number of bits per symbol required by the arithmetic coding algorithm approaches the minimum given by the entropy, provided the probability distribution of the symbols is correct.*

7.4.4 A decoding algorithm

We commented briefly on decoding at the end of section 7.4.1. In this section we will give a detailed decoding algorithm similar to algorithm 7.19.

We will need the linear function that maps an interval $[a, b]$ to the interval $[0, 1]$, i.e., the inverse of the function in observation 7.17.

> **Observation 7.23.** *Let $[a, b]$ be a given interval with $a < b$. The function*
>
> $$h(y) = \frac{y - a}{b - a}$$
>
> *will map any number y in $[a, b]$ to the interval $[0, 1]$. In particular the endpoints are mapped to the endpoints and the midpoint to the midpoint,*
>
> $$h(a) = 0, \quad h\bigl((a + b)/2\bigr) = 1/2, \quad h(b) = 1.$$

Linear functions like h in observation 7.23 play a similar role in decoding as the g_ks in algorithm 7.19; they help us avoid working with very small intervals. The decoding algorithm assumes that the number of symbols in the text is known and decodes the arithmetic code symbol by symbol. It stops when the correct number of symbols have been found.

Algorithm 7.24. *Let $C(\boldsymbol{x})$ be a given arithmetic code of an unknown text $\boldsymbol{x}$ of length m, based on an alphabet $\mathscr{A} = \{\alpha_1, \ldots, \alpha_n\}$ with known probabilities $p(\alpha_i)$ for $i = 1, \ldots, n$. The following algorithm determines the symbols of the text $\boldsymbol{x} = \{x_1, \ldots, x_m\}$ from the arithmetic code $C(\boldsymbol{x})$:*

  1. *Set $z_1 = C(\boldsymbol{x})$.*

  2. *For $k = 1, \ldots, m$*

      (a) *Find the integer i such that $L(\alpha_i) \le z_k < F(\alpha_i)$ and set*
      $$[a_k, b_k) = \big[L(\alpha_i), F(\alpha_i)\big).$$

      (b) *Output $x_k = \alpha_i$.*

      (c) *Determine the linear function $h_k(y) = (y - a_k)/(b_k - a_k)$.*

      (d) *Set $z_{k+1} = h_k(z_k)$.*

The algorithm starts by determining which of the n intervals

$$\big[0, F(\alpha_1)\big), \quad \big[F(\alpha_1), F(\alpha_2)\big), \ldots, \big[F(\alpha_{n-2}), F(\alpha_{n-1})\big), \quad \big[F(\alpha_{n-1}), 1\big)$$

it is that contains the arithmetic code $z_1 = C(\boldsymbol{x})$. This requires a search among the cumulative probabilities. When the index i of the interval is known, we know that $x_1 = \alpha_i$. The next step is to decide which subinterval of $[a_1, b_1) = \big[L(\alpha_i), F(\alpha_i)\big)$ that contains the arithmetic code. If we stretch this interval out to $[0, 1)$ with the function h_k, we can identify the next symbol just as we did with the first one. Let us see how this works by decoding the arithmetic code that we computed in example 7.16.

Example 7.25 (Decoding of an arithmetic code)**.** Suppose we are given the arithmetic code 1001 from example 7.16 together with the probabilities $p(0) = 0.8$ and $p(1) = 0.2$. We also assume that the length of the code is known, the probabilities, and how the probabilities were mapped into the interval $[0, 1]$; this is the typical output of a program for arithmetic coding. Since we are going to do this

manually, we start by converting the number to decimal; if we were to program arithmetic coding we would do everything in binary arithmetic.

The arithmetic code 1001 corresponds to the binary number 0.1001_2 which is the decimal number $z_1 = 0.5625$. Since this number lies in the interval $[0, 0.8)$ we know that the first symbol is $x_1 = 0$. We now map the interval $[0, 0.8)$ and the code back to the interval $[0, 1)$ with the function

$$h_1(y) = y/0.8.$$

We find that the code becomes

$$z_2 = h_1(z_1) = z_1/0.8 = 0.703125$$

relative to the new interval. This number lies in the interval $[0, 0.8)$ so the second symbol is $x_2 = 0$. Once again we map the current interval and arithmetic code back to $[0, 1)$ with the function h_2 and obtain

$$z_3 = h_2(z_2) = z_2/0.8 = 0.87890625.$$

This number lies in the interval $[0.8, 1)$, so our third symbol must be a $x_3 = 1$. At the next step we must map the interval $[0.8, 1)$ to $[0, 1)$. From observation 7.23 we see that this is done by the function $h_3(y) = (y - 0.8)/0.2$. This means that the code is mapped to

$$z_4 = h_3(z_3) = (z_3 - 0.8)/0.2 = 0.39453125.$$

This brings us back to the interval $[0, 0.8)$, so the fourth symbol is $x_4 = 0$. This time we map back to $[0, 1)$ with the function $h_4(y) = y/0.8$ and obtain

$$z_5 = h_4(z_4) = 0.39453125/0.8 = 0.493164.$$

Since we remain in the interval $[0, 0.8)$ the fifth and last symbol is $x_5 = 0$, so the original text was '00100'. ∎

7.4.5 Arithmetic coding in practice

Algorithms 7.19 and 7.24 are quite simple and appear to be easy to program. However, there is one challenge that we have not addressed. The typical symbol sequences that we may want to compress are very long, with perhaps millions or even billions of symbols. In the coding process the intervals that contain the arithmetic code become smaller for each symbol that is processed which means that the ends of the intervals must be represented with extremely high precision. A program for arithmetic coding must therefore be able to handle arbitrary precision arithmetic in an efficient way. For a time this prevented the method from

153

being used, but there are now good algorithms for handling this. The basic idea is to organise the computations of the endpoints of the intervals in such a way that early digits are not influenced by later ones. It is then sufficient to only work with a limited number of digits at a time (for example 32 or 64 binary digits). The details of how this is done is rather technical though.

Since the compression rate of arithmetic coding is close to the optimal rate predicted by the entropy, one would think that it is often used in practice. However, arithmetic coding is protected by many patents which means that you have to be careful with the legal details if you use the method in commercial software. For this reason, many prefer to use other compression algorithms without such restrictions, even though these methods may not perform quite so well.

In long texts the frequency of the symbols may vary within the text. To compensate for this it is common to let the probabilities vary. This does not cause problems as long as the coding and decoding algorithms compute and adjust the probabilities in exactly the same way.

7.5   Lempel-Ziv-Welch algorithm

The Lempel-Ziv-Welch algorithm is named after the three inventors and is usually referred to as the LZW algorithm. The original idea is due to Lempel and Ziv and is used in the LZ77 and LZ78 algorithms.

LZ78 constructs a *code book* during compression, with entries for combinations of several symbols as well as for individual symbols. If, say, the ten next symbols already have an entry in the code book as individual symbols, a new entry is added to represent the combination consisting of these next ten symbols. If this same combination of ten symbols appears later in the text, it can be represented by its code.

The LZW algorithm is based on the same idea as LZ78, with small changes to improve compression further.

LZ77 does not store a list of codes for previously encountered symbol combinations. Instead it searches previous symbols for matches with the sequence of symbols that are presently being encoded. If the next ten symbols match a sequence 90 symbols earlier in the symbol sequence, a code for the pair of numbers $(90, 10)$ will be used to represent these ten symbols. This can be thought of as a type of run-length coding.

7.6   Lossless compression programs

Lossless compression has become an important ingredient in many different contexts, often coupled with a lossy compression strategy. We will discuss this

in more detail in the context of digital sound and images in later chapters, but want to mention two general-purpose programs for lossless compression here.

7.6.1 Compress

The program `compress` is a much used compression program on UNIX platforms which first appeared in 1984. It uses the LZW-algorithm. After the program was published it turned out that part of the algorithm was covered by a patent.

7.6.2 gzip

To avoid the patents on `compress`, the alternative program `gzip` appeared in 1992. This program is based on the LZ77 algorithm, but uses Huffman coding to encode the pairs of numbers. Although `gzip` was originally developed for the Unix platform, it has now been ported to most operating systems, see `www.gzip.org`.

Exercises

7.1 In this exercise we are going to use Huffman coding to encode the text 'There are many people in the world', including the spaces.

   a) Compute the frequencies of the different symbols used in the text.

   b) Use algorithm 7.7 to determine the Huffman tree for the symbols.

   c) Determine the Huffman coding of the complete text. How does the result compare with the entropy?

7.2 We can generalise Huffman coding to numeral systems other than the binary system.

   a) Suppose we have a computer that works in the ternary (base-3) numeral system; describe a variant of Huffman coding for such machines.

   b) Generalise the Huffman algorithm so that it produces codes in the base-n numeral system.

7.3 In this exercise we are going to do Huffman coding for the text given by $x = \{ABACABCA\}$.

   a) Compute the frequencies of the symbols, perform the Huffman algorithm and determine the Huffman coding. Compare the result with the entropy.

   b) Change the frequencies to $f(A) = 1$, $f(B) = 1$, $f(C) = 2$ and compare the Huffman tree with the one from (a).

7.4 Recall from section 4.3.1 in chapter 4 that ASCII encodes the 128 most common symbols used in English with seven-bit codes. If we denote the alphabet by $\mathscr{A} = \{\alpha_i\}_{i=1}^{128}$, the codes are

$$c(\alpha_1) = 0000000, \quad c(\alpha_2) = 0000001, \quad c(\alpha_3) = 0000010, \quad \dots$$

$$c(\alpha_{127}) = 1111110, \quad c(\alpha_{128}) = 1111111.$$

Explain how these codes can be associated with a certain Huffman tree. What are the frequencies used in the construction of the Huffman tree?

7.5 In this exercise we use the two-symbol alphabet $\mathscr{A} = \{A, B\}$.

   a) Compute the frequencies $f(A)$ and $f(B)$ in the text

   $$\boldsymbol{x} = \{AAAAAAABAA\}$$

   and the probabilities $p(A)$ and $p(B)$.

   b) We want to use arithmetic coding to compress the sequence in (a); how many bits do we need in the arithmetic code?

   c) Compute the arithmetic code of the sequence in (a).

7.6 The four-symbol alphabet $\mathscr{A} = \{A, B, C, D\}$ is used throughout this exercise. The probabilities are given by $p(A) = p(B) = p(C) = p(D) = 0.25$.

   a) Compute the information entropy for this alphabet with the given probabilities.

   b) Construct the Huffman tree for the alphabet. How many bits per symbol is required if you use Huffman coding with this alphabet?

   c) Suppose now that we have a text $\boldsymbol{x} = \{x_1, \ldots, x_m\}$ consisting of m symbols taken from the alphabet $\mathscr{A}$. We assume that the frequencies of the symbols correspond with the probabilities of the symbols in the alphabet.

   How many bits does arithmetic coding require for this sequence and how many bits per symbol does this correspond to?

   d) The Huffman tree you obtained in (b) is not unique. Here we will fix a tree so that the Huffman codes are

   $$c(A) = 00, \quad c(B) = 01, \quad c(C) = 10, \quad c(D) = 11.$$

   Compute the Huffman coding of the sequence 'ACDBAC'.

   e) Compute the arithmetic code of the sequence in (d). What is the similarity with the result obtained with Huffman coding in (d)?

7.7 The three-symbol alphabet $\mathscr{A} = \{A, B, C\}$ with probabilities $p(A) = 0.1$, $p(B) = 0.6$ and $p(C) = 0.3$ is given. A text $\boldsymbol{x}$ of length 10 has been encoded by arithmetic coding and the code is 1001101. What is the text $\boldsymbol{x}$?

7.8 We have the two-symbol alphabet $\mathscr{A} = \{A, B\}$ with $p(A) = 0.99$ and $p(B) = 0.01$. Find the arithmetic code of the text

$$\overbrace{AAA \cdots AAA}^{99 \text{ times}} B.$$

7.9 The two linear functions in observations 7.17 and 7.23 are special cases of a more general construction. Suppose we have two nonempty intervals $[a, b]$ and $[c, d]$, find the linear function which maps $[a, b]$ to $[c, d]$.

Check that your solution is correct by comparing with observations 7.17 and 7.23.

CHAPTER 8

Audio compression in practice

In earlier chapters we have seen that digital sound is simply an array of numbers, where each number is a measure of the air pressure at a particular time. This representation makes it simple to manipulate sound on a computer by doing calculations with these numbers. We have seen examples of simple filters which change the sound in some preferred way like removing high or low frequencies or adding echo. We have also seen how an audio signal can be rewritten in terms of a multiresolution analysis which may be advantageous if we want to compress the data that represent the sound.

In this chapter we will very briefly review some practical aspects of sound processing. In particular we will briefly discuss how a sound may be represented in terms of trigonometric functions and review the most important digital audio formats.

8.1 Wavelet based compression

In chapter 10 in the Norwegian lecture notes, we described a way to decompose a set of data points by comparing representations at different resolutions with the help of piecewise linear functions, a so-called *multiresolution analysis*. This illustrated the idea behind a family of functions called *wavelets* (although wavelets should have some additional properties that we have ignored to keep the presentation simple). In this section we will try to compress sound with the multiresolution analysis.

Example 8.1. Figure 8.1 shows an example with real audio data. The data in (a) were decomposed once which resulted in the coarser signal in (b) and the error signal in (c). Then all error values smaller than 0.02 set to zero which leads to the signal in (d). The reconstruction algorithm was then run with the data in (b) and (d); the result is shown in (e). Figure (f) shows the difference between the two signals.

The same computations were performed with a larger data set consisting of 300 001 points from the same song. Out of 150 000 error values 141 844 were smaller than 0.02 in absolute value. When these were set to 0 and the sound reconstructed with the perturbed data, the music still sounded quite good. The
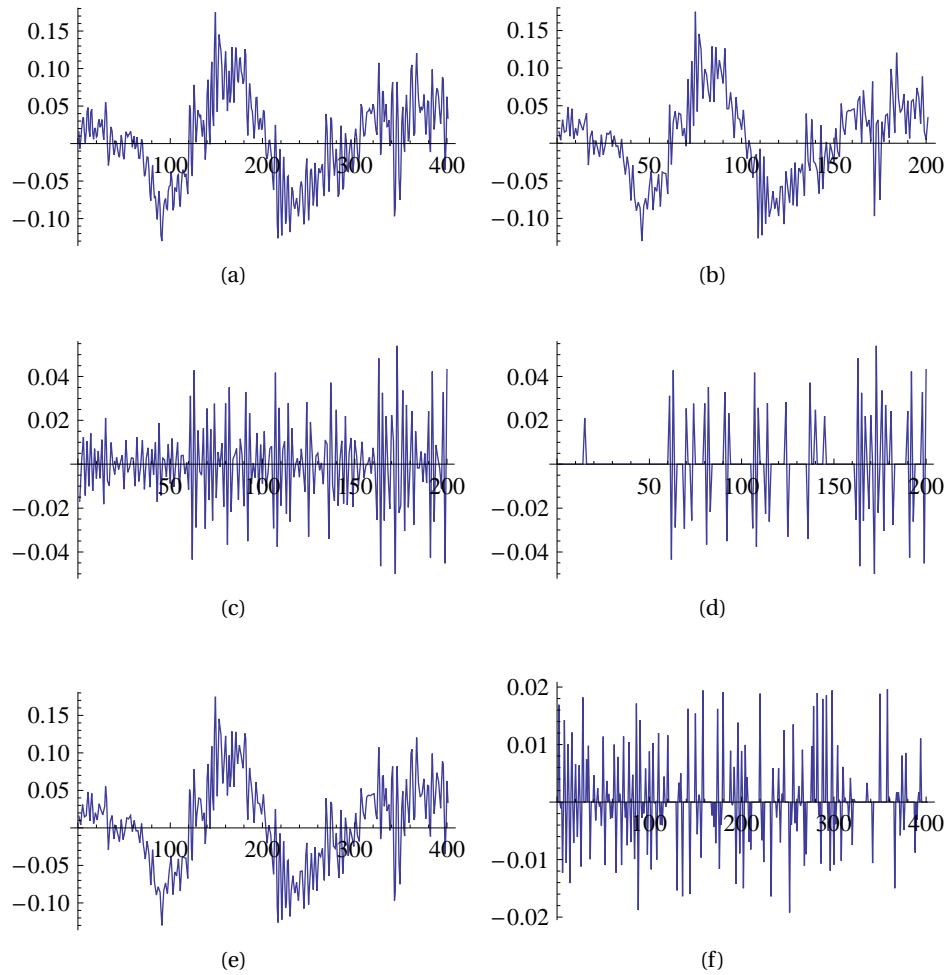
Figure 8.1. An example of data decomposition with the multiresolution analysis from chapter 10 of the Norwegian lecture notes. Figure (a) shows the piecewise linear interpolant to data taken from the song 'Here Comes the Sun' by the Beatles. Figure (b) shows the piecewise linear interpolant to every other data point, and figure (c) shows the error function. In (d) we have set all error values smaller than 0.02 to zero, altogether 148 out of 200 values. In (e) the data have been reconstructed from the functions in (c) and (d). Figure (f) shows the difference between the original data and the reconstructed data.

maximum difference between this perturbed signal and the original was just under 0.02.

To check the possibility of compression, the data were written to a file as 32-bit floating-point numbers and then compressed with gzip. This resulted in a file with 389 829 bytes. When the original data were written to file and com-

(a)

(b)
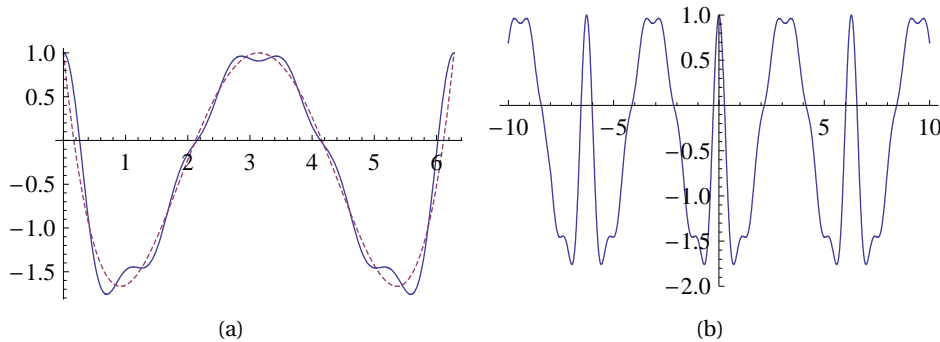
Figure 8.2. Trigonometric approximation of a cubic polynomial on the interval $[0, 2\pi]$. In (a) both functions are shown while in (b) the approximation is plotted on the interval $[-10, 10]$.

pressed with `gzip`, the file size was 705 475 bytes. In other words, the sound has been compressed down to 55 % with a very simple algorithm, without too much deterioration in the sound quality. ■

8.2 Fourier analysis and the DCT

As we have seen, multiresolution analysis and wavelets provide a convenient framework for compression of audio files. The theory behind wavelets was developed in the late 1980s and was not introduced into much commercial software until the late 1990s. Common audio formats like MP3, AAC and Ogg Vorbis were developed before this time and use a mathematical operation called the *Discrete Cosine Transform* (DCT) to transform the audio data to a form that lends itself well to compression. Before we consider the basics of the DCT, we need some background information.

A very common technique in mathematics is to approximate a general function by a function from some appropriate family. We have seen several examples of this, for example approximation by Taylor polynomials and approximation by piecewise linear functions. Another possibility is to approximate functions by combinations of trigonometric functions. This is a technique that is used in many different areas of mathematics and is usually referred to as *Fourier analysis*.

In Fourier analysis, general functions are approximated by combinations of the functions

$$1, \sin x, \cos x, \sin 2x, \cos 2x, \sin 3x, \cos 3x, \sin 4x, \cos 4x, \dots \qquad (8.1)$$

A simple example is shown in figure 8.2. There a cubic polynomial has been

159

approximated by a trigonometric function on the form

$$a_0 + b_1 \sin x + a_1 \cos x + b_2 \sin 2x + a_2 \cos 2x. \tag{8.2}$$

The coefficients have been determined in such a way that the error should be small on the interval $[0, 1]$. This has been quite successful in that we can hardly differentiate between the two functions in figure 8.2a. In figure 8.2b the approximation has been plotted on the interval $[-10, 10]$. From this plot we see that the approximation is periodic, as is the case for all trigonometric functions, and therefore very different from the polynomial as we come outside the interval $[0, 1]$. The fact that the error was small in figure 8.2a was no accident; it can be proved that quite general functions can be approximated arbitrarily well by trigonometric functions.

Recall that $\sin kt$ corresponds to a pure tone of frequency $k/(2\pi)$. If we think of a certain sound as a function, the fact that any function can be expressed as a combination of the trigonometric functions in (8.1) means that any sound can be generated by combining a set of pure tones with different weights, just as in (8.2). So given a sound, it can always de decomposed into different amounts of pure tones be represented by suitable trigonometric functions.

Fact 8.2 (Fourier analysis). *Any reasonable function f can be approximated arbitrarily well on the interval $[0, 2\pi]$ by a combination*

$$a_0 + \sum_{k=1}^{N} (a_k \cos kt + b_k \sin kt), \tag{8.3}$$

where N is a suitable integer. This means that any sound can be decomposed into different amounts (the coefficients $\{a_k\}$ and $\{b_k\}$) of pure tones (the functions $\{\cos kt\}$ and $\{\sin kt\}$). This is referred to by saying that f has been converted to the frequency domain and the coefficients $\{a_k\}$ and $\{b_k\}$ are referred to as the spectrum of f.

When a given sound has been decomposed as in (8.3), we can adjust the sound by adjusting the coefficients. The trigonometric functions with large values of k correspond to high frequencies and the ones with small values of k correspond to low frequencies. We can therefore adjust the treble and bass by manipulating the appropriate coefficients.

There is an extensive theory for audio processing based on Fourier analysis that is studied in the field of *signal processing*.

8.2.1   The Discrete Cosine Transform

We are particularly interested in digital audio, i.e., sound given by values sampled from an audio signal at regular intervals. We have seen how digital sound can be compressed by using a multiresolution analysis, but this can also be done with ideas from Fourier analysis.

For the purpose of compressing sound it is not so common to use Fourier analysis as indicated above. Instead only cosine functions are used for decomposition.

Definition 8.3 (Discrete Cosine Transform (DCT)). *Suppose the sequence of numbers $\boldsymbol{u} = \{u_s\}_{s=0}^{n-1}$ are given. The DCT of $\boldsymbol{u}$ is given by the sequence*

$$v_s = \frac{1}{\sqrt{n}} \sum_{r=0}^{n-1} u_r \cos\left(\frac{(2r+1)s\pi}{2n}\right), \quad \textit{for } s = 0, \ldots, n-1. \tag{8.4}$$

The new sequence $\boldsymbol{v}$ generated by the DCT tells us how much the sequence $\boldsymbol{u}$ contains of the different frequencies. For each $s = 0, 1, \ldots, n-1$, the function $\cos s\pi t$ is sampled at the points $t_r = (2r+1)/(2n)$ for $r = 0, 1, \ldots, n-1$ which results in the values

$$\cos\left(\frac{s\pi}{2n}\right), \quad \cos\left(\frac{3s\pi}{2n}\right), \quad \cos\left(\frac{5s\pi}{2n}\right), \quad \ldots, \quad \cos\left(\frac{(2n-1)s\pi}{2n}\right).$$

These are then multiplied by the u_r and everything is added together.

Plots of these values for $n = 6$ are shown in figure 8.3. We note that as s increases, the functions oscillate more and more. This means that v_0 gives a measure of how much constant content there is in the data, while v_5 gives a measure of how much content there is with maximum oscillation. In other words, the DCT of an audio signal shows the proportion of the different frequencies in the signal.

Once the DCT of $\boldsymbol{u}$ has been computed, we can analyse the frequency content of the signal. If we want to reduce the bass we can decrease the v_s-values with small indices and if we want to increase the treble we can increase the v_s-values with large indices. In a typical audio signal there will be most information in the lower frequencies, and some frequencies will be almost completely absent, i.e., some of the v_s-values will be almost zero. This can exploited for compression: We change the small v_s-values a little bit and set them to 0, and then store the signal by storing the DCT-values. This leaves us with one challenge: How to convert the perturbed DCT-values back to a normal signal. It turns out that this can be accomplished with formulas very similar to the DCT itself.
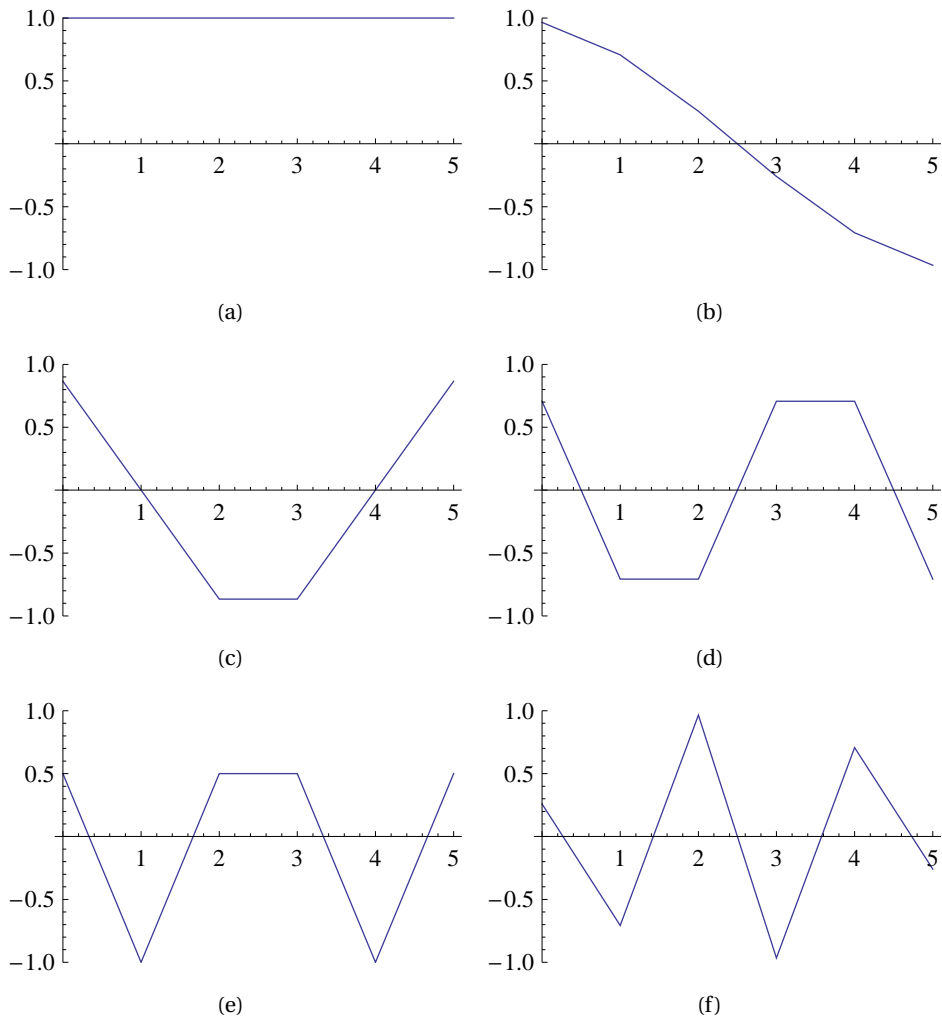
Figure 8.3. The 6 different versions of the cos function used in DCT for $n = 6$. The plots show piecewise linear functions, but this is just to make the plots more readable: Only the values at the integers $0, \ldots, 5$ are used.

Theorem 8.4 (Inverse Discrete Cosine Transform)**.** *Suppose that the sequence $\boldsymbol{v} = \{v_s\}_{s=0}^{n-1}$ is the DCT of the sequence $\boldsymbol{u} = \{u_r\}_{r=0}^{n-1}$ as in (8.4). Then $\boldsymbol{u}$ can be recovered from $\boldsymbol{v}$ via the formulas*

$$u_r = \frac{1}{\sqrt{n}}\left(v_0 + 2\sum_{s=1}^{n-1} v_s \cos\left(\frac{(2r+1)s\pi}{2n}\right)\right), \quad \textit{for } r = 0, \ldots, n-1. \tag{8.5}$$

162
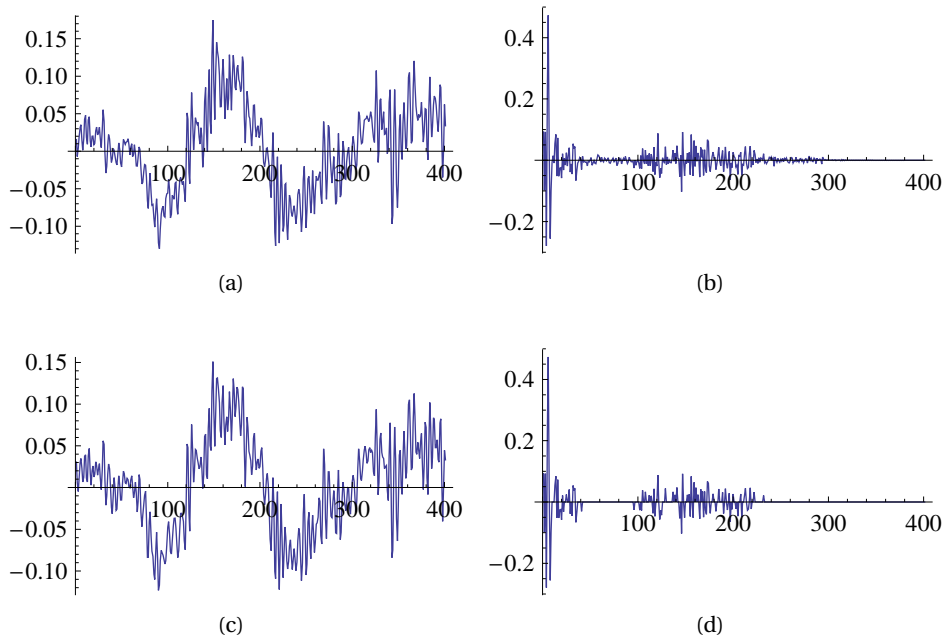
Figure 8.4. The signal in (a) is a small part of a song (the same as in figure 8.1b). The plot in (b) shows the DCT of the signal. In (c), all values of the DCT that are smaller than 0.02 in absolute value have been set to 0, a total of 309 values. In (c) the signal has been reconstructed from these perturbed values of the DCT. Note that all signals are discrete; the values have been connected by straight lines to make it easier to interpret the plots.

We now see the resemblance with multiresolution analysis. The DCT provides another way to rewrite a signal in an alternative form where many values become small or even 0, and this can then be exploited for compression.

Example 8.5. Let us test a naive compression strategy similar to the one in example 8.1 where we just replace the multiresolution decomposition and reconstruction formulas with the DCT and its inverse. The plots in figure 8.4 illustrate the principle. A signal is shown in (a) and its DCT in (b). In (d) all values of the DCT with absolute value smaller than 0.02 have been set to zero. The signal can then be reconstructed with the inverse DCT of theorem 8.4; the result of this is shown in (c). The two signals in (a) and (b) visually look almost the same even though the signal in (c) can be represented with less than 25 % of the information present in (a).

The larger data set from example 8.1 consists of 300 001 points. We compute the DCT and set all values smaller than a suitable tolerance to 0. It turns out that we need to use a tolerance as small as 0.004 to get an error in the signal itself that is comparable to the corresponding error in example 8.1. With a tolerance

of 0.04, a total of 142 541 values are set to zero. When we then reconstruct the sound with the inverse DCT, we obtain a signal that differs at most 0.019 from the original signal which is about the same as in example 8.1. We can store the signal by storing a `gzip`'ed version of the DCT-values (as 32-bit floating-point numbers) of the perturbed signal. This gives a file with 622 551 bytes, which is 88 % of the `gzip`'ed version of the original data.

Recall that when we used the wavelet transform in example 8.1, we ended up with a file size of 389 829, which is considerably less than what we obtained here, without the error being much bigger. From this simple example, wavelets therefore seem promising for compression of audio. ∎

The approach to compression that we have outlined in the above examples is essentially what is used in practice. The difference is that commercial software does everything in a more sophisticated way and thereby get better compression rates.

> **Fact 8.6** (Basic idea behind audio compression). *Suppose a digital audio signal u is given. To compress u, perform the following steps:*
>
> 1. *Rewrite the signal u in a new format where frequency information becomes accessible.*
>
> 2. *Remove those frequencies that only contribute marginally to human perception of the sound.*
>
> 3. *Store the resulting sound by coding the remaining frequency information with some lossless coding method.*

All the compression strategies used in the commercial formats that we review below, use the strategy in fact 8.6. In fact they all use a modified version of the DCT in step 1 and a variant of Huffman coding in step 3. Where they vary the most is probably in deciding what information to remove from the signal. To do this well requires some knowledge of human perception of sound.

8.3 Psycho-acoustic models

In the previous sections, we have outlined a simple strategy for compressing sound. The idea is to rewrite the audio signal in an alternative mathematical representation where many of the values are small, set the smallest values to 0, store this perturbed signal and compress it with a lossless compression method.

This kind of compression strategy works quite well, and is based on keeping the difference between the original signal and the compressed signal small. However, in certain situations a listener will not be able to perceive the sound as being different even if this difference is quite large. This is due to how our auditory system interprets audio signals and is referred to as *psycho-acoustic* effects.

When we hear a sound, there is a mechanical stimulation of the ear drum, and the amount of stimulus is directly related to the size of the sample values of the digital sound. The movement of the ear drum is then converted to electric impulses that travel to the brain where they are perceived as sound. The perception process uses a Fourier-like transformation of the sound so that a steady oscillation in air pressure is perceived as a sound with a fixed frequency. In this process certain kinds of perturbations of the sound are hardly noticed by the brain, and this is exploited in lossy audio compression.

The most obvious psycho-acoustic effect is that the human auditory system can only perceive frequencies in the range 20 Hz – 20 000 Hz. An obvious way to do compression is therefore to remove frequencies outside this range, although there are indications that these frequencies may influence the listening experience inaudibly.

Another phenomenon is *masking effects*. A simple example of this is that a loud sound will make a simultaneous quiet sound inaudible. For compression this means that if certain frequencies of a signal are very prominent, most of the other frequencies can be removed, even when they are quite large.

These kinds of effects are integrated into what is referred to as a *psycho-acoustic* model. This model is then used as the basis for simplifying the spectrum of the sound in way that is hardly noticeable to a listener, but which allows the sound to be stored with must less information than the original.

8.4 Digital audio formats

Digital audio first became commonly available when the CD was introduced in the early 1980s. As the storage capacity and processing speeds of computers increased, it became possible to transfer audio files to computers and both play and manipulate the data. However, audio was represented by a large amount of data and an obvious challenge was how to reduce the storage requirements. Lossless coding techniques like Huffman and Lempel-Ziv coding were known and with these kinds of techniques the file size could be reduced to about half of that required by the CD format. However, by allowing the data to be altered a little bit it turned out that it was possible to reduce the file size down to about ten percent of the CD format, without much loss in quality.

In this section we will give a brief description of some of the most common

165

digital audio formats, both lossy and lossless ones.

8.4.1 Audio sampling — PCM

The basis for all digital sound is sampling of an analog (continuous) audio signal. This is usually done with a technique called Pulse Code Modulation (PCM). The audio signal is sampled at regular intervals and the sampled values stored in a suitable number format. Both the sampling rate and the number format varies for different kinds of audio. For telephony it is common to sample the sound 8000 times per second and represent each sample value as a 13-bit integer. These integers are then converted to a kind of 8-bit floating-point format with a 4-bit mantissa. Telephony therefore generates 64 000 bits per second.

The classical CD-format samples the audio signal 44 100 times per second and stores the samples as 16-bit integers. This works well for music with a reasonably uniform dynamic range, but is problematic when the range varies. Suppose for example that a piece of music has a very loud passage. In this passage the samples will typically make use of almost the full range of integer values, from $-2^{15} - 1$ to 2^{15}. When the music enters a more quiet passage the sample values will necessarily become much smaller and perhaps only vary in the range -1000 to 1000, say. Since $2^{10} = 1024$ this means that in the quiet passage the music would only be represented with 10-bit samples. This problem can be avoided by using a floating-point format instead, but very few audio formats appear to do this.

Newer formats with higher quality are available. Music is distributed in various formats on DVDs (DVD-video, DVD-audio, Super Audio CD) with sampling rates up to 192 000 and up to 24 bits per pixel. These formats also support surround sound (up to seven channels as opposed to the two stereo channels on CDs).

Both the number of samples per second and the number of bits per sample influence the quality of the resulting sound. For simplicity the quality is often measured by the number of bits per second, i.e., the product of the sampling rate and the number of bits per sample. For standard telephony we saw that the bit rate is 64000 bits per second or 64 kb/s. The bit rate for CD-quality stereo sound is $44100 \times 2 \times 16$ bits/s = 1411.2 kb/s. This quality measure is particularly popular for lossy audio formats where the uncompressed audio usually is the same (CD-quality). However, it should be remembered that even two audio files in the same file format and with the same bit rate may be of very different quality because the encoding programs me be of different quality.

All the audio formats mentioned so far can be considered raw formats; it is a description of how the sound is digitised. When the information is stored on a computer, the details of how the data is organised must be specified, and there

are several popular formats for this.

8.4.2   Lossless formats

The two most common file formats for CD-quality audio are AIFF and WAV, which are both supported by most commercial audio programs. These formats specify in detail how the audio data should be stored in a file. In addition, there is support for including the title of the audio piece, album and artist name and other relevant data. All the other audio formats below (including the lossy ones) also have support for this kind of additional information.

AIFF.   *Audio Interchange File Format* was developed by Apple and published in 1988. AIFF supports different sample rates and bit lengths, but is most commonly used for storing CD-quality audio at 44 100 samples per second and 16 bits per sample. No compression is applied to the data, but there is also a variant that supports lossless compression, namely AIFF-C.

WAV.   *Waveform audio data* is a file format developed by Microsoft and IBM. As AIFF, it supports different data formats, but by far the most common is standard CD-quality sound. WAV uses a 32-bit integer to specify the file size at the beginning of the file which means that a WAV-file cannot be larger than 4 GB. Microsoft therefore developed the W64 format to remedy this.

Apple Lossless.   After Apple's iPods became popular, the company in 2004 introduced a lossless compressed file format called Apple Lossless. This format is used for reducing the size of CD-quality audio files. Apple has not published the algorithm behind the Apple Lossless format, but most of the details have been worked out by programmers working on a public decoder. The compression phase uses a two step algorithm:

1. When the nth sample value x_n is reached, an approximation y_n to x_n is computed, and the error $e_n = x_n - y_n$ is stored instead of x_n. In the simplest case, the approximation y_n would be the previous sample value x_{n-1}; better approximations are obtained by computing y_n as a combination of several of the previous sample values.

2. The error e_n is coded by a variant of the *Rice algorithm*. This is an algorithm which was developed to code integer numbers efficiently. It works particularly well when small numbers are much more likely than larger numbers and in this situation it achieves compression rates close to the

entropy limit. Since the sample values are integers, the step above produces exactly the kind of data that the Rice algorithm handles well.

FLAC. *Free Lossless Audio Code* is another compressed lossless audio format. FLAC is free and open source (meaning that you can obtain the program code). The encoder uses an algorithm similar to the one used for Apple Lossless, with prediction based on previous samples and encoding of the error with a variant of the Rice algorithm.

8.4.3 Lossy formats

All the lossy audio formats described below apply a modified version of the DCT to successive groups (frames) of sample values, analyse the resulting values, and perturb them according to a psycho-acoustic model. These perturbed values are then converted to a suitable number format and coded with some lossless coding method like Huffman coding. When the audio is to be played back, this process has to be reversed and the data translated back to perturbed sample values at the appropriate sample rate.

MP3. Perhaps the best known audio format is MP3 or more precisely *MPEG-1 Audio Layer 3*. This format was developed by Philips, CCETT (Centre commun d'études de télévision et télécommunications), IRT (Institut für Rundfunktechnik) and Fraunhofer Society, and became an international standard in 1991. Virtually all audio software and music players support this format. MP3 is just a sound format and does not specify the details of how the encoding should be done. As a consequence there are many different MP3 encoders available, of varying quality. In particular, an encoder which works well for higher bit rates (high quality sound) may not work so well for lower bit rates.

MP3 is based on applying a variant of the DCT (called the Modified Discrete Cosine Transform, MDCT) to groups of 576 (in special circumstances 192) samples. These MDCT values are then processed according to a psycho-acoustic model and coded efficiently with Huffman coding.

MP3 supports bit rates from 32 to 320 kbit/s and the sampling rates 32, 44.1, and 48 kHz. The format also supports variable bit rates (the bit rate varies in different parts of the file).

AAC. *Advanced Audio Coding* has been presented as the successor to the MP3 format by the principal MP3 developer, Fraunhofer Society. AAC can achieve better quality than MP3 at the same bit rate, particularly for bit rates below 192 kb/s. AAC became well known in April 2003 when Apple introduced this format

(at 128 kb/s) as the standard format for their iTunes Music Store and iPod music players. AAC is also supported by many other music players, including the most popular mobile phones.

The technologies behind AAC and MP3 are very similar. AAC supports more sample rates (from 8 kHz to 96 kHz) and up to 48 channels. AAC uses the MDCT, just like MP3, but AAC processes 1 024 samples at time. AAC also uses much more sophisticated processing of frequencies above 16 kHz and has a number of other enhancements over MP3. AAC, as MP3, uses Huffman coding for efficient coding of the MDCT values. Tests seem quite conclusive that AAC is better than MP3 for low bit rates (typically below 192 kb/s), but for higher rates it is not so easy to differentiate between the two formats. As for MP3 (and the other formats mentioned here), the quality of an AAC file depends crucially on the quality of the encoding program.

There are a number of variants of AAC, in particular AAC Low Delay (AAC-LD). This format was designed for use in two-way communication over a network, for example the Internet. For this kind of application, the encoding (and decoding) must be fast to avoid delays (a delay of at most 20 ms can be tolerated).

Ogg Vorbis. *Vorbis* is an open-source, lossy audio format that was designed to be free of any patent issues and free to use, and to be an improvement on MP3. At our level of detail Vorbis is very similar MP3 and AAC: It uses the MDCT to transform groups of samples to the frequency domain, it then applies a psychoacoustic model, and codes the final data with a variant of Huffman coding. In contrast to MP3 and AAC, Vorbis always uses variable length bit rates. The desired quality is indicated with an integer in the range -1 (worst) to 10 (best). Vorbis supports a wide range of sample rates from 8 kHz to 192 kHz and up to 255 channels. In comparison tests with the other formats, Vorbis appear to perform well, particularly at medium quality bit rates.

WMA. *Windows Media Audio* is a lossy audio format developed by Microsoft. WMA is also based on the MDCT and Huffman coding, and like AAC and Vorbis, it was explicitly designed to improve the deficiencies in MP3. WMA supports sample rates up to 48 kHz and two channels. There is a more advanced version, WMA Professional, which supports sample rates up to 96 kHz and 24 bit samples, but this has limited support in popular software and music players. There is also a lossless variant, WMA Lossless. At low bit rates, WMA generally appears to give better quality than MP3. At higher bit rates, the quality of WMA Pro seems to be comparable to that of AAC and Vorbis.

Part III

Functions

CHAPTER 9

Polynomial Interpolation

A fundamental mathematical technique is to approximate something compli-
cated by something simple, or at least less complicated, in the hope that the
simple can capture some of the essential information in the complicated. This
is the core idea of approximation with Taylor polynomials, a tool that has been
central to mathematics since the calculus was first discovered.

The wide-spread use of computers has made the idea of approximation even
more important. Computers are basically good at doing very simple operations
many times over. Effective use of computers therefore means that a problem
must be broken up into (possibly very many) simple sub-problems. The result
may provide only an approximation to the original problem, but this does not
matter as long as the approximation is sufficiently good.

The idea of approximation is often useful when it comes to studying func-
tions. Most mathematical functions only exist in quite abstract mathematical
terms and cannot be expressed as combinations of the elementary functions we
know from school. In spite of this, virtually all functions of practical interest can
be approximated arbitrarily well by simple functions like polynomials, trigono-
metric or exponential functions. Polynomials in particular are very appealing for
use on a computer since the value of a polynomial at a point can be computed
by utilising simple operations like addition and multiplication that computers
can perform extremely quickly.

A classical example is Taylor polynomials which is a central tool in calculus.
A Taylor polynomial is a simple approximation to a function that is based on in-
formation about the function at a single point only. In practice, the degree of a
Taylor polynomial is often low, perhaps only degree one (linear), but by increas-
ing the degree the approximation can in many cases become arbitrarily good

over large intervals.

In this chapter we first give a review of Taylor polynomials. We assume that you are familiar with Taylor polynomials already or that you are learning about them in a parallel calculus course, so the presentation is brief, with few examples. We do however take the time to derive the Taylor polynomial as this illustrates the importance of writing polynomials in an appropriate form. We also include a description of Taylor polynomials for functions of two variables as this is needed for the analysis of numerical methods for solving differential equations in chapter 12.

The second topic in this chapter is a related procedure for approximating general functions by polynomials. The polynomial approximation will be constructed by forcing the polynomial to take the same values as the function at a few distinct points; this is usually referred to as *interpolation*. Although polynomial interpolation can be used for practical approximation of functions, we are mainly going to use it in later chapters for constructing various numerical algorithms for approximate differentiation and integration of functions, and numerical methods for solving differential equations.

An important additional insight that should be gained from this chapter is that the form in which we write a polynomial is important. We can simplify algebraic manipulations greatly by expressing polynomials in the right form, and the accuracy of numerical computations with a polynomial is also influenced by how the polynomial is represented.

9.1 Taylor polynomials

Taylor polynomials are discussed extensively in all calculus books, so the description here is brief. The essential feature of a Taylor polynomial is that it approximates a given function well at a single point.

Definition 9.1 (Taylor polynomial)**.** *Suppose that the first n derivatives of the function f exist at $x = a$. The Taylor polynomial of f of degree n at a is written $T_n(f; a)$ and satisfies the conditions*

$$T_n(f; a)^{(i)}(a) = f^{(i)}(a), \quad \text{for } i = 0, 1, \ldots, n. \tag{9.1}$$

The conditions in (9.1) mean that $T_n(f; a)$ and f have the same value and first n derivatives at a. This makes it quite easy to derive an explicit formula for the Taylor polynomial.
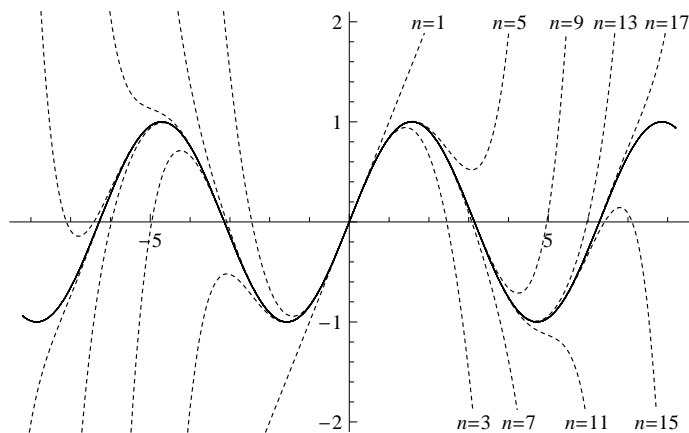
Figure 9.1. The Taylor polynomials of $\sin x$ (around $a = 0$) for degrees 1 to 17.

Theorem 9.2. *The Taylor polynomial of f of degree n at a is unique and can be written as*

$$T_n(f;a)(x) = f(a) + (x-a)f'(a) + \frac{(x-a)^2}{2}f''(a) + \cdots + \frac{(x-a)^n}{n!}f^{(n)}(a). \quad (9.2)$$

Figure 9.1 shows the Taylor polynomials of $\sin x$, generated about $a = 0$, for degrees up to 17. Note that the even degree terms for these Taylor polynomials are 0, so there are only 9 such Taylor polynomials. We observe that as the degree increases, the approximation improves on a larger interval.

Formula (9.2) is a classical result of calculus which is proved in most calculus books. We will include a proof here just to illustrate the usefulness of writing polynomials on nonstandard form.

9.1.1 Derivation of the Taylor formula

Suppose we are to prove (9.2) in the quadratic case. We are looking for a general quadratic polynomial

$$p_2(x) = c_0 + c_1 x + c_2 x^2 \quad (9.3)$$

that satisfies the conditions

$$p_2(a) = f(a), \quad p_2'(a) = f'(a), \quad p_2''(a) = f''(a). \quad (9.4)$$

175

The derivatives of p_2 are given by $p_2'(x) = c_1 + 2c_2 x$ and $p_2''(x) = 2c_2$. If we insert these formulas and p_2 itself in the interpolation conditions (9.1), we obtain

$$
\begin{aligned}
p_2(a) &= c_0 + c_1 a + c_2 a^2 = f(a), \\
p_2'(a) &= c_1 + 2c_2 a = f'(a), \\
p_2''(a) &= 2c_2 = f''(a).
\end{aligned}
\tag{9.5}
$$

This system is easy to solve if we start with the last equation. The result is

$$
\begin{aligned}
c_2 &= f''(a)/2, \\
c_1 &= f'(a) - 2c_2 a = f'(a) - f''(a) a, \\
c_0 &= f(a) - c_1 a - c_2 a^2 = f(a) - \big(f'(a) - f''(a) a\big) a - f''(a) a^2/2.
\end{aligned}
$$

If we insert these expression in (9.3) and collect the terms that multiply $f(a)$, $f'(a)$ and $f''(a)$, we obtain

$$
\begin{aligned}
p_2(x) &= f(a) - \big(f'(a) - f''(a) a\big) a - f''(a) a^2/2 + \big(f'(a) - f''(a) a\big) x + f''(a) x^2/2 \\
&= f(a) + (x - a) f'(a) + (x^2 - 2ax + a^2) f''(a)/2 \\
&= f(a) + (x - a) f'(a) + (x - a)^2 f''(a)/2,
\end{aligned}
$$

which agrees with (9.2).

The work involved in this derivation is not so much, but if we try the same strategy for general degree n, the algebra becomes quite involved. An alternative is to write the quadratic polynomial in the form

$$
p_2(x) = b_0 + b_1(x - a) + b_2(x - a)^2
\tag{9.6}
$$

straightaway. The derivatives can then be written as $p_2'(x) = b_1 + 2b_2(x - a)$ and $p_2''(x) = 2b_2$. If we insert the conditions (9.4) we now obtain

$$
\begin{aligned}
p_2(a) &= b_0 = f(a), \\
p_2'(a) &= b_1 = f'(a), \\
p_2''(a) &= 2b_2 = f''(a).
\end{aligned}
\tag{9.7}
$$

From this it is easy to find the coefficients, and if we insert this in (9.6) we immediately obtain

$$
p_2(x) = f(a) + (x - a) f'(a) + \frac{f''(a)}{2} (x - a)^2.
$$

The main difference is that by writing the polynomial in the form (9.6), the system of equations (9.7) becomes extremely simple, and we can just read off the

coefficients. When we used the traditional form of the polynomial (9.3), we obtained a more complicated system that required some algebraic manipulations when the solution was inserted in (9.3).

For general degree we write the polynomial as

$$p_n(x) = b_0 + b_1(x - a) + \cdots b_i(x - a)^i + \cdots + b_n(x - a)^n. \qquad (9.8)$$

The conditions to be satisfied by p_n are

$$p_n(a) = f(a), \quad p_n'(a) = f'(a), \quad ,\ldots, \quad p_n^{(i)}(a) = f^{(i)}(a), \quad \ldots, \quad p_n^{(n)}(a) = f^{(n)}(a).$$

The derivatives of p_n are

$$p'(x) = b_1 + 2b_2(x - a) + \cdots + ib_i(x - a)^{i-1} + \cdots + nb_n(x - a)^{n-1},$$

$$\vdots$$

$$p^{(i)}(x) = i!b_i + \cdots + n(n-1)\cdots(n-i+1)(x - a)^{n-i},$$

$$p^{(n)}(x) = n!b_n.$$

If we insert this in the conditions to be satisfied by p_n we obtain

$$b_0 = f(a), \quad b_1 = f'(a), \quad \ldots, \quad b_i = f^{(i)}(a)/i!, \quad \ldots, \quad b_n = f^{(n)}(a)/n!,$$

with hardly any work, which confirms formula (9.2).

The polynomial form (9.8) simplifies the computations because of the factor $(x - a)$ which cancels almost all of $p_n(x)$ and its derivatives when $x = a$. This illustrates that there is much to be gained by choosing carefully how the polynomial is written.

Observation 9.3. *In the derivation of the Taylor polynomial, the manipulations simplify if polynomials of degree n are written as*

$$p_n(x) = c_0 + c_1(x - a) + c_2(x - a)^2 + \cdots + c_n(x - a)^n.$$

9.1.2 The remainder

The Taylor polynomial $T_f(f)$ is an approximation to f, and in many situations it will be important to control the error in the approximation. The error can be expressed in a number of ways, and the following two are the most common.

Theorem 9.4. *Suppose that f is a function whose derivatives up to order $n+1$ exist and are continuous. Then the remainder in the Taylor expansion $R_n(f;a)(x) = f(x) - T_n(f;a)(x)$ is given by*

$$R_n(f;a)(x) = \frac{1}{n!}\int_a^x f^{(n+1)}(t)(x-t)^n\,dt. \qquad (9.9)$$

The remainder may also be written as

$$R_n(f;a)(x) = \frac{(x-a)^{n+1}}{(n+1)!}f^{(n+1)}(\xi), \qquad (9.10)$$

where ξ is a number in the interval (a,x) (the interval (x,a) if $x < a$).

The proof of this theorem is based on the fundamental theorem of calculus and integration by parts, and can be found in any standard calculus text.

We are going to make use of Taylor polynomials with remainder in future chapters to analyse the error in a number of numerical methods. Here we just consider one example of how we can use the remainder to control how well a function is approximated by a polynomial.

Example 9.5. We want to determine a polynomial approximation of the function $\sin x$ on the interval $[-1,1]$ with error smaller than 10^{-5}. We want to use Taylor polynomials about the point $a = 0$; the question is how high the degree needs to be in order to get the error to be small.

If we look at the error term (9.10), there is one factor that looks rather difficult to control, namely $f^{(n+1)}(\xi)$: Since we do not know the degree, we do not really know what this derivative is, and on top of this we do not know at which point it should be evaluated either. The solution is not so difficult if we realise that we do not need to control the error exactly, it is sufficient to make sure that the error is *smaller* than 10^{-5}.

The easiest error term to work with is (9.10), and what we want is to find the smallest n such that

$$\left|\frac{x^{n+1}}{(n+1)!}f^{(n+1)}(\xi)\right| \le 10^{-5}, \qquad (9.11)$$

where the function $f(x) = \sin x$ in our case and ξ is a number in the interval $(0,x)$. Here we demand that the absolute value of the error should be smaller than 10^{-5}. This is important since otherwise we could make the error small by making it negative, with large absolute value. The main ingredient in achieving what we want is the observation that since $f(x) = \sin x$, any derivative of f is either $\cos x$ or $\sin x$ (possibly with a minus sign which disappears when we take

absolute values). But then we certainly know that

$$\left| f^{(n+1)}(\xi) \right| \le 1. \tag{9.12}$$

This may seem like a rather crude estimate, which may be the case, but it was certainly very easy to derive; to estimate the correct value of ξ would certainly be much more difficult. If we insert the estimate (9.12) on the left in (9.11), we can also change our required inequality,

$$\left| \frac{x^{n+1}}{(n+1)!} f^{(n+1)}(\xi) \right| \le \frac{|x|^{n+1}}{(n+1)!} \le 10^{-5}.$$

If we manage to find an n such that this last inequality is satisfied, then (9.11) will also be satisfied. Since $x \in [-1, 1]$ we know that $|x| \le 1$ so this last inequality will be satisfied if

$$\frac{1}{(n+1)!} \le 10^{-5}. \tag{9.13}$$

The left-hand side of this inequality decreases with increasing n, so we can just determine n by computing $1/(n+1)!$ for the first few values of n, and use the first value of n for which the inequality holds. If we do this, we find that $1/8! \approx 2.5 \times 10^{-5}$ and $1/9! \approx 2.8 \times 10^{-6}$. This means that the smallest value of n for which (9.13) will be satisfied is $n = 8$. The Taylor polynomial we are looking for is therefore

$$p_8(x) = T_8(\sin; 0)(x) = x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040},$$

since the term of degree 8 is zero. If we evaluate this we find $p_8(1) \approx 0.841468$ with error roughly 2.73×10^{-6} which is close to the estimate $1/9!$ which we computed above.

Figure 9.1 shows the Taylor polynomials of $\sin x$ about $a = 0$ of degree up to 17. In particular we see that for degree 7, the approximation is indistinguishable from the original in the plot, at least up to $x = 2$. ∎

The error formula (9.10) will be the most useful one for us, and for easy reference we record the complete Taylor expansion in a corollary.

Corollary 9.6. *Any function f whose first $n + 1$ derivatives are continuous at $x = a$ can be expanded in a Taylor polynomial of degree n at $x = a$ with a corresponding error term,*

$$f(x) = f(a) + (x-a)f'(a) + \cdots + \frac{(x-a)^n}{n!} f^{(n)}(a) + \frac{(x-a)^{n+1}}{(n+1)!} f^{(n+1)}(\xi_x), \tag{9.14}$$

where ξ_x is a number in the interval (a, x) (the interval (x, a) if $x < a$) that depends on x. This is called a Taylor expansion *of f.*

The remainder term in (9.14) lets us control the error in the Taylor approximation. It turns out that the error behaves quite differently for different functions.

Example 9.7 (Taylor polynomials for $f(x) = \sin x$). If we go back to figure 9.1, it seems like the Taylor polynomials approximate $\sin x$ well on larger intervals as we increase the degree. Let us see if this observation can be derived from the error term

$$e(x) = \frac{(x-a)^{n+1}}{(n+1)!} f^{(n+1)}(\xi). \tag{9.15}$$

When $f(x) = \sin x$ we know that $|f^{(n+1)}(\xi)| \leq 1$, so the error is bounded by

$$|e(x)| \leq \frac{|x|^{n+1}}{(n+1)!}$$

where we have also inserted $a = 0$ which was used in figure 9.1. Suppose we want the error to be small on the interval $[-b, b]$. Then $|x| \leq b$, so on this interval the error is bounded by

$$|e(x)| \leq \frac{b^{n+1}}{(n+1)!}.$$

The question is what happens to the expression on the right when n becomes large; does it tend to 0 or does it not? It is not difficult to show that regardless of what the value of b is, the factorial $(n+1)!$ will tend to infinity more quickly, so

$$\lim_{n \to \infty} \frac{b^{n+1}}{(n+1)!} = 0$$

In other words, if we just choose the degree n to be high enough, we can get the Taylor polynomial to be an arbitrarily good approximation to $\sin x$ on an interval $[-b, b]$, regardless of the value of b. ∎

Example 9.8 (Taylor polynomials for $f(x) = e^x$). Figure 9.2 (a) shows a plot of the Taylor polynomial of degree 4 for the exponential function $f(x) = e^x$, expanded about the point $a = 1$. For this function it is easy to see that the Taylor polynomials will converge to e^x on any interval as the degree tends to infinity, just like we saw for $f(x) = \sin x$ in example 9.7. ∎

Example 9.9 (Taylor polynomials for $f(x) = \ln x$). The plot in figure 9.2 shows the function $f(x) = \ln x$ and its Taylor polynomial of degree 20 expanded at $a = 1$. The Taylor polynomial seems to be very close to $\ln x$ as long as x is a bit smaller than 2, but for $x > 2$ it seems to diverge quickly. Let us see if this can be deduced from the error term.
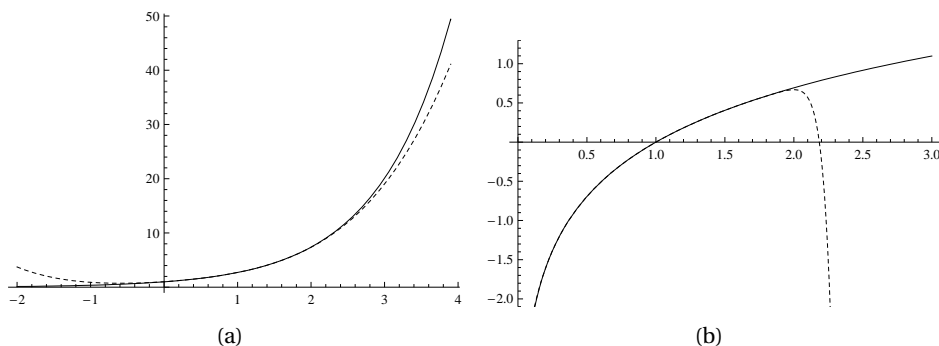
Figure 9.2. In (a) the Taylor polynomial of degree 4 about the point $a = 1$ for the function $f(x) = e^x$ is shown. Figure (b) shows the Taylor polynomial of degree 20 for the function $f(x) = \log x$, also about the point $a = 1$.

The error term involves the derivative $f^{(n+1)}(\xi)$ of $f(x) = \ln x$, so we need a formula for this. Since $f(x) = \ln x$, we have

$$f'(x) = \frac{1}{x} = x^{-1}, \quad f''(x) = -x^{-2}, \quad f'''(x) = 2x^{-3}$$

and from this we find that the general formula is

$$f^{(k)}(x) = (-1)^{k+1}(k-1)!\, x^{-k}. \tag{9.16}$$

From this we find that the general term in the Taylor polynomial is

$$\frac{(x-1)^k}{k!} f^{(k)}(1) = (-1)^{k+1} \frac{(x-1)^k}{k}$$

so the Taylor expansion (9.14) becomes

$$\ln x = \sum_{k=1}^{n} (-1)^{k+1} \frac{(x-1)^k}{k} + \frac{(x-1)^{n+1}}{n+1} \xi^{-n-1},$$

where ξ is some number in the interval $(1, x)$ (in $(x, 1)$ if $0 < x < 1$). The problematic area seems to be to the right of $x = 1$, so let us assume that $x > 1$. In this case $\xi > 1$ so $\xi^{-n-1} < 1$ so the error is bounded by

$$\left| \frac{(x-1)^{n+1}}{n+1} \xi^{-n-1} \right| \leq \frac{(x-1)^{n+1}}{n+1}.$$

When $x - 1 < 1$, i.e., when $x < 2$, we know that $(x-1)^{n+1}$ will tend to zero when n tends to infinity, and the denominator $n+1$ will just contribute to this happening even more quickly.

For $x > 2$, one can try and analyse the error term, and if one uses the integral form of the remainder (9.9) it is in fact possible to find an exact formula for the error. However, it is much simpler to consider the Taylor polynomial directly,

$$p_n(x) = T(\ln; 1)(x) = \sum_{k=1}^{n} (-1)^{k+1} \frac{(x-1)^k}{k}.$$

Note that for $x > 2$, the absolute value of the terms in the sum will become arbitrarily large since

$$\lim_{k \to \infty} \frac{c^k}{k} = \infty$$

when $c > 0$. This means that the sum will jump around more and more, so there is no way it can converge for $x > 2$, and it is this effect we see in figure 9.2 (b). ∎

9.2 Interpolation

A Taylor polynomial based at a point $x = a$ usually provides a very good approximation near a, but as we move away from this point, the error will increase. If we want a good approximation to a function f across a whole interval, it seems natural that we ought to utilise information about f from different parts of the interval. Polynomial interpolation lets us do just that.

9.2.1 Polynomial interpolation is possible

The idea behind polynomial interpolation is simple: We approximate a function f by a polynomial p by forcing p to have the same function values as f at a number of points. Suppose for instance that we have $n + 1$ distinct points $(x_i)_{i=0}^{n}$ scattered throughout the interval $[a, b]$ where f is defined. Since a polynomial of degree n has $n + 1$ free coefficients it is natural to try and find a polynomial of degree n with the same values as f at these points.

Problem 9.10 (Polynomial interpolation)**.** *Let f be a given function defined on an interval $[a, b]$, and let $(x_i)_{i=0}^{n}$ be $n + 1$ distinct points in $[a, b]$. The polynomial interpolation problem is to find a polynomial $p_n = P(f; x_0, \ldots, x_n)$ of degree n that matches f at the given points,*

$$p_n(x_i) = f(x_i), \quad \text{for } i = 0, 1, \ldots, n. \tag{9.17}$$

The points $(x_i)_{i=0}^{n}$ are called interpolation points, *the conditions* (9.17) *are called the* interpolation conditions, *and the polynomial $p_n = P(f; x_0, \ldots, x_n)$ is called a polynomial* interpolant.

The notation $P(f; x_0, \ldots, x_n)$ for the polynomial interpolant is similar to the notation $T_n(f; a)$ for the Taylor polynomial. However, it is a bit cumbersome, so we will often use p_n when no confusion is possible.

There are at least three questions raised by problem 9.10: Is there a polynomial of degree n that satisfies the interpolation conditions (9.17)? How many such polynomials are there? How can we find one such polynomial? As always it is instructive to consider the simplest cases to gain some insight into the problem.

Example 9.11 ($n = 0$). The case $n = 0$ corresponds to the situation where we only have one interpolation point x_0. Then we are looking for a polynomial p_0 of degree 0, i.e. a constant, such that $p_0(x_i) = f(x_i)$. This is easy to solve; we see that we must have $p_0(x) = f(x_0)$. ∎

Example 9.12 ($n = 1$). The next case is $n = 1$, i.e., we have two interpolation points x_0 and x_1. Now we are looking for a straight line $p_1(x) = c_0 + c_1 x$ such that $p_1(x_0) = f(x_0)$ and $p_1(x_1) = f(x_1)$. From school we know that this is possible, and one formula for the solution is

$$p_1(x) = \frac{x_1 - x}{x_1 - x_0} f(x_0) + \frac{x - x_0}{x_1 - x_0} f(x_1). \quad \blacksquare \tag{9.18}$$

Example 9.13 ($n = 2$). In this case we are given 3 points x_0, x_1 and x_2 and we are looking for a quadratic polynomial p_2 that may take the same values as f at these points. The representation in (9.18) is appealing since the dependence on the two function values is so explicit, so let us try and generalise this. In other words, we want to write p_2 in the form

$$p_2(x) = \ell_0(x) f(x_0) + \ell_1(x) f(x_1) + \ell_2(x) f(x_2) \tag{9.19}$$

where $\{\ell_i\}_{i=0}^2$ are quadratic polynomials. If we insert the two interpolation conditions at $x = x_1$ and $x = x_2$ we obtain the equations

$$f(x_1) = p_2(x_1) = \ell_0(x_1) f(x_0) + \ell_1(x_1) f(x_1) + \ell_2(x_1) f(x_2),$$
$$f(x_2) = p_2(x_2) = \ell_0(x_2) f(x_0) + \ell_1(x_2) f(x_1) + \ell_2(x_2) f(x_2).$$

From this we see that ℓ_0 should satisfy the equations $\ell_0(x_1) = \ell_0(x_2) = 0$ to make sure that $f(x_0)$ is cancelled out at x_1 and x_2. For this to happen, the quadratic polynomial p_2 should have two roots at x_1 and x_2, i.e., it should be on the form $\ell_0(x) = d_0(x - x_1)(x - x_2)$. The value of d_0 we can find from the interpolation condition at x_0,

$$f(x_0) = p_2(x_0) = d_0 \ell_0(x_0) f(x_0) = d_0(x_0 - x_1)(x_0 - x_2) f(x_0), \tag{9.20}$$

183
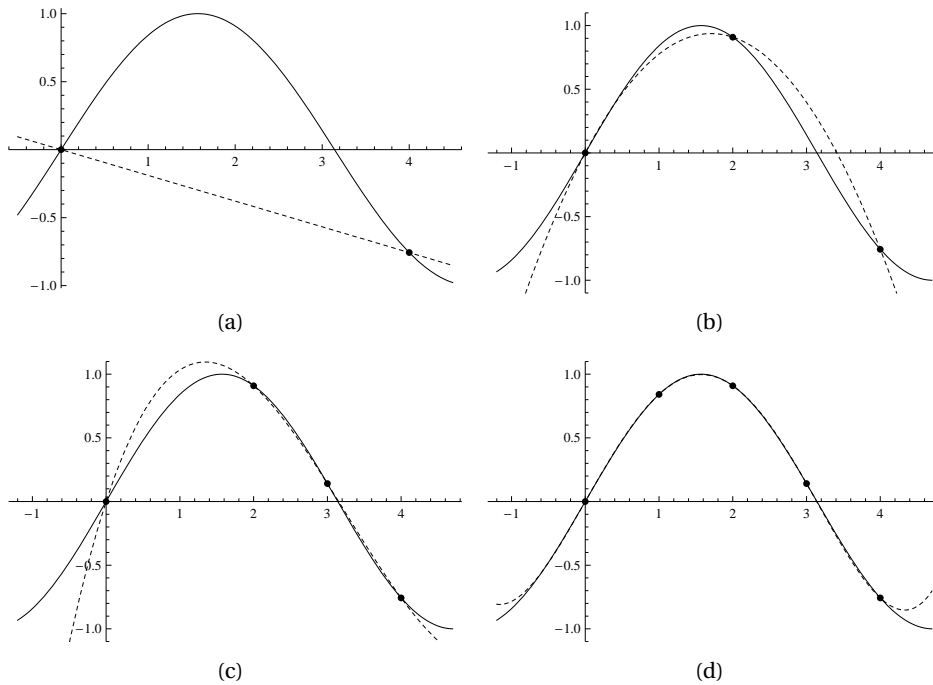
Figure 9.3. Interpolation of $\sin x$ with a line (a), a parabola (b), a cubic (c), and a quartic polynomial (d).

where we have used the fact that we must also have $\ell_1(x_0) = \ell_2(x_0) = 0$. From (9.20) we see that $d_0 = 1/\big((x_0 - x_1)(x_0 - x_2)\big)$ and therefore

$$\ell_0(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)}.$$

We can construct ℓ_1 and ℓ_2 in the same way and find

$$\ell_1(x) = \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)}, \quad \ell_2(x) = \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)}.$$

If we insert these polynomials in (9.19) it is easy to verify that the resulting polynomial p_2 indeed satisfies the 3 interpolation conditions. ■

Some examples of interpolation are shown in figure 9.3. Note how the quality of the approximation improves with increasing degree.

The crucial property possessed by the quadratic polynomials $\{\ell_i(x)\}_{i=0}^2$ at the end of example 9.13 is that they satisfy the relations

$$\ell_i(x_j) = \begin{cases} 1, & \text{if } i = j; \\ 0, & \text{otherwise.} \end{cases} \tag{9.21}$$

184

for any pair of integers in the range $0 \le i, j \le 2$. The two polynomials in the linear case (9.18) satisfy the same kind of relation. This kind of relation is so common that the right-hand side of (9.21) has been given a special name.

Definition 9.14 (Kronecker delta)**.** *The* Kronecker delta $\delta_{i,j}$ *is defined as*

$$\delta_{i,j} = \begin{cases} 1, & \text{if } i = j; \\ 0, & \text{otherwise.} \end{cases}$$

for any pair of integers (i, j).

The general idea for showing that the interpolation problem is solvable follows exactly as in the quadratic case. For degree n we have to construct $n + 1$ polynomials $\{\ell_{i,n}\}_{i=0}^{n}$ of degree n such that the interpolant p_n can be written

$$p_n(x) = \ell_{0,n}(x)f(x_0) + \ell_{1,n}(x)f(x_1) + \cdots + \ell_{n,n}(x)f(x_n)$$

where we have added a second subscript to the ℓ_i-polynomials to indicate the degree. This will work if the polynomial $\ell_{i,n}(x)$ is one at x_i and zero at all the other x_is, i.e., if $\ell_{i,n}(x_j) = \delta_{i,j}$. This is accomplished if

$$\ell_{i,n}(x) = \frac{(x - x_0) \cdots (x - x_{i-1})(x - x_{i+1}) \cdots (x - x_n)}{(x_i - x_0) \cdots (x_i - x_{i-1})(x_i - x_{i+1}) \cdots (x_i - x_n)} = \prod_{\substack{j=0 \\ j \ne i}}^{n} \frac{x - x_j}{x_i - x_j}. \qquad (9.22)$$

We have therefore proved most of the following basic theorem on interpolation.

Theorem 9.15. *Let f be a given function defined on an interval $[a, b]$, and let $(x_i)_{i=0}^{n}$ be $n + 1$ distinct points in $[a, b]$. Then the polynomial defined by*

$$p_n(x) = \ell_{0,n}(x)f(x_0) + \ell_{1,n}(x)f(x_1) + \cdots + \ell_{n,n}(x)f(x_n)$$

is the unique polynomial of degree n that satisfies the interpolation conditions

$$p_n(x_i) = f(x_i), \quad \text{for } i = 0, 1, \ldots, n.$$

Here $\{\ell_{i,n}\}_{i=0}^{n}$ are the polynomials of degree n defined by (9.22) which are usually referred to as the Lagrange polynomials *of degree n.*

Proof. The fact that p_n satisfies the interpolation conditions follows from the argument preceding the theorem, just like in the quadratic case. What remains
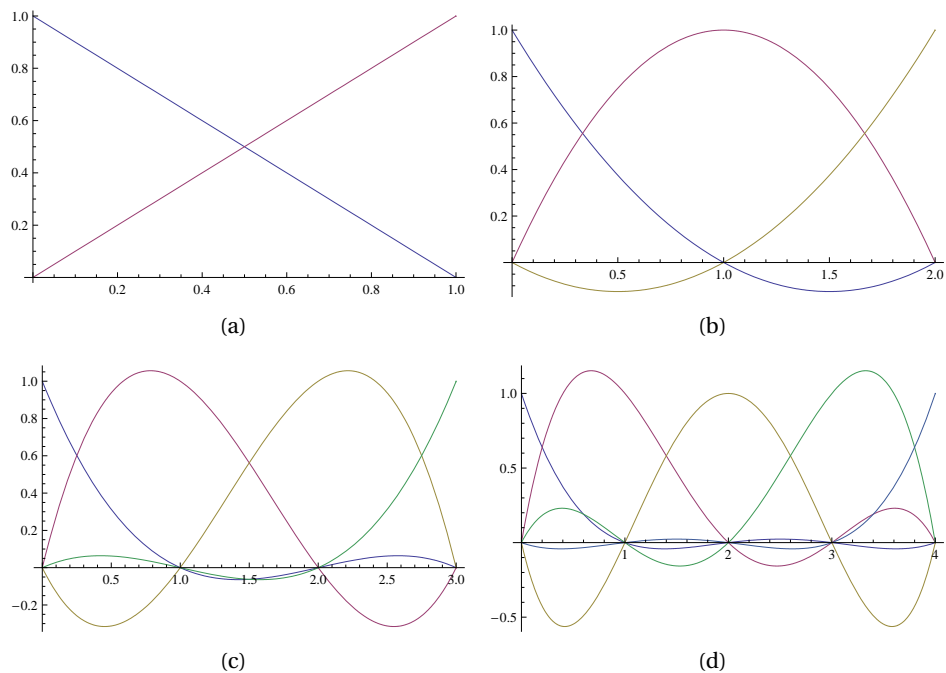
Figure 9.4. The Lagrange polynomials of degree 1 (in (a)), 2 (in (b)), 3 (in (c)), and 4 (in (d)). For degree n the interpolation points are $x_i = i$, for $i = 0, \ldots, n$.

is to prove is that this is the only polynomial that satisfies the interpolation conditions. For this, suppose that $\hat{p}_n$ is another polynomial that satisfies the interpolation conditions. Then the difference $e = p_n - \hat{p}_n$ is a polynomial of degree n that satisfies the conditions

$$e(x_i) = p_n(x_i) - \hat{p}_n(x_i) = f(x_i) - f(x_i) = 0, \quad \text{for } i = 0, 1, \ldots, n.$$

In other words, the polynomial e is of degree n and has $n + 1$ roots. The only polynomial of degree n that can have this many zeros is the polynomial which is identically zero. But if $e = p_n - \hat{p}_n$ is identically zero, then $\hat{p}_n$ must be identically equal to p_n, so there is only one polynomial of degree n that solves the interpolation problem. ∎

Some examples of Lagrange polynomials of different degrees are shown in figure 9.4.

Theorem 9.15 answers two of the questions raised above: Problem 9.10 has a solution and it is unique. The theorem itself does not tell us directly how to find the solution, but in the text preceding the theorem we showed how it could be constructed. One concrete example will illustrate the procedure.

186
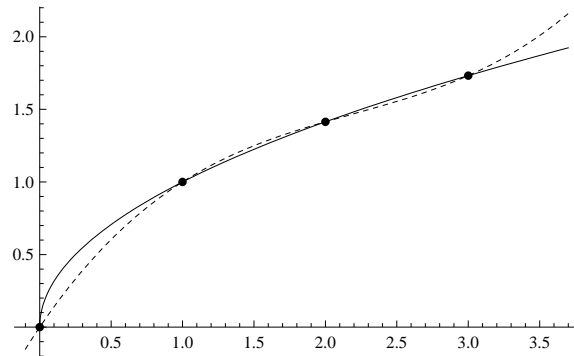
Figure 9.5. The function $f(x) = \sqrt{x}$ (solid) and its cubic interpolant at the four points 0, 1, 2, and 3 (dashed).

Example 9.16. Suppose we have the four points $x_i = i$, for $i = 0, \ldots, 3$, and we want to interpolate the function $\sqrt{x}$ at these points. We first construct the cubic Lagrange polynomials for the points $(x_i)_{i=0}^3$. For the first one, ℓ_0, we find (we drop the second subscript),

$$\ell_0(x) = \frac{(x - x_1)(x - x_2)(x - x_3)}{(x_0 - x_1)(x_0 - x_2)(x_0 - x_3)} = -\frac{(x - 1)(x - 2)(x - 3)}{6}.$$

The other three Lagrange polynomials can be constructed in the same way,

$$\ell_1(x) = \frac{x(x - 2)(x - 3)}{2}, \quad \ell_2(x) = -\frac{x(x - 1)(x - 3)}{2}, \quad \ell_3(x) = \frac{x(x - 1)(x - 2)}{6}.$$

When the Lagrange polynomials are known the interpolant is simply

$$p_3(x) = \ell_1(x) + \ell_2(x)\sqrt{2} + \ell_3(x)\sqrt{3}$$

(the first term disappears since $\sqrt{0} = 0$). Figure 9.5 shows a plot of this interpolant. ∎

9.2.2   The Newton form

Representation of polynomial interpolants provides another example of the fact that it may be worthwhile to write polynomials in alternative forms. The representation in terms of Lagrange polynomials makes it simple to write down the interpolant since no equations need to be solved.

Sometimes, it is desirable to improve the interpolating polynomial by adding one more interpolation point. If the Lagrange form is used, it is far form obvious how the earlier constructed interpolant can be exploited when computing the new one which uses one extra point. For this, the Newton form of the interpolating polynomial is more convenient.

187

Definition 9.17. *Let $(x_i)_{i=0}^n$ be $n+1$ distinct real numbers. The Newton form of the interpolating polynomial of degree n is written on the form*

$$p_n(x) = c_0 + c_1(x - x_0) + c_2(x - x_0)(x - x_1) + \cdots$$
$$+ c_n(x - x_0)(x - x_1)\cdots(x - x_{n-1}). \quad (9.23)$$

Note that there is no mention of the function f to be interpolated in definition 9.17. This information is contained in the coefficients $(c_i)_{i=0}^n$ which are not specified in the definition, but which must be computed. As usual, some examples are instructive.

Example 9.18 (Newton form for $n=0$)**.** Suppose we have only one interpolation point x_0. Then the Newton form is just $p_0(x) = c_0$. To interpolate f at x_0 we just have to choose $c_0 = f(x_0)$,

$$p_0(x) = f(x_0). \quad \blacksquare$$

Example 9.19 (Newton form for $n=1$)**.** With two points x_0 and x_1 the Newton form is $p_1(x) = c_0 + c_1(x - x_0)$. Interpolation at x_0 means that $f(x_0) = p_1(x_0) = c_0$ and interpolation at x_1,

$$f(x_1) = p_1(x_1) = f(x_0) + c_1(x_1 - x_0),$$

which means that

$$c_0 = f(x_0), \quad c_1 = \frac{f(x_1) - f(x_0)}{x_1 - x_0}. \quad (9.24)$$

Therefore c_0 remains the same as in the case $n=0$. $\quad \blacksquare$

Example 9.20 (Newton form for $n=2$)**.** We add another point and consider interpolation with a quadratic polynomial

$$p_2(x) = c_0 + c_1(x - x_0) + c_2(x - x_0)(x - x_1).$$

at the three points x_0, x_1, x_2. Interpolation at x_0 and x_1 gives the equations

$$f(x_0) = p_2(x_0) = c_0,$$
$$f(x_1) = p_2(x_1) = c_0 + c_1(x_1 - x_0),$$

which we note are the same equations as we solved in the case $n=1$. From the third condition

$$f(x_2) = p(x_2) = c_0 + c_1(x_2 - x_0) + c_2(x_2 - x_0)(x_2 - x_1),$$

we obtain

$$c_2 = \frac{f(x_2) - f(x_0) - \frac{f(x_1)-f(x_0)}{x_1-x_0}(x_2 - x_0)}{(x_2 - x_0)(x_2 - x_1)}.$$

Playing around a bit with this expression one finds that it can also be written as

$$c_2 = \frac{\frac{f(x_2)-f(x_1)}{x_2-x_1} - \frac{f(x_1)-f(x_0)}{x_1-x_0}}{x_2 - x_0}. \quad \blacksquare \tag{9.25}$$

It is easy to see that what happened in the quadratic case, happens in the general case: The equation that results from the interpolation condition at x_k involves only the points $(x_0, f(x_0))$, $(x_1, f(x_1))$, ..., $(x_k, f(x_k))$. If we write down all the equations we find

$$f(x_0) = c_0,$$
$$f(x_1) = c_0 + c_1(x_1 - x_0),$$
$$f(x_2) = c_0 + c_1(x_2 - x_0) + c_2(x_2 - x_0)(x_2 - x_1),$$
$$\vdots \tag{9.26}$$
$$f(x_k) = c_0 + c_1(x_k - x_0) + c_2(x_k - x_0)(x_k - x_1) + \cdots$$
$$+ c_{k-1}(x_k - x_0)\cdots(x_k - x_{k-2}) + c_k(x_k - x_0)\cdots(x_k - x_{k-1}).$$

This is an example of a *triangular system* where each new equation introduces one new variable and one new point. This means that each coefficient c_k only depends on the data $(x_0, f(x_0))$, $(x_1, f(x_1))$, ..., $(x_k, f(x_k))$, and the following lemma is immediate.

Lemma 9.21. *Let f be a given function and x_0, ..., x_n given interpolation points. If the interpolating polynomial of degree n is written in Newton form,*

$$p_n(x) = c_0 + c_1(x - x_0) + \cdots + c_n(x - x_0)(x - x_1)\cdots(x - x_{n-1}), \tag{9.27}$$

then c_k is only dependent on $(x_0, f(x_0))$, $(x_1, f(x_1))$, ..., $(x_k, f(x_k))$ and is written

$$c_k = f[x_0, \ldots, x_k] \tag{9.28}$$

for $k = 0, 1, \ldots, n$. The interpolating polynomials p_n and p_{n-1} are related by

$$p_n(x) = p_{n-1}(x) + f[x_0, \ldots, x_n](x - x_0)\cdots(x - x_{n-1}).$$

Note that with the notation (9.28) for the coefficients, the interpolation formula (9.27) becomes

$$p_n(x) = f[x_0] + f[x_0, x_1](x - x_0) + \cdots + f[x_0, \ldots, x_n](x - x_0) \cdots (x - x_{n-1}). \quad (9.29)$$

The coefficients (c_k) can be computed from the system (9.26). The big advantage compared with the Lagrange form is the simplicity of adding an extra point: An extra point adds an extra equation to (9.26) from which the additional coefficient can be computed, while the remaining coefficients are the same as in the case with one point less.

9.2.3 Divided differences

The coefficients $c_k = f[x_0, \ldots, x_k]$ have certain properties that are useful both for computation and understanding. When doing interpolation at the points $x_0, \ldots,$ x_k we can consider two smaller problems, namely interpolation at the points $x_0,$ $\ldots, x_{k-1}$ as well as interpolation at the points $x_1, \ldots, x_k$.

Suppose that the polynomial q_0 interpolates f at the points $x_0, \ldots, x_{k-1}$ and that q_1 interpolates f at the points $x_1, \ldots, x_k$, and consider the polynomial defined by the formula

$$p(x) = \frac{x_k - x}{x_k - x_0} q_0(x) + \frac{x - x_0}{x_k - x_0} q_1(x). \quad (9.30)$$

Our claim is that $p(x)$ interpolates f at the points $x_0, \ldots, x_k$, which means that $p = p_k$ since a polynomial interpolant of degree $k - 1$ which interpolates $k + 1$ points is unique.

We first check that p interpolates f at an interior point x_i with $0 < i < k$. In this case $q_0(x_i) = q_1(x_i) = f(x_i)$ so

$$p(x_i) = \frac{x_k - x}{x_k - x_0} f(x_i) + \frac{x - x_0}{x_k - x_0} f(x_i) = f(x_i).$$

At $x = x_0$ we have

$$p(x_0) = \frac{x_k - x_0}{x_k - x_0} q_0(x_0) + \frac{x_0 - x_0}{x_k - x_0} q_1(x_0) = q_0(x_0) = f(x_0),$$

as required, and in a similar way we also find that $p(x_k) = f(x_k)$.

Let us rewrite (9.30) in a more explicit way.

Lemma 9.22. *Let $P(f; x_0, \ldots, x_k)$ denote the polynomial of degree $k - 1$ that interpolates the function f at the points $x_0, \ldots, x_k$. Then*

$$P(f; x_0, \ldots, x_k)(x) = \frac{x_k - x}{x_k - x_0} P(f; x_0, \ldots, x_{k-1})(x) + \frac{x - x_0}{x_k - x_0} P(f; x_1, \ldots, x_k)(x).$$

From this lemma we can deduce a useful formula for the coefficients of the interpolating polynomial. (Recall that the leading coefficient of a polynomial is the coefficient of the highest degree term.)

Theorem 9.23. *Let $c_k = f[x_0,\ldots,x_k]$ denote the leading coefficient of the interpolating polynomial $P(f; x_0,\ldots,x_k)$. This is called a kth order divided difference of f and satisfies the relations $f[x_0] = f(x_0)$, and*

$$f[x_0,\ldots,x_k] = \frac{f[x_1,\ldots,x_k] - f[x_0,\ldots,x_{k-1}]}{x_k - x_0} \tag{9.31}$$

for $k > 0$.

Proof. The relation (9.31) follows from the relation in lemma 9.22 if we consider the leading coefficients on both sides. On the left the leading coefficient is $f[x_0,\ldots,x_k]$. The right-hand side has the form

$$\frac{x_k - x}{x_k - x_0}\left(f[x_0,\ldots,x_{k-1}]x^{k-1} + \text{lower degree terms}\right) +$$

$$\frac{x - x_0}{x_k - x_0}\left(f[x_1,\ldots,x_k]x^{k-1} + \text{lower degree terms}\right)$$

$$= \frac{f[x_1,\ldots,x_k] - f[x_0,\ldots,x_k]}{x_k - x_0}x^k + \text{lower degree terms}$$

and from this (9.31) follows. ∎

The significance of theorem 9.23 is that it provides a simple formula for computing the coefficients of the interpolating polynomial in Newton form. The relation (9.31) also explains the name 'divided difference', and it should not come as a surprise that $f[x_0,\ldots,x_k]$ is related to the kth derivative of f, as we will see below.

It is helpful to organise the computations of divided differences in a table,

$$
\begin{array}{llll}
x_0 & f[x_0] \\
x_1 & f[x_1] & f[x_0,x_1] \\
x_2 & f[x_2] & f[x_1,x_2] & f[x_0,x_1,x_2] \\
x_3 & f[x_3] & f[x_2,x_3] & f[x_1,x_2,x_3] & f[x_0,x_1,x_2,x_3] \\
\vdots
\end{array}
\tag{9.32}
$$

Here an entry in the table (except for the first two columns) is computed by subtracting the entry to the left and above from the entry to the left, and dividing by the last minus the first x_i involved. Then all the coefficients of the Newton form can be read off from the diagonal. An example will illustrate how this is used.
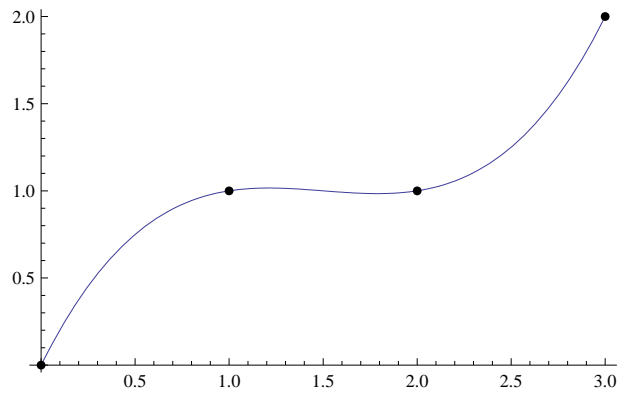
Figure 9.6. The data points and the interpolant in example 9.24.

Example 9.24. Suppose we have the data

x	0	1	2	3
$f(x)$	0	1	1	2

We want to compute the divided differences using (9.31) and organise the computations as in (9.32),

x	$f(x)$			
0	0			
1	1	1		
2	1	0	$-1/2$	
3	2	1	$1/2$	$1/3$

This means that the interpolating polynomial is

$$p_3(x) = 0 + 1(x-0) - \frac{1}{2}(x-0)(x-1) + \frac{1}{3}(x-0)(x-1)(x-2)$$
$$= x - \frac{1}{2}x(x-1) + \frac{1}{3}x(x-1)(x-2).$$

A plot of this polynomial with the interpolation points is shown in figure 9.6. ∎

There is one more important property of divided differences that we need to discuss. If we look back on equation (9.24), we see that

$$c_1 = f[x_0, x_1] = \frac{f(x_1) - f(x_0)}{x_1 - x_0}.$$

From the mean value theorem for derivatives we can conclude from this that $f[x_0, x_1] = f'(\xi)$ for some number ξ in the interval (x_0, x_1), provided f' is contin-

uous in this interval. The relation (9.31) shows that higher order divided differences are built form lower order ones in a similar way, so it should come as no surprise that divided differences can be related to derivatives in general.

Theorem 9.25. *Let f be a function whose first k derivatives are continuous in the smallest interval $[a, b]$ that contains all the numbers $x_0, \ldots, x_k$. Then*

$$f[x_0, \ldots, x_k] = \frac{f^{(k)}(\xi)}{k!} \tag{9.33}$$

where ξ is some number in the interval (a, b).

We skip the proof of this theorem, but return to the Newton form of the interpolating polynomial,

$$p_n = f[x_0] + f[x_0, x_1](x - x_0) + \cdots + \cdots f[x_0, \ldots, x_n](x - x_0) \cdots (x - x_{n-1}).$$

Theorem 9.25 shows that divided differences can be associated with derivatives, so this formula is very similar to the formula for a Taylor polynomial. In fact, if we let all the interpolation points x_i approach a common number z, it is quite easy to show that the interpolating polynomial p_n approaches the Taylor polynomial

$$T_n(f; z)(x) = f(z) + f'(z)(x - z) + \cdots + f^{(n)}(z) \frac{(x - z)^n}{n!}.$$

9.2.4 Computing with the Newton form

Our use of polynomial interpolation will primarily be as a tool for developing numerical methods for differentiation, integration, and solution of differential equations. For such purposes the interpolating polynomial is just a step on the way to a final computational formula, and is not computed explicitly. There are situations though where one may need to determine the interpolating polynomial explicitly, and then the Newton form is usually preferred.

To use the Newton form in practice, we need two algorithms: One for determining the divided differences involved in the formula (9.29), and one for computing the value $p_n(x)$ of the interpolating polynomial for a given number x. We consider each of these in turn.

The Newton form of the polynomial that interpolates a given function f at the $n + 1$ points $x_0, \ldots, x_n$ is given by

$$p_n(x) = f[x_0] + f[x_0, x_1](x - x_0) + \cdots + f[x_0, \ldots, x_n](x - x_0) \cdots (x - x_{n-1}),$$

and to represent this polynomial we need to compute the divided differences $f[x_0]$, $f[x_0, x_1]$, ..., $f[x_0, ..., x_n]$. The obvious way to do this is as indicated in the table (9.32) which we repeat here for convenience,

$$
\begin{array}{llll}
x_0 & f[x_0] & & \\
x_1 & f[x_1] & f[x_0, x_1] & \\
x_2 & f[x_2] & f[x_1, x_2] & f[x_0, x_1, x_2] \\
x_3 & f[x_3] & f[x_2, x_3] & f[x_1, x_2, x_3] & f[x_0, x_1, x_2, x_3] \\
\vdots & & & &
\end{array}
$$

The natural way to do this is to start with the interpolation points x_0, ..., x_n and the function f, and then compute the values in the table column by column. Let us denote the entries in the table by $(d_{i,k})_{i=0,k=0}^{n}$, where i runs down the columns and k indicates the column number, starting with 0 for the column with the function values. The first column of function values is special and must be computed separately. Otherwise we note that a value $d_{i,k}$ in column k is given by the two neighbouring values in column $k-1$,

$$
d_{i,k} = \frac{d_{i,k-1} - d_{i-1,k-1}}{x_i - x_{i-k}}, \tag{9.34}
$$

for $i \geq k$, while the other entries in column k are not defined. We start by computing the first column of function values. Then we can use the formula (9.34) to compute the next column. It would be natural to start by computing the diagonal entry and then proceed down the column. However, it is better to start with the last entry in the column, and proceed up, to the diagonal entry. The reason is that once an entry $d_{i,k}$ has been computed, the entry $d_{i,k-1}$ immediately to the left is not needed any more. Therefore, there is no need to use a two dimensional array; we can just start with the one dimensional array of function values and let every new column overwrite the one to the left. Since no column contains values above the diagonal, we end up with the correct divided differences in the one dimensional array at the end, see exercise 3.

Algorithm 9.26 (Computing divided differences)**.** *Let f be a given function, and x_0, ..., x_n given interpolation points for some nonnegative integer n. After the code*

```
for i := 0, 1, ..., n
    f_i := f(x_i);
for k := 1, 2, ... n
    for i := n, n-1, ..., k
        f_i := (f_i - f_{i-1})/(x_i - x_{i-k});
```

Note that this algorithm has two nested for-loops, so the number of subtractions is

$$\sum_{k=1}^{n}\sum_{i=k}^{n} 2 = \sum_{k=1}^{n} 2(n - k + 1) = 2\sum_{k=1}^{n} k = n(n + 1) = n^2 + n$$

which follows from the formula for the sum on the first n integers. We note that this grows with the square of the degree n, which is a consequence of the double for-loop. This is much faster than linear growth, which is what we would have if there was only the outer for-loop. However, for this problem the quadratic growth is not usually a problem since the degree tends to be low — rarely more than 10. If one has more points than this the general advice is to use some other approximation method which avoids high degree polynomials since these are also likely to lead to large rounding-errors.

The second algorithm that is needed is evaluation of the interpolation polynomial (9.35). Let us consider a specific example,

$$p_3(x) = f_0 + f_1(x - x_0) + f_2(x - x_0)(x - x_1) + f_3(x - x_0)(x - x_1)(x - x_2). \quad (9.36)$$

Given a number x, there is an elegant algorithm for computing the value of the polynomial which is based on rewriting (9.36) slightly as

$$p_3(x) = f_0 + (x - x_0)\Big(f_1 + (x - x_1)\big(f_2 + (x - x_2)f_3\big)\Big). \quad (9.37)$$

To compute $p_3(x)$ we start from the inner-most parenthesis and then repeatedly multiply and add,

$$s_3 = f_3,$$
$$s_2 = (x - x_2)s_3 + f_2,$$
$$s_1 = (x - x_1)s_2 + f_1,$$
$$s_0 = (x - x_0)s_1 + f_0.$$

After this we see that $s_0 = p_3(x)$. This can easily be generalised to a more formal algorithm. Note that there is no need to keep the different (s_i)-values; we can just use one variable s and accumulate the calculations in this.

Algorithm 9.27 (Horner's rule). *Let x_0, ..., x_n be given numbers, and let $(f_k)_{k=0}^n$ be the coefficients of the polynomial*

$$p_n(x) = f_0 + f_1(x - x_0) + \cdots + f_n(x - x_0) \cdots (x - x_{n-1}). \qquad (9.38)$$

After the code

```
s := f_n;
for k := n - 1, n - 2, ... 0
    s := (x - x_k) * s + f_k;
```

the variable s will contain the value of $p_n(x)$.

The total number of arithmetic operations in this algorithm is n additions, subtractions and multiplications. On the other hand, the last term in (9.38) on its own requires n multiplications. The simple reformulation (9.37) is therefore almost like magic.

9.2.5 Interpolation error

The interpolating polynomial p_n is an approximation to f, but unless f itself is a polynomial of degree n, there will be a nonzero error $e(x) = f(x) - p_n(x)$, see exercise 4. At times it is useful to have an explicit expression for the error.

Theorem 9.28. *Suppose f is interpolated by a polynomial of degree n at $n+1$ distinct points x_0, ..., x_n. Let $[a, b]$ be the smallest interval that contains all the interpolation points as well as the number x, and suppose that the function f has continuous derivatives up to order $n + 1$ in $[a, b]$. Then the error $e(x) = f(x) - p_n(x)$ is given by*

$$e(x) = f[x_0, \ldots, x_n, x](x - x_0) \cdots (x - x_n) = \frac{f^{(n+1)}(\xi_x)}{(n+1)!}(x - x_0) \ldots (x - x_n), \quad (9.39)$$

where ξ_x is a number in the interval (a, b) that depends on x.

Proof. The second equality in (9.39) follows from (9.33), so we need to prove the first equality. For this we add the (arbitrary) number x as an interpolation point and consider interpolation with a polynomial of degree $n + 1$ at the points x_0, ..., x_n, x. We use t as the free variable to avoid confusion with x. Then we know that

$$p_{n+1}(t) = p_n(t) + f[x_0, \ldots, x_n, x](t - x_0) \cdots (t - x_n).$$

196
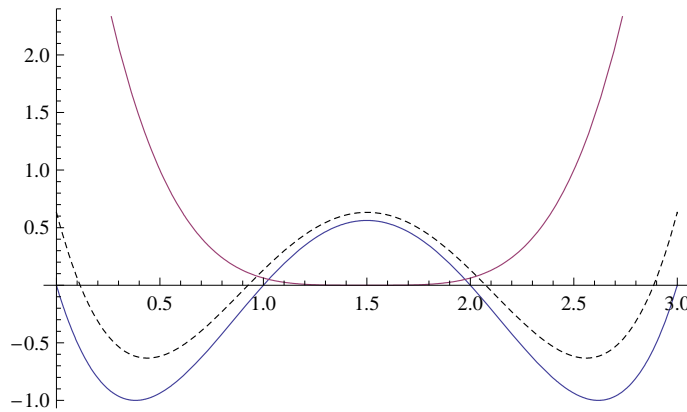
Figure 9.7. The solid, nonnegative graph is the polynomial factor $(x-3/2)^4$ in the error term for Taylor expansion of degree 3 about the $a = 3/2$, while the other solid graph is the polynomial part $x(x-1)(x-2)(x-3)$ of the error term for interpolation at 0, 1, 2, 3. The dashed graph is the smallest possible polynomial part of an error terms for interpolation at 4 points in $[0,3]$.

Since p_{n+1} interpolates f at $t = x$ we have $p_{n+1}(x) = f(x)$ so

$$f(x) = p_n(x) + f[x_0, \ldots, x_n, x](x - x_0) \cdots (x - x_n)$$

which proves the first relation in (9.39). ∎

Theorem 9.28 has obvious uses for assessing the error in polynomial interpolation and will prove very handy in later chapters.

The error term in (9.39) is very similar to the error term in the Taylor expansion (9.14). A natural question to ask is therefore: Which approximation method will give the smallest error, Taylor expansion or interpolation? Since the only essential difference between the two error terms is the factor $(x - a)^{n+1}$ in the Taylor case and the factor $(x - x_0) \cdots (x - x_n)$ in the interpolation case, a reasonable way to compare the methods is to compare the two polynomials $(x - a)^{n+1}$ and $(x - x_0) \cdots (x - x_n)$.

In reality, we do not just have two approximation methods but infinitely many, since there are infinitely many ways to choose the interpolation points. In figure 9.7 we compare the two most obvious choices in the case $n = 3$ for the interval $[0,3]$: Taylor expansion about the midpoint $a = 3/2$ and interpolation at the integers 0, 1, 2, and 3. In the Taylor case, the polynomial $(x-3/2)^4$ is nonnegative and small in the interval $[1,2]$, but outside this interval it grows quickly and soon becomes larger than the polynomial $x(x - 1)(x - 2)(x - 3)$ corresponding to interpolation at the integers. We have also included a plot of a third polynomial which corresponds to the best possible interpolation points in the sense

197
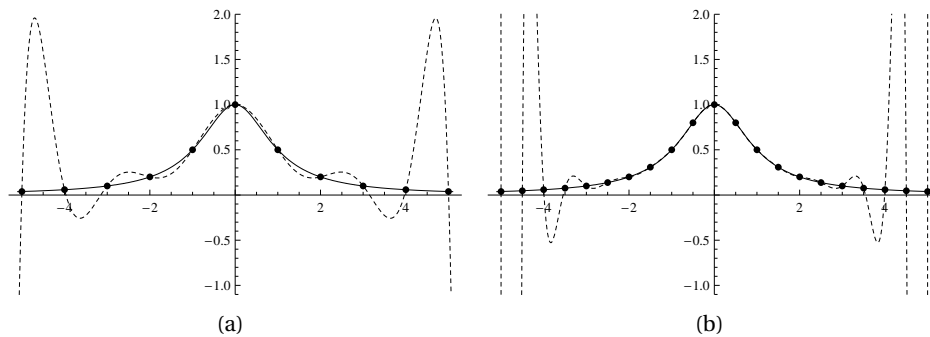
Figure 9.8. Interpolation of the function $f(x) = 1/(1 + x^2)$ on the interval $[-5, 5]$ with polynomials of degree 10 in (a), and degree 20 in (b). The points are uniformly distributed in the interval in each case.

that the maximum value of this polynomial is as small as possible in the interval $[0, 3]$, given that its leading coefficient should be 1.

If used sensibly, polynomial interpolation will usually provide a good approximation to the underlying data. As the distance between the data points decreases, either by increasing the number of points or by moving the points closer together, the approximation can be expected to become better. However, we saw that there are functions for which Taylor approximation does not work well, and the same may happen with interpolation. As for Taylor approximation, the problem arises when the derivatives of the function to be approximated become large. A famous example is the so-called Runge function $1/(1 + x^2)$ on the interval $[-5, 5]$. Figure 9.8 shows the interpolants for degree 10 and degree 20. In the middle of the interval, the error becomes smaller when the degree is increased, but towards the ends of the interval the error becomes larger when the degree increases.

9.3 Summary

In this chapter we have considered two different ways of constructing polynomial interpolants. We first reviewed Taylor polynomials briefly, and then studied polynomial interpolation in some detail. Taylor polynomials are for the main part a tool that is used for various pencil and paper investigations, while interpolation is often used as a tool for constructing numerical methods, as we will see in later chapters. Both Taylor polynomials and polynomial interpolation are methods of approximation and so it is important to keep track of the error, which is why the error formulas are important.

In this chapter we have used polynomials all the time, but have written them in different forms. This illustrates the important principle that there are many

198

different ways to write polynomials, and a problem may simplify considerably by adapting the form of the polynomial to the problem at hand.

Exercises

9.1 The Taylor polynomials of e^x, $\cos x$ and $\sin x$ expanded around zero are

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + \frac{x^6}{720} + \frac{x^7}{5040} + \cdots$$

$$\cos x = 1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} + \cdots$$

$$\sin x = x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + \cdots$$

Calculate the Taylor polynomial of the complex exponential e^{ix}, compare with the Taylor polynomials above, and explain why Euler's formula $e^{ix} = \cos x + i \sin x$ is reasonable.

9.2 The data

x	0	1	3	4
$f(x)$	1	0	2	1

are given.

a) Determine the Lagrange form of the cubic polynomial that interpolates the data.

b) Determine the Newton form of the interpolating polynomial.

c) Verify that the solutions in (a) and (b) are the same.

9.3 Use algorithm 9.26 to compute the divided differences needed to determine the Newton form of the interpolating poynomial in exercise 2. Verify that no data are lost when variables are overwritten.

9.4   **a)** We have the data

x	0	1	2
$f(x)$	2	1	0

which have been sampled from the straight line $y = 2 - x$. Determine the Newton form of the quadratic, interpolating polynomial, and compare it to the straight line. What is the difference?

b) Suppose we are doing interpolation at x_0, ..., x_n with polynomials of degree n. Show that if the function f to be interpolated is a polynomial p of degree n, then the interpolant p_n will be identically equal to p. How does this explain the result in (a)?

9.5 Suppose we have the data

$$(0, y_0), \quad (1, y_1), \quad (2, y_2), \quad (3, y_3) \tag{9.40}$$

where we think of $y_i = f(i)$ as values being sampled from an unknown function f. In this problem we are going to find formulas that approximate f at various points using cubic interpolation.

199

a) Determine the straight line p_1 that interpolates the two middle points in (9.40), and use $p_1(3/2)$ as an approximation to $f(3/2)$. Show that

$$f(3/2) \approx p_1(3/2) = \frac{1}{2}\big(f(1) + f(2)\big).$$

Find an expression for the error.

b) Determine the cubic polynomial p_3 that interpolates the data (9.40) and use $p_3(3/2)$ as an approximation to $f(3/2)$. Show that then

$$f(3/2) \approx p_3(3/2) = \frac{-y_0 + 9y_1 - 9y_2 + y_3}{16}.$$

What is the error?

c) Sometimes we need to estimate f outside the interval that contains the interpolation points; this is called *extrapolation*. Use the same approach as in (a), but find an approximation to $f(4)$. What is the error?

9.6 **a)** The data

x	0	1	2	3
$f(x)$	0	1	4	9

is sampled from the function $f(x) = x^2$. Determine the third order divided difference $f[0,1,2,3]$.

b) Explain why we always have $f[x_0,\dots,x_n] = 0$ if f is a polynomial of degree at most $n-1$.

CHAPTER 10

Zeros of Functions

An important part of the maths syllabus in secondary school is equation solving. This is important for the simple reason that equations are important — a wide range of problems can be translated into an equation, and by solving the equation we solve the problem. We will discuss a couple of examples in section 10.1.

In school you should have learnt to solve linear and quadratic equations, some trigonometric equations, some equations involving logarithms and exponential functions, as well as systems of two or three equations. Solving these equations usually follow a fixed recipe, and it may well be that you managed to solve all the equations that you encountered in school. For this reason you may believe that the problem of solving equations is — a solved problem.

The truth is that most equations cannot be solved by traditional pencil-and-paper methods. And even for equations that can be solved in this way, the expressions may become so complicated that they are almost useless for many purposes. Consider for example the equation

$$x^3 - 3x + 1 = 0. \tag{10.1}$$

The Norwegian mathematician Niels Henrik Abel proved that all polynomial equations of degree less than five can be solved by extracting roots, so we know there is a formula for the solutions. The program Mathematica will tell us that there are three solutions, one real and two complex. The real solution is given by

$$\frac{-20\sqrt[3]{\dfrac{3}{-9+\sqrt{12081}}} + \sqrt[3]{2\left(-9+\sqrt{12081}\right)}}{6^{2/3}}.$$

Although more complicated than the solution of a quadratic equation, this is not

201

so bad. However, the solution becomes much more complicated for equations of degree 4. For example, the equation

$$x^4 - 3x + 1 = 0$$

has two complex and two real solutions, and one of the real solutions is given by

$$\frac{\sqrt{\sqrt[3]{81 - \sqrt{5793}} + \sqrt[3]{81 + \sqrt{5793}}}}{2\sqrt[6]{2}\sqrt[3]{3}}$$

$$+ \frac{1}{2}\sqrt{\frac{1}{3}\left(-\sqrt[3]{\frac{3}{2}}\left(81 - \sqrt{5793}\right) - \sqrt[3]{\frac{3}{2}}\left(81 + \sqrt{5793}\right)\right.}$$

$$\left. + \frac{18\sqrt[6]{2}\sqrt[3]{3}}{\sqrt{\sqrt[3]{81 - \sqrt{5793}} + \sqrt[3]{81 + \sqrt{5793}}}}\right)$$

(the square root in the second line extends to end of the third line).

In this chapter we are going to approach the problem of solving equations in a completely different way. Instead of looking for exact solutions, we are going to derive numerical methods that can compute approximations to the roots, with whatever accuracy is desired (or possible with the computer resources you have available). In most situations numerical approximations are also preferable for equations where the exact solutions can be found. For example the given root of the cubic equation above with 20 correct digits is -0.09990029880547284209, while the given solution of the quartic equation is 1.3074861009619814743 with the same accuracy. For most purposes this is much more informative than the large expressions above.

10.1 The need for numerical root finding

In this chapter we are going to derive three numerical methods for solving equations: the Bisection method, the Secant method and Newton's method. Before deriving these methods, we consider two practical examples where there is a need to solve equations.

10.1.1 Analysing difference equations

In chapter 6 we studied difference equations and saw that they can easily be simulated on a computer. However, we also saw that the computed solution may be completely overwhelmed by round-off errors so that the true solution is completely lost. Whether or not this will happen depends on the size of the

roots of the characteristic equation of the difference equation. As an example, consider the difference equation

$$16x_{n+5} + 5x_{n+4} - 70x_{n+3} - 24x_{n+2} + 56x_{n+1} - 16x_n = 0$$

whose characteristic equation is

$$16r^5 + 5r^4 - 70r^3 - 24r^2 + 56r + 16 = 0.$$

It is impossible to find exact formulas for the roots of this equation. However, by using numerical methods like the ones we are going to derive in this chapter, one quite easily finds that the five roots are (with five-digit accuracy)

$$r_1 = -1.7761, \quad r_2 = -1.0985, \quad r_3 = -0.27959, \quad r_4 = 0.99015, \quad r_5 = 1.8515.$$

From this we see that the largest root is $r_5 \approx 1.85$. This means that regardless of the initial values, the computed (simulated) solution will eventually be dominated by the term r_5^n.

10.1.2   Labelling plots

A completely different example where there is a need for finding zeros of functions is illustrated in figure 10.1 which is taken from chapter 9. This figure has nine labels of the form $n = 2k - 1$ for $k = 1, \ldots, 9$, that are placed either directly above the point where the corresponding graph intersects the horizontal line $y = 2$ or below the point where the graph intersects the line $y = -2$. It would be possible to use an interactive drawing program and place the labels manually, but this is both tedious and time consuming, and it would be difficult to place all the labels consistently. With access to an environment for producing plots that is also programmable, it is possible to compute the exact position of the label.

Consider for example the label $n = 9$ which is to be placed above the point where the Taylor polynomial of $\sin x$, expanded about $a = 0$, intersects the line $y = 2$. The Taylor polynomial is given by

$$p(x) = x - \frac{x^3}{6} + \frac{x^5}{720} - \frac{x^7}{5040} + \frac{x^9}{362880}$$

so the x-value at the intersection point is given by the equation $p(x) = 2$, i.e., we have to solve the equation

$$x - \frac{x^3}{6} + \frac{x^5}{720} - \frac{x^7}{5040} + \frac{x^9}{362880} - 2 = 0.$$
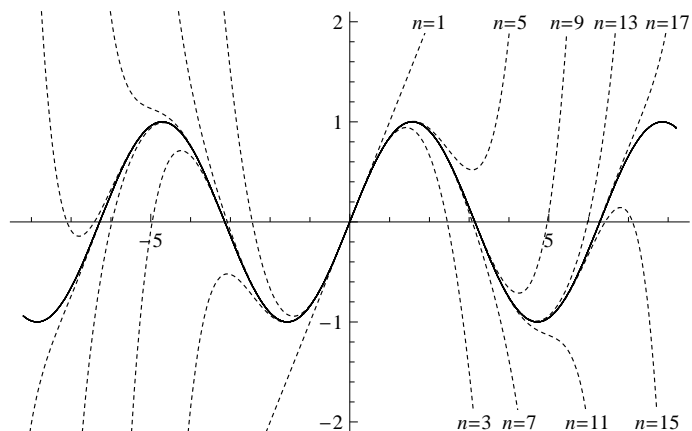
Figure 10.1. Plot with automatically placed labels.

This equation may have as many as nine real solutions, but from the plot we see that the one we are interested in is close to $x = 5$. Numerical methods for finding roots usually require a starting value near the root, and in our case it is reasonable to use 5 as starting value. If we do this and use a method like one of those derived later in this chapter, we find that the intersection between $p(x)$ and the horizontal line $y = 2$ is at $x = 5.4683$. This means that the label $n = 9$ should be drawn at the point with position $(5.4683, 2)$.

The position of the other labels may be determined similarly. In fact, this procedure may be incorporated in a program with a loop where k runs from 1 to 9. For each k, we determine the Taylor polynomial p_{2k-1} and plot it, compute the intersection x_k with $y = (-1)^{k+1}2$, and draw the label $n = 2k - 1$ at $(x_k, (-1)^{k+1}2)$. This is exactly how figure 10.1 was produced, using Mathematica.

10.2   The Bisection method

There are a large number of numerical methods for computing roots of equations, but the simplest of all is the *Bisection method*. Before we describe the method, let us review a basic mathematical result which forms the basis for the method.

10.2.1   The intermediate value theorem

The mean value theorem is illustrated in figure 10.2a. It basically says that if a function is positive at one point and negative at another, it must be zero somewhere in between.
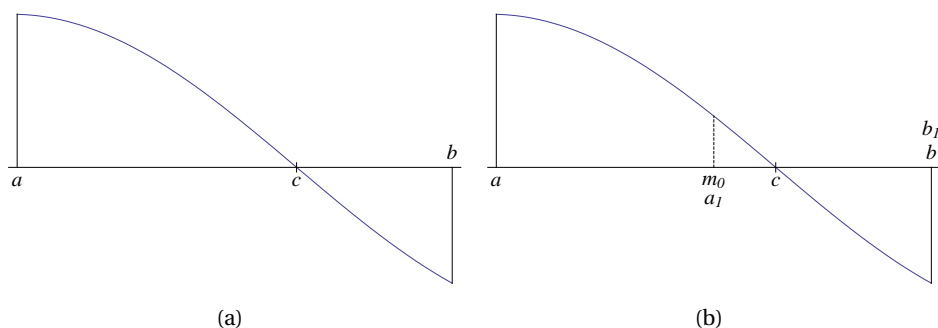
Figure 10.2. Illustration of the mean value theorem (a), and the first step of the Bisection method (b).

Theorem 10.1 (Intermediate value theorem). *Suppose f is a function that is continuous on the interval $[a, b]$ and has opposite signs at a and b. Then there is a real number $c \in (a, b)$ such that $f(c) = 0$.*

This result seems obvious since f is assumed to be continuous, but the proof, which can be found in a standard calculus book, is not so simple. It may be easier to appreciate if we try to work with rational numbers only. Consider for example the function $f(x) = x^2 - 2$ on the interval $[a, b] = [0, 2]$. This function satisfies $f(0) < 0$ and $f(2) > 0$, but there is no rational number c such that $f(c) = 0$. The zero in this case is of course $c = \sqrt{2}$, which is irrational, so the main content of theorem is basically that there are no gaps in the real numbers.

10.2.2   Derivation of the Bisection method

The intermediate value theorem only tells that f must have a zero, but it says nothing about how to find this zero. However, based on the theorem it is easy to devise a method for finding good approximations to the zero.

To start with, we know that f has opposite signs at the two ends of the interval $[a, b]$. Our aim is to find a new interval $[a_1, b_1]$, which is smaller than $[a, b]$, such that f also has opposite signs at the two ends of $[a_1, b_1]$. But this is not difficult: We use the midpoint $m_0 = (a + b)/2$ of the interval $[a, b]$ and compute $f(m_0)$. If $f(m_0) = 0$, we are very happy because we have found the zero. If this is not the case, the sign of $f(m_0)$ must either be equal to the sign of $f(a)$ or the sign of $f(b)$. If $f(m_0)$ and $f(a)$ have the same sign, we set $[a_1, b_1] = [m_0, b]$; if $f(m_0)$ and $f(b)$ have the same sign, we set $[a_1, b_1] = [a, m_0]$. The construction is illustrated in figure 10.2b.

The discussion in the previous paragraph shows how we may construct a new interval $[a_1, b_1]$, with a width that is half that of $[a, b]$, with the property that

205

(a)

(b)

(c)

(d)
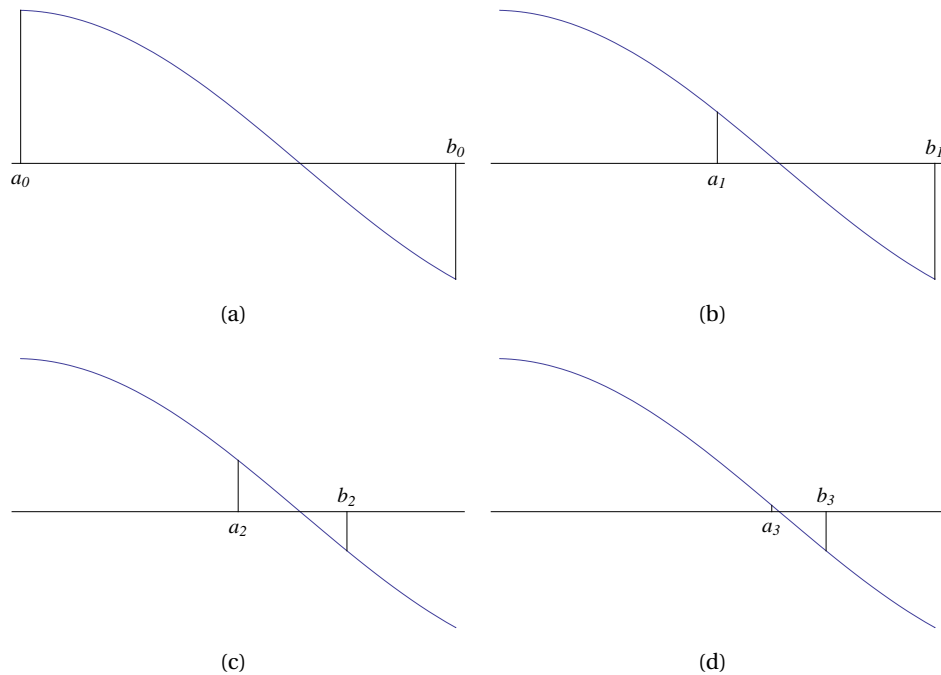
Figure 10.3. The first four steps of the bisection algorithm.

f is also guaranteed to have a zero in $[a_1, b_1]$. But then we may of course continue the process in the same way and determine another interval $[a_2, b_2]$ that is half the width of $[a_1, b_1]$, and such that f is guaranteed to have a zero in $[a_2, b_2]$. This process can obviously be continued until we hit a zero or the interval has become so small that the zero is determined with sufficient accuracy.

Algorithm 10.2 (Bisection method). *Let f be a continuous function that has opposite signs at the two ends of the interval $[a, b]$. The following algorithm computes an approximation m_N to a zero $c \in (a, b)$ after N bisections:*

> $a_0 := a;$
> $b_0 := b;$
> **for** $i := 1, 2, \ldots, N$
> $m_{i-1} := (a_{i-1} + b_{i-1})/2;$
>   **if** $f(m_{i-1}) = 0$
> $a_i := b_i := m_{i-1};$
>   **if** $f(a_{i-1})f(m_{i-1}) < 0$
> $a_i := a_{i-1};$

206
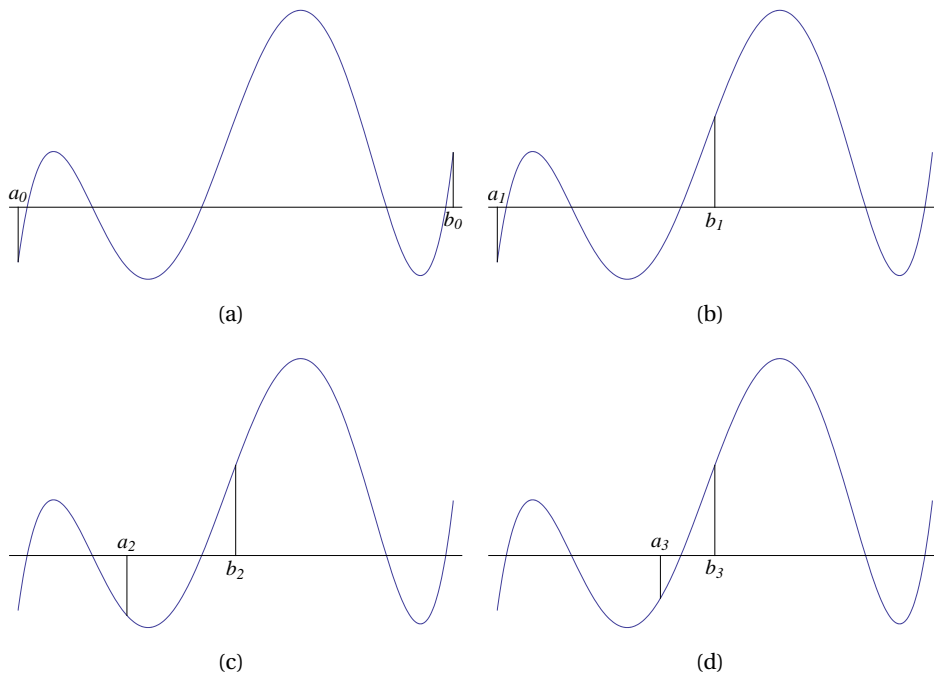
Figure 10.4. The first four steps of the bisection algorithm for a function with five zeros in the initial interval.

$$b_i := m_{i-1};$$
$$\textbf{else}$$
$$a_i := m_{i-1};$$
$$b_i := b_{i-1};$$

$$m_N := (a_N + b_N)/2;$$

This algorithm is just a formalisation of the discussion above. The for loop starts with an interval $[a_{i-1}, b_{i-1}]$ with the property that $f(a_{i-1})f(b_{i-1}) < 0$. It usually produces a new interval $[a_i, b_i]$ of half the width of $[a_{i-1}, b_{i-1}]$, such that $f(a_i)f(b_i) < 0$. The exception is if we hit a zero c, then the width of the interval becomes 0. Initially, we start with $[a_0, b_0] = [a, b]$.

The first four steps of the Bisection method for the example in figure 10.2 are shown in figure 10.3. An example where there are several zeros in the original interval is shown in figure 10.4. In general, it is difficult to predict which zero the algorithm zooms in on, so it is best to choose the initial interval such that it only contains one zero.

10.2.3 Error analysis

Algorithm 10.18 does N subdivisions and then stops, but it would be more desirable if the loop runs until the error is sufficiently small. In order to do this, we need to know how large the error is.

If we know that a function f has a zero c in the interval $[a, b]$, and we use the midpoint $m = (a + b)/2$ as an approximation to the zero, what can we say about the error? The worst situation is if the zero is far from the midpoint, and the furthest from the midpoint we can get, is a or b in which case the error is $(b - a)/2$. This gives the following lemma.

Lemma 10.3. *Suppose f is a function with a zero c in the interval $[a, b]$. If the midpoint $m = (a + b)/2$ of $[a, b]$ is used as an approximation to c, the error is at most*
$$|c - m| \leq \frac{b - a}{2}.$$

This simple tool is what we need to estimate the error in the Bisection method. Each bisection obviously halves the width of the interval, so the error is also halved each time.

Theorem 10.4. *Suppose f is a function with only one zero c in the interval $[a, b]$ and let $\{m_i\}$ denote the successive midpoints computed by the Bisection method. After N iterations, the error is bounded by*

$$|c - m_N| \leq \frac{b - a}{2^{N+1}}. \tag{10.2}$$

As N tends to infinity, the midpoints m_N will converge to the zero c.

Here we have emphasised that f should have only one zero in $[a, b]$. If there are several zeros, an estimate like (10.2) still holds for one of the zeros, but it is difficult to say in advance which one.

This simple result allows us to control the number of steps necessary to achieve a certain error ϵ. For in order to ensure that the error is smaller than ϵ it is clearly sufficient to demand that the upper bound in the inequality (10.2) is smaller than ϵ,
$$|c - m_N| \leq \frac{b - a}{2^{N+1}} \leq \epsilon.$$
The second inequality can be solved for N by taking logarithms. This yields
$$\ln(b - a) - (N + 1)\ln 2 \leq \ln \epsilon$$

which leads to the following observation.

A simple word of advice: Do not try and remember the formula (10.3). It is much better to understand (and thereby remember) how it was derived.

Example 10.6. Suppose we want to find the zero $\sqrt{2}$ with error smaller than 10^{-10} by solving the equation $f(x) = x^2 - 2$. We have $f(1) = -1$ and $f(2) = 2$, so we can use the Bisection method, starting with the interval $[1,2]$. To get the error to be smaller than 10^{-10}, we know that N should be larger than

$$\frac{\ln(b-a) - \ln\epsilon}{\ln 2} - 1 = \frac{10\ln 10}{\ln 2} - 1 \approx 32.2.$$

Since N needs to be an integer this shows that $N = 33$ is guaranteed to make the error smaller than 10^{-10}. If we run algorithm 10.18 we find

$$m_0 = 1.50000000000,$$
$$m_1 = 1.25000000000,$$
$$m_2 = 1.37500000000,$$
$$\vdots$$
$$m_{33} = 1.41421356233.$$

We have $\sqrt{2} \approx 1.41421356237$ with eleven correct digits, and the actual error in m_{33} is approximately 4.7×10^{-11}. ∎

Recall that when we are working with floating-point numbers, the relative error is a better error measure than the absolute error. The relative error after i iterations is given by

$$\frac{|c - m_i|}{|c|}.$$

From the inequality (10.2) we have an upper bound on the numerator. Recall also that generally one is only interested in a rough estimate of the relative error. It is therefore reasonable to approximate c by m_i.

> **Observation 10.7.** *After i iterations with the Bisection method, the relative error is approximately bounded by*
>
> $$\frac{b-a}{|m_i|2^{i+1}}. \tag{10.4}$$

One may wonder if it is possible to estimate beforehand how many iterations are needed to make the relative error smaller than some given tolerance, like we did in observation 10.5 for the absolute error. This would require some a priori estimate of the zero c or the approximation m_i, which is hardly possible.

Recall from observation 5.3 that if the relative error in an approximation $\tilde{c}$ to c is of magnitude 10^{-m}, then c and $\tilde{c}$ have roughly m decimal digits in common. This is easily generalised to the fact that if the relative error is roughly 2^{-m}, then c and $\tilde{c}$ have roughly m binary digits in common. Observation 10.7 shows that the relative error in the Bisection method is roughly halved during each iteration (the variation in m_i will not vary much in magnitude with i). But this means that the number of correct bits increases by one in each iteration. Since 32-bit floating-point numbers use 24 bits for the significand and 64-bit floating-point numbers 54 bits, we can make the following observation.

> **Observation 10.8.** *The number of correct bits in the approximations to a zero generated by the Bisection method increases by 1 per iteration. With 32-bit floating-point numbers, full accuracy is obtained after 24 iterations, while full accuracy is obtained after 54 iterations with 64-bit floating-point iterations.*

10.2.4   Revised algorithm

If we look back on algorithm 10.18, there are several improvements we can make. We certainly do not need to keep track of all the subintervals and midpoints, we only need the last one. It is therefore sufficient to have the variables a, b and m for this purpose. More importantly, we should use the idea from the previous section and let the number of iterations be determined by the requested accuracy. Given some tolerance $\epsilon > 0$, we could then estimate N as in observation 10.5. This is certainly possible, but remember that the absolute error may be an inadequate measure of the error if the numbers are far from 1 in size. Instead we should use the relative error. We use an integer counter i and the expression in (10.4) (with N replaced by i) to estimate the relative error. We stop the computations when i becomes so large that

$$\frac{b-a}{|m_i|2^{i+1}} \leq \epsilon.$$

This condition becomes problematic if m_i should become 0. We therefore use the equivalent test

$$\frac{b-a}{2^{i+1}} \leq \epsilon|m_i|$$

instead. If m_i should become 0 for an i, this inequality will be virtually impossible to satisfy, so the computations will just continue.

Algorithm 10.9 (Revised Bisection method). *Let f be a continuous function that has opposite signs at the two ends of the interval $[a, b]$. The following algorithm attempts to compute an approximation m to a zero $c \in (a, b)$ with relative error at most ϵ, using at most N bisections:*

    $i = 0;$
    $m := (a + b)/2;$
    $abserr := (b - a)/2;$
    while $i \leq N$ **and** $abserr > \epsilon|m|$
        if $f(m) = 0$
            $a := b := m;$
        if $f(a)f(m) < 0$
            $b := m;$
        else
            $a := m;$
        $i := i + 1;$
        $m := (a + b)/2;$
        $abserr := (b - a)/2;$

In the while loop we have also added a test which ensures that the while loop does not run forever. This is good practice to ensure that your program does not enter an infinite loop because some unforeseen situation occurs that prevents the error from becoming smaller than ϵ.

It must be emphasised that algorithm 10.9 lacks many details. For instance, the algorithm should probably terminate if $|f(m)|$ becomes small in some sense, not just when it becomes zero. And there is no need to perform more iterations than roughly the number of bits in the significand of the type of floating-point numbers used. However, the basic principle of the Bisection method is illustrated by the algorithm, and the extra details belong to the area of professional software development.
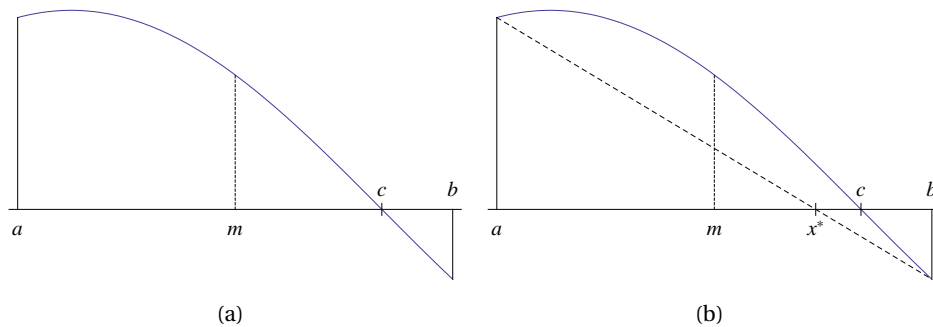
Figure 10.5. An example of the first step with the Bisection method (a), and the alternative approximation to the zero provided by the secant (b).

10.3 The Secant method

The Bisection method is robust and uses only the sign of $f(x)$ at the end points and the successive midpoints to compute an approximation to a zero. In many cases though, the method appears rather unintelligent. An example is shown in figure 10.5a. The values of f at a and b indicate that the zero should be close to b, but still the Bisection method uses the midpoint as the guess for the zero.

10.3.1 Basic idea

The idea behind the Secant method is to use the zero of the secant between $(a, f(a))$ and $(b, f(b))$ as an approximation to the zero instead of the midpoint, as shown in figure 10.5b.

Idea 10.10 (Secant idea). *Let f be a continuous function, let a and b be two points in its domain, and let*

$$s(x) = f(a) + \frac{f(b) - f(a)}{b - a}(x - a)$$

be the secant between the two points $(a, f(a))$ and $(b, f(b))$. The Secant method uses the zero

$$x^* = b - \frac{b - a}{f(b) - f(a)} f(b) \tag{10.5}$$

of the secant as an approximation to a zero of f.

We observe that the secant is symmetric in the two numbers a and b, so the

212

(a)

(b)

(c)

(d)
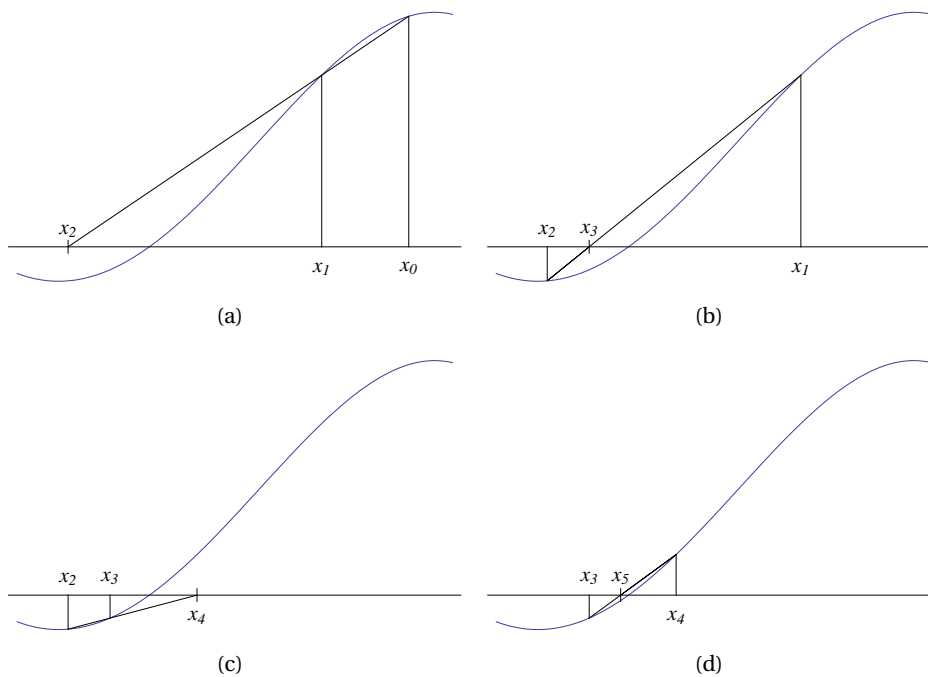
Figure 10.6. An example of the first four steps of the Secant method.

formula (10.8) may also be written

$$x^* = a - \frac{b-a}{f(b) - f(a)} f(a).$$

In the Secant method it is convenient to label a and b as $x_0 = a$ and $x_1 = b$ and denote the zero x^* by x_2. We are then in a position where we may repeat the formula: From the two numbers x_0 and x_1, we compute the approximate zero x_2, then from the two numbers x_1 and x_2 we compute the approximate zero x_3, from x_2 and x_3 we compute the approximate zero x_4, and so on. This is the basic Secant method, and an example of the first few iterations of the method is shows in figure 10.6. Note how the method quite quickly zooms in on the zero.

Algorithm 10.11 (Basic Secant method). *Let f be a continuous function and let x_0 and x_1 be two given numbers in its domain. The sequence $\{x_i\}_{i=0}^{N}$ given by*

$$x_i = x_{i-1} - \frac{x_{i-1} - x_{i-2}}{f(x_{i-1}) - f(x_{i-2})} f(x_{i-1}), \quad i = 2, 3, \ldots, N, \qquad (10.6)$$
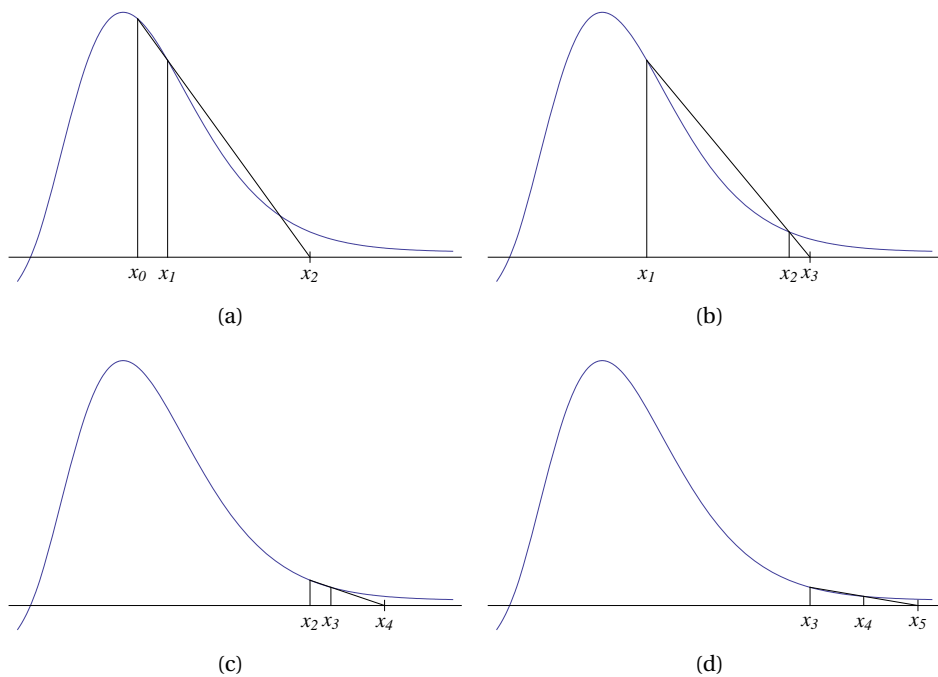
Figure 10.7. An example where the Secant method fails.

> *will in certain situations converge to a zero of f.*

It is important to realise that unlike the Bisection method, the Secant method may fail. One such example is shown in figure 10.7. The problem here is that the two starting values are too far away from the zero to the left in the plot, and the algorithm gets stuck in the area to the right where the function is small, without ever becoming 0. This explains the expression "will in certain situations converge" in algorithm 10.11.

10.3.2   Testing for convergence

Algorithm 10.11 provides the basis for the secant algorithm. However, rather than just do N iterations, it would be more useful to stop the iterations when a certain accuracy has been attained. This turns out to be more difficult for the Secant method than for the Bisection method since there is not such an explicit error estimate available for the Secant method.

The Secant method produces a sequence of approximations x_0, x_1, ... to a zero, and we want to decide when we are within a tolerance ϵ of the zero. We

will often find ourselves in this kind of situation: Some algorithm produces a sequence of approximations, and we want to check whether we have convergence.

When we come close to the zero, the difference between successive approximations will necessarily become small. If we are working with the absolute error, it is therefore common to use the number $|x_n - x_{n-1}|$ as a measure of the absolute error at iteration no. n. If we want to stop when the absolute error is smaller than ϵ, the condition then becomes $|x_n - x_{n-1}| \le \epsilon$.

Usually, it is preferable to work with the relative error, and then we need an estimate for the zero as well. At step n of the algorithm, the best approximation we have for the zero is the latest approximation, x_n. The estimate for the relative error at step n is therefore

$$\frac{|x_n - x_{n-1}|}{|x_n|}.$$

To test whether the relative error is less than or equal to ϵ, we would then use the condition $|x_n - x_{n-1}| \le \epsilon |x_n|$. We emphasise that this is certainly not exact, and this kind of test cannot guarantee that the error is smaller than ϵ. But in the absence of anything better, this kind of strategy is often used.

Observation 10.12. *Suppose that an algorithm generates a sequence $\{x_n\}$. The absolute error in the terms x_n is then often estimated by $|x_n - x_{n-1}|$, and the relative error by $|x_n - x_{n-1}|/|x_n|$. To test whether the relative error is smaller than ϵ, the condition*

$$|x_n - x_{n-1}| \le \epsilon |x_n|$$

is often used.

When computing zeros of functions, there is one more ingredient that is often used. At the zero c we obviously have $f(c) = 0$. It is therefore reasonable to think that if $f(x_n)$ is small, then x_n is close to a zero. It is easy to construct functions where this is not the case. Consider for example the function $f(x) = x^2 + 10^{-30}$. This function is positive everywhere, but becomes as small as 10^{-30} at $x = 0$. Without going into further detail, we therefore omit this kind of convergence testing, although it may work well in certain situations.

10.3.3 Revised algorithm

The following is a more detailed algorithm for the Secant method, where the test for convergence is based on the discussion above.

Algorithm 10.13 (Revised Secant method). *Let f be a continuous function, and let x_0 and x_1 be two distinct initial approximations to a zero of f. The following algorithm attempts to compute an approximation z to a zero with relative error less than $\epsilon < 1$, using at most N iterations:*

> $i = 0;$
> $xpp := x_0;$
> $xp := z := x_1;$
> $abserr := |z|;$
> **while** $i \le N$ **and** $abserr \ge \epsilon|z|$
> $\quad z := xp - f(xp)(xp - xpp)/\big(f(xp) - f(xpp)\big);$
> $\quad abserr := |z - xp|;$
> $\quad xpp := xp;$
> $\quad xp := z;$
> $\quad i := i + 1;$

Since we are only interested in the final approximation of the root, there is no point in keeping track of all the approximations. All we need to compute the next approximation z, is the two previous approximations which we call xp and xpp, just like in simulation of second order difference equations (in fact, the iteration (10.6) in the Secant method can be viewed as the simulation of a non-linear, second-order, difference equation). Before we enter the while loop, we have to make sure that the test of convergence does not become true straight-away. The first time through the loop, the test for convergence is $|z| \ge \epsilon|z|$ which will always be true (even if $z = 0$), since ϵ is assumed to be smaller than 1.

10.3.4   Convergence and convergence order of the Secant method

So far we have focused on the algorithmic aspect of the Secant method, but an important question is obviously whether or not the sequence generated by the algorithm converges to a zero. As we have seen, this is certainly not always the case, but if f satisfies some reasonable conditions and we choose the starting values near a zero, the sequence generated by the algorithm will converge to the zero.

Theorem 10.14 (Convergence of the Secant method). *Suppose that f and its first two derivatives are continuous in an interval I that contains a zero c of f, and suppose that there exists a positive constant γ such that $|f'(x)| > \gamma > 0$ for all x in I. Then there exists a constant K such that for all starting values x_0*

216

and x_1 *sufficiently close to c, the sequence produced by the Secant method will converge to c and the error $e_n = c - x_n$ will satisfy*

$$|e_n| \leq K|e_{n-1}|^r, \quad n = 2, 3, \ldots, \tag{10.7}$$

where

$$r = \frac{1}{2}(1 + \sqrt{5}) \approx 1.618.$$

We are not going to prove this theorem which may appear rather overwhelming, but let us comment on some of the details.

First of all we note the assumptions: The function f and its first two derivatives must be continuous in an interval I that contains the zero c. In addition $|f'(x)|$ must be positive in this interval. This is always the case as long as $f'(c) \neq 0$, because then $f'(x)$ must also be nonzero near c. However, the Secant method works even if $f'(c) = 0$, it will just require more iterations than in the case when $f'(c)$ is nonzero.

The other assumption is that the starting values x_0 and x_1 are "sufficiently close to c". This is imprecise, but means that it is in fact possible to write down precisely how close x_0 and x_1 must be to c.

Provided the assumptions are satisfied, theorem 10.14 guarantees that the Secant method will converge to the zero. However, the inequality (10.7) also says something about how quickly the error goes to zero. Suppose that at some stage we have $e_k = 10^{-1}$ and that K is some number near 1. Then we find that

$$e_{k+1} \lesssim e_k^r = 10^{-r} \approx 10^{-1.618} \approx 2.41 \times 10^{-2},$$
$$e_{k+2} \lesssim e_{k+1}^r \lesssim e_k^{r^2} = 10^{-r^2} \approx 2.41 \times 10^{-3},$$
$$e_{k+3} \lesssim 5.81 \times 10^{-5},$$
$$e_{k+4} \lesssim 1.40 \times 10^{-7},$$
$$e_{k+5} \lesssim 8.15 \times 10^{-12},$$
$$e_{k+6} \lesssim 1.43 \times 10^{-18}.$$

This means that if the size of the root is approximately 1, and we manage to get the error to become 0.1, it will be as small as 10^{-18} (machine precision with 64-bit floating-point numbers) only six iterations later.

Example 10.16. Let us see if the predictions above happen in practice. We test the Secant method on the function $f(x) = x^2 - 2$ and attempt to compute the zero $c = \sqrt{2} \approx 1.41421356237309505$. We start with $x_0 = 2$ and $x_1 = 1.5$ and obtain

$$x_2 \approx 1.42857142857142857, \quad e_2 \approx 1.4 \times 10^{-2},$$
$$x_3 \approx 1.41463414634146341, \quad e_3 \approx 4.2 \times 10^{-4},$$
$$x_4 \approx 1.41421568627450980, \quad e_4 \approx 2.1 \times 10^{-6},$$
$$x_5 \approx 1.41421356268886964, \quad e_5 \approx 3.2 \times 10^{-10},$$
$$x_6 \approx 1.41421356237309529, \quad e_6 \approx 2.4 \times 10^{-16}.$$

This confirms the claim in observation 10.15. ∎

10.4 Newton's method

We are going to study a third method for finding roots of equations, namely Newton's method. This method is quite similar to the Secant method, and the description is quite similar, so we will be brief.

10.4.1 Basic idea

In the Secant method we used the secant as an approximation to f and the zero of the tangent as an approximation to the zero of f. In Newton's method we use the tangent of f instead.

Idea 10.17 (Newton's method)**.** *Let f be a continuous, differentiable function, let a be a point in its domain, and let*

$$T(x) = f(a) + f'(a)(x - a)$$

be the tangent of f at a. Newton's method uses the zero

$$x^* = a - \frac{f(a)}{f'(a)} \tag{10.8}$$

of the tangent as an approximation to a zero of f.

(a)   (b)

(c)   (d)
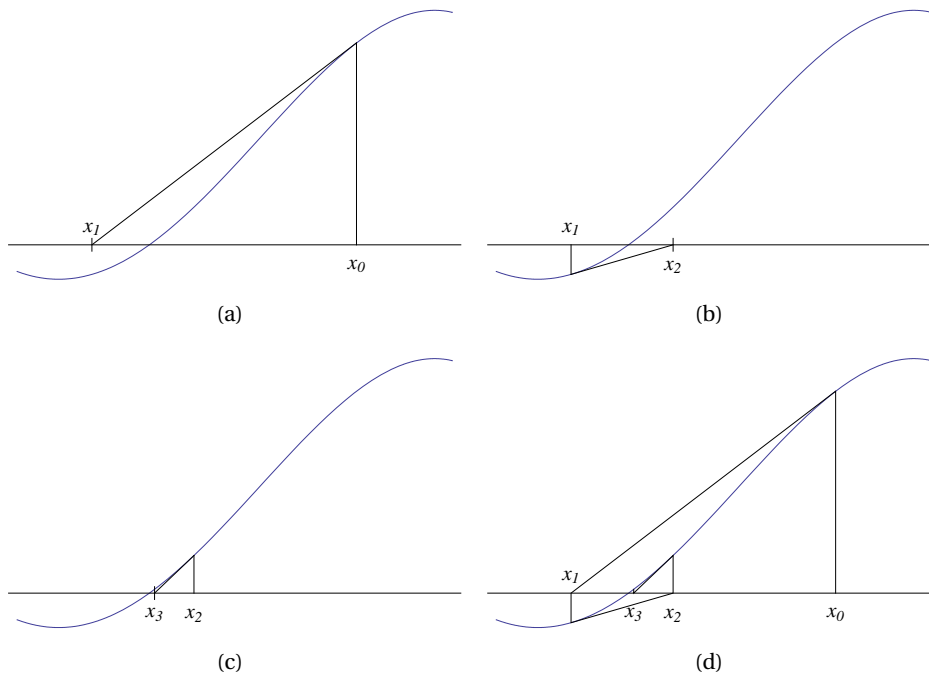
Figure 10.8. An example of the first three steps of Newton's method (a)–(c). The plot in shows a standard way of illustrating all three steps in one figure.

Newton's method is usually iterated, just like the Secant method. So if we start with $x_0 = a$, we compute the zero x_1 of the tangent at x_0. Then we repeat and compute the zero x_2 of the tangent at x_1, and so on,

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}, \quad n = 1, 2, \dots \tag{10.9}$$

The hope is that the resulting sequence $\{x_n\}$ will converge to a zero of f. Figure 10.8 illustrates the first three iterations with Newton's method for the example in figure 10.6.

An advantage of Newton's method compared to the Secant method is that only one starting value is needed since the iteration (10.9) is a first-order (non-linear) difference equation. On the other hand, it is sometimes a disadvantage that an explicit expression for the derivative is required.

10.4.2 Algorithm

Newton's method is very similar to the Secant method, and so is the algorithm. We measure the relative error in the same way, and therefore the stopping crite-
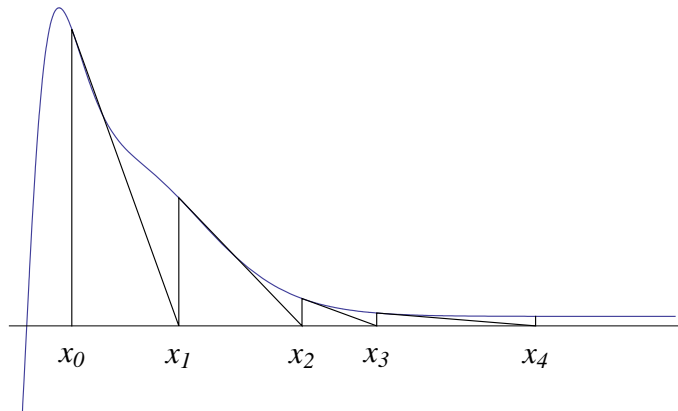
Figure 10.9. An example where Newton's method fails to converge because of a bad starting value.

rion is also exactly the same.

Algorithm 10.18 (Newton's method). *Let f be a continuous, differentiable function, and let x_0 be an initial approximation to a zero of f. The following algorithm attempts to compute an approximation z to a zero with relative error less than $\epsilon < 1$, using at most N iterations:*

$i = 0;$
$xp := z := x_0;$
$abserr := |z|;$
while $i \le N$ **and** $abserr \ge \epsilon|z|$
  $z := xp - f(xp)/f'(xp);$
  $abserr := |z - xp|;$
  $xp := z;$
  $i := i + 1;$

What may go wrong with this algorithm is that, like the Secant method, it may not converge, see the example in figure 10.9. Another possible problem is that we may in rare cases get division by zero in the first statement in the while loop.

10.4.3 Convergence and convergence order

The behaviour of Newton's method is very similar to that of the Secant method. One difference is that Newton's method is in fact easier to analyse since it is a

220

first-order difference equation. The equivalent of theorem 10.14 is therefore easier to prove in this case. The following lemma is a consequence of Taylor's formula.

Lemma 10.19. *Let c be a zero of f which is assumed to have continuous derivatives up to order 2, and let $e_n = x_n - c$ denote the error at iteration n in Newton's method. Then*

$$e_{n+1} = e_n^2 \frac{f''(\xi_n)}{2f'(x_n)}, \tag{10.10}$$

where ξ_n is a number in the interval (c, x_n) (the interval (x_n, c) if $x_n < c$).

Proof. The basic relation in Newton's method is

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

If we subtract the zero c on both sides we obtain

$$e_{n+1} = e_n - \frac{f(x_n)}{f'(x_n)} = \frac{e_n f'(x_n) - f(x_n)}{f'(x_n)}. \tag{10.11}$$

Consider now the Taylor exansion

$$f(c) = f(x_n) + (c - x_n)f'(x_n) + \frac{(c - x_n)^2}{2}f''(\xi_n),$$

where ξ_n is a number in the interval (x_n, c). Since $f(c) = 0$, this may be rewritten as

$$-f(x_n) + (x_n - c)f'(x_n) = \frac{(c - x_n)^2}{2}f''(\xi_n).$$

If we insert this in (10.11) we obtain the relation

$$e_{n+1} = e_n^2 \frac{f''(\xi_n)}{2f'(x_n)},$$

as required. ∎

Lemma 10.19 is the basis for proving that Newton's method converges. The result is the following theorem.

Theorem 10.20. *Suppose that f and its first two derivatives are continuous in an interval I that contains a zero c of f, and suppose that there exists a positive constant γ such that $|f'(x)| > \gamma > 0$ for all x in I. Then there exists a constant K such that for all initial values x_0 sufficiently close to c, the sequence produced by Newton's method will converge to c and the error $e_n = x_n - c$ will satisfy*

$$|e_{n+1}| \le K|e_n|^2, \quad n = 1, 2, \ldots \tag{10.12}$$

where K is some nonzero constant.

We will not prove this theorem, just comment on a few details. First of all we note that the assumptions are basically the same as the assumptions in the similar theorem 10.14 for the Secant method. The essential condition is that $f'(c) \ne 0$. Without this, the method still works, but the convergence is very slow.

The inequality (10.12) is obtained from (10.10) by taking the maximum of the expression $f''(x)/f'(y)$ on the right in (10.12), for all x and y in the interval I. If $f'(c) = 0$ this constant will not exist.

When we know that Newton's method converges, the relation (10.12) tells us how quickly it converges. If at some stage we have obtained $e_k \approx 10^{-1}$ and $K \approx 1$, we see that

$$e_{k+1} \approx e_k^2 \approx 10^{-2},$$
$$e_{k+2} \approx e_{k+1}^2 \approx 10^{-4},$$
$$e_{k+3} \approx e_{k+2}^2 \approx 10^{-8},$$
$$e_{k+4} \approx e_{k+3}^2 \approx 10^{-16}.$$

This means that if the root is approximately 1 in size and we somehow manage to reach an error of about 10^{-1}, we only need four more iterations to reach machine accuracy with 64-bit floating-point numbers. This shows the power of the relation (10.12). An algorithm for which the error satisfies this kind of relation is said to be *quadratically convergent.*

Observation 10.21. *When Newon's method converges to a zero c for which $f'(c) \ne 0$, the number of correct digits doubles per iteration.*

Let us end by redoing example 10.16 with Newton's method and checking observation 10.21 on a practical example.

Example 10.22. The equation is $f(x) = x^2 - 2$ which has the solution $c = \sqrt{2} \approx 1.41421356237309505$. If we run Newton's method with the initial value $x_0 = 1.7$,

we find

$$x_1 \approx 1.43823529411764706, \qquad e_2 \approx 2.3 \times 10^{-1},$$
$$x_2 \approx 1.41441417057620594, \qquad e_3 \approx 2.4 \times 10^{-2},$$
$$x_3 \approx 1.41421357659935635, \qquad e_4 \approx 2.0 \times 10^{-4},$$
$$x_4 \approx 1.41421356237309512, \qquad e_5 \approx 1.4 \times 10^{-8},$$
$$x_5 \approx 1.41421356237309505, \qquad e_6 \approx 7.2 \times 10^{-17}.$$

We see that although we only use one starting value, which is further from the root than best of the two starting values used with the Secant method, we still end up with a smaller error than with the Secant method after five iterations. ∎

10.5 Summary

We have considered three methods for computing zeros of functions, the Bisection method, the Secant method, and Newton's method. The Bisection method is robust and works for almost any kind of equations and any kind of zeros, but the convergence is relatively slow. The other two methods converge much faster when the root c is simple, i.e., when $f'(c) \neq 0$. The Secant method is then a bit slower than Newton's method, but it does not require knowledge of the derivative of f.

If $f'(c) = 0$, the Bisection method still converges with the same speed, as long as an interval where f has opposite signs at the ends can be found. In this situation the Secant method and Newton's method are not much faster than the Bisection method.

A major problem with all three methods is the need for starting values. This is especially true for the Secant method and Newton's method which may easily diverge if the starting values are not good enough. There are other, more advanced, methods available which converge as quickly as Newton's method, without requiring precise starting values.

If you try the algorithms in this chapter on some examples, you are very likely to discover that they do not always behave like you expect. The most likely problem is going to be the estimation of the (relative) error and therefore the stopping criteria for the while loops. We therefore emphasise that the algorithms given here are not at all fool-proof codes, but are primarily meant to illustrate the ideas behind the methods.

Out of the many other methods available for solving equations, one deserves to be mentioned specially. The *Regula Falsi method* is a mix between the Secant method and the Bisection method. It is reminiscent of the Bisection method in that it generates a sequence of intervals for which f has opposite signs at the

ends, but the intervals are not bisected at the midpoints. Instead, they are sub-divided at the point where the secant between the graph at the two endpoints is zero. This may seem like a good idea, but it is easy to construct examples where this does not work particularly well. However, there are standard ways to improve the method to avoid these problems.

Exercises

10.1 In this exercise we are going to test the three numerical methods that are discussed in this chapter. We use the equation $f(x) = \sin x = 0$, and want to compute the zero $x = \pi \approx 3.14159265358979932385$.

    a) Determine an approximation to π by performing ten manual steps with the Bisecton method, starting with the interval $[3, 4]$. Compute the error in each step by comparing with the exact value.

    b) Determine an approximation by performing four steps with the Secant method with starting values $x_0 = 4$ and $x_1 = 3$. Compute the error in each step.

    c) Determine an approximation by performing four steps with Newton's method with initial value $x_0 = 3$. Compute the error in each step.

    d) Compare the errors for the three methods. Which one converges the fastest?

10.2 Repeat exercise 1 with the equation $(x - 10/3)^5 = 0$. Why do you think the error behaves differently than in exercise 1? Hint: Take a careful look at the conditions in theorems 10.14 and 10.20.

10.3 The equation $f(x) = x - \cos x = 0$ has a zero at $x \approx 0.739085133215160642$.

    a) Use the Bisection method to find an approximation to the zero, starting with $[a, b] = [0, 1]$. How many correct digits do you have after ten steps?

    b) How many steps do you need to get ten correct digits with the Bisection method?

    c) Write a program that implements the Bisection method, and compute an approximation to the root with the number of iterations that you found in (b). How does the actual error compare with the requirement of ten correct digits?

10.4 Repeat exercise 3, but use Newtons method instead of the bisection method. Make a plot giving a visual image of the iteration process

10.5   **a)** Find an equation which is zero at $c = \sqrt{3}$. Use the Secant method and determine an approximation to c with 2 correct digits by hand. Program the Secant method, and determine an approximation with 15 correct digits using the program.

    b) Repeat (a), but with $c = 2^{1/12}$.

    c) Repeat (a) but with $c = e$, where $e = 2.71828\cdots$ is the base for natural logarithms.

10.6 In this exercise we will analyse the behaviour of the Secant method and Newton's method applied to the equation $f(x) = x^2 - 2 = 0$.

a) Let $\{x_n\}$ denote the sequence generated by Newton's method, and set $e_n = x_n - \sqrt{2}$. Derive the formula

$$e_{n+1} = \frac{e_n^2}{2x_n} \qquad (10.13)$$

directly (do not use lemma 10.19), and verify that the values computed in example 10.22 satisfy this equation.

b) Derive a relation similar to (10.13) for the Secant method, and verify that the numbers computed in example 10.16 satisfy this relation.

10.7 Some computers do not have hardware for division, and need to compute numbers like $1/R$ in some other way.

a) Set $f(x) = 1/x - R$. Verify that the Newton iteration for this function is

$$x_{n+1} = x_n(2 - Rx_n),$$

and explain how this can be used to compute $1/R$ without division.

b) Use the idea in (a) to compute $1/7$ with an accuracy of ten decimal digits.

10.8 Suppose that you are working with a function f where both f, f' and f'' are continous on all of $\mathbb{R}$. Suppose also that f has a zero at c, that $f'(c) \neq 0$ and that Newton's method generates a sequence $\{x_n\}$ that converges to c. From lemma 10.19 we know that the error $e_n = x_n - c$ satisfies the relation

$$e_{n+1} = \frac{1}{2} \frac{f''(\xi_n)}{f'(x_n)} e_n^2 \quad \text{for } n \geq 0, \qquad (10.14)$$

where ξ_n is a number in the interval (c, x_n) (or the interval (x_n, c) if $x_n < c$).

a) Use 10.14 to show that if $f''(c) \neq 0$, there exists an N such that either $x_n > c$ for all $n > N$ or $x_n < c$ for all $n > N$. (Hint: Use the fact that $\{x_n\}$ converges to c and that neither f' nor f'' changes sign in sufficiently small intervals around c.)

b) Suppose that $f'(c) > 0$, but that $f''(c) = 0$ and that the sign of f'' changes from positive to negative at c (when we move from left to right). Show that there exists an N such that $(x_{n+1} - z)(x_n - z) < 0$ for all $n > N$. In other words, the approximations x_n will alternately lie to the left and right of c when n becomes sufficiently large.

c) Find examples that illustrate each of the three types of convergence found in (a) and (b), and verify that the behaviour is as expected by performing the computations (with a computer program).

CHAPTER 11

Numerical Differentiation and Integration

Differentiation and integration are basic mathematical operations with a wide range of applications in many areas of science. It is therefore important to have good methods to compute and manipulate derivatives and integrals. You probably learnt the basic rules of differentiation and integration in school — symbolic methods suitable for pencil-and-paper calculations. These are important, and most derivatives can be computed this way. Integration however, is different, and most integrals cannot be determined with symbolic methods like the ones you learnt in school.

Another complication is the fact that in practical applications a function is only known at a few points. For example, we may measure the position of a car every minute via a GPS (Global Positioning System) unit, and we want to compute its speed. If the position is known as a continuous function of time, we can find the speed by differentiating this function. But when the position is only known at isolated times, this is not possible. The same applies to integrals.

The solution, both when it comes to integrals that cannot be determined by the usual methods, and functions that are only known at isolated points, is to use approximate methods of differentiation and integration. In our context, these are going to be numerical methods. We are going to present a number of methods for doing numerical integration and differentiation, but more importantly, we are going to present a general strategy for deriving such methods. In this way you will not only have a number of methods available to you, but you will also be able to develop new methods, tailored to special situations that you may encounter.

We use the same general strategy for deriving both numerical integration and numerical differentiation methods. The basic idea is to evaluate a function at a few points, find the polynomial that interpolates the function at these points, and use the derivative or integral of the polynomial as an approximation to the function. This technique also allows us to keep track of the so-called *truncation error*, the mathematical error committed by integrating or differentiating the polynomial instead of the function itself. However, when it comes to round-off error, we have to treat differentiation and integration differently: Numerical integration is very *insensitive* to round-off errors, while numerical differentiation behaves in the opposite way; it is very *sensitive* to round-off errors.

11.1 A simple method for numerical differentiation

We start by studying numerical differentiation. We first introduce the simplest method, derive its error, and its sensitivity to round-off errors. The procedure used here for deriving the method and analysing the error is used over again in later sections to derive and analyse additional methods.

Let us first make it clear what numerical differentiation is.

Problem 11.1 (Numerical differentiation). *Let f be a given function that is only known at a number of isolated points. The problem of numerical differentiation is to compute an approximation to the derivative f' of f by suitable combinations of the known values of f.*

A typical example is that f is given by a computer program (more specifically a function, procedure or method, depending on you choice of programming language), and you can call the program with a floating-point argument x and receive back a floating-point approximation of $f(x)$. The challenge is to compute an approximation to $f'(a)$ for some real number a when the only aid we have at our disposal is the program to compute values of f.

11.1.1 The basic idea

Since we are going to compute derivatives, we must be clear about they are defined. Recall that $f'(a)$ is defined by

$$f'(a) = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h}. \tag{11.1}$$

In the following we will assume that this limit exists; i.e., that f is differentiable. From (11.1) we immediately have a natural approximation to $f'(a)$; we simply

pick a positive h and use the approximation

$$f'(a) \approx \frac{f(a+h) - f(a)}{h}. \qquad (11.2)$$

Note that this corresponds to approximating f by the straight line p_1 that interpolates f at a and $a - h$, and then using $p_1'(a)$ as an approximation to $f'(a)$.

Observation 11.2. *The derivative of f at a can be approximated by*

$$f'(a) \approx \frac{f(a+h) - f(a)}{h}.$$

In a practical situation, the number a would be given, and we would have to locate the two nearest values a_1 and a_2 to the left and right of a such that $f(a_1)$ and $f(a_2)$ can be found. Then we would use the approximation

$$f'(a) \approx \frac{f(a_2) - f(a_1)}{a_2 - a_1}.$$

In later sections, we will derive several formulas like (11.2). Which formula to use for a specific example, and exactly how to use it, will have to be decided in each case.

Example 11.3. Let us test the approximation (11.2) for the function $f(x) = \sin x$ at $a = 0.5$ (using 64-bit floating-point numbers). In this case we have $f'(x) = \cos x$ so $f'(a) = 0.87758256$. This makes it is easy to check the accuracy. We try with a few values of h and find

h	$(f(a+h) - f(a))/h$	$E_1(f; a, h)$
10^{-1}	0.8521693479	2.5×10^{-2}
10^{-2}	0.8751708279	2.4×10^{-3}
10^{-3}	0.8773427029	2.4×10^{-4}
10^{-4}	0.8775585892	2.4×10^{-5}
10^{-5}	0.8775801647	2.4×10^{-6}
10^{-6}	0.8775823222	2.4×10^{-7}

where $E_1(f; a, h) = f(a) - (f(a+h) - f(a))/h$. In other words, the approximation seems to improve with decreasing h, as expected. More precisely, when h is reduced by a factor of 10, the error is reduced by the same factor. ∎

229

11.1.2 The truncation error

Whenever we use approximations, it is important to try and keep track of the error, if at all possible. To analyse the error in numerical differentiation, Taylor polynomials with remainders are useful. To analyse the error in the approximation above, we do a Taylor expansion of $f(a+h)$. We have

$$f(a+h) = f(a) + hf'(a) + \frac{h^2}{2}f''(\xi_h),$$

where ξ_h lies in the interval $(a, a+h)$. If we rearrange this formula, we obtain

$$f'(a) - \frac{f(a+h) - f(a)}{h} = -\frac{h}{2}f''(\xi_h). \qquad (11.3)$$

This is often referred to as the *truncation error* of the approximation, and is a reasonable error formula, but it would be nice to get rid of ξ_h. We first take absolute values in (11.3),

$$\left| f'(a) - \frac{f(a+h) - f(a)}{h} \right| = \frac{h}{2}\left| f''(\xi_h) \right|.$$

Recall from the Extreme value theorem that if a function is continuous, then its maximum always exists on any closed and bounded interval. In our setting here, it is natural to let the closed and bounded interval be $[a, a+h]$. This leads to the following lemma.

Lemma 11.4. *Suppose that f has continuous derivatives up to order two near a. If the derivative $f'(a)$ is approximated by*

$$\frac{f(a+h) - f(a)}{h},$$

then the truncation error *is bounded by*

$$E(f; a, h) = \left| f'(a) - \frac{f(a+h) - f(a)}{h} \right| \le \frac{h}{2} \max_{x \in [a, a+h]} \left| f''(x) \right|. \qquad (11.4)$$

Let us check that the error formula (11.3) confirms the numerical values in example 11.3. We have $f''(x) = -\sin x$, so the right-hand side in (11.4) becomes

$$E(\sin; 0.5, h) = \frac{h}{2}\sin \xi_h,$$

where $\xi_h \in (0.5, 0.5 + h)$. For $h = 0.1$ we therefore have that the error must lie in the interval

$$[0.05 \sin 0.5, \, 0.05 \sin 0.6] = [2.397 \times 10^{-2}, 2.823 \times 10^{-2}],$$

and the right end of the interval is the maximum value of the right-hand side in (11.4). When h is reduced by a factor of 10, the factor $h/2$ is reduced by the same factor. This means that ξ_h will approach 0.5 so $\sin \xi_h$ will approach the lower value $\sin 0.5 \approx 0.479426$. For $h = 10^{-n}$, the error will therefore tend to $10^{-n} \sin 0.5/2 \approx 10^{-n} 0.2397$, which is in complete agreement with example 11.3.

This is true in general. If f'' is continuous, then ξ_h will approach a when h goes to zero. But even when $h > 0$, the error in using the approximation $f''(\xi_h) \approx f''(a)$ is small. This is the case since it is usually only necessary to know the magnitude of the error, i.e., it is sufficient to know the error with one or two correct digits.

> **Observation 11.5.** *The truncation error is approximately given by*
>
> $$\left| f'(a) - \frac{f(a+h) - f(a)}{h} \right| \approx \frac{h}{2} \left| f''(a) \right|.$$

11.1.3 The round-off error

So far, we have just considered the mathematical error committed when $f'(a)$ is approximated by $\big(f(a+h) - f(a)\big)/h$. But what about the round-off error? In fact, when we compute this approximation we have to perform the one critical operation $f(a+h) - f(a)$ — subtraction of two almost equal numbers — which we know from chapter 5 may lead to large round-off errors. Let us continue example 11.3 and see what happens if we use smaller values of h.

Example 11.6. Recall that we estimated the derivative of $f(x) = \sin x$ at $a = 0.5$ and that the correct value with ten digits is $f'(0.5) \approx 0.8775825619$. If we check values of h from 10^{-7} and smaller we find

h	$\big(f(a+h) - f(a)\big)/h$	$E(f; a, h)$
10^{-7}	0. 8775825372	2.5×10^{-8}
10^{-8}	0.8775825622	-2.9×10^{-10}
10^{-9}	0.8775825622	-2.9×10^{-10}
10^{-11}	0.8775813409	1.2×10^{-6}
10^{-14}	0.8770761895	5.1×10^{-4}
10^{-15}	0.8881784197	-1.1×10^{-2}
10^{-16}	1.110223025	-2.3×10^{-1}
10^{-17}	0.000000000	8.8×10^{-1}

231

This shows very clearly that something quite dramatic happens, and when we come to $h = 10^{-17}$, the derivative is computed as zero. ∎

If $\overline{f(a)}$ is the floating-point number closest to $f(a)$, we know from lemma 5.6 that the relative error will be bounded by 5×2^{-53} since floating-point numbers are represented in binary ($\beta = 2$) with 53 bits for the significand ($m = 53$). We therefore have $|\epsilon| \leq 5 \times 2^{-53} \approx 6 \times 10^{-16}$. In practice, the real upper bound on ϵ is usually smaller, and in the following we will denote this upper bound by ϵ^*. This means that a definite upper bound on ϵ^* is 6×10^{-16}.

Notation 11.7. *The maximum relative error when a real number is represented by a floating-point number is denoted by ϵ^*.*

There is a handy way to express the relative error in $f(a)$. If we denote the computed value of $f(a)$ by $\overline{f(a)}$, we will have

$$\overline{f(a)} = f(a)(1 + \epsilon)$$

which corresponds to the relative error being $|\epsilon|$.

Observation 11.8. *Suppose that $f(a)$ is computed with 64-bit floating-point numbers and that no underflow or overflow occurs. Then the computed value $\overline{f(a)}$ satisfies*

$$\overline{f(a)} = f(a)(1 + \epsilon) \tag{11.5}$$

where $|\epsilon| \leq \epsilon^*$, and ϵ depends on both a and f.

The computation of $f(a + h)$ is of course also affected by round-off error, so we have

$$\overline{f(a)} = f(a)(1 + \epsilon_1), \quad \overline{f(a+h)} = f(a+h)(1 + \epsilon_2) \tag{11.6}$$

where $|\epsilon_i| \leq \epsilon^*$ for $i = 1, 2$. Here we should really write $\epsilon_2 = \epsilon_2(h)$, because the exact round-off error in $\overline{f(a+h)}$ will inevitably depend on h in a rather random way.

The next step is to see how these errors affect the computed approximation of $f'(a)$. Recall from example 5.11 that the main source of round-off in subtraction is the replacement of the numbers to be subtracted by the nearest floating-point numbers. We therefore consider the computed approximation to be

$$\frac{\overline{f(a+h)} - \overline{f(a)}}{h}.$$

If we insert the expressions (11.6), and also make use of lemma 11.4, we obtain

$$
\begin{aligned}
\frac{\overline{f(a+h)} - \overline{f(a)}}{h} &= \frac{f(a+h) - f(a)}{h} + \frac{f(a+h)\epsilon_2 - f(a)\epsilon_1}{h} \\
&= f'(a) + \frac{h}{2} f''(\xi_h) + \frac{f(a+h)\epsilon_2 - f(a)\epsilon_1}{h}
\end{aligned}
\tag{11.7}
$$

where $\xi_h \in (a, a+h)$. This leads to the following important observation.

Theorem 11.9. *Suppose that f and its first two derivatives are continuous near a. When the derivative of f at a is approximated by*

$$
\frac{f(a+h) - f(a)}{h},
$$

the error in the computed approximation is given by

$$
\left| f'(a) - \frac{\overline{f(a+h)} - \overline{f(a)}}{h} \right| \le \frac{h}{2} M_1 + \frac{2\epsilon^*}{h} M_2,
\tag{11.8}
$$

where

$$
M_1 = \max_{x \in [a, a+h]} \left| f''(a) \right|, \quad M_2 = \max_{x \in [a, a+h]} \left| f(a) \right|.
$$

Proof. To get to (11.8) we have rearranged (11.7) and used the triangle inequality. We have also replaced $\left| f''(\xi_h) \right|$ by its maximum on the interval $[a, a+h]$, as in (11.4). Similarly, we have replaced $f(\xi_h)$ and $f(a+h)$ by their common maximum on $[a, a+h]$. The last term then follows by applying the triangle inequality to the last term in (11.7) and replacing $|\epsilon_1|$ and $|\epsilon_2(h)|$ by the upper bound ϵ^*. ∎

The inequality (11.8) can be replaced by an approximate equality by making the approximations $M_1 \approx \left| f''(a) \right|$ and $M_2 \approx \left| f(a) \right|$, just like in observation 11.8 and using the maximum of ϵ_1 and ϵ_2 in (11.7), which we denote $\epsilon(h)$.

Observation 11.10. *The inequality* (11.8) *is approxmately equivalent to*

$$
\left| f'(a) - \frac{\overline{f(a+h)} - \overline{f(a)}}{h} \right| \approx \frac{h}{2} \left| f''(a) \right| + \frac{2 |\epsilon(h)|}{h} \left| f(a) \right|.
\tag{11.9}
$$

Let us check how well observation 11.10 agrees with the computations in examples 11.3 and 11.6.

Example 11.11. For large values of h the first term on the right in (11.9) will dominate the error and we have already seen that this agrees very well with the computed values in example 11.3. The question is how well the numbers in example 11.6 can be modelled when h becomes smaller.

To estimate the size of $\epsilon(h)$, we consider the case when $h = 10^{-16}$. Then the observed error is -2.3×10^{-1} so we should have

$$\frac{10^{-16}}{2} \sin 0.5 - \frac{2\epsilon(10^{-16})}{10^{-16}} = -2.3 \times 10^{-1}.$$

We solve this equation and find

$$\epsilon(10^{-16}) = \frac{10^{-16}}{2}\left(2.3 \times 10^{-1} + \frac{10^{-16}}{2}\sin 0.5\right). = 1.2 \times 10^{-17}.$$

If we try some other values of h we find

$$\epsilon(10^{-11}) = -6.1 \times 10^{-18}, \quad \epsilon(10^{-13}) = 2.4 \times 10^{-18}, \quad \epsilon(10^{-15}) = 5.3 \times 10^{-18}.$$

We observe that all these values are considerably smaller than the upper limit 6×10^{-16} which we mentioned above.

Figure 11.1 shows plots of the error. The numerical approximation has been computed for the values $n = 0.01i$, $i = 0, \ldots, 200$ and plotted in a log-log plot. The errors are shown as isolated dots, and the function

$$g(h) = \frac{h}{2} \sin 0.5 + \tilde{\epsilon}\frac{2}{h} \sin 0.5 \tag{11.10}$$

with $\tilde{\epsilon} = 7 \times 10^{-17}$ is shown as a solid graph. It seems like this choice of $\tilde{\epsilon}$ makes $g(h)$ a reasonable upper bound on the error. ∎

11.1.4 Optimal choice of h

Figure 11.1 indicates that there is an optimal value of h which minimises the total error. We can find this mathematically by minimising the upper bound in (11.9), with $|\epsilon(h)|$ replaced by the upper bound ϵ^*. This gives

$$g(h) = \frac{h}{2}\left|f''(a)\right| + \frac{2\epsilon^*}{h}\left|f(a)\right|. \tag{11.11}$$

To find the value of h which minimises this expression, we differentiate with respect to h and set the derivative to zero. We find

$$g'(h) = \frac{\left|f''(a)\right|}{2} - \frac{2\epsilon^*}{h^2}\left|f(a)\right|.$$

If we solve the equation $g(h) = 0$, we obtain the approximate optimal value.
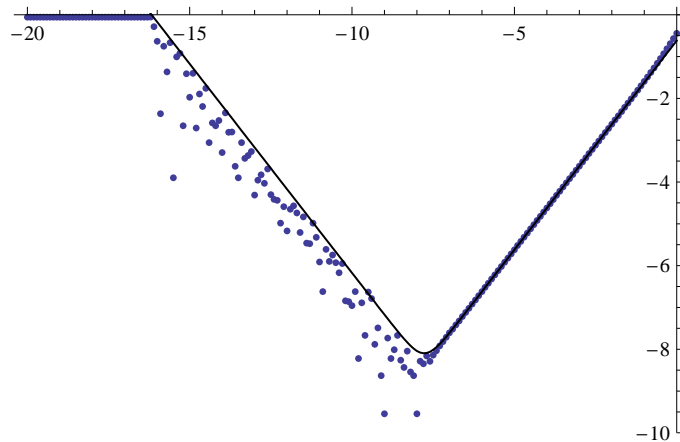
Figure 11.1. Numerical approximation of the derivative of $f(x) = \sin x$ at $x = 0.5$ using the approximation in lemma 11.4. The plot is a $\log_{10}$-$\log_{10}$ plot which shows the logarithm to base 10 of the absolute value of the total error as a function of the logarithm to base 10 of h, based on 200 values of h. The point -10 on the horizontal axis therefore corresponds $h = 10^{-10}$, and the point -6 on the vertical axis corresponds to an error of 10^{-6}. The plot also includes the function given by (11.10).

> **Lemma 11.12.** *Let f be a function with continuous derivatives up to order 2. If the derivative of f at a is approximated as in lemma 11.4, then the value of h which minimises the total error (truncation error + round-off error) is approximately*
>
> $$h^* \approx 2 \frac{\sqrt{\epsilon^* \, |f(a)|}}{\sqrt{|f''(a)|}}.$$

It is easy to see that the optimal value of h is the value that balances the two terms in (11.11)l, i.e., the truncation error and the round-off error are equal. In the example with $f(x) = \sin x$ and $a = 0.5$ we can use $\epsilon^* = 7 \times 10^{-17}$ which gives

$$h^* = 2\sqrt{\epsilon} = 2\sqrt{7 \times 10^{-17}} \approx 1.7 \times 10^{-8}.$$

11.2   Summary of the general strategy

Before we continue, let us sum up the derivaton and analysis of the numerical differentiation method in section 11.1, since we will use this over and over again.

The first step was to derive the numerical method. In section 11.1 this was very simple since the method came straight out of the definition of the derivative. Just before observation 11.2 we indicated that the method can also be de-

235

rived by approximating f by a polynomial p and using $p'(a)$ as an approximation to $f'(a)$. This is the general approach that we will use below.

Once the numerical method is known, we estimate the mathematical error in the approximation, *the truncation error*. This we do by performing Taylor expansions with remainders. For numerical differentiation methods which provide estimates of a derivative at a point a, we replace all function values at points other than a by Taylor polynomials with remainders. There may be a challenge to choose the degree of the Taylor polynomial.

The next task is to estimate the total error, including round-off error. We consider the difference between the derivative to be computed and the computed approximation, and replace the computed function evaluations by expressions like the ones in observation 11.8. This will result in an expression involving the mathematical approximation to the derivative. This can be simplified in the same way as when the truncation error was estimated, with the addition of an expression involving the relative round-off errors in the function evaluations. These expressions can then be simplified to something like (11.8) or (11.9).

As a final step, the optimal value of h can be found by minimising the total error.

Algorithm 11.13. *To derive and analyse a numerical differentiation method, the following steps are necessary:*

1. *Derive the method using polynomial interpolation.*

2. *Estimate the truncation error using Taylor series with remainders.*

3. *Estimate the total error (truncation error + round-off error) by assuming all function evaluations are replaced by the nearest floating-point numbers.*

4. *Estimate the optimal value of h.*

11.3   A simple, symmetric method

The numerical differentiation method in section 11.1 is not symmetric about a, so let us try and derive a symmetric method.

11.3.1   Construction of the method

We want to find an approximation to $f'(a)$ using values of f near a. To obtain a symmetric method, we assume that $f(a-h)$, $f(a)$, and $f(a+h)$ are known

values, and we want to find an approximation to $f'(a)$ using these values. The strategy is to determine the quadratic polynomial p_2 that interpolates f at $a - h$, a and $a + h$, and then we use $p_2'(a)$ as an approximation to $f'(a)$.

We write p_2 in Newton form,

$$p_2(x) = f[a - h] + f[a - h, a](x - (a - h))$$
$$+ f[a - h, a, a + h](x - (a - h))(x - a). \quad (11.12)$$

We differentiate and find

$$p_2'(x) = f[a - h, a] + f[a - h, a, a + h](2x - 2a + h).$$

Setting $x = a$ yields

$$p_2'(a) = f[a - h, a] + f[a - h, a, a + h]h.$$

To get a practically useful formula we must express the divided differences in terms of function values. If we expand the second expression we obtain

$$p_2'(a) = f[a - h, a] + \frac{f[a, a + h] - f[a - h, a]}{2h}h = \frac{f[a, a + h] + f[a - h, a]}{2} \quad (11.13)$$

The two first order differences are

$$f[a - h, a] = \frac{f(a) - f(a - h)}{h}, \quad f[a, a + h] = \frac{f(a + h) - f(a)}{h},$$

and if we insert this in (11.13) we end up with

$$p_2'(a) = \frac{f(a + h) - f(a - h)}{2h}.$$

Lemma 11.14. *Let f be a given function, and let a and h be given numbers. If $f(a - h)$, $f(a)$, $f(a + h)$ are known values, then $f'(a)$ can be approximated by $p_2'(a)$ where p_2 is the quadratic polynomial that interpolates f at $a - h$, a, and $a + h$. The approximation is given by*

$$f'(a) \approx p_2'(a) = \frac{f(a + h) - f(a - h)}{2h}. \quad (11.14)$$

Let us test this approximation on the function $f(x) = \sin x$ at $a = 0.5$ so we can compare with the method in section 11.1.

237

Example 11.15. We test the approximation (11.14) with the same values of h as in examples 11.3 and 11.6. Recall that $f'(0.5) \approx 0.8775825619$ with ten correct decimals. The results are

h	$\big(f(a+h) - f(a-h)\big)/(2h)$	$E(f; a, h)$
10^{-1}	0.8761206554	1.5×10^{-3}
10^{-2}	0.8775679356	1.5×10^{-5}
10^{-3}	0.8775824156	1.5×10^{-7}
10^{-4}	0.8775825604	1.5×10^{-9}
10^{-5}	0.8775825619	1.8×10^{-11}
10^{-6}	0.8775825619	-7.5×10^{-12}
10^{-7}	0.8775825616	2.7×10^{-10}
10^{-8}	0.8775825622	-2.9×10^{-10}
10^{-11}	0.8775813409	1.2×10^{-6}
10^{-13}	0.8776313010	-4.9×10^{-5}
10^{-15}	0.8881784197	-1.1×10^{-2}
10^{-17}	0.0000000000	8.8×10^{-1}

If we compare with examples 11.3 and 11.6, the errors are generally smaller. In particular we note that when h is reduced by a factor of 10, the error is reduced by a factor of 100, at least as long as h is not too small. However, when h becomes smaller than about 10^{-6}, the error becomes larger. It therefore seems like the truncation error is smaller than in the first method, but the round-off error makes it impossible to get accurate results for small values of h. The optimal value of h seems to be $h^* \approx 10^{-6}$, which is larger than for the first method, but the error is then about 10^{-12}, which is smaller than the best we could do with the first method. ∎

11.3.2 Truncation error

Let us attempt to estimate the truncation error for the method in lemma 11.14. The idea is to do replace $f(a-h)$ and $f(a+h)$ with Taylor expansions about a. We use the Taylor expansions

$$f(a+h) = f(a) + hf'(a) + \frac{h^2}{2}f''(a) + \frac{h^3}{6}f'''(\xi_1),$$

$$f(a-h) = f(a) - hf'(a) + \frac{h^2}{2}f''(a) - \frac{h^3}{6}f'''(\xi_2),$$

where $\xi_1 \in (a, a+h)$ and $\xi_2 \in (a-h, a)$. If we subtract the second formula from the first and divide by $2h$, we obtain

$$\frac{f(a+h) - f(a-h)}{2h} = f'(a) + \frac{h^2}{12}\big(f'''(\xi_1) + f'''(\xi_2)\big). \qquad (11.15)$$

This leads to the following lemma.

Lemma 11.16. *Suppose that f and its first three derivatives are continuous near a, and suppose we approximate $f'(a)$ by*

$$\frac{f(a+h)-f(a-h)}{2h}. \tag{11.16}$$

The truncation error in this approximation is bounded by

$$\left|E_2(f;a,h)\right| = \left|f'(a) - \frac{f(a+h)-f(a-h)}{2h}\right| \le \frac{h^2}{6} \max_{x\in[a-h,a+h]} \left|f'''(x)\right|. \tag{11.17}$$

Proof. What remains is to simplify the last term in (11.15) to the term on the right in (11.17). This follows from

$$\left|f'''(\xi_1) + f'''(\xi_2)\right| \le \max_{x\in[a,a+h]}\left|f'''(x)\right| + \max_{x\in[a-h,a]}\left|f'''(x)\right|$$

$$\le \max_{x\in[a-h,a+h]}\left|f'''(x)\right| + \max_{x\in[a-h,a+h]}\left|f'''(x)\right|$$

$$= 2\max_{x\in[a-h,a+h]}\left|f'''(x)\right|.$$

The last inequality is true because the width of the intervals over which we take the maximums are increased, so the maximum values may also increase. ∎

The error formula (11.17) confirms the numerical behaviour we saw in example 11.15 for small values of h since the error is proportional to h^2: When h is reduced by a factor of 10, the error is reduced by a factor 10^2.

11.3.3   Round-off error

The round-off error may be estimated just like for the first method. When the approximation (11.16) is computed, the values $f(a-h)$ and $f(a+h)$ are replaced by the nearest floating point numbers $\overline{f(a-h)}$ and $\overline{f(a+h)}$ which can be expressed as

$$\overline{f(a+h)} = f(a+h)(1+\epsilon_1), \quad \overline{f(a-h)} = f(a-h)(1+\epsilon_2),$$

where both ϵ_1 and ϵ_2 depend on h and satisfy $|\epsilon_i| \le \epsilon^*$ for $i = 1, 2$. Using these expressions we obtain

$$\frac{\overline{f(a+h)} - \overline{f(a-h)}}{2h} = \frac{f(a+h)-f(a-h)}{2h} + \frac{f(a+h)\epsilon_1 - f(a-h)\epsilon_2}{2h}.$$

We insert (11.15) and get the relation

$$\frac{\overline{f(a+h)} - \overline{f(a-h)}}{2h} = f'(a) + \frac{h^2}{12}\big(f'''(\xi_1) + f'''(\xi_2)\big) + \frac{f(a+h)\epsilon_1 - f(a-h)\epsilon_2}{2h}.$$

This leads to an estimate of the total error if we use the same technique as in the proof of lemma 11.8.

Theorem 11.17. *Let f be a given function with continuous derivatives up to order three, and let a and h be given numbers. Then the error in the approximation*

$$f'(a) \approx \frac{f(a+h) - f(a-h)}{2h},$$

including round-off error and truncation error, is bounded by

$$\left| f'(a) - \frac{\overline{f(a+h)} - \overline{f(a-h)}}{2h} \right| \le \frac{h^2}{6}M_1 + \frac{\epsilon^*}{h}M_2 \qquad (11.18)$$

where

$$M_1 = \max_{x \in [a-h,a+h]} \left| f'''(x) \right|, \quad M_2 = \max_{x \in [a-h,a+h]} \left| f(x) \right|. \qquad (11.19)$$

In practice, the interesting values of h will usually be so small that there is very little error in making the approximations

$$M_1 = \max_{x \in [a-h,a+h]} \left| f'''(x) \right| \approx \left| f'''(a) \right|, \quad M_2 = \max_{x \in [a-h,a+h]} \left| f(x) \right| \approx \left| f(a) \right|.$$

If we make this simplification in (11.18) we obtain a slightly simpler error estimate.

Observation 11.18. *The error (11.18) is approximately bounded by*

$$\left| f'(a) - \frac{\overline{f(a+h)} - \overline{f(a-h)}}{2h} \right| \lesssim \frac{h^2}{6}\left| f'''(a) \right| + \frac{\epsilon^* \left| f(a) \right|}{h}. \qquad (11.20)$$

A plot of how the error behaves in this approximation, together with the estimate of the error on the right in (11.20), is shown in figure 11.2.
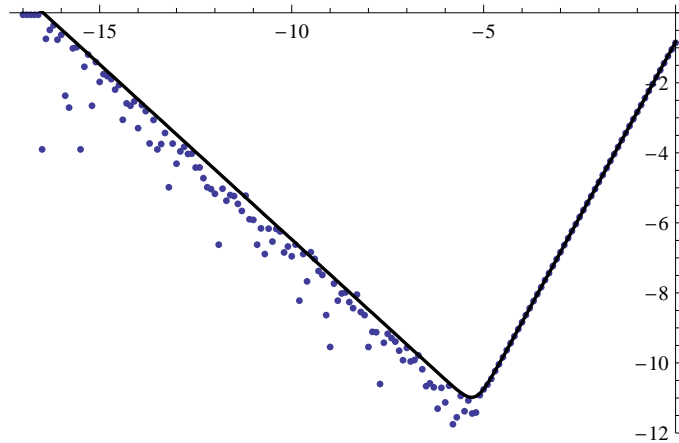
Figure 11.2. Log-log plot of the error in the approximation to the derivative of $f(x) = \sin x$ at $x = 1/2$ for values of h in the interval $[0, 10^{-17}]$, using the method in theorem 11.17. The function plotted is the right-hand side of (11.20) with $\epsilon^* = 7 \times 10^{-17}$, as a function of h.

11.3.4 Optimal choice of h

As for the first numerical differentiation method, we can find an optimal value of h which minimises the error. The error is minimised when the truncation error and the round-off error have the same magnitude. We can find this value of h if we differentiate the right-hand side of (11.18) with respect to h and set the derivative to 0. This leads to the equation

$$\frac{h}{3}M_1 - \frac{\epsilon^*}{h^2}M_2 = 0$$

which has the solution

$$h^* = \frac{\sqrt[3]{3\epsilon^* M_2}}{\sqrt[3]{M_1}} \approx \frac{\sqrt[3]{3\epsilon^* |f(a)|}}{\sqrt[3]{|f'''(a)|}}.$$

At the end of section 11.1.4 we saw that a reasonable value for ϵ^* was $\epsilon^* = 7 \times 10^{-17}$. The optimal value of h in example 11.15, where $f(x) = \sin x$ and $a = 1/2$, then becomes $h = 4.6 \times 10^{-6}$. For this value of h the approximation is $f'(0.5) \approx 0.877582561887$ with error 3.1×10^{-12}.

11.4 A four-point method for differentiation

In a way, the two methods for numerical differentiation that we have considered so far are the same. If we use a step length of $2h$ in the first method, the

241

approximation becomes

$$f'(a) \approx \frac{f(a+2h) - f(a)}{2h}.$$

The analysis of the symmetric method shows that the approximation is considerably better if we associate the approximation with the midpoint between a and $a + h$,

$$f'(a + h) \approx \frac{f(a+2h) - f(a)}{2h}.$$

At the point $a + h$ the approximation is proportional to h^2 rather than h, and this makes a big difference as to how quickly the error goes to zero, as is evident if we compare examples 11.3 and 11.15. In this section we derive another method for which the truncation error is proportional to h^4.

The computations below may seem overwhelming, and have in fact been done with the help of a computer to save time and reduce the risk of miscalculations. The method is included here just to illustrate that the principle for deriving both the method and the error terms is just the same as for the simple symmetric method in the previous section. To save space we have only included one highlight, of the approximation method and the total error.

11.4.1 Derivation of the method

We want better accuracy than the symmetric method which was based on interpolation with a quadratic polynomial. It is therefore natural to base the approximation on a cubic polynomial, which can interpolate four points. We have seen the advantage of symmetry, so we choose the interpolation points $x_0 = a - 2h$, $x_1 = a - h$, $x_2 = a + h$, and $x_3 = a + 2h$. The cubic polynomial that interpolates f at these points is

$$p_3(x) = f(x_0) + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1)$$
$$+ f[x_0, x_1, x_2, x_3](x - x_0)(x - x_1)(x - x_2).$$

and its derivative is

$$p_3'(x) = f[x_0, x_1] + f[x_0, x_1, x_2](2x - x_0 - x_1)$$
$$+ f[x_0, x_1, x_2, x_3]\big((x - x_1)(x - x_2) + (x - x_0)(x - x_2) + (x - x_0)(x - x_1)\big).$$

If we evaluate this expression at $x = a$ and simplify (this is quite a bit of work), we find that the resulting approximation of $f'(a)$ is

$$f'(a) \approx p_3'(a) = \frac{f(a-2h) - 8f(a-h) + 8f(a+h) - f(a+2h)}{12h}. \tag{11.21}$$

11.4.2 Truncation error

To estimate the error, we expand the four terms in the numerator in (11.21) in Taylor series,

$$f(a-2h) = f(a) - 2hf'(a) + 2h^2 f''(a) - \frac{4h^3}{3} f'''(a) + \frac{2h^4}{3} f^{(iv)}(a) - \frac{4h^5}{15} f^{(v)}(\xi_1),$$

$$f(a-h) = f(a) - hf'(a) + \frac{h^2}{2} f''(a) - \frac{h^3}{6} f'''(a) + \frac{h^4}{24} f^{(iv)}(a) - \frac{h^5}{120} f^{(v)}(\xi_2),$$

$$f(a+h) = f(a) + hf'(a) + \frac{h^2}{2} f''(a) + \frac{h^3}{6} f'''(a) + \frac{h^4}{24} f^{(iv)}(a) + \frac{h^5}{120} f^{(v)}(\xi_3),$$

$$f(a+2h) = f(a) + 2hf'(a) + 2h^2 f''(a) + \frac{4h^3}{3} f'''(a) + \frac{2h^4}{3} f^{(iv)}(a) + \frac{4h^5}{15} f^{(v)}(\xi_4),$$

where $\xi_1 \in (a-2h, a)$, $\xi_2 \in (a-h, a)$, $\xi_3 \in (a, a+h)$, and $\xi_4 \in (a, a+2h)$. If we insert this into the formula for $p_3'(a)$ we obtain

$$\frac{f(a-2h) - 8f(a-h) + 8f(a+h) - f(a+2h)}{12h} =$$
$$f'(a) - \frac{h^4}{45} f^{(v)}(\xi_1) + \frac{h^4}{180} f^{(v)}(\xi_2) + \frac{h^4}{180} f^{(v)}(\xi_3) - \frac{h^4}{45} f^{(v)}(\xi_4).$$

If we use the same trick as for the symmetric method, we can combine all last four terms in and obtain an upper bound on the truncation error. The result is

$$\left| f'(a) - \frac{f(a-2h) - 8f(a-h) + 8f(a+h) - f(a+2h)}{12h} \right| \leq \frac{h^4}{18} M \qquad (11.22)$$

where

$$M = \max_{x \in [a-2h, a+2h]} \left| f^{(v)}(x) \right|.$$

11.4.3 Round-off error

The truncation error is derived in the same way as before. The quantities we actually compute are

$$\overline{f(a-2h)} = f(a-2h)(1+\epsilon_1), \quad \overline{f(a+2h)} = f(a+2h)(1+\epsilon_3),$$
$$\overline{f(a-h)} = f(a-h)(1+\epsilon_2), \qquad \overline{f(a+h)} = f(a+h)(1+\epsilon_4).$$

We estimate the difference between $f'(a)$ and the computed approximation, make use of the estimate (11.22), combine the function values that are multiplied by ϵs, and approximate the maximum values by function values at a. We sum up the result.

243

Figure 11.3. Log-log plot of the error in the approximation to the derivative of $f(x) = \sin x$ at $x = 1/2$, using the method in observation 11.19, with h in the interval $[0, 10^{-17}]$. The function plotted is the right-hand side of (11.23) with $\epsilon^* = 7 \times 10^{-17}$.

Observation 11.19. *Suppose that f and its first five derivatives are continuous. If $f'(a)$ is approximated by*

$$f'(a) \approx \frac{f(a-2h) - 8f(a-h) + 8f(a+h) - f(a+2h)}{12h},$$

the total error is approximately bounded by

$$\left| f'(a) - \frac{\overline{f(a-2h)} - 8\overline{f(a-h)} + 8\overline{f(a+h)} - \overline{f(a+2h)}}{12h} \right| \lesssim$$

$$\frac{h^4}{18} \left| f^{(v)}(a) \right| + \frac{3\epsilon^*}{h} \left| f(a) \right|. \quad (11.23)$$

A plot of the error in the approximation for the $\sin x$ example is shown in figure 11.3.

11.4.4   Optimal value of h

From observation 11.19 we can compute the optimal value of h by differentiating the right-hand side with respect to h and setting it to zero,

$$\frac{2h^3}{9} \left| f^{(v)}(a) \right| - \frac{3\epsilon^*}{h^2} \left| f(a) \right| = 0$$

244

which has the solution

$$h^* = \frac{\sqrt[5]{27\epsilon^* |f(a)|}}{\sqrt[5]{2|f^{(v)}(a)|}}.$$

For the case above with $f(x) = \sin x$ and $a = 0.5$ the solution is $h^* \approx 8.8 \times 10^{-4}$. For this value of h the actual error is 10^{-14}.

11.5   Numerical approximation of the second derivative

We consider one more method for numerical approximation of derivatives, this time of the second derivative. The approach is the same: We approximate f by a polynomial and approximate the second derivative of f by the second derivative of the polynomial. As in the other cases, the error analysis is based on expansion in Taylor series.

11.5.1   Derivation of the method

Since we are going to find an approximation to the second derivative, we have to approximate f by a polynomial of degree at least two, otherwise the second derivative is identically 0. The simplest is therefore to use a quadratic polynomial, and for symmetry we want it to interpolate f at $a - h$, a, and $a + h$. The resulting polynomial p_2 is the one we used in section 11.3 and it is given in equation (11.12). The second derivative of p_2 is constant, and the approximation of $f'(a)$ is

$$f''(a) \approx p_2''(a) = f[a - h, a, a + h].$$

The divided difference is easy to expand.

Lemma 11.20. *The second derivative of a function f at a can be approximated by*

$$f''(a) \approx \frac{f(a + h) - 2f(a) + f(a - h)}{h^2}. \tag{11.24}$$

11.5.2   The truncation error

Estimation of the error goes as in the other cases. The Taylor series of $f(a - h)$ and $f(a + h)$ are

$$f(a - h) = f(a) - hf'(a) + \frac{h^2}{2}f''(a) - \frac{h^3}{6}f'''(a) + \frac{h^4}{24}f^{(iv)}(\xi_1),$$

$$f(a + h) = f(a) + hf'(a) + \frac{h^2}{2}f''(a) + \frac{h^3}{6}f'''(a) + \frac{h^4}{24}f^{(iv)}(\xi_2),$$

where $\xi_1 \in (a - h, a)$ and $\xi_2 \in (a, a + h)$. If we insert these Taylor series in (11.24) we obtain

$$\frac{f(a+h) - 2f(a) + f(a-h)}{h^2} = f''(a) + \frac{h^2}{24}\left(f^{(iv)}(\xi_1) + f^{(iv)}(\xi_2)\right).$$

From this obtain an expression for the truncation error.

Lemma 11.21. *Suppose f and its first three derivatives are continuous near a. If the second derivative $f''(a)$ is approximated by*

$$f''(a) \approx \frac{f(a+h) - 2f(a) + f(a-h)}{h^2},$$

the error is bounded by

$$\left| f''(a) - \frac{f(a+h) - 2f(a) + f(a-h)}{h^2} \right| \leq \frac{h^2}{12} \max_{x \in [a-h, a+h]} \left| f'''(x) \right|. \qquad (11.25)$$

11.5.3   Round-off error

The round-off error can also be estimated as before. Instead of computing the exact values, we compute $\overline{f(a-h)}$, $\overline{f(a)}$, and $\overline{f(a+h)}$, which are linked to the exact values by

$$\overline{f(a-h)} = f(a-h)(1 + \epsilon_1), \quad \overline{f(a)} = f(a)(1 + \epsilon_2), \quad \overline{f(a+h)} = f(a+h)(1 + \epsilon_3),$$

where $|\epsilon_i| \leq \epsilon^*$ for $i = 1, 2, 3$. The difference between $f''(a)$ and the computed approximation is therefore

$$f''(a) - \frac{\overline{f(a+h)} - 2\overline{f(a)} + \overline{f(a-h)}}{h^2}$$

$$= -\frac{h^2}{24}\left(f'''(\xi_1) + f'''(\xi_2)\right) - \frac{\epsilon_1 f(a-h) - \epsilon_2 f(a) + \epsilon_3 f(a+h)}{h^2}.$$

If we combine terms on the right as before, we end up with the following theorem.

Theorem 11.22. *Suppose f and its first three derivatives are continuous near a, and that $f''(a)$ is approximated by*
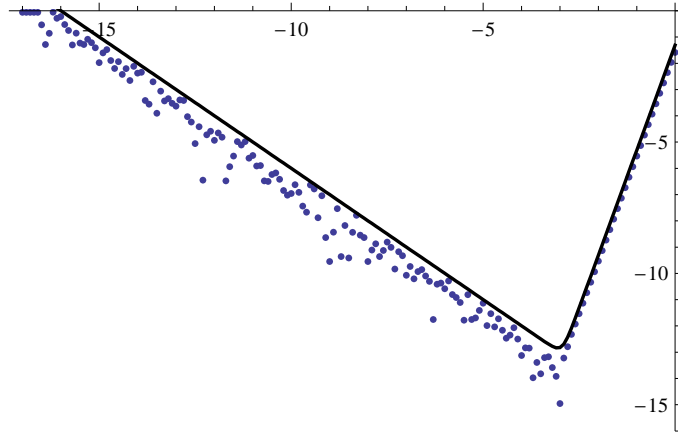
$$f''(a) \approx \frac{f(a+h) - 2f(a) + f(a-h)}{h^2}.$$

246
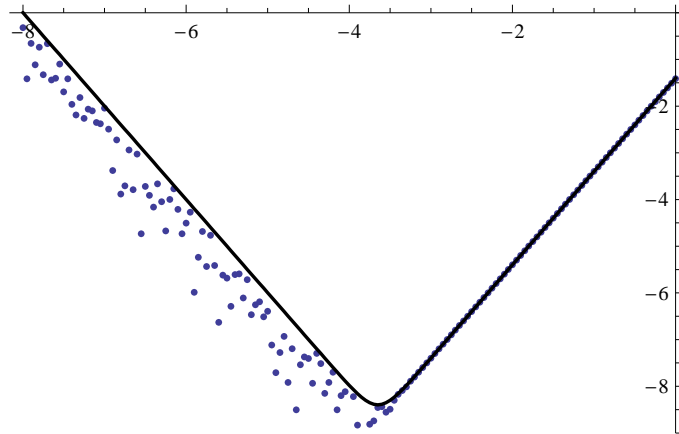
Figure 11.4. Log-log plot of the error in the approximation to the derivative of $f(x) = \sin x$ at $x = 1/2$ for h in the interval $[0, 10^{-8}]$, using the method in theorem 11.22. The function plotted is the right-hand side of (11.23) with $\epsilon^* = 7 \times 10^{-17}$.

> *Then the total error (truncation error + round-off error) in the computed approximation is bounded by*
>
> $$\left| f''(a) - \frac{\overline{f(a+h)} - 2\overline{f(a)} + \overline{f(a-h)}}{h^2} \right| \leq \frac{h^2}{12} M_1 + \frac{3\epsilon^*}{h^2} M_2. \qquad (11.26)$$
>
> *where*
> $$M_1 = \max_{x \in [a-h, a+h]} \left| f^{(iv)}(x) \right|, \quad M_2 = \max_{x \in [a-h, a+h]} \left| f(x) \right|.$$

As before, we can simplify the right-hand side to

$$\frac{h^2}{12} \left| f^{(iv)}(a) \right| + \frac{3\epsilon^*}{h^2} \left| f(a) \right| \qquad (11.27)$$

if we can tolerate a slightly approximate upper bound.

Figure 11.4 shows the errors in the approximation to the second derivative given in theorem 11.22 when $f(x) = \sin x$ and $a = 0.5$ and for h in the range $[0, 10^{-8}]$. The solid graph gives the function in (11.27) which describes the upper limit on the error as function of h, with $\epsilon^* = 7 \times 10^{-17}$. For h smaller than 10^{-8}, the approximation becomes 0, and the error constant. Recall that for the approximations to the first derivative, this did not happen until h was about 10^{-17}. This illustrates the fact that the higher the derivative, the more problematic is the round-off error, and the more difficult it is to approximate the derivative with numerical methods like the ones we study here.

247
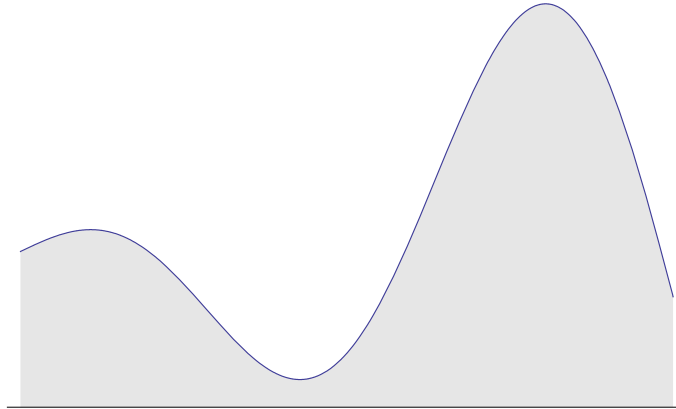
Figure 11.5. The area under the graph of a function.

11.5.4   Optimal value of h

Again, we find the optimal value of h by minimising the right-hand side of (11.26). To do this we find the derivative with respect to h and set it to 0,

$$\frac{h}{6} M_1 - \frac{6\epsilon^*}{h^3} M_2 = 0.$$

As usual it does not make much difference if we use the approximations $M_1 \approx \left| f'''(a) \right|$ and $M_2 = \left| f(a) \right|$.

> **Observation 11.23.** *The upper bound on the total error* (11.26) *is minimised when h has the value*
> $$h^* = \frac{\sqrt[4]{36\epsilon^* \left| f(a) \right|}}{\sqrt[4]{\left| f^{(iv)}(a) \right|}}.$$

When $f(x) = \sin x$ and $a = 0.5$ this gives $h^* = 2.2 \times 10^{-4}$ if we use the value $\epsilon^* = 7 \times 10^{-17}$. Then the approximation to $f''(a) = -\sin a$ is -0.4794255352 with an actual error of 3.4×10^{-9}.

11.6   General background on integration

Our next task is to develop methods for numerical integration. Before we turn to this, it is necessary to briefly review the definition of the integral.

(a)   (b)

(c)   (d)
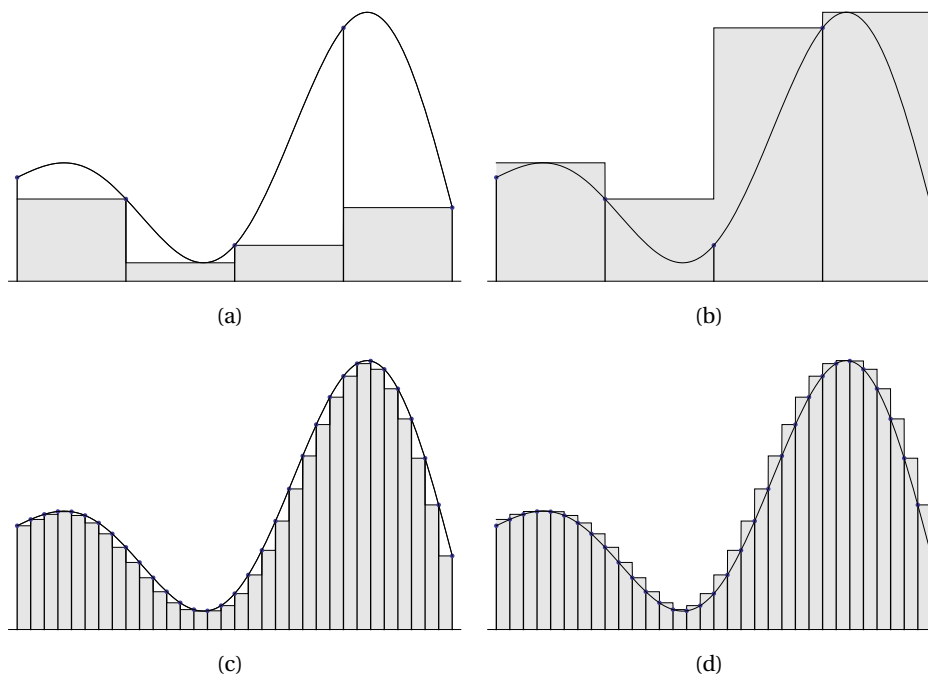
Figure 11.6. The definition of the integral via inscribed and circumsribed step functions.

Recall that if $f(x)$ is a function, then the integral of f from $x = a$ to $x = b$ is written

$$\int_a^b f(x)\,dx.$$

This integral gives the area under the graph of f, with the area under the positive part counting as positive area, and the area under the negative part of f counting as negative area, see figure 11.5.

Before we continue, we need to define a term which we will use repeatedly in our description of integration.

Definition 11.24. *Let a and b be two real numbers with $a < b$. A* partition *of $[a, b]$ is a finite sequence $\{x_i\}_{i=0}^n$ of increasing numbers in $[a, b]$ with $x_0 = a$ and $x_n = b$,*

$$a = x_0 < x_1 < x_2 \cdots < x_{n-1} < x_n = b.$$

The partition is said to be uniform *if there is a fixed number h, called the* step length, *such that $x_i - x_{i-1} = h = (b - a)/n$ for $i = 1, \ldots, n$.*

The traditional definition of the integral is based on a numerical approximation to the area. We pick a partition $\{x_i\}$ of $[a, b]$, and in each subinterval $[x_{i-1}, x_i]$ we determine the maximum and minimum of f (for convenience we assume that these values exist),

$$m_i = \min_{x \in [x_{i-1}, x_i]} f(x), \quad M_i = \max_{x \in [x_{i-1}, x_i]} f(x),$$

for $i = 1, 2, \ldots, n$. We use these values to compute the two sums

$$\underline{I} = \sum_{i=1}^{n} m_i (x_i - x_{i-1}), \quad \overline{I} = \sum_{i=1}^{n} M_i (x_i - x_{i-1}).$$

To define the integral, we consider larger partitions and consider the limits of $\underline{I}$ and $\overline{I}$ as the distance between neighbouring x_is goes to zero. If those limits are the same, we say that f is integrable, and the integral is given by this limit. More precisely,

$$I = \int_a^b f(x)\,dx = \sup \underline{I} = \inf \overline{I},$$

where the sup and inf are taken over all partitions of the interval $[a, b]$. This process is illustrated in figure 11.6 where we see how the piecewise constant approximations become better when the rectangles become narrower.

The above definition can be used as a numerical method for computing approximations to the integral. We choose to work with either maxima or minima, select a partition of $[a, b]$ as in figure 11.6, and add together the areas of the rectangles. The problem with this technique is that it can be both difficult and time consuming to determine the maxima or minima, even on a computer.

However, it can be shown that the integral has a very convenient property: If we choose a point t_i in each interval $[x_{i-1}, x_i]$, then the sum

$$\tilde{I} = \sum_{i=1}^{n} f(t_i)(x_i - x_{i-1})$$

will also converge to the integral when the distance between neighbouring x_is goes to zero. If we choose t_i equal to x_{i-1} or x_i, we have a simple numerical method for computing the integral. An even better choice is the more symmetric $t_i = (x_i + x_{i-1})/2$ which leads to the approximation

$$I \approx \sum_{i=1}^{n} f\big((x_i + x_{i-1})/2\big)(x_i - x_{i-1}). \tag{11.28}$$

This is the so-called *midpoint method* which we will study in the next section.
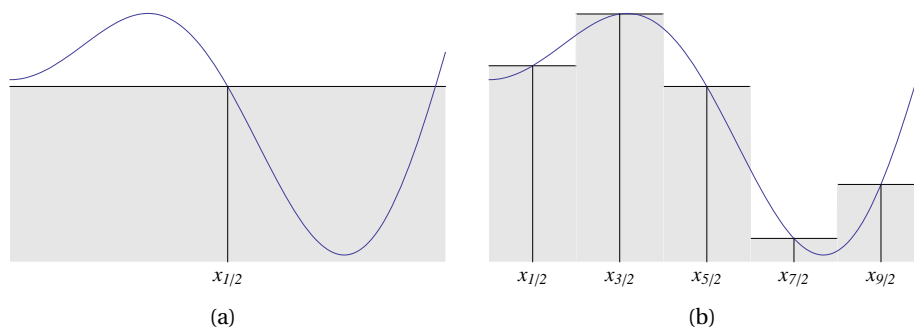
Figure 11.7. The midpoint rule with one subinterval (a) and five subintervals (b).

In general, we can derive numerical integration methods by splitting the interval $[a, b]$ into small subintervals, approximate f by a polynomial on each subinterval, integrate this polynomial rather than f, and then add together the contributions from each subinterval. This is the strategy we will follow, and this works as long as f can be approximated well by polynomials on each subinterval.

11.7 The midpoint method for numerical integration

We have already introduced the midpoint rule (11.28) for numerical integration. In our standard framework for numerical methods based on polynomial approximation, we can consider this as using a constant approximation to the function f on each subinterval. Note that in the following we will always assume the partition to be uniform.

> **Algorithm 11.25.** *Let f a function which is integrable on the interval $[a, b]$ and let $\{x_i\}_{i=0}^n$ be a uniform partition of $[a, b]$. In the midpoint rule, the integral of f is approximated by*
>
> $$\int_a^b f(x)\, dx \approx I_{mid}(h) = h \sum_{i=1}^n f(x_{i-1/2}), \tag{11.29}$$
>
> *where*
>
> $$x_{i-1/2} = (x_i + x_{i-1})/2 = a + (i - 1/2)h.$$

This may seem like a strangely formulated algorithm, but all there is to it is to compute the sum on the right in (11.29). The method is illustrated in figure 11.7 in the cases where we have one and five subintervals.

251

Example 11.26. Let us try the midpoint method on an example. As usual, it is wise to test on an example where we know the answer, so we can easily check the quality of the method. We choose the integral

$$\int_0^1 \cos x \, dx = \sin 1 \approx 0.8414709848$$

where the exact answer is easy to compute by traditional, symbolic methods. To test the method, we split the interval into 2^k subintervals, for $k = 1, 2, \ldots, 10$, i.e., we halve the step length each time. The result is

h	$I_{mid}(h)$	Error
0.500000	0.85030065	-8.8×10^{-3}
0.250000	0.84366632	-2.2×10^{-3}
0.125000	0.84201907	-5.5×10^{-4}
0.062500	0.84160796	-1.4×10^{-4}
0.031250	0.84150523	-3.4×10^{-5}
0.015625	0.84147954	-8.6×10^{-6}
0.007813	0.84147312	-2.1×10^{-6}
0.003906	0.84147152	-5.3×10^{-7}
0.001953	0.84147112	-1.3×10^{-7}
0.000977	0.84147102	-3.3×10^{-8}

By error, we here mean

$$\int_0^1 f(x) \, dx - I_{mid}(h).$$

Note that each time the step length is halved, the error seems to be reduced by a factor of 4. ∎

11.7.1   Local error analysis

As usual, we should try and keep track of the error. We first focus on what happens on one subinterval. In other words we want to study the error

$$\int_a^b f(x) \, dx - f(a_{1/2})(b-a), \quad a_{1/2} = (a+b)/2. \tag{11.30}$$

Once again, Taylor polynomials with remainders help us out. We expand both $f(x)$ and $f(a_{1/2})$ about the the left endpoint

$$f(x) = f(a) + (x-a)f'(a) + \frac{(x-a)^2}{2} f''(\xi_1),$$

$$f(a_{1/2}) = f(a) + (a_{1/2} - a)f'(a) + \frac{(a_{1/2} - a)^2}{2} f''(\xi_2),$$

where ξ_1 is a number in the interval (a, x) that depends on x, and ξ_2 is in the interval $(a, a_{1/2})$. If we multiply the second Taylor expansion by $(b-a)$, we obtain

$$f(a_{1/2})(b-a) = f(a)(b-a) + \frac{(b-a)^2}{2}f'(a) + \frac{(b-a)^3}{8}f''(\xi_2). \qquad (11.31)$$

Next, we integrate the Taylor expansion and obtain

$$\int_a^b f(x)\,dx = \int_a^b \left(f(a) + (x-a)f'(a) + \frac{(x-a)^2}{2}f''(\xi_1) \right) dx$$

$$= f(a)(b-a) + \frac{1}{2}\left[(x-a)^2 \right]_a^b f'(a) + \frac{1}{2}\int_a^b (x-a)^2 f''(\xi_1)\,dx \quad (11.32)$$

$$= f(a)(b-a) + \frac{(b-a)^2}{2}f'(a) + \frac{1}{2}\int_a^b (x-a)^2 f''(\xi_1)\,dx.$$

We then see that the error can be written

$$\left| \int_a^b f(x)\,dx - f(a_{1/2})(b-a) \right| = \left| \frac{1}{2}\int_a^b (x-a)^2 f''(\xi_1)\,dx - \frac{(b-a)^3}{8}f''(\xi_2) \right|$$

$$\leq \frac{1}{2}\left| \int_a^b (x-a)^2 f''(\xi_1)\,dx \right| + \frac{(b-a)^3}{8}\left| f''(\xi_2) \right|.$$

$$(11.33)$$

For the last term, we use our standard trick,

$$\left| f''(\xi_2) \right| \leq M = \max_{x \in [a,b]} \left| f''(x) \right|. \qquad (11.34)$$

Note that since $\xi_2 \in (a, a_{1/2})$, we could just have taken the maximum over the interval $[a, a_{1/2}]$, but we will see later that it is more convenient to maximise over the whole interval $[a, b]$.

The first term in (11.33) needs some massaging. Let us do the work first, and explain afterwords,

$$\frac{1}{2}\left| \int_a^b (x-a)^2 f''(\xi_1)\,dx \right| \leq \frac{1}{2}\int_a^b \left| (x-a)^2 f''(\xi_1) \right| dx$$

$$= \frac{1}{2}\int_a^b (x-a)^2 \left| f''(\xi_1) \right| dx$$

$$\leq \frac{M}{2}\int_a^b (x-a)^2\,dx \qquad (11.35)$$

$$= \frac{M}{2}\frac{1}{3}\left[(x-a)^3 \right]_a^b$$

$$= \frac{M}{6}(b-a)^3.$$

253

The first inequality is valid because when we move the absolute value sign inside the integral sign, the function that we integrate becomes nonnegative everywhere. This means that in the areas where the integrand in the original expression is negative, everything is now positive, and hence the second integral is larger than the first.

Next there is an equality which is valid because $(x-a)^2$ is never negative. The next inequality follows because we replace $\left|f''(\xi_1)\right|$ with its maximum on the interval $[a, b]$. The last step is just the evaluation of the integral of $(x-a)^2$.

We have now simplified both terms on the right in (11.33), so we have

$$\left|\int_a^b f(x)\,dx - f\left(a_{1/2}\right)(b-a)\right| \le \frac{M}{6}(b-a)^3 + \frac{M}{8}(b-a)^3.$$

The result is the following lemma.

Lemma 11.27. *Let f be a continuous function whose first two derivatives are continuous on the interval $[a.b]$. The the error in the midpoint method, with only one interval, is bounded by*

$$\left|\int_a^b f(x)\,dx - f\left(a_{1/2}\right)(b-a)\right| \le \frac{7M}{24}(b-a)^3,$$

where $M = \max_{x\in[a,b]}\left|f''(x)\right|$ and $a_{1/2} = (a+b)/2$.

The importance of this lemma lies in the factor $(b-a)^3$. This means that if we reduce the size of the interval to half its width, the error in the midpoint method will be reduced by a factor of 8.

Perhaps you feel completely lost in the work that led up to lemma 11.27. The wise way to read something like this is to first focus on the general idea that was used: Consider the error (11.30) and replace both $f(x)$ and $f(a_{1/2})$ by its quadratic Taylor polynomials with remainders. If we do this, a number of terms cancel out and we are left with (11.33). At this point we use some standard techniques that give us the final inequality.

Once you have an overview of the derivation, you should check that the details are correct and make sure you understand each step.

11.7.2   Global error analysis

Above, we analysed the error on one subinterval. Now we want to see what happens when we add together the contributions from many subintervals; it should not surprise us that this may affect the error.

We consider the general case where we have a partition that divides $[a, b]$ into n subintervals, each of width h. On each subinterval we use the simple midpoint rule that we analysed in the previous section,

$$I = \int_a^b f(x)\,dx = \sum_{i=1}^n \int_{x_{i-1}}^{x_i} f(x)\,dx \approx \sum_{i=1}^n f(x_{i-1/2})h.$$

The total error is then

$$I - I_{mid} = \sum_{i=1}^n \left(\int_{x_{i-1}}^{x_i} f(x)\,dx - f(x_{i-1/2})h \right).$$

But the expression inside the parenthesis is just the local error on the interval $[x_{i-1}, x_i]$. We therefore have

$$
\begin{aligned}
|I - I_{mid}| &= \left| \sum_{i=1}^n \left(\int_{x_{i-1}}^{x_i} f(x)\,dx - f(x_{i-1/2})h \right) \right| \\
&\leq \sum_{i=1}^n \left| \int_{x_{i-1}}^{x_i} f(x)\,dx - f(x_{i-1/2})h \right| \\
&\leq \sum_{i=1}^n \frac{7h^3}{24} M_i
\end{aligned}
\tag{11.36}
$$

where M_i is the maximum of $\left|f''(x)\right|$ on the interval $[x_{i-1}, x_i]$. To simiplify the expression (11.36), we extend the maximum on $[x_{i-1}, x_i]$ to all of $[a, b]$. This will usually make the maximum larger, so for all i we have

$$M_i = \max_{x \in [x_{i-1}, x_i]} \left|f''(x)\right| \leq \max_{x \in [a,b]} \left|f''(x)\right| = M.$$

Now we can simplify (11.36) further,

$$\sum_{i=1}^n \frac{7h^3}{24} M_i \leq \sum_{i=1}^n \frac{7h^3}{24} M = \frac{7h^3}{24} nM. \tag{11.37}$$

Here, we need one final little observation. Recall that $h = (b-a)/n$, so $hn = b-a$. If we insert this in (11.37), we obtain our main result.

Theorem 11.28. *Suppose that f and its first two derivatives are continuous on the interval $[a, b]$, and that the integral of f on $[a, b]$ is approximated by the midpoint rule with n subintervals of equal width,*

$$I = \int_a^b f(x)\,dx \approx I_{mid} = \sum_{i=1}^n f(x_{i-1/2})h.$$

Then the error is bounded by

$$|I - I_{mid}| \le (b-a)\frac{7h^2}{24} \max_{x \in [a,b]} |f''(x)| \qquad (11.38)$$

where $x_{i-1/2} = a + (i - 1/2)h$.

This confirms the error behaviour that we saw in example 11.26: If h is reduced by a factor of 2, the error is reduced by a factor of $2^2 = 4$.

One notable omission in our discussion of the midpoint method is round-off error, which was a major concern in our study of numerical differentiation. The good news is that round-off error is not usually a problem in numerical integration. The only situation where round-off may cause problems is when the value of the integral is 0. In such a situation we may potentially add many numbers that sum to 0, and this may lead to cancellation effects. However, this is so rare that we will not discuss it here.

You should be aware of the fact that the error estimate (11.38) is not the best possible in that the constant 7/24 can be reduced to 1/24, but then the derivation becomes much more complicated.

11.7.3   Estimating the step length

The error estimate (11.38) lets us play a standard game: If someone demands that we compute an integral with error smaller than ϵ, we can find a step length h that guarantees that we meet this demand. To make sure that the error is smaller than ϵ, we enforce the inequality

$$(b-a)\frac{7h^2}{24} \max_{x \in [a,b]} |f''(x)| \le \epsilon$$

which we can easily solve for h,

$$h \le \sqrt{\frac{24\epsilon}{7(b-a)M}}, \qquad M = \max_{x \in [a,b]} |f''(x)|.$$

This is not quite as simple as it may look since we will have to estimate M, the maximum value of the second derivative. This can be difficult, but in some cases it is certainly possible, see exercise 4.

11.7.4   A detailed algorithm

Algorithm 11.25 describes the midpoint method, but lacks a lot of detail. In this section we give a more detailed algorithm.

Whenever we compute a quantity numerically, we should try and estimate the error, otherwise we have no idea of the quality of our computation. A standard way to do this for numerical integration is to compute the integral for decreasing step lengths, and stop the computations when difference between two successive approximations is less than the tolerance. More precisely, we choose an initial step length h_0 and compute the approximations

$$I_{mid}(h_0), I_{mid}(h_1), \dots, I_{mid}(h_k), \dots,$$

where $h_k = h_0/2^k$. Suppose $I_{mid}(h_k)$ is our latest approximation. Then we estimate the relative error by the number

$$\frac{|I_{mid}(h_k) - I_{mid}(h_{k-1}|}{|I_{mid}(h_k)|}$$

and stop the computations if this is smaller than ϵ. To avoid potential division by zero, we use the test

$$|I_{mid}(h_k) - I_{mid}(h_{k-1})| \leq \epsilon |I_{mid}(h_k)|.$$

As always, we should also limit the number of approximations that are computed.

Algorithm 11.29. *Suppose the function f, the interval $[a, b]$, the length n_0 of the intitial partition, a positive tolerance $\epsilon < 1$, and the maximum number of iterations M are given. The following algorithm will compute a sequence of approximations to $\int_a^b f(x)\,dx$ by the midpoint rule, until the estimated relative error is smaller than ϵ, or the maximum number of computed approximations reach M. The final approximation is stored in I.*

```
n := n_0;   h := (b − a)/n;
I := 0;   x := a + h/2;
for k := 1, 2, …, n
    I := I + f(x);
    x := x + h;
j := 1;
I := h * I;
abserr := |I|;
while j < M and abserr > ϵ * |I|
    j := j + 1;
    Ip := I;
```
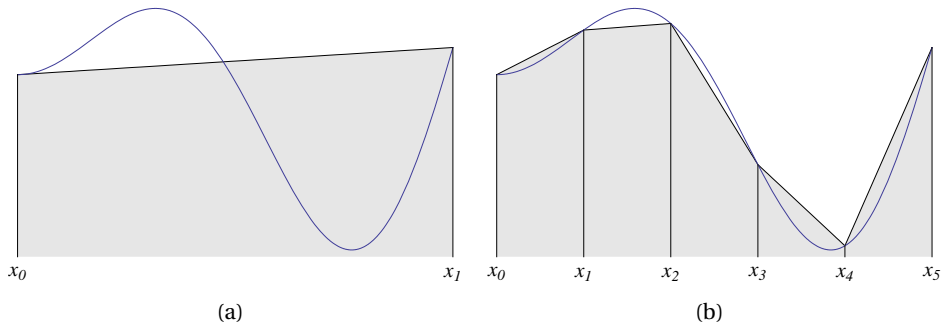
Figure 11.8. The trapezoid rule with one subinterval (a) and five subintervals (b).

$$n := 2n; \quad h := (b-a)/n;$$
$$I := 0; \quad x := a + h/2;$$
$$\textbf{for } k := 1, 2, \dots, n$$
$$\quad I := I + f(x);$$
$$\quad x := x + h;$$
$$I := h * I;$$
$$abserr := |I - Ip|;$$

Note that we compute the first approximation outside the main loop. This is necessary in order to have meaningful estimates of the relative error (the first time we reach the top of the while loop we will always get past the condition). We store the previous approximation in Ip so that we can estimate the error.

In the coming sections we will describe two other methods for numerical integration. These can be implemented in algorithms similar to Algorithm 11.29. In fact, the only difference will be how the actual approximation to the integral is computed.

11.8   The trapezoid rule

The midpoint method is based on a very simple polynomial approximation to the function f to be integrated on each subinterval; we simply use a constant approximation by interpolating the function value at the middle point. We are now going to consider a natural alternative; we approximate f on each subinterval with the secant that interpolates f at both ends of the subinterval.

The situation is shown in figure 11.8a. The approximation to the integral is

the area of the trapezoidal figure under the secant so we have

$$\int_a^b f(x)\,dx \approx \frac{f(a)+f(b)}{2}(b-a). \tag{11.39}$$

To get good accuracy, we will have to split $[a,b]$ into subintervals with a partition and use this approximation on each subinterval, see figure 11.8b. If we have a uniform partition $\{x_i\}_{i=0}^n$ with step length h, we get the approximation

$$\int_a^b f(x)\,dx = \sum_{i=1}^n \int_{x_{i-1}}^{x_i} f(x)\,dx \approx \sum_{i=1}^n \frac{f(x_{i-1})+f(x_i)}{2}h. \tag{11.40}$$

We should always aim to make our computational methods as efficient as possible, and in this case an improvement is possible. Note that on the interval $[x_{i-1}, x_i]$ we use the function values $f(x_{i-1})$ and $f(x_i)$, and on the next interval we use the values $f(x_i)$ and $f(x_{i+1})$. All function values, except the first and last, therefore occur twice in the sum on the right in (11.40). This means that if we implement this formula directly we do a lot of unnecessary work. From the explanation above the following observation follows.

Observation 11.30 (Trapezoid rule)**.** *Suppose we have a function f defined on an interval $[a,b]$ and a partition $\{x_i\}_{i=0}^n$ of $[a.b]$. If we approximate f by its secant on each subinterval and approximate the integral of f by the integral of the resulting piecewise linear approximation, we obtain the approximation*

$$\int_a^b f(x)\,dx \approx h\left(\frac{f(a)+f(b)}{2} + \sum_{i=1}^{n-1} f(x_i)\right). \tag{11.41}$$

In the formula (11.41) there are no redundant function evaluations.

11.8.1 Local error analysis

Our next step is to analyse the error in the trapezoid method. We follow the same recipe as for the midpoint method and use Taylor series. Because of the similarities with the midpoint method, we will skip some of the details.

We first study the error in the approximation (11.39) where we only have one secant. In this case the error is given by

$$\left| \int_a^b f(x)\,dx - \frac{f(a)+f(b)}{2}(b-a) \right|, \tag{11.42}$$

and the first step is to expand the function values $f(x)$ and $f(b)$ in Taylor series about a,

$$f(x) = f(a) + (x-a)f'(a) + \frac{(x-a)^2}{2} f''(\xi_1),$$

$$f(b) = f(a) + (b-a)f'(a) + \frac{(b-a)^2}{2} f''(\xi_2),$$

where $\xi_1 \in (a, x)$ and $\xi_2 \in (a, b)$. The integration of the Taylor series for $f(x)$ we did in (11.32) so we just quote the result here,

$$\int_a^b f(x)\, dx = f(a)(b-a) + \frac{(b-a)^2}{2} f'(a) + \frac{1}{2} \int_a^b (x-a)^2 f''(\xi_1)\, dx.$$

If we insert the Taylor series for $f(b)$ we obtain

$$\frac{f(a) + f(b)}{2}(b-a) = f(a)(b-a) + \frac{(b-a)^2}{2} f'(a) + \frac{(b-a)^3}{4} f''(\xi_2).$$

If we insert these expressions into the error (11.42), the first two terms cancel against each other, and we obtain

$$\left| \int_a^b f(x)\, dx - \frac{f(a) + f(b)}{2}(b-a) \right| \le \left| \frac{1}{2} \int_a^b (x-a)^2 f''(\xi_1)\, dx - \frac{(b-a)^3}{4} f''(\xi_2) \right|$$

These expressions can be simplified just like in (11.34) and (11.35), and this yields

$$\left| \int_a^b f(x)\, dx - \frac{f(a) + f(b)}{2}(b-a) \right| \le \frac{M}{6}(b-a)^3 + \frac{M}{4}(b-a)^3.$$

Let us sum this up in a lemma.

Lemma 11.31. *Let f be a continuous function whose first two derivatives are continuous on the interval $[a.b]$. The the error in the trapezoid rule, with only one line segment on $[a, b]$, is bounded by*

$$\left| \int_a^b f(x)\, dx - \frac{f(a) + f(b)}{2}(b-a) \right| \le \frac{5M}{12}(b-a)^3,$$

where $M = \max_{x \in [a,b]} |f''(x)|$.

This lemma is completely analogous to lemma 11.27 which describes the local error in the midpoint method. We particularly notice that even though the trapezoid rule uses two values of f, the error estimate is slightly larger than the

estimate for the midpoint method. The most important feature is the exponent on $(b-a)$, which tells us how quickly the error goes to 0 when the interval width is reduced, and from this point of view the two methods are the same. In other words, we have gained nothing by approximating f by a linear functions instead of a constant. This does not mean that the trapezoid rule is bad, it rather means that the midpoint rule is unusually good.

11.8.2 Global error

We can find an expression for the global error in the trapezoid rule in exactly the same way as we did for the midpoint rule, so we skip the proof. We sum everything up in a theorem about the trapezoid rule.

Theorem 11.32. *Suppose that f and its first two derivatives are continuous on the interval $[a, b]$, and that the integral of f on $[a, b]$ is approximated by the trapezoid rule with n subintervals of equal width h,*

$$I = \int_a^b f(x)\,dx \approx I_{trap} = h\left(\frac{f(a)+f(b)}{2} + \sum_{i=1}^{n-1} f(x_i)\right).$$

Then the error is bounded by

$$\left|I - I_{trap}\right| \le (b-a)\frac{5h^2}{12} \max_{x\in[a,b]} \left|f''(x)\right|. \tag{11.43}$$

As we mentioned when we commented on the midpoint rule, the error estimates that we obtain are not best possible in the sense that it is possible to derive better error estimates (using other techniques) with smaller constants. In the case of the trapezoid rule, the constant can be reduced from 5/12 to 1/12. However, the fact remains that the trapezoid rule is a disappointing method compared to the midpoint rule.

11.9 Simpson's rule

The final method for numerical integration that we consider is *Simpson's rule.* This method is based on approximating f by a parabola on each subinterval, which makes the derivation a bit more involved. The error analysis is essentially the same as before, but because the expressions are more complicated, it pays off to plan the analysis better. You may therefore find the material in this section more challenging than the treatment of the other two methods, and should

make sure that you have a good understanding of the error analysis for these methods before you start studying section 11.9.2.

11.9.1   Deriving Simpson's rule

As for the other methods, we derive Simpson's rule in the simplest case where we use one parabola on all of $[a, b]$. We find the polynomial p_2 that interpolates f at a, $a_{1/2} = (a + b)/2$ and b, and approximate the integral of f by the integral of p_2. We could find p_2 via the Newton form, but in this case it is easier to use the Lagrange form. Another simiplification is to first contstruct Simpson's rule in the case where $a = -1$, $a_{1/2} = 0$, and $b = 1$, and then use this to generalise the method.

The Lagrange form of the polyomial that interpolates f at -1, 0, 1, is given by

$$p_2(x) = f(-1)\frac{x(x-1)}{2} - f(0)(x+1)(x-1) + f(1)\frac{(x+1)x}{2},$$

and it is easy to check that the interpolation conditions hold. To integrate p_2, we must integrate each of the three polynomials in this expression. For the first one we have

$$\frac{1}{2}\int_{-1}^{1} x(x-1)\,dx = \frac{1}{2}\int_{-1}^{1}(x^2 - x)\,dx = \frac{1}{2}\left[\frac{1}{3}x^3 - \frac{1}{2}x^2\right]_{-1}^{1} = \frac{1}{3}.$$

Similarly, we find

$$-\int_{-1}^{1}(x+1)(x-1)\,dx = \frac{4}{3}, \qquad \frac{1}{2}\int_{-1}^{1}(x+1)x\,dx = \frac{1}{3}.$$

On the interval $[-1, 1]$, Simpson's rule therefore gives the approximation

$$\int_{-1}^{1} f(x)\,dx \approx \frac{1}{3}\big(f(-1) + 4f(0) + f(1)\big). \tag{11.44}$$

To obtain an approximation on the interval $[a, b]$, we use a standard technique. Suppose that x and y are related by

$$x = (b - a)\frac{y+1}{2} + a. \tag{11.45}$$

We see that if y varies in the interval $[-1, 1]$, then x will vary in the interval $[a, b]$. We are going to use the relation (11.45) as a substitution in an integral, so we note that $dx = (b - a)dy/2$. We therefore have

$$\int_{a}^{b} f(x)\,dx = \int_{-1}^{1} f\left(\frac{b-a}{2}(y+1) + a\right)dy = \frac{b-a}{2}\int_{-1}^{1} \tilde{f}(y)\,dy, \tag{11.46}$$
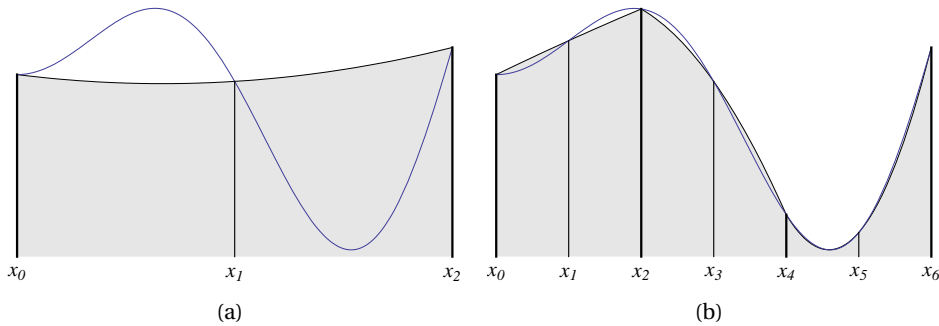
Figure 11.9. Simpson's rule with one subinterval (a) and three subintervals (b).

where

$$\tilde{f}(y) = f\left(\frac{b-a}{2}(y+1) + a\right).$$

To determine an approximaiton to the intgeral of $\tilde{f}$ on the interval $[-1,1]$, we can use Simpson's rule (11.44). The result is

$$\int_{-1}^{1} \tilde{f}(y)\,dy \approx \frac{1}{3}(\tilde{f}(-1) + 4\tilde{f}(0) + \tilde{f}(1)) = \frac{1}{3}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right),$$

since the relation in (11.45) maps -1 to a, the midpoint 0 to $(a+b)/2$, and the right endpoint b to 1. If we insert this in (11.46), we obtain Simpson's rule for the general interval $[a,b]$, see figure 11.9a. In practice, we will usually divide the interval $[a,b]$ into smaller ntervals and use Simpson's rule on each subinterval, see figure 11.9b.

Observation 11.33. *Let f be an integrable function on the interval $[a,b]$. If f is interpolated by a quadratic polynomial p_2 at the points a, $(a+b)/2$ and b, then the integral of f can be approximated by the integral of p_2,*

$$\int_a^b f(x)\,dx \approx \int_a^b p_2(x)\,dx = \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right). \qquad (11.47)$$

We could just as well have derived this formula by doing the interpolation directly on the interval $[a,b]$, but then the algebra becomes quite messy.

263

11.9.2   Local error analysis

The next step is to analyse the error. We follow the usual recipe and perform Taylor expansions of $f(x)$, $f\big((a+b)/2\big)$ and $f(b)$ around the left endpoint a. However, those Taylor expansions become long and tedious, so we are gong to see how we can predict what happens. For this, we define the error function,

$$E(f) = \int_a^b f(x)\,dx - \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right). \qquad (11.48)$$

Note that if $f(x)$ is a polynomial of degree 2, then the interpolant p_2 will be exactly the same as f. Since the last term in (11.48) is the integral of p_2, we see that the error $E(f)$ will be 0 for any quadratic polynomial. We can check this by calculating the error for the three functions 1, x, and x^2,

$$E(1) = (b-a) - \frac{b-a}{6}(1+4+1) = 0,$$

$$E(x) = \frac{1}{2}\big[x^2\big]_a^b - \frac{b-a}{6}\left(a + 4\frac{a+b}{2} + b\right) = \frac{1}{2}(b^2-a^2) - \frac{(b-a)(b+a)}{2} = 0,$$

$$E(x^2) = \frac{1}{3}\big[x^3\big]_a^b - \frac{b-a}{6}\left(a^2 + 4\frac{(a+b)^2}{4} + b^2\right)$$

$$= \frac{1}{3}(b^3-a^3) - \frac{b-a}{3}(a^2+ab+b^2)$$

$$= \frac{1}{3}\big(b^3 - a^3 - (a^2b + ab^2 + b^3 - a^3 - a^2b - ab^2)\big)$$

$$= 0.$$

Let us also check what happens if $f(x) = x^3$,

$$E(x^3) = \frac{1}{4}\big[x^4\big]_a^b - \frac{b-a}{6}\left(a^3 + 4\frac{(a+b)^3}{8} + b^3\right)$$

$$= \frac{1}{4}(b^4-a^4) - \frac{b-a}{6}\left(a^3 + \frac{a^3 + 3a^3b + 3ab^2 + b^3}{2} + b^3\right)$$

$$= \frac{1}{4}(b^4-a^4) - \frac{b-a}{6} \times \frac{3}{2}\big(a^3 + 3a^2b + 3ab^2 + b^3\big)$$

$$= 0.$$

The fact that the error is zero when $f(x) = x^3$ comes as a pleasant surprise; by the construction of the method we can only expect the error to be 0 for quadratic polynomials.

The above computations mean that

$$E\big(c_0 + c_1 x + c_2 x^2 + c_3 x^3\big) = c_0 E(1) + c_1 E(x) + c_2 E(x^2) + c_3 E(x^3) = 0$$

for any real numbers $\{c_i\}_{i=0}^3$, i.e., the error is 0 whenever f is a cubic polynomial.

> **Lemma 11.34.** *Simpson's rule is exact for cubic polynomials.*

To obtain an error estimate, suppose that f is a general function which can be expanded in a Taylor polynomial of degree 3 about a with remainder

$$f(x) = T_3(f;x) + R_3(f;x).$$

Then we see that

$$E(f) = E\big(T_3(f) + R_3(f)\big) = E\big(T_3(f)\big) + E\big(R_3(f)\big) = E\big(R_3(f)\big).$$

The second equality follows from simple properties of the integral and function evaluations, while the last equality follows because the error in Simpson's rule is 0 for cubic polynomials.

The Lagrange form of the error term is given by

$$R_3(f;x) = \frac{(x-a)^4}{24} f^{(iv)}(\xi_x),$$

where $\xi_x \in (a,x)$. We then find

$$E\big(R_3(f;x)\big) = \frac{1}{24} \int_a^b (x-a)^4 f^{(iv)}(\xi_x)\, dx$$

$$- \frac{b-a}{6}\left(0 + 4\frac{(b-a)^4}{24 \times 16} f^{(iv)}(\xi_{(a+b)/2}) + \frac{(b-a)^4}{24} f^{(iv)}(\xi_b)\right)$$

$$= \frac{1}{24} \int_a^b (x-a)^4 f^{(iv)}(\xi_x)\, dx - \frac{(b-a)^5}{576}\big(f^{(iv)}(\xi_{(a+b)/2}) + 4f^{(iv)}(\xi_b)\big),$$

where $\xi_1 = \xi_{(a+b)/2}$ and $\xi_2 = \xi_b$ (the error is 0 at $x = a$). If we take absolute values, use the triangle inequality, the standard trick of replacing the function values by maxima over the whole interval $[a,b]$, and evaluate the integral, we obtain

$$|E(f)| \leq \frac{(b-a)^5}{5 \times 24} M + \frac{(b-a)^5}{576}(M + 4M).$$

This gives the following estimate of the error.

> **Lemma 11.35.** *If f is continuous and has continuous derivatives up to order 4 on the interval $[a,b]$, the error in Simpson's rule is bounded by*
>
> $$|E(f)| \leq \frac{49}{2880}(b-a)^5 \max_{x \in [a,b]} \left|f^{(iv)}(x)\right|.$$
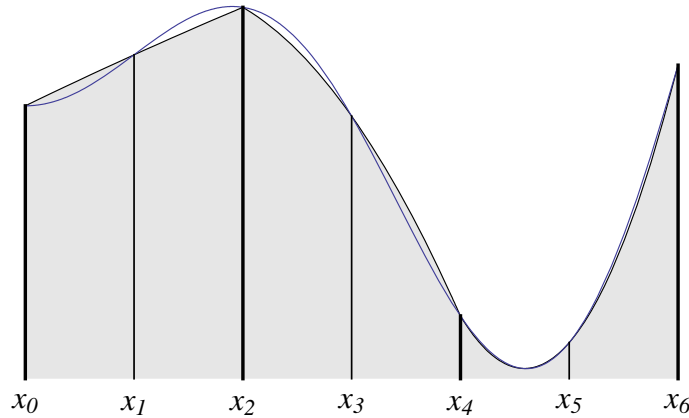
Figure 11.10. Simpson's rule with three subintervals.

We note that the error in Simpson's rule depends on $(b-a)^5$, while the error in the midpoint rule and trapezoid rule depend on $(b-a)^3$. This means that the error in Simpson's rule goes to zero much more quickly than for the other two methods when the width of the interval $[a,b]$ is reduced. More precisely, a reduction of h by a factor of 2 will reduce the error by a factor of 32.

As for the other two methods the constant $49/2880$ is not best possible; it can be reduced to $1/2880$ by using other techniques.

11.9.3 Composite Simpson's rule

Simpson's rule is used just like the other numerical integration techniques we have studied: The interval over which f is to be integrated is split into subintervals, and Simpson's rule is applied on neighbouring pairs of intervals, see figure 11.10. In other words, each parabola is defined over *two* subintervals which means that the total number of subintervals must be even and the number of given values of f must be odd.

If the partition is $(x_i)_{i=0}^{2n}$ with $x_i = a + ih$, Simpson's rule on the interval $[x_{2i-2}, x_{2i}]$ is

$$\int_{x_{2i-2}}^{x_{2i}} f(x)\,dx \approx \frac{h}{3}\big(f(x_{2i-2}) + 4f(x_{2i-1}) + f(x_{2i})\big).$$

The approximation to the total integral is therefore

$$\int_a^b f(x)\,dx \approx \frac{h}{3}\sum_{i=1}^{n}\big((f(x_{2i-2}) + 4f(x_{2i-1}) + f(x_{2i})\big).$$

266

In this sum we observe that the right endpoint of one subinterval becomes the left endpoint of the following subinterval to the right. Therefore, if this is implemented directly, the function values at the points with an even subscript will be evaluated twice, except for the extreme endpoints a and b which only occur once in the sum. We can therefore rewrite the sum in a way that avoids these redundant evaluations.

Observation 11.36. *Suppose f is a function defined on the interval $[a,b]$, and let $\{x_i\}_{i=0}^{2n}$ be a uniform partition of $[a,b]$ with step length h. The composite Simpson's rule approximates the integral of f by*

$$\int_a^b f(x)\,dx \approx \frac{h}{3}\left(f(a) + f(b) + 2\sum_{i=1}^{n-1} f(x_{2i}) + 4\sum_{i=1}^{n} f(x_{2i-1})\right).$$

With the midpoint rule, we computed a sequence of approximations to the integral by successively halving the width of the subintervals. The same is often done with Simpson's rule, but then care should be taken to avoid unnecessary function evaluations since all the function values computed at one step will also be used at the next step, see exercise 7.

11.9.4   The error in the composite Simpson's rule

The approach we used to deduce the global error for the midpoint rule, can also be used for Simpson's rule, see theorem 11.28. The following theorem sums this up.

Theorem 11.37. *Suppose that f and its first four derivatives are continuous on the interval $[a,b]$, and that the integral of f on $[a,b]$ is approximated by Simpson's rule with $2n$ subintervals of equal width h. Then the error is bounded by*

$$\left|E(f)\right| \le (b-a)\frac{49h^4}{2880} \max_{x\in[a,b]}\left|f^{(iv)}(x)\right|. \tag{11.49}$$

11.10   Summary

In this chapter we have derived a three methods for numerical differentiation and three methods for numerical integration. All these methods and their error analyses may seem rather overwhelming, but they all follow a common thread:

> **Procedure 11.38.** *The following is a general procedure for deriving numerical methods for differentiation and integration:*
>
> 1. *Interpolate the function f by a polynomial p at suitable points.*
>
> 2. *Approximate the derivative or integral of f by the derivative or integral of p. This makes it possible to express the approximation in terms of function values of f.*
>
> 3. *Derive an estimate for the error by expanding the function values (other than the one at a) in Taylor series with remainders.*
>
> 4D. *For numerical differentiation, derive an estimate of the round-off error by assuming that the relative errors in the function values are bounded by ϵ^*. By minimising the total error, an optimal step length h can be determined.*
>
> 4I. *For numerical integration, the global error can easily be derived from the local error using the technique leading up to theorem 11.28.*

Perhaps the most delicate part of the above procedure is to choose the degree of the Taylor polynomials. This is discussed in exercise 6.

It is procedure 11.38 that is the main content of this chapter. The individual methods are important in practice, but also serve as examples of how this procedure is implemented, and should show you how to derive other methods more suitable for your specific needs.

Exercises

11.1   **a)** Write a program that implements the numerical differentiation method

$$f'(a) \approx \frac{f(a+h) - f(a-h)}{2h},$$

and test the method on the function $f(x) = e^x$ at $a = 1$.

b) Determine the optimal value of h given in section 11.3.4 which minimises the total error. Use $\epsilon^* = 7 \times 10^{-17}$.

c) Use your program to determine the optimal value h of experimentally.

d) Use the optimal value of h that you found in (c) to determine a better value for ϵ^* in this specific example.

11.2 Repeat exercise 1, but compute the second derivative using the approximation

$$f''(a) \approx \frac{f(a+h) - 2f(a) + f(a-h)}{h^2}.$$

In (b) you should use the value of h given in observation 11.23.

11.3   **a)** Suppose that we want to derive a method for approximating the derivative of f at a which has the form

$$f'(a) \approx c_1 f(a-h) + c_2 f(a+h), \quad c_1, c_2 \in \mathbb{R}.$$

We want the method to be exact when $f(x) = 1$ and $f(x) = x$. Use these conditions to determine c_1 and c_2.

   b) Show that the method in (a) is exact for all polynomials of degree 1, and compare it to the methods we have discussed in this chapter.

   c) Use the procedure in (a) and (b) to derive a method for approximating the second derivative of f,

$$f''(a) \approx c_1 f(a-h) + c_2 f(a) + c_3 f(a+h), \quad c_1, c_2, c_3 \in \mathbb{R},$$

by requiring that the method should be exact when $f(x) = 1$, x and x^2.

   d) Show that the method in (c) is exact for all quadratic polynomials.

11.4   **a)** Write a program that implements the midpoint method as in algorithm 11.29 and test it on the integral

$$\int_0^1 e^x \, dx = e - 1.$$

   b) Determine a value of h that guarantees that the absolute error is smaller than 10^{-10}. Run your program and check what the actual error is for this value of h. (You may have to adjust algorithm 11.29 slightly and print the absolute error.)

11.5 Repeat exercise 4, but use Simpson's rule instead of the midpoint method.

11.6 It may sometimes be difficult to judge how many terms to include in the Taylor series used in the analysis of numerical methods. In this exercise we are going to see how this can be done. We use the numerical approximation

$$f'(a) \approx \frac{f(a+h) - f(a-h)}{2h}$$

in section 11.3 for our experiments.

   a) Do the same derivation as section 11.3.2, but include only two terms in the Taylor series (plus remainder). What happens?

   b) Do the same derivation as section 11.3.2, but include four terms in the Taylor series (plus remainder). What happens now?

11.7 When h is halved in the trapezoid method, some of the function values used with step length $h/2$ are the same as those used for step length h. Derive a formula for the trapezoid method with step length $h/2$ that makes use of the function values that were computed for step length h.

CHAPTER 12

Numerical Solution of Differential Equations

We have considered numerical solution procedures for two kinds of equations: In chapter 10 the unknown was a real number; in chapter 6 the unknown was a sequence of numbers. In a differential equation the unknown is a function, and the differential equation relates the function to its derivative(s).

In this chapter we start by considering how the simplest differential equations, the first order ones which only involve the unknown function and its first derivative, can be solved numerically by the simplest method, namely Euler's method. We analyse the error in Euler's method, and then introduce some more advanced methods with better accuracy. After this we show that the methods for handling one equation in one unknown generalise nicely to systems of several equations in several unknowns. What about equations that involve higher order derivatives? It turns out that even systems of higher order equations can be rewritten as a system of first order equations. At the end we discuss briefly the important concept of stability.

12.1   What are differential equations?

Differential equations is an essential tool in a wide range of applications. The reason for this is that many phenomena can be modelled by a relationship between a function and its derivatives. Let us consider a simple example.

12.1.1   An example from physics

Consider an object moving through space. At time $t = 0$ it is located at a point P and after a time t its distance to P corresponds to a number $f(t)$. In other words,

271

the distance can be described by a function of time. The divided difference

$$\frac{f(t + \Delta t) - f(t)}{\Delta t} \tag{12.1}$$

then measures the average speed during the time interval from t to $t + \Delta t$. If we take the limit in (12.1) as Δt approaches zero, we obtain the speed $v(t)$ at time t,

$$v(t) = \lim_{\Delta t \to 0} \frac{f(t + \Delta t) - f(t)}{\Delta t}. \tag{12.2}$$

Similarly, the divided difference of the speed is given by $\big(v(t + \Delta t) - v(t)\big)/\Delta t$. This is the average acceleration from time t to time $t + \Delta t$, and if we take the limit as Δt tends to zero we get the acceleration $a(t)$ at time t,

$$a(t) = \lim_{\Delta t \to 0} \frac{v(t + \Delta t) - v(t)}{\Delta t}. \tag{12.3}$$

If we compare the above definitions of speed and acceleration with the definition of the derivative we notice straightaway that

$$v(t) = f'(t), \qquad a(t) = v'(t) = f''(t). \tag{12.4}$$

Newton's second law states that if an object is influenced by a force, its acceleration is proportional to the force. More precisely, if the total force is F, Newton's second law can be written

$$F = ma \tag{12.5}$$

where the proportionality factor m is the mass of the object.

As a simple example of how Newton's law is applied, we can consider an object with mass m falling freely towards the earth. It is then influenced by two opposite forces, gravity and friction. The gravitational force is $F_g = mg$, where g is acceleration due to gravitation alone. Friction is more complicated, but in many situations it is reasonable to say that it is proportional to the square of the speed of the object, or $F_f = cv^2$ where c is a suitable proportionality factor. The two forces pull in opposite directions so the total force acting on the object is $F = F_g - F_f$. From Newton's law $F = ma$ we then obtain the equation

$$mg - cv^2 = ma.$$

Gravity g is constant, but both v and a depend on time and are therefore functions of t. In addition we know from (12.4) that $a(t) = v'(t)$ so we have the equation

$$mg - cv(t)^2 = mv'(t)$$

which would usually be shortened and rearranged as

$$mv' = mg - cv^2. \tag{12.6}$$

The unknown here is the function $v(t)$, the speed, but the equation also involves the derivative (the acceleration) $v'(t)$, so this is a differential equation. This equation is just a mathematical formulation of Newton's second law, and the hope is that we can solve the equation and determine the speed $v(t)$.

12.1.2   General use of differential equations

The simple example above illustrates how differential equations are typically used in a variety of contexts:

Procedure 12.1 (Modelling with differential equations).

1. *A quantity of interest is modelled by a function x.*

2. *From some known principle a relation between x and its derivatives is derived, in other words, a differential equation.*

3. *The differential equation is solved by a mathematical or numerical method.*

4. *The solution of the equation is interpreted in the context of the original problem.*

There are several reasons for the success of this procedure. The most basic reason is that many naturally occurring quantities can be represented as mathematical functions. This includes physical quantities like position, speed and temperature, which may vary in both space and time. It also includes quantities like 'money in the bank' and even vaguer, but quantifiable concepts like for instance customer satisfaction, both of which will typically vary with time.

Another reason for the popularity of modelling with differential equations is that such equations can usually be solved quite effectively. For some equations it is possible to find an explicit expression for the unknown function, but this is rare. For a large number of equations though, it is possible to compute good approximations to the solution via numerical algorithms, and this is the main topic in this chapter.

12.1.3 Different types of differential equations

Before we start discussing numerical methods for solving differential equations, it will be helpful to classify different types of differential equations. The simplest equations only involve the unknown function x and its first derivative x', as in (12.6); this is called a *first order differential equation*. If the equation involves higher derivatives up ot order p it is called a *pth order differential equation*. An important subclass are given by *linear differential equations*. A linear differential equation of order p is an equation on the form

$$x^{(p)}(t) = f(t) + g_0(t)x(t) + g_1(t)x'(t) + g_2(t)x''(t) + \cdots + g_{p-1}(t)x^{(p-1)}(t).$$

For all the equations we study here, the unknown function depends on only one variable which we usually label as t. Such equations are referred to as *ordinary differential equations*. This is in contrast to equations where the unknown function depends on two or more variables, like the three coordinates of a point in space, these are referred to as *partial differential equations*.

12.2 First order differential equations

A first order differential equation is an equation on the form

$$x' = f(t, x).$$

Here $x = x(t)$ is the unknown function, and t is the free variable. The function f tells us how x' depends on both t and x and is therefore a function of two variables. Some examples may be helpful.

Example 12.2. Some examples of first order differential equations are

$$x' = 3, \qquad x' = 2t, \qquad x' = x, \qquad x' = t^3 + \sqrt{x}, \qquad x' = \sin(tx).$$

The first three equations are very simple. In fact the first two can be solved by integration and have the solutions $x(t) = 3t + C$ and $x(t) = t^2 + C$ where C is an arbitrary constant in both cases. The third equation cannot be solved by integration, but it is easy to check that the function $x(t) = Ce^t$ is a solution for any value of the constant C. It is worth noticing that all the first three equations are linear.

For the first three equations there are simple procedures that lead to the solutions. On the other hand, the last two equations do not have solutions given by simple formulas. In spite of this, we shall see that there are simple numerical methods that allow us to compute good approximations to the solutions. ∎

The situation described in example 12.2 is similar to what we had for non-linear equations and integrals: There are analytic solution procedures that work in some special situations, but in general the solutions can only be determined approximately by numerical methods.

In this chapter our main concern will be to derive numerical methods for solving differential equations on the form $x' = f(t, x)$ where f is a given function of two variables. The description may seem a bit vague since f is not known explicitly, but the advantage is that once the method has been deduced we may plug in almost any f.

When we solve differential equations numerically we need a bit more information than just the differential equation itself. If we look back on example 12.2, we notice that the solution in the first three cases involved a general constant C, just like when we determine indefinite integrals. This ambiguity is present in all differential equations, and cannot be handled very well by numerical solution methods. We therefore need to supply an extra condition that will specify the value of the constant. The standard way of doing this for first order equations is to specify one point on the solution of the equation. In other words, we demand that the solution should satisfy the equation $x(a) = x_0$ for some real numbers a and x_0.

Example 12.3. Let us consider the differential equation $x' = 2x$. It is easy to check that $x(t) = Ce^{2t}$ is a solution for any value of the constant C. If we add the initial value $x(0) = 1$, we are led to the equation $1 = x(0) = Ce^0 = C$, so $C = 1$ and the solution becomes $x(t) = e^{2t}$.

If we instead impose the initial condition $x(1) = 2$, we obtain the equation $2 = x(1) = Ce^2$ which means that $C = 2e^{-2}$. In this case the solution is therefore $x(t) = 2e^{-2} e^t = 2e^{2(t-1)}$.

The general initial condition is $x(a) = x_0$. This leads to $x_0 = x(a) = Ce^{2a}$ or $C = x_0 e^{-2a}$. The solution is therefore

$$x(t) = x_0 e^{2(t-a)}. \quad \blacksquare$$

Adding an initial condition to a differential equation is not just a mathematical trick to pin down the exact solution; it usually has a concrete physical interpretation. Consider for example the differential equation (12.6) which describes the speed of an object with mass m falling towards earth. The speed at a certain time is clearly dependent on how the motion started — there is a difference between just dropping a ball and throwing it towards the ground, but note that there is nothing in equation (12.6) to reflect this difference. If we measure time such that $t = 0$ when the object starts falling, we would have $v(0) = 0$ in the situation where it is simply dropped, we would have $v(0) = v_0$ if it is thrown down-

(a) (b)
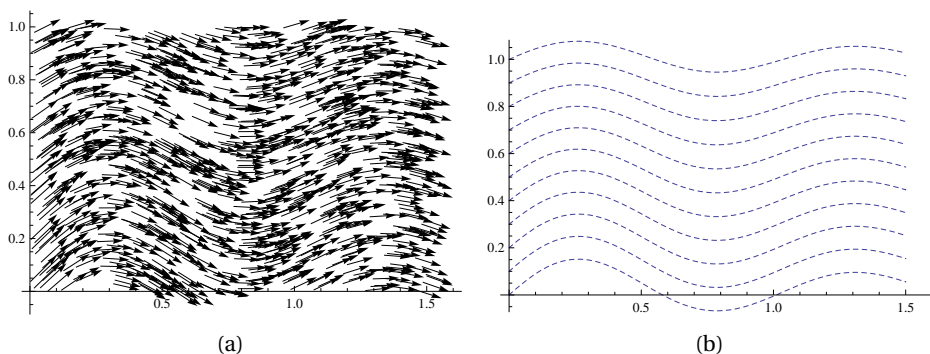
Figure 12.1. Figure (a) shows the tangents to the solutions of the differential equation $x' = \cos 6t / (1 + t + x^2)$ at 1000 random points in the square $[0, 1.5] \times [0, 1]$. Figure (b) shows the 11 solutions corresponding to the initial values $x(0) = i/10$, for $i = 0, 1, \ldots, 10$.

wards with speed v_0, and we would have $v(0) = -v_0$ if it was thrown upwards with speed v_0. Let us sum this up in an observation.

> **Observation 12.4** (First order differential equation). *A first order differential equation is an equation on the form $x' = f(t, x)$, where $f(t, x)$ is a function of two variables. In general, this kind of equation has many solutions, but a specific solution is obtained by adding an initial condition $x(a) = x_0$. A complete formulation of a first order differential equation is therefore*
>
> $$x' = f(t, x), \qquad x(a) = x_0. \tag{12.7}$$

It is equations of this kind that we will be studying in most of the chapter, with special emphasis on deriving numerical solution algorithms.

12.2.1 A geometric interpretation of first order differential equations

The differential equation in (12.7) has a natural geometric interpretation: At any point (t, x), the equation $x' = f(t, x)$ prescribes the slope of the solution through this point. This is illustrated in figure 12.1a for the differential equation

$$x' = f(t, x) = \frac{\cos 6t}{1 + t + x^2}. \tag{12.8}$$

A typical arrow starts at a point (t, x) and has slope given by $x' = f(t, x)$, and therefore shows the tangent to the solution that passes through the point. The image was obtained by picking 1000 points at random and drawing the corresponding tangent at each of the points.

Behind the many arrows in figure 12.1 we perceive a family of wave-like functions. This is shown much more clearly in figure 12.1b. The 11 functions in this figure represent solutions of the differential equation (12.8), each corresponding to one of the initial conditions $x(0) = i/10$ for $i = 0, \ldots, 10$.

> **Observation 12.5** (Geomteric interpretation of differential equation). *The differential equation $x' = f(t, x)$ describes a family of functions whose tangent at the point (t, x) has slope $f(t, x)$. By adding an initial condition $x(a) = x_0$, a particular solution, or solution curve, is selected from the family of solutions.*

12.2.2 Conditions that guarantee existence of one solution

The class of differential equations described by (12.7) is quite general since we have not placed any restrictions on the function f, and this may lead to some problems. Consider for example the equation

$$x' = \sqrt{1 - x^2}. \tag{12.9}$$

Since we are only interested in solutions that are real functions, we have to be careful so we do not select initial conditions that lead to square roots of negative numbers. The initial condition $x(0) = 0$ would be fine, as would $x(1) = 1/2$, but $x(0) = 2$ would mean that $x'(0) = \sqrt{1 - x(0)^2} = \sqrt{-3}$ which does not make sense.

For the general equation $x' = f(t, x)$ there are many potential pitfalls. As in the example, the function f may involve roots which require the expressions under the roots to be nonnegative, there may be logarithms which require the arguments to be positive, inverse sines or cosines which require the arguments to not exceed 1 in absolute value, fractions which do not make sense if the denominator becomes zero, and combinations of these and other restrictions. On the other hand, there are also many equations that do not require any restrictions on the values of t and x. This is the case when $f(t, x)$ is a polynomial in t and x, possibly combined with sines, cosines and exponential functions.

The above discussion suggests that the differential equation $x' = f(t, x)$ may not have a solution. Or it may have more than one solution if f has certain kinds of problematic behaviour. The most common problem that may occur is that there may be one or more points (t, x) for which $f(t, x)$ is not defined, as with equation (12.9) above. So-called *existence and uniqueness theorems* specify conditions on f which guarantee that a unique solutions can be found. Such theorems may appear rather abstract, and their proofs are often challenging. We are going to quote one such theorem, but the proof requires techniques which are beyond the scope of these notes.

Before we state the theorem, we need to introduce some notation. It turns out that how $f(t,x)$ depends on x influences the solution in an essential way. We therefore need to restrict the behaviour of the derivative of $f(t,x)$ when viewed as a function of x. We will denote this derivative by $\partial f/\partial x$, or sometimes just f_x to save space. If for instance $f(t,x) = t + x^2$, then $f_x(t,x) = 2x$, while if $f(t,x) = \sin(tx)$ then $f_x(t,x) = t\cos(tx)$.

The theorem talks about a rectangle. This is just a set in the plane on the form $\mathbb{A} = [\alpha,\beta] \times [\gamma,\delta]$ and a point (t,x) lies in $\mathbb{A}$ if $t \in [\alpha,\beta]$ and $x \in [\delta,\gamma]$. A point (t,x) is an interior point of $\mathbb{A}$ if it does not lie on the boundary, i.e., if $\alpha < t < \beta$ and $\gamma < x < \delta$.

Theorem 12.6. *Suppose that the functions f and f_x are continuous in the rectangle $\mathbb{A} = [\alpha,\beta] \times [\gamma,\delta]$. If the point (a,x_0) lies in the interior of $\mathbb{A}$ there exists a number $\tau > 0$ such that the differential equation*

$$x' = f(t,x), \quad x(a) = x_0 \tag{12.10}$$

has a unique solution on the interval $[a-\tau, a+\tau]$ which is contained in $[\alpha,\beta]$.

Theorem 12.6 is positive and tells us that if a differential equation is 'nice' near an initial condition, it will have a unique solution that extends both to the left and right of the initial condition. 'Nice' here means that both f and f_x are continuous in a rectangle $\mathbb{A}$ which contains the point (a,x_0) in its interior, i.e., they should have no jumps, should not blow up to infinity, and so on, in $\mathbb{A}$. This is sufficient to prove that the equation has a unique solution, but it is generally not enough to guarantee that a numerical method will converge to the solution. In fact, it is not even sufficient to guarantee that a numerical method will avoid areas where the function f is not defined. For this reason we will strengthen the conditions on f when we state and analyse the numerical methods below and assume that $f(t,x)$ and $f_x(t,x)$ (and sometimes more derivatives) are continuous and bounded for t in some interval $[\alpha,\beta]$ and any real number x.

Notice also that we seek a solution on an interval $[a,b]$. The left end is where the initial condition is and the right end limits the area in which we seek a solution. It should be noted that it is not essential that the initial condition is at the left end; the numerical methods can be easily adapted to work in a situation where the initial condition is at the right end, see exercise 1.

Assumption 12.7. *In the numerical methods for solving the equation*

$$x' = f(t,x), \quad x(a) = x_0,$$

278

to be introduced below, it is assumed that the function $f(t, x)$ and its derivative $f_x(t, x)$ with respect to x are well-defined, continuous, and bounded in a set $[\alpha, \beta] \times \mathbb{R}$, i.e., for all (t, x) such that $\alpha \leq t \leq \beta$ and $x \in \mathbb{R}$. It is also assumed that a solution $x(t)$ is sought for t in an interval $[a, b]$ that is strictly contained in $[\alpha, \beta]$, i.e., that $\alpha < a < b < \beta$.

The conditions in assumption 12.7 are quite restrictive and leave out many differential equations of practical interest. However, our focus is on introducing the ideas behind the most common numerical methods and analysing their error, and not on establishing exactly when they will work. It is especially the error analysis that depends on the functions f and f_x (and possibly other derivatives of f) being bounded for all values of t and x that may occur during the computations. In practice, the methods will work for many equations that do not satisfy assumption 12.7.

12.2.3   What is a numerical solution of a differential equation?

In earlier chapters we have derived numerical methods for solving nonlinear equations, for differentiating functions, and for computing integrals. A common feature of all these methods is that the answer is a single number. However, the solution of a differential equation is a function, and we cannot expect to find a single number that can approximate general functions well.

All the methods we derive compute the same kind of approximation: They start at the initial condition $x(a) = x_0$ and then compute successive approximations to the solution at a sequence of points t_1, t_2, t_3, ..., t_n where $a = t_0 < t_1 < t_2 < t_3 < \cdots < t_n = b$.

Fact 12.8 (General strategy for numerical solution of differential equations)**.**
Suppose the differential equation and initial condition

$$x' = f(t, x), \quad x(a) = x_0$$

are given together with an interval $[a, b]$ where a solution is sought. Suppose also that an increasing sequence of t-values $(t_k)_{k=0}^{n}$ are given, with $a = t_0$ and $b = t_n$, which in the following will be equally spaced with step length h, i.e.,

$$t_k = a + kh, \quad \text{for } k = 0, \ldots, n.$$

A numerical method for solving the equation is a recipe for computing a sequence of numbers x_0, x_1, ..., x_n such that x_k is an approximation to the true

12.3 Euler's method

Most methods for finding analytical solutions of differential equations appear rather tricky and unintuitive. In contrast, many numerical methods are based on simple, often geometric ideas. The simplest of these methods is *Euler's method* which is based directly on the geometric interpretation in observation 12.5.

12.3.1 Basic idea

We assume that the differential equation is

$$x' = f(t, x), \quad x(a) = x_0,$$

and our aim is to compute a sequence of approximations $(t_k, x_k)_{k=0}^n$ where $t_k = a + kh$. The initial condition provides us with one point on the true solution, so our first point is (t_0, x_0). We compute the slope of the tangent at (t_0, x_0) as $x_0' = f(t_0, x_0)$ which gives us the tangent $T(t) = x_0 + (t - t_0)x_0'$. As the approximation x_1 at t_1 we use the value of the tangent which is given by

$$x_1 = T(t_1) = x_0 + hx_0' = x_0 + hf(t_0, x_0).$$

But now we have a new approximate solution point (t_1, x_1), and from this we can compute the slope $x_1' = f(t_1, x_1)$. This allows us to compute an approximation $x_2 = x_1 + hx_1' = x_1 + hf(t_1, x_1)$ to the solution at t_2. If we continue this we can compute an approximation x_3 to the solution at t_3, then an approximation x_4 at t_4, and so on.

From this description we see that the basic idea is how to advance the approximate solution from a point (t_k, x_k) to a point (t_{k+1}, x_{k+1}).

Idea 12.9. *In Euler's method, an approximate solution (t_k, x_k) is advanced to (t_{k+1}, x_{k+1}) by following the tangent*

$$T(t) = x_k + (t - t_k)x_k' = x_k + (t - t_k)f(t_k, x_k)$$

to $t_{k+1} = t_k + h$. This results in the approximation

$$x_{k+1} = x_k + hf(t_k, x_k) \tag{12.11}$$

to $x(t_{k+1})$.

(a)   (b)
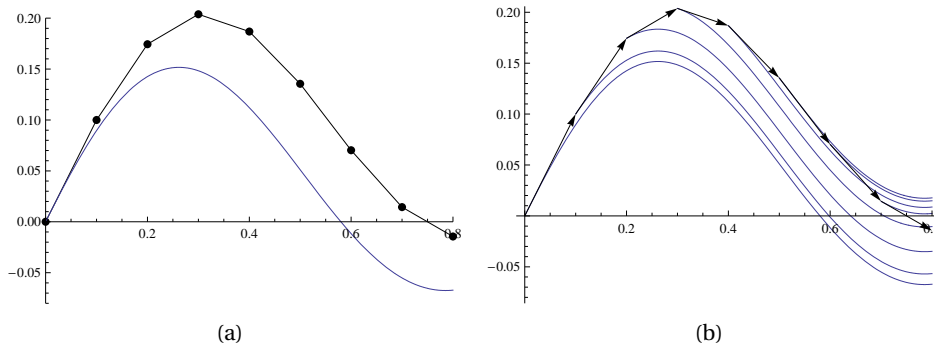
Figure 12.2. The plot in (a) shows the approximation produced by Euler's method to the solution of the differential equation $x' = \cos 6t/(1 + t + x^2)$ with initial condition $x(0) = 0$ (smooth graph). The plot in (b) shows the same solution augmented with the solution curves that pass through the points produced by Euler's method.

Idea 12.9 shows how we can get from one point on the approximation to the next, while the initial condition $x(a) = x_0$ provides us with a starting point. We therefore have all we need to compute a sequence of approximate points on the solution of the differential equation.

Algorithm 12.10 (Euler's method). *Let the differential equation $x' = f(t,x)$ be given together with the initial condition $x(a) = x_0$, the solution interval $[a,b]$, and the number of steps n. If the following algorithm is performed*

    h:=(b-a)/n;
    for $k := 0, 1, \ldots, n-1$
        $x_{k+1} := x_k + hf(t_k, x_k)$;
        $t_{k+1} := a + (k+1)h$;

the value x_k will be an approximation to the solution $x(t_k)$ of the differential equation, for each $k = 0, 1, \ldots, n$.

Figure 12.2 illustrates the behaviour of Euler's method for the differential equation

$$x' = \frac{\cos 6t}{1 + t + x^2}, \quad x(0) = 0.$$

This is just a piecewise linear approximation to the solution, see the figure in (a), but the figure in (b) illustrates better how the approximation is obtained. We start off by following the tangent at the initial condition $(0,0)$. This takes us to a point that is slightly above the graph of the true solution. At this point we compute a new tangent and follow this to the next point. However, there is a solution

281

curve that passes through this second point, and the line from the second to the third point is tangent to the solution curve which has the second point as initial condition. We therefore see that as we compute new approximate points on the solution, we jump between different solution curves of the differential equation $x' = f(t, x)$.

Note that Euler's method can also be obtained via a completely different argument. A common approximation to the derivative of x is given by

$$x'(t) \approx \frac{x(t+h) - x(t)}{h}.$$

If we rewrite this and make use of the fact that $x'(t) = f(t, x(t))$, we find that

$$x(t+h) \approx x(t) + hf(t, x(t))$$

which is the basis for Euler's method.

12.3.2   Error analysis

We know that Euler's method in most cases just produces an approximation to the true solution of the differential equation, but how accurate is the approximation? To answer this question we need to think more carefully about the various approximations involved.

The basic idea in Euler's method is to advance the solution from (t_k, x_k) to (t_{k+1}, x_{k+1}) with the relation

$$x_{k+1} = x_k + hf(t_k, x_k) \tag{12.12}$$

which stems from the approximation $x(t_{k+1}) \approx x(t_k) + hx'(t_k)$. If we include the error term in this simple Taylor polynomial, we obtain the identity

$$x(t_{k+1}) = x(t_k) + hx'(t_k) + \frac{h^2}{2}x''(\xi_k) = x(t_k) + hf(t_k, x(t_k)) + \frac{h^2}{2}x''(\xi_k), \tag{12.13}$$

where ξ_k is a number in the interval (t_k, t_{k+1}). We subtract (12.12) and end up with

$$x(t_{k+1}) - x_{k+1} = x(t_k) - x_k + h(f(t_k, x(t_k)) - f(t_k, x_k)) + \frac{h^2}{2}x''(\xi_k). \tag{12.14}$$

The number $\epsilon_{k+1} = x(t_{k+1}) - x_{k+1}$ is the global error accumulated by Euler's method at t_{k+1}. This error has two sources:

1. The *global error* $\epsilon_k = x(t_k) - x_k$ accumulated up to the previous step. This also leads to an error in computing $x'(t_k)$ since we use the value $f(t_k, x_k)$ instead of the correct value $f(t_k, x(t_k))$.

282

2. The *local error* we commit when advancing from (t_k, x_k) to (t_{k+1}, x_{k+1}) and ignore the remainder in Taylor's formula,

$$\frac{h^2}{2} x''(\xi_k).$$

The right-hand side of (12.14) can be simplified a little bit by noting that

$$f\big(t_k, x(t_k)\big) - f(t_k, x_k) = f_x(t_k, \theta_k)\big(x(t_k) - x_k\big) = f_x(t_k, \theta_k)\epsilon_k,$$

where θ_k is a number in the interval $\big(x_k, x(t_k)\big)$. The result is summarised in the following lemma.

Lemma 12.11. *If the two first derivatives of f exist, the error in using Euler's method for solving $x' = f(t, x)$ develops according to the relation*

$$\epsilon_{k+1} = \big(1 + h f_x(t_k, \theta_k)\big)\epsilon_k + \frac{h^2}{2} x''(\xi_k). \qquad (12.15)$$

where ξ_k is a number in the interval (t_k, t_{k+1}) and θ_k is a number in the interval $\big(x_k, x(t_k)\big)$. In other words, the global error at step $k + 1$ has two sources:

1. *The advancement of the global error at step k to the next step*

$$\big(1 + h f_x(t_k, \theta_k)\big)\epsilon_k.$$

2. *The local truncation error committed by only including two terms in the Taylor polynomial,*

$$h^2 x''(\xi_k)/2.$$

The lemma tells us how the error develops from one stage to the next, but we would really like to know explicitly what the global error at step k is. For this we need to simplify (12.15) a bit. The main complication is the presence of the two numbers θ_k and ξ_k which we know very little about. We use a standard trick: We take absolute values in (12.15) and replace the two terms $|f_x(t_k, \theta_k)|$ and $|x''(\xi_k)|$ by their maximum values,

$$|\epsilon_{k+1}| = \left|\big(1 + h f_x(t_k, \theta_k)\big)\epsilon_k + \frac{h^2}{2} x''(\xi_k)\right|$$

$$\leq \left|1 + h f_x(t_k, \theta_k)\right||\epsilon_k| + \frac{h^2}{2}|x''(\xi_k)|$$

$$\leq (1 + hC)|\epsilon_k| + \frac{h^2}{2} D.$$

283

This is where the restrictions on f and f_x that we mentioned in assumption 12.7 are needed: We need the two maximum values used to define the constants $D = \max_{t\in[a,b]} |x''(t)|$ and $C = \max_{t\in[a,b]} |f_x(t, x(t))|$ to exist. To simplify notation we write $\tilde{C} = 1 + hC$ and $\tilde{D} = Dh^2/2$, so the final inequality is

$$|\epsilon_{k+1}| \leq \tilde{C}|\epsilon_k| + \tilde{D}$$

which is valid for $k = 0, 1, \ldots, n-1$. This is a 'difference inequality' which can be solved quite easily. We do this by unwrapping the error terms,

$$
\begin{aligned}
|\epsilon_{k+1}| &\leq \tilde{C}|\epsilon_k| + \tilde{D} \\
&\leq \tilde{C}\big(\tilde{C}|\epsilon_{k-1}| + \tilde{D}\big) + \tilde{D} = \tilde{C}^2|\epsilon_{k-1}| + \big(1 + \tilde{C}\big)\tilde{D} \\
&\leq \tilde{C}^2\big(\tilde{C}|\epsilon_{k-2}| + \tilde{D}\big) + \big(1 + \tilde{C}\big)\tilde{D} \\
&\leq \tilde{C}^3|\epsilon_{k-2}| + \big(1 + \tilde{C} + \tilde{C}^2\big)\tilde{D} \\
&\vdots \\
&\leq \tilde{C}^{k+1}|\epsilon_0| + \big(1 + \tilde{C} + \tilde{C}^2 + \cdots + \tilde{C}^k\big)\tilde{D}.
\end{aligned}
\tag{12.16}
$$

We note that $\epsilon_0 = x(a) - x_0 = 0$ because of the initial condition, and the sum we recognise as a geometric series. This means that

$$|\epsilon_{k+1}| \leq \tilde{D} \sum_{i=0}^{k} \tilde{C}^i = \tilde{D}\frac{\tilde{C}^{k+1} - 1}{\tilde{C} - 1}.$$

We insert the values for $\tilde{C}$ and $\tilde{D}$ and obtain

$$|\epsilon_{k+1}| \leq hD\frac{(1 + hC)^{k+1} - 1}{2C}.
\tag{12.17}$$

Let us sum up our findings and add some further refinements.

Theorem 12.12 (Error in Euler's method)**.** *Suppose that f, f_t and f_x are continuous and bounded functions on the rectangle $\mathbb{A} = [\alpha, \beta] \times \mathbb{R}$ and that the interval $[a, b]$ satisfies $\alpha < a < b < \beta$. Let $\epsilon_k = x(t_k) - x_k$ denote the error at step k in applying Euler's method with n steps of length h to the differential equation $x' = f(t, x)$ on the interval $[a, b]$, with initial condition $x(a) = x_0$. Then*

$$|\epsilon_k| \leq h\frac{D}{2C}\left(e^{(t_k - a)C} - 1\right) \leq h\frac{D}{2C}\left(e^{(b-a)C} - 1\right)
\tag{12.18}$$

for $k = 0, 1, \ldots, n$ where the constants C and D are given by

$$C = \max_{(t,x)\in\mathbb{A}} |f_x(t,x)|,$$

$$D = \max_{t\in[a,b]} |x''(t)|.$$

Proof. From Taylor's formula with remainder we know that $e^t = 1+t+t^2 e^\eta/2$ for any positive, real number t, with η some real number in the interval $(0, t)$ (the interval $(t, 0)$ if $t < 0$). We therefore have $1 + t \le e^t$ and therefore $(1 + t)^k \le e^{kt}$. If we apply this to (12.17), with $k + 1$ replaced by k, we obtain

$$|\epsilon_k| \le \frac{hD}{2C} e^{khC},$$

and from this the first inequality in (12.18) follows since $kh = t_k - a$. The last inequality is then immediate since $t_k - a \le b - a$.

We will see in lemma 12.18 that $x'' = f_t + f_x f$. By assuming that f, f_t and f_x are continuous and bounded we are therefore assured that x'' is also continuous, and therefore that the constant D exists. ∎

The error estimate (12.18) depends on the quantities h, D, C, a and b. Of these, all except h are given by the differential equation itself, and therefore beyond our control. The step length h, however, can be varied as we wish, and the most interesting feature of the error estimate is therefore how the error depends on h. This is often expressed as

$$|\epsilon_k| \le O(h)$$

which simply means that $|\epsilon_k|$ is bounded by a constant times the step length h, just like in (12.18), without any specification of what the constant is. The error in numerical methods for solving differential equations typically behave like this.

Definition 12.13 (Accuracy of a numerical method)**.** *A numerical method for solving differential equations with step length h is said to be of order p if the error ϵ_k at step k satisfies*
$$|\epsilon_k| \le O(h^p).$$

The significance of the concept of order is that it tells us how quickly the error goes to zero with h. If we first try to run the numerical method with step

length h and then reduce the step length to $h/2$ we see that the error will roughly be reduced by a factor $1/2^p$. So the larger the value of p, the better the method, at least from the point of view of accuracy.

The accuracy of Euler's method can now be summed up quite concisely.

Corollary 12.14. *Euler's method is of order 1.*

In other words, if we halve the step length we can expect the error in Euler's method to also be halved. This may be a bit surprising in view of the fact that the local error in Euler's method is $O(h^2)$, see lemma 12.11. The explanation is that although the error committed in replacing $x(t_{k+1})$ by $x_k + hf(t_k, x_k)$ is bounded by Kh^2 for a suitable constant K, the error accumulates so that the global order becomes 1 even though the local approximation order is 2.

12.4 Differentiating the differential equation

Our next aim is to develop a whole family of numerical methods that can attain any order of accuracy, namely the Taylor methods. For these methods however, we need to know how to determine higher order derivatives of the solution of a differential equation at a point, and this is the topic of the current section.

We consider the standard equation

$$x' = f(t, x), \quad x(a) = x_0. \tag{12.19}$$

The initial condition explicitly determines a point on the solution, namely the point given by $x(a) = x_0$, and we want to compute the derivatives $x'(a)$, $x''(a)$, $x'''(a)$ and so on. It is easy to determine the derivative of the solution at $x = a$ since

$$x'(a) = f(a, x(a)) = f(a, x_0).$$

To determine higher derivatives, we simply differentiate the differential equation. This is best illustrated by an example.

Example 12.15. Suppose the equation is $x' = t + x^2$, or more explicitly,

$$x'(t) = t + x(t)^2, \quad x(a) = x_0. \tag{12.20}$$

At $x = a$ we know that $x(a) = x_0$, while the derivative is given by the differential equation

$$x'(a) = a + x_0^2.$$

If we differentiate the differential equation, the chain rule yields

$$x''(t) = 1 + 2x(t)x'(t) = 1 + 2x(t)\left(t + x(t)^2\right) \tag{12.21}$$

286

where we have inserted the expression for $x'(t)$ given by the differential equation (12.20). This means that at any point t where $x(t)$ (the solution) and $x'(t)$ (the derivative of the solution) is known, we can also determine the second derivative of the solution. In particular, at $x = a$, we have

$$x''(a) = 1 + 2x(a)x'(a) = 1 + 2x_0(a + x_0^2).$$

Note that the relation (12.21) is valid for any value of t, but since the right-hand side involves $x(t)$ and $x'(t)$ these quantities must be known. The derivative in turn only involves $x(t)$, so at a point where $x(t)$ is known, we can determine both $x'(t)$ and $x''(t)$.

What about higher derivatives? If we differentiate (12.21) once more, we find

$$x'''(t) = 2x'(t)x'(t) + 2x(t)x''(t) = 2\big(x'(t)^2 + x(t)x''(t)\big). \tag{12.22}$$

The previous formulas express $x'(t)$ and $x''(t)$ in terms of $x(t)$ and if we insert this at $x = a$ we obtain

$$x'''(a) = 2\big(x'(a)^2 + x(a)x''(a)\big) = 2\Big(\big(a + x_0^2\big)^2 + x_0\big(1 + 2x_0(a + x_0^2)\big)\Big).$$

In other words, at any point t where the solution $x(t)$ is known, we can also determine $x'(t)$, $x''(t)$ and $x'''(t)$. And by differentiating (12.22) the required number of times, we see that we can in fact determine any derivative $x^{(n)}(t)$ at a point where $x(t)$ is known. ∎

It is important to realise the significance of example 12.15. Even though we do not have a general formula for the solution $x(t)$ of the differential equation, we can easily find explicit formulas for the derivatives of x at a single point where the solution is known. One particular such point is the point where the initial condition is given. One obvious restriction is that the derivatives must exist.

Lemma 12.16 (Determining derivatives)**.** *Let $x' = f(t, x)$ be a differential equation with initial condition $x(a) = x_0$, and suppose that the derivatives of $f(t, x)$ of order $p - 1$ exist at the point (a, x_0). Then the pth derivative of the solution $x(t)$ at $x = a$ can be expressed in terms of a and x_0, i.e.,*

$$x^{(p)}(a) = F_p(a, x_0), \tag{12.23}$$

where F_p is a function defined by f and its derivatives of order less than p.

Proof. The proof is essentially the same as in example 12.15, but since f is not known explicitly, the argument becomes a bit more abstract. We use induction

on n, the order of the derivative. For $p = 1$, equation (12.23) just says that $x'(a) = F_1(a, x_0)$. In this situation we therefore have $F_1 = f$ and the result is immediate.

Suppose now that the result has been shown to be true for $p = k$, i.e., that

$$x^{(k)}(a) = F_k(a, x_0), \qquad (12.24)$$

where F_k depends on f and its derivatives of order less than k; we must show that it is also true for $p = k + 1$. To do this we differentiate both sides of (12.24) and obtain

$$x^{(k+1)}(a) = \frac{\partial F_k}{\partial t}(a, x_0) + \frac{\partial F_k}{\partial x}(a, x_0)x'(a). \qquad (12.25)$$

The right-hand side of (12.25) defines F_{k+1},

$$F_{k+1}(a, x_0) = \frac{\partial F_k}{\partial t}(a, x_0) + \frac{\partial F_k}{\partial x}(a, x_0)f(a, x_0),$$

where we have also used the fact that $x'(a) = f(a, x_0)$. Since F_k involves partial derivatives of f up to order $k - 1$, it follows that F_{k+1} involves partial derivatives of f up to order k. This proves all the claims in the theorem. ∎

Example 12.17. The function F_p that appears in Lemma 12.16 may seem a bit mysterious, but if we go back to example 12.15, we see that it is in fact quite straightforward. In this specific case we have

$$x' = F_1(t, x) = f(t, x) = t + x^2, \qquad (12.26)$$

$$x'' = F_2(t, x) = 1 + 2xx' = 1 + 2tx + 2x^3, \qquad (12.27)$$

$$x''' = F_3(t, x) = 2(x'^2 + xx'') = 2\big((t + x^2)^2 + x(1 + 2tx + 2x^3)\big). \qquad (12.28)$$

This shows the explicit expressions for F_1, F_2 and F_3. The expressions can usually be simplified by expressing x'' in terms of t, x and x', and by expressing x''' in terms of t, x, x' and x'', as shown in the intermediate formulas in (12.26)–(12.28). ∎

For later reference we record the general formulas for the first three derivatives of x in terms of f.

Lemma 12.18. *Let $x(t)$ be a solution of the differential equation $x' = f(t, x)$. Then*

$$x' = f, \quad x'' = f_t + f_x f, \quad x''' = f_{tt} + 2f_{tx}f + f_{xx}f^2 + f_t f_x + f_x^2 f,$$

at any point where the derivatives exist.

Lemma 12.16 tells us that at some point t where we know the solution $x(t)$, we can also determine all derivatives of the solution, just as we did in example 12.15. The obvious place where this can be exploited is at the initial condition. But this property also means that if in some way we have determined an approximation $\hat{x}$ to $x(t)$ we can compute approximations to all derivatives at t as well. Consider again example 12.15 and let us imagine that we have an approximation $\hat{x}$ to the solution $x(t)$ at t. We can then successively compute the approximations

$$x'(t) \approx \hat{x}' = F_1(t, \hat{x}) = f(t, \hat{x}) = x + \hat{x}^2,$$
$$x''(t) \approx \hat{x}'' = F_2(t, \hat{x}) = 1 + 2\hat{x}\hat{x}',$$
$$x'''(t) \approx \hat{x}''' = F_3(t, \hat{x}) = 2(\hat{x}'^2 + \hat{x}\hat{x}'').$$

This corresponds to finding the exact derivatives of the solution curve that has the value $\hat{x}'$ at t. The same is of course true for a general equation.

The fact that all derivatives of the solution of a differential equation at a point can be computed as in lemma 12.16 is the foundation of a whole family of numerical methods for solving differential equations.

12.5 Taylor methods

In this section we are going to derive the family of numerical methods that are usually referred to as Taylor methods. An important ingredient in these methods is the computation of derivatives of the solution at a single point which we discussed in section 12.4. We first introduce the idea behind the methods and the resulting algorithms, and then discuss the error. We focus on the quadratic case as this is the simplest, but also illustrates the general principle.

12.5.1 Derivation of the Taylor methods

The idea behind Taylor methods is to approximate the solution by a Taylor polynomial of a suitable degree. In Euler's method, which is the simplest Taylor method, we used the approximation

$$x(t + h) \approx x(t) + hx'(t).$$

The quadratic Taylor method is based on the more accurate approximation

$$x(t + h) \approx x(t) + hx'(t) + \frac{h^2}{2} x''(t). \tag{12.29}$$

To describe the algorithm, we need to specify how the numerical solution can be advanced from a point (t_k, x_k) to a new point (t_{k+1}, x_{k+1}) with $t_{k+1} = t_k + h$.

The idea is to use (12.29) and compute x_{k+1} as

$$x_{k+1} = x_k + hx'_k + \frac{h^2}{2} x''_k. \tag{12.30}$$

The numbers x_k, x'_k and x''_k are approximations to the function value and derivatives of the solution at t. These are obtained via the recipe in lemma 12.16. An example should make this clear.

Example 12.19. Let us consider the differential equation

$$x' = f(t,x) = F_1(t,x) = t - \frac{1}{1+x}, \quad x(0) = 1, \tag{12.31}$$

which we want to solve on the interval $[0,1]$. To illustrate the method, we choose a large step length $h = 0.5$ and attempt to find an approximate numerical solution at $x = 0.5$ and $x = 1$ using a quadratic Taylor method.

From (12.31) we obtain

$$x''(t) = F_2(t,x) = 1 + \frac{x'(t)}{\left(1 + x(t)\right)^2}. \tag{12.32}$$

To compute an approximation to $x(h)$ we use the quadratic Taylor polynomial

$$x(h) \approx x_1 = x(0) + hx'(0) + \frac{h^2}{2} x''(0).$$

The differential equation (12.31) and (12.32) yield

$$\begin{aligned} x(0) &= x_0 = 1, \\ x'(0) &= x'_0 = 0 - 1/2 = -1/2, \\ x''(0) &= x''_0 = 1 - 1/8 = 7/8, \end{aligned}$$

which leads to the approximation

$$x(h) \approx x_1 = x_0 + hx'_0 + \frac{h^2}{2} x''_0 = 1 - \frac{h}{2} + \frac{7h^2}{16} = 0.859375.$$

To prepare for the next step we need to determine approximations to $x'(h)$ and $x''(h)$ as well. From the differential equation (12.31) and (12.32) we find

$$\begin{aligned} x'(h) &\approx x'_1 = F_1(t_1, x_1) = t_1 - 1/(1+x_1) = -0.037815126, \\ x''(h) &\approx x''_1 = F_2(t_1, x_1) = 1 + x'_1/(1+x_1)^2 = 0.98906216, \end{aligned}$$

rounded to eight digits. From this we can compute the approximation

$$x(1) = x(2h) \approx x_2 = x_1 + hx'_1 + \frac{h^2}{2} x''_1 = 0.96410021.$$

The result is shown in figure 12.3a. ∎

(a)   (b)
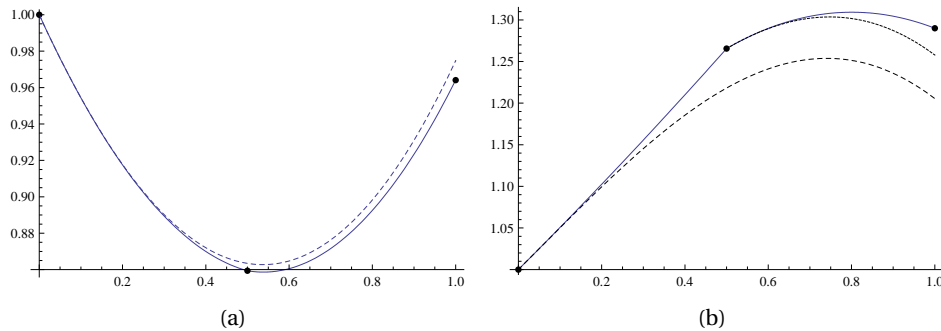
Figure 12.3. The plots show the result of solving a differential equation numerically with the quadratic Taylor method. The plot in (a) show the first two steps for the equation $x' = t - 1/(1 + x)$ with $x(0) = 1$ and $h = 0.5$, while the plot in (b) show the first two steps for the equation $x' = \cos(3t/2) - 1/(1 + x)$ with $x(0) = 1$ and $h = 0.5$. The dots show the computed approximations, while the solid curves show the parabolas that are used to compute the approximations. The exact solution is shown by the dashed curve in both cases.

Figure 12.3 illustrates the first two steps of the quadratic Talor method for two equations. The solid curve shows the two parabolas used to compute the approximate solution points in both cases. In figure (a) it seems like the two parabolas join together smoothly, but this is just a feature of the underlying differential equation. The behaviour in (b), where the two parabolas meet at a slight corner is more representative, although in this case, the first parabola is almost a straight line. In practice the solution between two approximate solution points will usually be approximated by a straight line, not a parabola.

Let us record the idea behind the quadratic Taylor method.

Idea 12.20 (Quadratic Taylor method). *The quadratic Taylor method advances the solution from a point (t_k, x_k) to a point (t_{k+1}, x_{k+1}) by evaluating the approximate Taylor polynomial*

$$x(t) \approx x_k + (t - t_k)x'_k + \frac{(t - t_k)^2}{2}x''_k$$

at $x = t_{k+1}$. In other words, the new value x_{k+1} is given by

$$x_{k+1} = x_k + hx'_k + \frac{h^2}{2}x''_k$$

where the values x_k, x'_k and x''_k are obtained as described in lemma 12.16 and $h = t_{k+1} - t_k$.

291

This idea is easily translated into a simple algorithm. At the beginning of a new step, we know the previous approximation x_k, but need to compute the approximations to x'_k and x''_k. Once these are known we can compute x'_{k+1} and t_{k+1} before we proceed with the next step. Note that in addition to the function $f(t, x)$ which defines the differential equation we also need the function F_2 which defines the second derivative, as in lemma 12.16. This is usually determined by manual differentiation as in the examples above.

Algorithm 12.21 (Quadratic Taylor method)**.** *Let the differential equation* $x' = f(t, x)$ *be given together with the initial condition* $x(a) = x_0$, *the solution interval* $[a, b]$ *and the number of steps n, and let the function F_2 be such that* $x''(t) = F_2\big(t, x(t)\big)$. *The quadratic Taylor method is given by the algorithm*

$h := (b - a)/n;$

$t_0 := a;$

for $k := 0, 1, \ldots, n - 1$

　　　$x'_k := f(t_k, x_k);$

　　　$x''_k := F_2(t_k, x_k);$

　　　$x_{k+1} := x_k + h x'_k + h^2 x''_k / 2;$

　　　$t_{k+1} := a + (k + 1)h;$

After these steps the value x_k will be an approximation to the solution $x(t_k)$ of the differential equation, for each $k = 0, 1, \ldots, n$.

The quadratic Taylor method is easily generalised to higher degrees by including more terms in the Taylor polynomial. The *Taylor method of degree p* uses the formula

$$x_{k+1} = x_k + h x'_k + \frac{h^2}{2} x''_k + \cdots + \frac{h^{p-1}}{(p-1)!} x_k^{(p-1)} + \frac{h^p}{p!} x_k^{(p)} \tag{12.33}$$

to advance the solution from the point (t_k, x_k) to (t_{k+1}, x_{k+1}). Just like for the quadratic method, the main challenge is the determination of the derivatives, whose complexity may increase quickly with the degree. It is possible to make use of software for symbolic computation to produce the derivatives, but it is much more common to use a numerical method that mimics the behaviour of the Taylor methods by evaluating $f(t, x)$ at intermediate steps instead of computing higher order derivatives, like the *Runge-Kutta methods* in section 12.6.3.

12.5.2　Error analysis for Taylor methods

In section 12.3.2 we discussed the error in Euler's method. In this section we use the same technique to estimate the error in the Taylor methods.

The Taylor method of degree p advances the numerical solution from the point (t_k, x_k) to (t_{k+1}, x_{k+1}) with the formula (12.33). This is based on the exact relation

$$x(t_{k+1}) = x(t_k) + h x'(t_k) + \cdots + \frac{h^p}{p!} x^{(p)}(t_k) + \frac{h^{p+1}}{(p+1)!} x^{(p+1)}(\xi_k), \qquad (12.34)$$

where ξ_k is a number in the interval (t_k, t_{k+1}). When we omit the last term on the right and use (12.33) instead we have a local truncation error given by

$$\frac{h^{p+1}}{(p+1)!} x^{(p+1)}(\xi_k),$$

i.e., of order $O(h^{p+1})$. As for Euler's method, the challenge is to see how this local error at each step filters through and leads to the global error.

We will follow a procedure very similar to the one that we used to estimate the error in Euler's method. We start by rewriting (12.33) as

$$x_{k+1} = x_k + h \Phi(t_k, x_k, h) \qquad (12.35)$$

with the function Φ given by

$$\Phi(t_k, x_k, h) = x'_k + \frac{h}{2} x''_k + \cdots + \frac{h^{p-1}}{(p-1)!} x_k^{(p-1)}$$
$$= F_1(t_k, x_k) + \frac{h^2}{2} F_2(t_k, x_k) + \cdots + \frac{h^{p-1}}{(p-1)!} F_{p-1}(t_k, x_k), \qquad (12.36)$$

where F_1, F_2, ..., F_{p-1} are the functions given in lemma 12.16. With the same notation we can write (12.34) as

$$x(t_{k+1}) = x(t_k) + h \Phi \big(t_k, x(t_k), h \big) + \frac{h^{p+1}}{(p+1)!} x^{(p+1)}(\xi_k). \qquad (12.37)$$

The first step is to derive the relation which corresponds to (12.14) by subtracting (12.35) from (12.37),

$$x\big(t_{k+1}\big) - x_{k+1} = x(t_k) - x_k + h \Big(\Phi \big(t_k, x(t_k), h \big) - \Phi(t_k, x_k, h) \Big) + \frac{h^{p+1}}{(p+1)!} x^{(p+1)}(\xi_k).$$

We introduce the error $\epsilon_k = x(t_k) - x_k$ and rewrite this as

$$\epsilon_{k+1} = \epsilon_k + h \Big(\Phi \big(t_k, x(t_k), h \big) - \Phi(t_k, x_k, h) \Big) + \frac{h^{p+1}}{(p+1)!} x^{(p+1)}(\xi_k). \qquad (12.38)$$

In the following we assume that the derivatives of the functions $\{F_i\}_{i=1}^{p-1}$ with respect to x exist. This means that the derivative of $\Phi(t, x)$ with respect to x also exist, so the mean value theorem means that (12.38) can be written as

$$
\begin{aligned}
\epsilon_{k+1} &= \epsilon_k + h\frac{\partial \Phi}{\partial x}(t_k, \theta_k, h)\epsilon_k + \frac{h^{p+1}}{(p+1)!}x^{(p+1)}(\xi_k) \\
&= \left(1 + h\frac{\partial \Phi}{\partial x}(t_k, \theta_k, h)\right)\epsilon_k + \frac{h^{p+1}}{(p+1)!}x^{(p+1)}(\xi_k),
\end{aligned}
\tag{12.39}
$$

where θ_k is a number in the interval $\big(x_k, x(t_k)\big)$. This relation is similar to equation (12.15), and the rest of the analysis can be performed as in section 12.3.2. We take absolute values, use the triangle inequality, and introduce the constants C and D. This gives us the inequality

$$
|\epsilon_{k+1}| \le (1 + hC)|\epsilon_k| + \frac{h^{p+1}}{(p+1)!}D.
\tag{12.40}
$$

Proceedings just as in section 12.3.2 we end up with an analogous result.

Theorem 12.22 (Error in Taylor method of degree p). *Let $\epsilon_k = x(t_k) - x_k$ denote the error at step k in applying the Taylor method of degree p with n steps of length h to the differential equation $x' = f(t, x)$ on the interval $[a, b]$, with initial condition $x(a) = x_0$. Suppose that the derivatives of f of order p exist and are continuous in a set $[\alpha, \beta] \times \mathbb{R}$ with $\alpha < a < b < \beta$. Then*

$$
|\epsilon_k| \le h^p \frac{D}{C(p+1)!}\left(e^{(t_k-a)C} - 1\right) \le h^p \frac{D}{C(p+1)!}\left(e^{(b-a)C} - 1\right)
\tag{12.41}
$$

for $k = 0, 1, \ldots, n$ where

$$
C = \max_{(t,x)\in\mathbb{A}}\left|\frac{\partial \Phi}{\partial x}(t, x(t))\right|,
$$
$$
D = \max_{t\in[a,b]}|x^{(p+1)}(t)|.
$$

Note that even though the local truncation error in (12.34) is $O(h^{p+1})$, the global approximation order is p. In other words, the local errors accumulate so that we lose one approximation order in passing from local error to global error, just like for Euler's method. In fact the error analysis we have used here both for Euler's method and for the Taylor methods (which include Euler's method as a special case), work in quite general situations, and below we will use it to analyse other methods as well.

We end the section with a shorter version of theorem 12.22.

> **Corollary 12.23.** *The Taylor method of degree p has global approximation order p.*

12.6   Other methods

The big advantage of the Taylor methods is that they can attain any approximation order. Their disadvantage is that they require symbolic differentiation of the differential equation (except for Euler's method). In this section we are going to develop some methods of higher order than Euler's method that do not require differentiation of the differential equation. Instead they advance from (t_k, x_k) to (t_{k+1}, x_{k+1}) by evaluating $f(t, x)$ at intermediate points in the interval $[t_k, t_{k+1}]$.

12.6.1   Euler's midpoint method

The first method we consider is a simple extension of Euler's method. If we look at the plots in figure 12.2, we notice how the tangent is a good approximation to a solution curve at the initial condition, but the quality of the approximation deteriorates as we move to the right. One way to improve on Euler's method is therefore to estimate the slope of each line segment better. In *Euler's midpoint method* this is done via a two-step procedure which aims to estimate the slope at the midpoint between the two solution points. In proceeding from (t_k, x_k) to (t_{k+1}, x_{k+1}) we would like to use the tangent to the solution curve at the midpoint $t_k + h/2$. But since we do not know the value of the solution curve at this point, we first compute an approximation $x_{k+1/2}$ to the solution at $t_k + h/2$ using the traditional Euler's method. Once we have this approximation, we can determine the slope of the solution curve that passes through the point and use this as the slope for a straight line that we follow from t_k to t_{k+1} to determine the new approximation x_{k+1}. This idea is illustrated in figure 12.4.

> **Idea 12.24** (Euler's midpoint method). *In Euler's midpoint method the solution is advanced from (t_k, x_k) to $(t_k + h, x_{k+1})$ in two steps: First an approximation to the solution is computed at the midpoint $t_k + h/2$ by using Euler's method with step length $h/2$,*
>
> $$x_{k+1/2} = x_k + \frac{h}{2} f(t_k, x_k).$$
>
> *Then the solution is advanced to t_{k+1} by following the straight line from (t_k, x_k) with slope given by $f(t_k + h/2, x_{k+1/2})$,*
>
> $$x_{k+1} = x_k + h f(t_k + h/2, x_{k+1/2}).$$
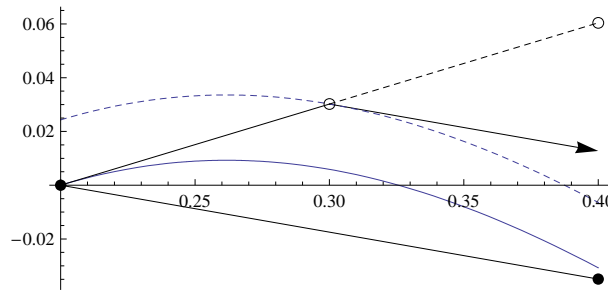
Figure 12.4. The figure illustrates the first step of the midpoint Euler method, starting at $x = 0.2$ and with step length $h = 0.2$. We start by following the tangent at the starting point ($x = 0.2$) to the midpoint ($x = 0.3$). Here we determine the slope of the solution curve that passes through this point and use this as the slope for a line through the starting point. We then follow this line to the next t-value ($x = 0.4$) to determine the first approximate solution point. The solid curve is the correct solution and the open circle shows the approximation produced by Euler's method.

Once the basic idea is clear it is straightforward to translate this into a complete algorithm for computing an approximate solution to the differential equation.

Algorithm 12.25 (Euler's midpoint method)**.** *Let the differential equation $x' = f(t, x)$ be given together with the initial condition $x(a) = x_0$, the solution interval $[a, b]$ and the number of steps n. Euler's midpoint method is given by*

$$h := (b - a)/n;$$
$$\text{for } k := 0, 1, \ldots, n-1$$
$$\quad x_{k+1/2} := x_k + h f(t_k, x_k)/2;$$
$$\quad x_{k+1} := x_k + h f(t_k + h/2, x_{k+1/2});$$
$$\quad t_{k+1} := a + (k+1)h;$$

After these steps the value x_k will be an approximation to the solution $x(t_k)$ of the differential equation, for each $k = 0, 1, \ldots, n$.

As an alternative viewpoint, let us recall the two approximations for numerical differentiation given by

$$x'(t) \approx \frac{x(t+h) - x(t)}{h},$$
$$x'(t + h/2) \approx \frac{x(t+h) - x(t)}{h}.$$
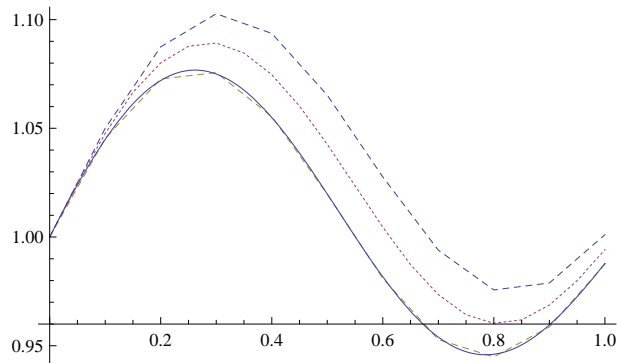
Figure 12.5. Comparison of Euler's method and Euler's midpoint method for the differential equation $x' = \cos(6t)/(1 + t + x^2)$ with initial condition $x(0) = 1$ with step length $h = 0.1$. The solid curve is the exact solution and the two approximate solutions are dashed. The dotted curve in the middle is the approximation produced by Euler's method with step length $h = 0.05$. The approximation produced by Euler's midpoint method appears almost identical to the exact solution.

As we saw above, the first one is the basis for Euler's method, but we know from our study of numerical differentiation that the second one is more accurate. If we solve for $x(t + h)$ we find

$$x(t + h) \approx x(t) + hx'(t + h/2)$$

and this relation is the basis for Euler's midpoint method.

In general Euler's midpoint method is more accurate than Euler's method since it is based on a better approximation of the first derivative, see Figure 12.5 for an example. However, this extra accuracy comes at a cost: the midpoint method requires two evaluations of $f(t, x)$ per iteration instead of just one for the regular method. In many cases this is insignificant, although there may be situations where f is extremely complicated and expensive to evaluate, or the added evaluation may just not be feasible. But even then it is generally better to use Euler's midpoint method with a double step length, see figure 12.5.

12.6.2   Error analysis for Euler's midpoint method

In this section we are going to analyse the error in Euler's midpoint method with the same technique as was used to analyse Euler's method and the Taylor methods. From idea 12.24 we recall that the approximation is advanced from (t_k, x_k) to (t_{k+1}, x_{k+1}) with the formula

$$x_{k+1} = x_k + hf\big(t_k + h/2, x_k + hf(t_k, x_k)/2\big). \tag{12.42}$$

The idea behind the analysis is to apply Taylor's formula and replace the outer evaluation of f to an evaluation of f and its derivatives at (t_k, x_k), and then sub-

tract a Taylor expansion of f, in analogy to the analysis of Euler's method. We first do a Taylor expansion with respect to the first variable in (12.42),

$$f(t_k + h/2, x) = f(t_k, x) + \frac{h}{2} f_t(t_k, x) + \frac{h^2}{8} f_{tt}(t_k, x) + O(h^3),$$ (12.43)

where $x = x_k + hf(t_k, x_k)/2$, and the error is indicated by the $O(h^3)$ term. We then replace each of the function terms by Taylor expansions with respect to the second variable about x_k,

$$f(t_k, x) = f + \frac{hf}{2} f_x + \frac{h^2 f^2}{8} f_{xx} + O(h^3),$$

$$f_t(t_k, x) = f_t + \frac{hf}{2} f_{tx} + O(h^2),$$

$$f_{tt}(t_k, x) = f_{tt} + O(h),$$

where the functions with no arguments are evaluated at (t_k, x_k). If we insert this in (12.43) we obtain

$$f(t_k + h/2, x) = f + \frac{h}{2} f_x f + \frac{h^2}{8} f_{xx} f^2 + \frac{h}{2} f_t + \frac{h^2}{4} f_{tx} f + \frac{h^2}{8} f_{tt} + O(h^3)$$

$$= f + \frac{h}{2} (f_t + f_x f) + \frac{h^2}{8} (f_{tt} + 2 f_{tx} f + f_{xx} f^2) + O(h^3).$$

This means that (12.42) can be written

$$x_{k+1} = x_k + hf + \frac{h^2}{2} (f_t + f_x f) + \frac{h^3}{8} (f_{tt} + 2 f_{tx} f + f_{xx} f^2) + O(h^4).$$ (12.44)

On the other hand, a standard Taylor expansion of $x(t_{k+1})$ about t_k with remainder yields

$$x(t_{k+1}) = x(t_k) + hx'(t_k) + \frac{h^2}{2} x''(t_k) + \frac{h^3}{6} x'''(\xi_k)$$

$$= x(t_k) + hf\big(t_k, x(t_k)\big) + \frac{h^2}{2} \Big(\big(f_t(t_k, x(t_k)) + f_x\big(t_k, x(t_k)\big) \big) f\big(t_k, x(t_k)\big)$$

$$+ \frac{h^3}{6} x'''(\xi_k).$$ (12.45)

If we compare this with (12.44) we notice that the first three terms are similar. We follow the same recipe as for the Taylor methods and introduce the function

$$\Phi(t, x, h) = f(t, x) + \frac{h}{2} \big(f_t(t, x) + f_x(t, x) f(t, x) \big).$$ (12.46)

The equations (12.44) and (12.45) can then be written as

$$x_{k+1} = x_k + h\Phi(t_k, x_k, h) + \frac{h^3}{8}(f_{tt} + 2f_{tx}f + f_{xx}f^2) + O(h^4), \qquad (12.47)$$

$$x(t_{k+1}) = x(t_k) + h\Phi(t_k, x(t_k), h) + \frac{h^3}{6}x'''(\xi_k). \qquad (12.48)$$

We subtract (12.47) from (12.48) and obtain

$$x(t_{k+1}) - x_{k+1} = x(t_k) - x_k + h\Big(\Phi(t_k, x(t_k), h) - \Phi(t_k, x_k, h)\Big) + O(h^3), \qquad (12.49)$$

where all the terms of degree higher than 2 have been collected together in the $O(h^3)$ term.

Theorem 12.26. *The global error $\epsilon_k = x(t_k) - x_k$ in Euler's midpoint method is advanced from step k to step $k+1$ by the relation*

$$\epsilon_{k+1} = \epsilon_k + h\Big(\Phi(t_k, x(t_k), h) - \Phi(t_k, x_k, h)\Big) + O(h^3), \qquad (12.50)$$

where Φ is defined in equation (12.46). This means that $|\epsilon_k| = O(h^2)$, i.e., the global error in Euler's midpoint method is of second order, provided f, f_t, f_x and f_{xx} are all continuous and bounded on a set $[\alpha, \beta] \times \mathbb{R}$ such that $\alpha < a < b < \delta$.

Proof. The relation (12.50) is completely analogous to relation (12.38) for the Taylor methods. We can therefore proceed in the same way and end up with an inequality like (12.40),

$$|\epsilon_{k+1}| \leq (1 + hC)|\epsilon_k| + Dh^3.$$

As for Euler's method and the Taylor methods we lose one order of approximation when we account for error accumulation which means that $|\epsilon_k| = O(h^2)$. ∎

Theorem 12.26 shows that Euler's midpoint method is of second order, just like the second order Taylor method, but without the need for explicit formulas for the derivatives of f. Instead the midpoint method uses an extra evaluation of f halfway between t_k and t_{k+1}. The derivation of the error formula (12.49) illustrates why the method works; the formula (12.42), which is equivalent to (12.44), reproduces the first three terms of the Taylor expansion (12.45). We therefore see that the accuracy of the Taylor methods may be mimicked by performing extra evaluations of f between t_k and t_{k+1}. The Runge-Kutta methods achieve the same accuracy as higher order Taylor methods in this way.

In our error analysis we did not compare the $O(h^3)$ terms in (12.44) and (12.45), so one may wonder if perhaps these match as well? Lemma 12.18 gives an expression for x''' in terms of f and its derivatives and we see straightaway that (12.44) only matches some of these terms.

12.6.3 Runge-Kutta methods

Runge-Kutta methods is a family of methods that generalise the midpoint Euler method. The methods use several evaluations of f between each step in a clever way which leads to higher accuracy.

In the simplest Runge-Kutta methods, the new value x_{k+1} is computed from x_k with the formula

$$x_{k+1} = x_k + h\big(\lambda_1 f(t_k, x_k) + \lambda_2 f(t_k + r_1 h, x_k + r_2 h f(t_k, x_k))\big), \qquad (12.51)$$

where λ_1, λ_2, r_1, and r_2 are constants to be determined. The idea is to choose the constants in such a way that (12.51) mimics a Taylor method of the highest possible order. This can be done by following the recipe that was used in the analysis of Euler's midpoint method: Replace the outer function evaluation in (12.51) by a Taylor polynomial and choose the constants such that this matches as many terms as possible in the Taylor polynomial of $x(t)$ about $x = t_k$, see (12.44) and (12.45). It turns out that the first three terms in the Taylor expansion can be matched. This leaves one parameter free (we choose this to be $\lambda = \lambda_2$), and determines the other three in terms of λ,

$$\lambda_1 = 1 - \lambda, \quad \lambda_2 = \lambda, \quad r_1 = r_2 = \frac{1}{2\lambda}.$$

This determines a whole family of second order accurate methods.

Theorem 12.27 (Second order Runge-Kutta methods). *Let the differential equation $x' = f(t, x)$ with initial condition $x(a) = x_0$ be given. Then the numerical method which advances from (t_k, x_k) to $(t_{k+1}, x_{k+1}$ according to the formula*

$$x_{k+1} = x_k + h\left((1-\lambda)f(t_k, x_k) + \lambda f\left(t_k + \frac{h}{2\lambda}, x_k + \frac{hf(t_k, x_k)}{2\lambda}\right)\right), \qquad (12.52)$$

is second order accurate for any nonzero value of the parameter λ, provided f, f_t, f_x and f_{xx} are continuous and bounded in a set $[\alpha, \beta] \times \mathbb{R}$ with $\alpha < a < b < \beta$.

The proof is completely analogous to the argument used to establish the convergence rate of Euler's midpoint method. In fact, Euler's midpoint method

corresponds to the particular second order Runge-Kutta method with $\lambda = 1$. Another commonly used special case is $\lambda = 1/2$. This results in the iteration formula

$$x_{k+1} = x_k + \frac{h}{2}\Big(f(t_k, x_k) + f\big((t_k, x_k + h(t_k, x_k))\big)\Big),$$

which is often referred to as *Heun's method* or the improved Euler's method. Note also that the original Euler's may be considered as the special case $\lambda = 0$, but then the accuracy drops to first order.

It is possible to devise methods that reproduce higher degree polynomials at the cost of more intermediate evaluations of f. The derivation is analogous to the procedure used for the second order Runge-Kutta method, but more involved because the degree of the Taylor polynomials are higher. One member of the family of fourth order methods is particularly popular.

Theorem 12.28 (Fourth order Runge-Kutta method). *Suppose the differential equation $x' = f(t, x)$ with initial condition $x(a) = x_0$ is given. The numerical method given by the formulas*

$$\left.\begin{aligned}
k_0 &= f(t_k, x_k), \\
k_1 &= f(t_k + h/2, x_k + hk_0/2), \\
k_2 &= f(t_k + h/2, x_k + hk_1/2), \\
k_3 &= f(t_k + h, x_k + hk_2), \\
x_{k+1} &= x_k + \frac{h}{6}(k_0 + k_1 + k_2 + k_3),
\end{aligned}\right\} \quad k = 0, 1, \ldots, n$$

is fourth order accurate provided the derivatives of f up to order four are continuous and bounded in the set $[\alpha, \beta] \times \mathbb{R}$ with $a < \alpha < \beta < b$.

It can be shown that Runge-Kutta methods which use p evaluations pr. step are pth order accurate for $p = 1, 2, 3$, and 4. However, it turns out that 6 evaluations pr. step are necessary to get a method of order 5. This is one of the reasons for the popularity of the fourth order Runge-Kutta methods—they give the most orders of accuracy pr. evaluation.

12.6.4   Multi-step methods

The methods we have discussed so far are all called *one-step methods* since they advance the solution by just using information about one previous step. This is in contrast to an *order m multi-step method* which computes x_{k+1} based on x_k,

$x_{k-1}, \ldots, x_{k+1-m}$. The methods are based on integrating the differential equation,

$$x(t_{k+1}) = x(t_k) + \int_{t_k}^{t_{k+1}} x'(t)\,dt = x(t_k) + \int_{t_k}^{t_{k+1}} f\big(t, x(t)\big)\,dt.$$

The idea is to replace $x'(t)$ with a polynomial that interpolates $x'(t) = f(t, x)$ at previously computed points. More specifically, suppose that the m approximate values $x_{k-(m-1)}, \ldots, x_k$ have been computed. We then determine the polynomial $p_m(t)$ of degree $m-1$ such that

$$p_m(t_{k-i}) = f_{k-i} = f(t_{k-i}, x_{k-i}), \quad i = 0, 1, \ldots, m-1 \tag{12.53}$$

and compute x_{k+1} by integrating p_m instead of $f\big(t, x(t)\big)$,

$$x_{k+1} = x_k + \int_{t_k}^{t_{k+1}} p_m(t)\,dt.$$

Recall from chapter 9 that the interpolating polynomial may be written as

$$p_m(t) = \sum_{i=0}^{m-1} f_{k-i}\ell_{k-i}(t) \tag{12.54}$$

where ℓ_{k-i} is the polynomial of degree $m-1$ that has the value 1 at t_{k-i} and is 0 at all the other interpolation points $t_{k-m+1}, \ldots, t_{k-i-1}, t_{k-i+1}, \ldots, t_k$. We integrate (12.54) and obtain

$$\int_{t_k}^{t_{k+1}} p_m(t)\,dt = \sum_{i=0}^{m-1} f_{k-i} \int_{t_k}^{t_{k+1}} \ell_{k-i}(t)\,dt = h \sum_{i=0}^{m-1} c_{k-i}\, f_{k-i}$$

where

$$c_{k-i} = \frac{1}{h} \int_{t_k}^{t_{k+1}} \ell_{k-i}(x)\,dt.$$

The division by h has the effect that the coefficients are independent of h. The final formula for solving the differential equation becomes

$$x_{k+1} = x_k + h \sum_{i=0}^{m-1} c_{k-i} f_{k-i}. \tag{12.55}$$

The advantage of multi-step methods is that they achieve high accuracy but just require one new evaluation of f each time the solution is advanced one step. However, multi-step methods must be supplemented with alternative methods with the same accuracy during the first iterations as there are then not sufficiently many previously computed values.

12.6.5 Implicit methods

All the methods we have considered so far advance the solution via a formula which is based on computing an approximation at a new time step from approximations computed at previous time steps. This is not strictly necessary though. The simplest example is the *backward Euler method* given by

$$x_{k+1} = x_k + hf(t_{k+1}, x_{k+1}), \quad k = 1, \ldots, n. \tag{12.56}$$

Note that the value x_{k+1} to be computed appears on both sides of the equation which means that in general we are left with a nonlinear equation or *implicit equation* for x_{k+1}. To determine x_{k+1} this equation must be solved by some nonlinear equation solver like Newton's method.

Example 12.29. Suppose that $f(t, x) = t + \sin x$. In this case the implicit equation (12.56) becomes

$$x_{k+1} = x_k + h(t_{k+1} + \sin x_{k+1})$$

which can only be solved by a numerical method.

Another example is $x' = 1/(t + x)$. Then (12.56) becomes

$$x_{k+1} = x_k + \frac{h}{t_{k+1} + x_{k+1}}.$$

In this case we obtain a quadratic equation for x_{k+1},

$$x_{k+1}^2 - (x_k - t_{k+1})x_{k+1} - t_{k+1}x_k - h = 0.$$

This can be solved with some nonlinear equation solver or the standard formula for quadratic equations. ∎

The idea of including x_{k+1} in the estimate of itself can be used for a variety of methods. An alternative midpoint method is given by

$$x_{k+1} = x_k + \frac{h}{2}\big(f(t_k, x_k) + f(t_{k+1}, x_{k+1})\big),$$

and more generally x_{k+1} can be included on the right-hand side to yield implicit Runge-Kutta like methods. A very important class of methods is implicit multi-step methods where the degree of the interpolating polynomial in (12.53) is increased by one and the next point $\big(t_{k+1}, f(t_{k+1}, x_{k+1})\big)$ is included as an interpolation point. More specifically, if the interpolating polynomial is q_m, the interpolation conditions are taken to be

$$q_m(t_{k+1-i}) = f_{k+1-i} = f(t_{k+1-i}, x_{k+1-i}), \quad i = 0, 1, \ldots, m.$$

303

At t_{k+1} , the value x_{k+1} is unknown so the equation (12.55) is replaced by

$$x_{k+1} = x_k + hc_{k+1}f(t_{k+1}, x_{k+1}) + h\sum_{i=0}^{m-1} c_{k-i}f_{k-i} \qquad (12.57)$$

where $(c_{k+1-i})_{i=0}^{m}$ are coefficients that can be computed in the same way as indicated in section 12.6.4 (they will be different though, since the degree is different). This shows clearly the implicit nature of the method. The same idea can be used to adjust most other explicit methods as well.

It turns out that an implicit method has quite superior convergence properties and can achieve a certain accuracy with considerably larger time steps than a comparable explicit method. The obvious disadvantage of implicit methods is the need to solve a nonlinear equation at each time step. However, this disadvantage is not as bad as it may seem. Consider for example equation (12.57). If in some way we can guess a first approximation x_{k+1}^0 for x_{k+1}, we can insert this on the right and compute a hopefully better approximation x_{k+1}^1 as

$$x_{k+1}^1 = x_k + hc_{k+1}f(t_{k+1}, x_{k+1}^0) + h\sum_{i=0}^{m-1} c_{k-i}f_{k-i}.$$

But now we can compute a new approximation to x_{k+1}^2 by inserting x_{k+1}^1 on the right, and in this way we obtain a tailor-made numerical method for solving (12.57).

In practice the first approximation x_{k+1}^0 is obtained via some explicit numerical method like a suitable multi-step method. This is often referred to as a *predictor* and the formula (12.57) as the *corrector*; the combination is referred to as a *predictor-corrector method*. In many situations it turns out that it is sufficient to just use the corrector formula once.

12.7   Stability

An important input parameter for a differential equation, and therefore for any numerical method for finding an approximate solution, is the value x_0 that enters into the initial condition $x(a) = x_0$. In general, this is a real number that cannot be be represented exactly by floating point numbers, so an initial condition like $x_\epsilon(a) = x_0 + \epsilon$ will be used instead, where ϵ is some small number. This will obviously influence the computations in some way; the question is by how much?

12.7.1 Stability of a differential equation

Initially, we just focus on the differential equation and ignore effects introduced by particular numerical methods. A simple example will illustrate what can happen.

Example 12.30. Consider the differential equation

$$x' = \lambda\left(x - \sqrt{2}\right), \quad x(0) = \sqrt{2}$$

for t in some interval $[0, b]$ where $b > 0$. It is quite easy to see that the exact solution of this equation is

$$x(t) = \sqrt{2}. \tag{12.58}$$

On the other hand, if the initial condition is changed to $x_\epsilon(0) = \sqrt{2} + \epsilon$, where ϵ is a small number, the solution becomes

$$x_\epsilon(t) = \sqrt{2} + \epsilon e^{\lambda t}. \tag{12.59}$$

If we try to solve the equation numerically using floating point numbers, and commit no errors other than replacing $\sqrt{2}$ by the nearest floating point number, we end up with the solution given by (12.59) rather than the correct solution (12.58).

For small values of t, the solution given by (12.59) will be a good approximation to the correct solution. However, if $\lambda > 0$, the function $e^{\lambda t}$ grows very quickly with t even for moderate values of λ, so the second term in (12.59) will dominate. If for example $\lambda = 2$, then $e^{\lambda t} \approx 5 \times 10^{21}$ already for $t = 25$. If we use 64 bit floating point numbers we will have $\epsilon \approx 10^{-17}$ and therefore

$$x_\epsilon(25) \approx \sqrt{2} + 10^{-17} \times 5 \times 10^{21} \approx 5 \times 10^4$$

which is way off the correct value.

This kind of error is unavoidable whichever numerical method we choose to use. In practice we will also commit other errors which complicates matters further. ∎

Example 12.30 shows that it is possible for a simple differential equation to be highly sensitive to perturbations of the initial value. As we know, different initial values pick different solutions from the total family of solution curves. Therefore, if the solution curves separate more and more when t increases, the equation will be sensitive to perturbations of the initial values, whereas if the solution curves come closer together, the equation will not be sensitive to perturbations. This phenomenon is called *stability* of the equation, and it can be shown that stability can be measured by the size of the derivative of $f(t, x)$ with respect to x.
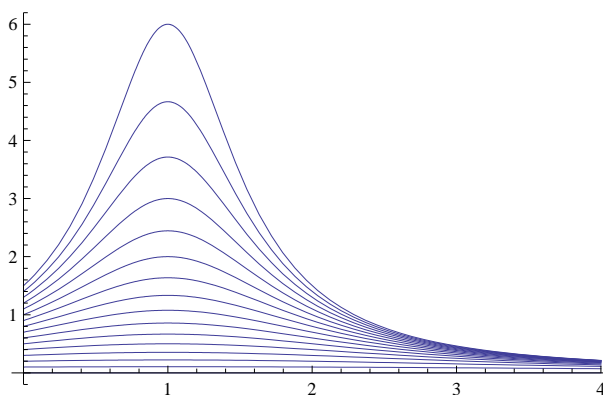
Figure 12.6. Solutions of the equation $x' = (1 - t)x$ with initial values $x(0) = i/10$ for $i = 0, \ldots, 15$.

> **Definition 12.31** (Stability of differential equation)**.** *The differential equation* $x' = f(t, x)$ *is said to be stable in an area where the derivative $f_x(t, x)$ is negative (the solution curves approach each other), while it is said to be unstable (the solution curves separate) if $f_x(t, x) > 0$. Here $f_x(t, x)$ denotes the derivative of f with respect to x.*

If we return to example 12.30, we see that the equation considered there is stable when $\lambda < 0$ and unstable when $\lambda > 0$. For us the main reason for studying stability is to understand why the computed solution of some equations may blow up and become completely wrong, like the one in (12.59). However, even if $\lambda > 0$, the instability will not be visible for small t. This is true in general: It takes some time for instability to develop, and for many unstable equations, the instability may not be very strong. Likewise, there may be equations where f_x is negative in certain areas, but very close to 0, which means that the effect of dampening the errors is not very pronounced. For this reason it often makes more sense to talk about *weakly unstable* or *weakly stable* equations.

Example 12.32. Consider the differential equation

$$y' = f(t, x) = (1 - t)x^2.$$

The solutions of this equation with initial values $x(0) = i/10$ for $i = 0, 1, \ldots, 15$ are shown in figure 12.6. If we differentiate f with respect to x we obtain

$$f_x(t, x) = (1 - t)x.$$

This equation is therefore unstable for $t < 1$ and stable for $t > 1$ which corresponds well with the plots in figure 12.6. However, the stability effects visible in

306

the plot are not extreme and it makes more sense to call this weak stability and instability. ∎

12.7.2 Stability of Euler's method

Stability is concerned with a differential equation's sensitivity to perturbations of the initial condition. Once we use a numerical method to solve the differential equation, there are additional factors that can lead to similar behaviour. The perturbation of the initial condition is usually insignificant compared to the error we commit when we step from one solution point to another with some approximate formula; and these errors accumulate as we step over the total solution interval $[a, b]$. The effect of all these errors is that we keep jumping from one solution curve to another, so we must expect the stability of the differential equation to be amplified further by a numerical method. It also seems inevitable that different numerical methods will behave differently with respect to stability since they use different approximations. However, no method can avoid instabilities in the differential equation itself.

We will only consider stability for Euler's method. The crucial relation is (12.15) which relates the global error $\epsilon_k = x(t_k) - x_k$ to the corresponding error at time t_{k+1}. The important question is whether the error is magnified or not as it is filtered through many time steps.

We rewrite (12.15) as

$$\epsilon_{k+1} = \left(1 + hL_k\right)\epsilon_k + G_k,$$

with $L_k = f_x(t_k, \theta_k)$ with θ_k some real number between x_k and $x(t_k)$, and $G_k = h^2 x''(\xi_k)/2$ with ξ_k some number between t_k and t_{k+1}. The decisive part of this relation is the factor that multiplies ϵ_k. We see this quite clearly if we unwrap the error as in (12.16),

$$
\begin{aligned}
\epsilon_{k+1} &= (1 + hL_k)\epsilon_k + G_k \\
&= (1 + hL_k)((1 + hL_{k-1})\epsilon_{k-1} + G_{k-1}) + G_k \\
&= \prod_{j=k-1}^{k} (1 + hL_j)\epsilon_{k-1} + (1 + hL_k)G_{k-1} + G_k \\
&= \prod_{j=k-1}^{k} (1 + hL_j)((1 + hL_{k-2})\epsilon_{k-2} + G_{k-2}) + (1 + hL_k)G_{k-1} + G_k \\
&= \prod_{j=k-2}^{k} (1 + hL_j)\epsilon_{k-2} + \prod_{j=k-1}^{k} (1 + hL_j)G_{k-2} + (1 + hL_k)G_{k-1} + G_k
\end{aligned}
$$

307

$$= \prod_{j=k-2}^{k} (1 + hL_j)\epsilon_{k-2} + \sum_{i=k-2}^{k} \prod_{j=i+1}^{k} (1 + hL_j)G_i$$

$$\vdots$$

$$= \prod_{j=0}^{k} (1 + hL_j)\epsilon_0 + \sum_{i=0}^{k} \prod_{j=i+1}^{k} (1 + hL_j)G_i.$$

This is a bit more complicated than what we did in (12.16) since we have taken neither absolute nor maximum values, as this would hide some of the information we are looking for. On the other hand, we have an identity which deserves to be recorded in a lemma.

Lemma 12.33. *The error in Euler's method for the equation $x' = f(t, x)$ at time t_{k+1} is given by*

$$\epsilon_{k+1} = \prod_{j=0}^{k} (1 + hL_j)\epsilon_0 + \sum_{i=0}^{k} \prod_{j=i+1}^{k} (1 + hL_j)G_i, \qquad (12.60)$$

for $k = 0, \ldots, n - 1$. Here $L_j = f_x(t_j, \theta_j)$ where θ_j is a number between x_j and $x(t_j)$, while $G_j = h^2 x''(\xi_j)/2$ with ξ_j some number between t_k and t_{k+1}. The number ϵ_0 is the error committed in implementing the initial condition $x(a) = x_0$.

We can now more easily judge what can go wrong with Euler's method. We see that the error in the initial condition, ϵ_0, is magnified by the factor $\prod_{j=0}^{k}(1 + hL_j)$. If each of the terms in this product has absolute value larger than 1, this magnification factor can become very large, similarly to what we saw in example 12.30. Equation 12.60 also shows that similar factors multiply the truncation errors (the remainders in the Taylor expansions) which are usually much larger than ϵ_0, so these may also be magnified in the same way.

What does this say about stability in Euler's method? If $|1 + hL_j| > 1$, i.e., if either $hL_j > 0$ or $hL_j < -2$ for all j, then the errors will be amplified and Euler's method will be unstable. On the other hand, if $|1 + hL_j| < 1$, i.e., if $-2 < hL_j < 0$, then the factors will dampen the error sources and Eulers's method will be stable.

This motivates a definition of stability for Euler's method.

> **Definition 12.34.** *Euler's method for the equation*
>
> $$x' = f(t, x), \quad x(a) = x_0,$$
>
> *is said to be stable in an area where $|1 + hf_x(t, x)| > 1$ and unstable in an area where $|1 + hf_x(t, x)| < 1$.*

We observe that Euler's method has no chance of being stable if the differential equation is unstable, i.e., if $f_x > 0$. On the other hand, it is not necessarily stable even if the differential equation is stable, that is if $f_x < 0$; we must then avoid $1 + hf_x$ becoming smaller than -1. This means that we must choose h so small that $-1 - hf_x(t, x) > -1$ or

$$h < \frac{2}{|f_x(t, x)|}$$

for all t and x. Otherwise, Euler's method will be unstable, although in many cases it is more correct to talk about *weak stability* or *weak instability*, just like for differential equations.

Example 12.35. The prototype of a stable differential equation is

$$x' = -\lambda x, \quad x(0) = 1$$

with the exact solution $x(t) = e^{-\lambda t}$ which approaches 0 quickly with increasing t. We will try and solve this with Euler's method when $\lambda = -10$, on the interval $[0, 2]$. In this case $f_x(t, x) = -10$ so the stability estimate demands that $|1 - 10h| < 1$ or $h < 1/5 = 0.2$. To avoid instability in Euler's method we therefore need to use at least 11 time steps on the interval $[0, 2]$. Figure 12.7 illustrates how Euler's method behaves. In figure (a) we have used a step length of $2/9$ which is just above the requirement for stability and we notice how the size of the computed solution grows with t, a clear sign of instability. In figure (b), the step length is chosen so that $|1 + hf_x(t, x)| = 1$ and we see that the computed solution does not converge to 0 as it should, but neither is the error amplified. In figure (c), the step length is just below the limit and the solution decreases, but rather slowly. Finally, in figure (d), the step size is well below the limit and the numerical solution behaves as it should. ∎

Stability of other numerical methods can be studied in much the same way as for Euler's method. It is essentially the derivative of the function Φ (see (12.36) and (12.46)) with respect to x that decides the stability of a given method. This is a vast area and the reader is referred to advanced books on numerical analysis to learn more.
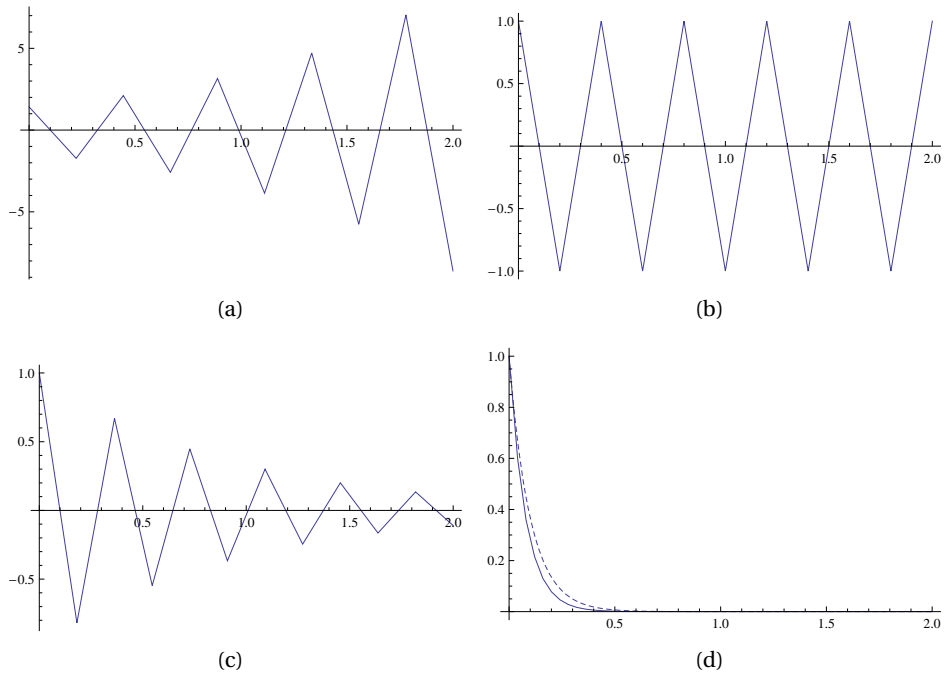
Figure 12.7. The plots illustrate the result of using Euler's method for solving the equation $x' = -10x$ with initial condition $x(0) = 1$. In (a) a step length of 2/9 was used, in (b) the step length was 2/10, and in (c) it was 2/11. The dashed curve in (d) shows the exact solution $x(t) = e^{-10t}$ while the solid curve shows the result produced by Euler's method with a step length of 1/25.

12.8 Systems of differential equations

So far we have focused on how to solve a single first order differential equation. In practice two or more such equations, coupled together, are necessary to model a problem, and perhaps even equations of higher order. In this section we are going to see how the methods we have developed above can easily be adapted to deal with both systems of equations and equations of higher order.

12.8.1 Vector notation and existence of solution

Many practical problems involve not one, but two or more differential equations. For example many processes evolve in three dimensional space, with separate differential equations in each space dimension.

Example 12.36. At the beginning of this chapter we saw that a vertically falling object subject to gravitation and friction can be modelled by the differential

310

equation

$$v' = g - \frac{c}{m} v^2, \tag{12.61}$$

where $v = v(t)$ is the speed at time t. How can an object that also has a horizontal speed be modelled? A classical example is that of throwing a ball. In the vertical direction, equation (12.61) is still valid, but since the y-axis points upwards, we change signs on the right-hand side and label the speed by a subscript 2 to indicate that this is movement along the y- (the second) axis,

$$v_2' = \frac{c}{m} v_2^2 - g.$$

In the x-direction a similar relation holds, except there is no gravity. If we assume that the positive x-axis is in the direction of the movement we therefore have

$$v_1' = -\frac{c}{m} v_1^2.$$

In total we have

$$v_1' = -\frac{c}{m} v_1^2, \qquad v_1(0) = v_{0_x}, \tag{12.62}$$

$$v_2' = \frac{c}{m} v_2^2 - g, \quad v_2(0) = v_{0_y}, \tag{12.63}$$

where v_{0_x} is the initial speed of the object in the x-direction and v_{0_y} is the initial speed of the object in the y-direction. If we introduce the vectors $\boldsymbol{v} = (v_1, v_2)$ and $\boldsymbol{f} = (f_1, f_2)$ where

$$f_1(t, \boldsymbol{v}) = f_1(t, v_1, v_2) = -\frac{c}{m} v_1^2,$$

$$f_2(t, \boldsymbol{v}) = f_2(t, v_1, v_2) = \frac{c}{m} v_2^2 - g,$$

and the initial vector $\boldsymbol{v}_0 = (v_{0_x}, v_{0_y})$, the equations (12.62)–(12.63) may be rewritten more compactly as

$$\boldsymbol{v}' = \boldsymbol{f}(t, \boldsymbol{v}), \quad \boldsymbol{v}(0) = \boldsymbol{v}_0.$$

Apart from the vector symbols, this is exactly the same equation as we have studied throughout this chapter. ∎

The equations in example 12.36 are quite specialised in that the time variable does not appear on the right, and the two equations are independent of each other. The next example is more general.

311

Example 12.37. Consider the three equations with initial conditions

$$x' = xy + \cos z, \qquad x(0) = x_0, \tag{12.64}$$

$$y' = 2 - t^2 + z^2 y, \quad x(0) = y_0, \tag{12.65}$$

$$z' = \sin t - x + y, \quad z(0) = z_0. \tag{12.66}$$

If we introduce the vectors $\boldsymbol{x} = (x, y, z)$, $\boldsymbol{x}_0 = (x_0, y_0, z_0)$, and the vector of functions $\boldsymbol{f}(t, \boldsymbol{x}) = \big(f_1(t, \boldsymbol{x}), f_2(t, \boldsymbol{x}), f_3(t, \boldsymbol{x})\big)$ defined by

$$x' = f_1(t, \boldsymbol{x}) = f_1(t, x, y, z) = xy + \cos z,$$

$$y' = f_2(t, \boldsymbol{x}) = f_2(t, x, y, z) = 2 - t^2 + z^2 y,$$

$$z' = f_3(t, \boldsymbol{x}) = f_3(t, x, y, z) = \sin t - x + y,$$

we can write (12.64)–(12.66) simply as

$$\boldsymbol{x}' = \boldsymbol{f}(t, \boldsymbol{x}), \quad \boldsymbol{x}(0) = \boldsymbol{x}_0. \quad \blacksquare$$

Examples 12.36–12.37 illustrate how vector notation may camouflage a system of differential equations as a single equation. This is helpful since it makes it quite obvious how the theory for scalar equations can be applied to systems of equations. Let us first be precise about what we mean with a system of differential equations.

Definition 12.38. *A system of M first order differential equations in M unknowns with corresponding initial conditions is given by a vector relation on the form*

$$\boldsymbol{x}' = \boldsymbol{f}(t, \boldsymbol{x}), \quad \boldsymbol{x}(0) = \boldsymbol{x}_0. \tag{12.67}$$

Here $\boldsymbol{x} = \boldsymbol{x}(t) = \big(x_1(t), \dots, x_M(t)\big)$ is a vector of M unknown scalar functions and $\boldsymbol{f}(t, \boldsymbol{x}) : \mathbb{R}^{M+1} \to \mathbb{R}^M$ is a vector function of the $M + 1$ variables t and $\boldsymbol{x} = (x_1, \dots, x_M)$, i.e.,

$$\boldsymbol{f}(t, \boldsymbol{x}) = \big(f_1(t, \boldsymbol{x}), \dots, f_M(t, \boldsymbol{x})\big).$$

The notation $\boldsymbol{x}'$ denotes the vector of derivatives of the components of $\boldsymbol{x}$ with respect to t,

$$\boldsymbol{x}' = \boldsymbol{x}'(t) = \big(x_1'(t), \dots, x_M'(t)\big).$$

It may be helpful to write out the vector equation (12.67) in detail,

$$x_1' = f_1(t, \boldsymbol{x}) = f_1(t, x_1, \dots, x_M), \qquad x_1(0) = x_{1,0}$$

$$\vdots$$

$$x_M' = f_M(t, \boldsymbol{x}) = f_M(t, x_1, \dots, x_M), \quad x_M(0) = x_{M,0}.$$

We see that both the examples above fit into this setting, with $M = 2$ for example 12.36 and $M = 3$ for example 12.37.

Before we start considering numerical solutions of systems of differential equations, we need to know that solutions exist. This is settled by the following theorem which is completely analogous to theorem 12.6.

Theorem 12.39. *Suppose that the function $f(t, x)$ and its first derivatives with respect to the components of x are continuous on the set*

$$\mathbb{A} = [\alpha, \beta] \times [\gamma_1, \delta_1] \times [\gamma_2, \delta_2] \times \cdots \times [\gamma_M, \delta_M]. \qquad (12.68)$$

If the point (a, x_0) lies in the interior of $\mathbb{A}$ there exists an $\tau > 0$ such that the differential equation

$$x' = f(t, x), \quad x(a) = x_0 \qquad (12.69)$$

has a unique solution on the interval $[a - \tau, a + \tau]$ which is contained in $[\alpha, \beta]$.

The set $\mathbb{A}$ defined in (12.68) may seem mysterious. It is just the collection of all points $(t, x_1, \ldots, x_M)$ such that $t \in [\alpha, \beta]$ and $x_i \in [\gamma_i, \delta_i]$ for $i = 1, \ldots, M$. The precise derivatives of the components of f that are required to be continuous are

$$
\begin{array}{cccc}
\dfrac{\partial f_1}{\partial x_1}, & \dfrac{\partial f_1}{\partial x_2}, & \cdots & \dfrac{\partial f_1}{\partial x_M}, \\[2mm]
\dfrac{\partial f_2}{\partial x_1}, & \dfrac{\partial f_2}{\partial x_2}, & \cdots & \dfrac{\partial f_2}{\partial x_M}, \\[2mm]
\vdots & \vdots & \ddots & \vdots \\[2mm]
\dfrac{\partial f_M}{\partial x_1}, & \dfrac{\partial f_M}{\partial x_2}, & \cdots & \dfrac{\partial f_M}{\partial x_M}.
\end{array}
$$

This may all seem complicated, but the conclusion is rather simple: the system of equations (12.69) has a solution near the initial value (t, x_0) provided all the component functions are reasonably well behaved near the point.

12.8.2 Numerical methods for systems of first order equations

There are very few analytic methods for solving systems of differential equations, so numerical methods are essential. It turns out that most of the methods for a single equation generalise to systems. A simple example illustrates the general principle

Example 12.40 (Euler's method for a system)**.** We consider the equations in example 12.37,

$$x' = f(t, x), \quad x(0) = x_0,$$

where

$$\boldsymbol{f}(t,\boldsymbol{x}) = \big(f_1(t,x_1,x_2,x_3), f_2(t,x_1,x_2,x_3), f_3(t,x_1,x_2,x_3)\big)$$
$$= (x_1 x_2 + \cos x_3, 2 - t^2 + x_3^2 x_2, \sin t - x_1 + x_2).$$

Euler's method is easily generalised to vector equations as

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + h\boldsymbol{f}(t_k, \boldsymbol{x}_k), \quad k = 0, 1, \dots, n-1. \tag{12.70}$$

If we write out the three components explicitly, this becomes

$$\left. \begin{aligned} x_1^{k+1} &= x_1^k + h f_1(t_k, x_1^k, x_2^k, x_3^k) = x_1^k + h\big(x_1^k x_2^k + \cos x_3^k\big), \\ x_2^{k+1} &= x_2^k + h f_2(t_k, x_1^k, x_2^k, x_3^k) = x_2^k + h\big(2 - t_k^2 + (x_3^k)^2 x_2^k\big), \\ x_3^{k+1} &= x_3^k + h f_3(t_k, x_1^k, x_2^k, x_3^k) = x_3^k + h\big(\sin t_k - x_1^k + x_2^k\big), \end{aligned} \right\} \tag{12.71}$$

for $k = 0, 1, \dots, n-1$, with the starting values (x_1^0, x_2^0, x_3^0) given by the initial condition. Although they look rather complicated, these formulas can be programmed quite easily. The trick is to make use of the vector notation in (12.70), since it nicely hides the details in (12.71). ∎

Example 12.40 illustrates Euler's method for a system of equations, and for most of the numerical methods we have encountered earlier in the chapter it is equally straightforward to generalise to systems of equations.

> **Observation 12.41** (Generalisation to systems). *Euler's method, Euler's midpoint method, and the Runge-Kutta methods all generalise naturally to systems of differential equations.*

For example the formula for advancing one time step with Euler's midpoint method becomes

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + h\boldsymbol{f}\big(t_k + h/2, \boldsymbol{x}_k + h\boldsymbol{f}(t_k, \boldsymbol{x}_k)/2\big),$$

while the fourth order Runge-Kutta method becomes

$$\begin{aligned} \boldsymbol{k}_0 &= \boldsymbol{f}(t_k, \boldsymbol{x}_k), \\ \boldsymbol{k}_1 &= \boldsymbol{f}(t_k + h/2, \boldsymbol{x}_k + h\boldsymbol{k}_0/2), \\ \boldsymbol{k}_2 &= \boldsymbol{f}(t_k + h/2, \boldsymbol{x}_k + h\boldsymbol{k}_1/2), \\ \boldsymbol{k}_3 &= \boldsymbol{f}(t_k + h, \boldsymbol{x}_k + h\boldsymbol{k}_2), \\ \boldsymbol{x}_{k+1} &= \boldsymbol{x}_k + \frac{h}{6}(\boldsymbol{k}_0 + \boldsymbol{k}_1 + \boldsymbol{k}_2 + \boldsymbol{k}_3). \end{aligned}$$

Systems of differential equations is an example where the general mathematical formulation is simpler than most concrete examples. In fact, if each component of these formulas are written out in detail, the details quickly become overwhelming, so it is important to stick with the vector notation. This also applies to implementation in a program: It is wise to use the vector formalism and mimic the mathematical formulation as closely as possible.

In principle the Taylor methods also generalise to systems of equations, but because of the need for manual differentiation of each component equation, the details swell up even more than for the other methods. Multi-step methods generalise nicely to systems as well. Implicit methods are not so easy to generalise though, since nonlinear solutions methods are considerably more complicated for systems than for scalar equations. However, a predictor-corrector approach works quite well for systems.

12.8.3 Higher order equations as systems of first order equations

Many practical modelling problems lead to systems of differential equations, and sometimes higher order equations are necessary. It turns out that these can be reduced to systems of first order equations as well.

Example 12.42. Consider the second order equation

$$x'' = t^2 + \sin(x + x'), \quad x(0) = 1, \quad x'(0) = 0. \tag{12.72}$$

This equation is nonlinear and cannot be solved with any of the standard analytical methods. If we introduce the new function $x_2 = x'$, we notice that $x_2' = x''$, so the differential equation can be written

$$x_2' = t^2 + \sin(x + x_2), \quad x(0) = 1, \quad x_2(0) = 0.$$

If we also rename x as $x_1 = x$ we see that the second order equation in (12.72) can be written as the system

$$x_1' = x_2, \qquad\qquad x_1(0) = 1, \tag{12.73}$$
$$x_2' = t^2 + \sin(x_1 + x_2), \quad x_2(0) = 0. \tag{12.74}$$

In other words, equation (12.72) can be written as the system (12.73)–(12.74). We also see that this system can be expressed as the single equation in (12.72), so the two equations (12.73)–(12.74) and the single equation (12.72) are in fact equivalent in the sense that a solution of one automatically gives a solution of the other. ∎

The technique used in example 12.42 works in general—a pth order equation can be rewritten as a system of p first order equations.

Theorem 12.43. *The pth order differential equation*

$$x^{(p)} = g\left(t, x, x', \ldots, x^{(p-1)}\right) \tag{12.75}$$

with initial conditions

$$x(a) = d_0, \; x'(a) = d_1, \ldots, \; x^{(p-2)}(0) = d_{p-2}, \; x^{(p-1)}(0) = d_{p-1} \tag{12.76}$$

is equivalent to the system of p equations in the p unknown functions x_1, x_2, ..., x_p,

$$
\begin{aligned}
x_1' &= x_2, & x_1(a) &= d_0, \\
x_2' &= x_3, & x_2(a) &= d_1, \\
&\;\;\vdots & & \\
x_{p-1}' &= x_p, & x_{p-1}(a) &= d_{p-2}, \\
x_p' &= g(t, x_1, x_2, \ldots, x_{p-1}), & x_p(a) &= d_{p-1},
\end{aligned}
\tag{12.77}
$$

in the sense that the component solution $x_1(t)$ of (12.77) agrees with the solution $x(t)$ of (12.75)–(12.76).

Proof. The idea of the proof is just like in example 12.42. From the first $p - 1$ relations in (12.77) we see that

$$x_2 = x_1', \quad x_3 = x_2' = x_1'', \quad \ldots, \quad x_p = x_{p-1}' = x_{p-2}'' = \cdots = x_1^{(p-1)}.$$

If we insert this in the last equation in (12.77) we obtain a pth order equation for x_1 that is identical to (12.75). In addition, the initial values in (12.77) translate into initial values for x_1 that are identical to (12.76) so x_1 must solve (12.75)–(12.76). Conversely, if x is a solution of (12.75)–(12.76) it is easy to see that the functions

$$x_1 = x, \quad x_2 = x', \quad x_3 = x'', \quad \ldots, \quad x_{p-1} = x^{(p-2)}, \quad x_p = x^{(p-1)}$$

solve the system (12.77). ∎

Theorem 12.43 shows that if we can solve systems of differential equations we can also solve single equations of order higher than one. We can also handle systems of higher order equations in this way.

Example 12.44 (System of higher order equations)**.** Consider the system of differential equations given by

$$
\begin{aligned}
x'' &= t + x' + y', & x(0) &= 1, & x'(0) &= 2, \\
y''' &= x'y'' + x, & y(0) &= -1, & y'(0) &= 1, & y''(0) &= 2.
\end{aligned}
$$

We introduce the new functions $x_1 = x$, $x_2 = x'$, $y_1 = y$, $y_2 = y'$, and $y_3 = y''$. Then the above system can be written as

$$
\begin{aligned}
x_1' &= x_2, & x_1(0) &= 1, \\
x_2' &= t + x_2 + y_2, & x_2(0) &= 2, \\
y_1' &= y_2, & y_1(0) &= -1, \\
y_2' &= y_3, & y_2(0) &= 1, \\
y_3' &= x_2 y_3 + x_1, & y_3(0) &= 2. \quad \blacksquare
\end{aligned}
$$

Example 12.44 illustrates how a system of higher order equations may be expressed as a system of first order equations. Perhaps not surprisingly, a general system of higher order equations can be converted to a system of first order equations. The main complication is in fact notation. We assume that we have r equations involving r unknown functions $x_1, \ldots, x_r$. Equation no. i expresses some derivative of x_i on the left in terms of derivatives of itself and the other functions on the right,

$$
x_i^{(p_i)} = g_i\left(t, x_1, x_1', \ldots, x_1^{(p_1-1)}, \ldots, x_r, x_r', \ldots, x_r^{(p_r-1)}\right), \quad i = 1, \ldots, r. \tag{12.78}
$$

In other words, the integer p_i denotes the derivative of x_i on the left in equation no. i, and it is assumed that in the other equations the highest derivative of x_i is $p_i - 1$ (this is not an essential restriction, see exercise 20).

To write the system (12.78) as a system of first order equations, we just follow the same strategy as in example 12.44: For each variable x_i, we introduce the p_i variables

$$
x_{i,1} = x_i, \quad x_{i,2} = x_i', \quad x_{i,3} = x_i'', \quad \ldots, \quad x_{i,p_i} = x_i^{(p_i-1)}.
$$

Equation no. i in (12.78) can then be replaced by the p_i first order equations

$$
\begin{aligned}
x_{i,1}' &= x_{i,2}, \\
x_{i,2}' &= x_{i,3}, \\
&\vdots \\
x_{i,p_i-1}' &= x_{i,p_i}, \\
x_{i,p_i}' &= g_i\left(t, x_{1,1}, \ldots, x_{1,p_1}, \ldots, x_{r,1}, \ldots, x_{r,p_r}\right)
\end{aligned}
$$

for $i = 1, \ldots, r$. We emphasise that the general procedure is exactly the same as the one used in example 12.44, it is just that the notation becomes rather heavy in the general case.

We record the conclusion in a non-technical theorem.

> **Theorem 12.45.** *A system of differential equations can always be written as a system of first order equations.*

12.9 Final comments

Our emphasis in this chapter has been to derive some of the best-known methods for numerical solution of first order ordinary differential equations, including a basic error analysis, and treatment of systems of equations. There are a number of additional issues we have not touched upon.

There are numerous other numerical methods in addition to the ones we have discussed here. The universal method that is optimal for all kinds of applications does not exist; you should choose the method that works best for your particular kind of application.

We have assumed that the step size h remains fixed during the solution process. This is convenient for introducing the methods, but usually too simple for solving realistic problems. A good method will use a small step size in areas where the solution changes quickly and longer step sizes in areas where the solution varies more slowly. A major challenge is therefore to detect, during the computations, how quickly the solution varies, or equivalently, how large the error is locally. If the error is large in an area, it means that the local step size needs to be reduced; it may even mean that another numerical method should be used in the area in question. This kind of monitoring of the error, coupled with local control of the step size and choice of method, is an important and challenging characteristic of modern software for solving differential equations. Methods like these are called *adaptive methods*.

We have provided a basic error analysis of the simplest methods, and this kind of analysis can be extended to a number of other methods without much change. The analysis accounts for the error committed by making use of certain mathematical approximations. In most cases this kind of error analysis is adequate, but in certain situations it may also be necessary to pay attention to the round-off error.

Exercises

12.1 Initial condition at the right end of $[a, b]$.

12.2 Compute numerical solutions to $x(1)$ for the equations below using two steps with Euler's method, the quadratic Taylor method and the quartic Taylor method. For comparison the correct solution to 14 decimal digits is given in each case.

a) $x' = t^5 + 4$, $\quad x(0) = 1$,
$x(1) = 31/6 \approx 5.166666666667$.

b) $x' = x + t$, $\quad x(0) = 1$,
$x(1) \approx 3.4365636569181$.

c) $x' = x + t^3 - 3(t^2 + 1) - \sin t + \cos t$, $\quad x(0) = 7$,
$x(1) \approx 13.714598298644$.

12.3 We are given the differential equation

$$x' = e^{-t^2}, \quad x(0) = 0.$$

Compute an estimate of $x(0.5)$ using each of the methods below, and find an upper bound on the absolute error in each case.

    a) Euler's method.

    b) The quadratic Taylor method.

    c) The cubic Taylor method.

12.4 Suppose we perform one step of Euler's method for the differential equation

$$x' = \sin x, \quad x(0) = 1.$$

Find an upper bound for absolute the error.

12.5 Estimate the error in exercise 2 (a) (using Euler's method). Hint: Use equation 12.15.

12.6 This exercise is based on example 12.36 in which we modelled the movement of a ball thrown through air with the equations

$$v_1' = -\frac{c}{m} v_1^2, \quad v_1(0) = v_{0_x},$$
$$v_2' = \frac{c}{m} v_2^2 - g, \quad v_2(0) = v_{0_y},$$

We now consider the launch of a rocket. In this case, the constants g and c will become complicated functions of the height y, and possibly also of x. We make the (rather unrealistic) assumption that

$$\frac{c}{m} = c_0 - ay$$

where c_0 is the air resistance constant at the surface of the earth and y is the height above the earth given in kilometers. We will also use the fact that gravity varies with the height according to the formula

$$g = \frac{g_0}{(y+r)^2},$$

where g_0 is the gravitational constant times the mass of the earth, and r is the radius of the earth. Finally, we use the facts that $x' = v_1$ and $y' = v_2$.

    a) Find the second order differential equation for the vertical motion (make sure that the positive direction is upwards).

    b) Rewrite the differential equation for the horisontal motion as a second order differential equation that depends on x, x', y and y'.

c) Rewrite the coupled second order equations from (a) and (b) as a system of four first order differential equations.

d) Optional: Use a numerical method to find a solution at $t = 1$ hour for the initial conditions $x(0) = y(0) = 0$, $x'(0) = 200$ km/h and $y'(0) = 300$ km/h. Use $a = 1.9 * 10^{-4} \frac{\text{Nh}^2}{\text{km}^3\text{kg}}$, $g_0 = 3.98 * 10^8 \frac{(\text{km})^2\text{m}}{\text{s}^2}$ and $c_0 = 0.19 \frac{\text{Nh}^2}{\text{km}^2\text{kg}}$. These units are not so important, but mean that distances can be measured in km and speeds in km/h.

12.7 Consider the first order differential equation

$$x' = x^2, \quad x(0) = 1.$$

a) Estimate $x(1)$ by using one step with Euler's method.

b) Estimate $x(1)$ by using one step with the quadratic Taylor method.

c) Estimate $x(1)$ by using one step with Euler's midpoint method.

d) Estimate $x(1)$ by using one step with the Runge Kutta fourth order method.

e) Estimate $x(1)$ by using two steps with the Runge Kutta fourth order method.

f) Optional: Write a computer program that implements one of the above mentioned methods and use it to estimate the value of $y(1)$ with 10, 100, 1000 and 10000 steps?

g) Do the estimates seem to converge?

h) Solve the equation analytically and explain your numerical results.

12.8 Solve the differential equation
$$x' + x\sin t = \sin t$$

and plot the solution on the interval $t \in [-2\pi, 2\pi]$ for the following initial values:

a) $x(0) = 1 - e$.

b) $x(4) = 1$.

c) $x(\pi/2) = 2$.

d) $x(-\pi/2) = 3$.

12.9 Rn-222 is a common radioactive isotope. It decays to 218-Po through α-decay with a half-life of 3.82 days. The average concentration is about 150 atoms per mL of air. Radon emanates naturally from the ground, and so is typically more abundant in cellars than in a sixth floor apartment. Certain rocks like granite emanates much more radon than other substances.

In this exercise we assume that we have collected air samples from different places, and these samples have been placed in special containers so that no new Rn-222 (or any other element) may enter the sample after the sampling has been completed. We now want to measure the Rn-222 abundance as a function of time, $f(t)$.

a) The abundance $x(t)$ of Rn-222 is governed the differential equation $x' = \lambda x$. Solve the differential equation analytically and determine λ from the half-life given above.

b) Make a plot of the solution for the first 10 days for the initial conditions $x(0) = 100$, 150, 200 and 300 atoms per mL.

c) The different initial conditions give rise to a family of functions. Do any of the functions cross each other? Can you find a reason why they do/do not?

d) The four initial conditions correspond to four different air samples. Two of them were taken from two different cellars, one was taken from an upstairs bedroom, and the fourth is an average control sample. Which is which?

12.10 In this problem we are going to use Euler's method to solve the differential equation you found in exercise 9 with the inital condition $x(0) = 300$ atoms per mL sample over a time period from 0 to 6 days.

a) Use 3 time steps and make a plot where the points (t_i, x_i) for each time step are marked. What is the relative error at each point? (Compare with the exact solution.)

b) For each point computed by Euler's method, there is an exact solution curve that passes through the point. Determine these solutions and draw them in the plot you made in (a).

c) Use Euler's midpoint method with 3 time steps to find the concentration of Rn-222 in the 300 atoms per mL sample after 6 days. Compare with the exact result, and your result from exercise 10. What are the relative errors at the computed points?

d) Repeat (a), but use the quadratic Taylor method instead.

12.11 In this exercise we are going to derive the quartic (degree four) Taylor method and use it to solve the equation for radioactive decay in exercise 9.

a) Derive the quartic Taylor method.

b) Use the quartic Taylor method to find the concentration of RN-222 in the 300 atoms per mL sample after 6 days using 3 time steps and compare your results with those produced by the quadratic Taylor method in exercise 10. How much has the solution improved (in terms of absolute and relative errors)?

c) How many time steps would you have to use in the two Taylor methods to achive a relative error smaller than 10^{-5}?

d) What order would the Taylor order have to be to make sure that the relative error is smaller than 10^{-5} with only 3 steps?

12.12 Write a program that implements Euler's method for first order differential equations on the form

$$x' = f(t, x), \quad x(a) = x_0,$$

on the interval $[a, b]$, with n time steps. You may assume that the function f and the numbers a, b, x_0, and n are given. Test the program with the equation $x' = x$ and $x(0) = 1$ on the interval $[0, 1]$. Plot the exact solution $y(x) = e^x$ alongside the approximation and experiment with different values of n.

12.13 In this problem we are going to solve the equation

$$x' = f(t, x) = -x \sin t + \sin t, \quad x(0) = 2 + e,$$

numerically on the interval $[0, 2\pi]$.

a) Use Euler's method with 1, 2, 5, and 10 steps and plot the results. How does the solution evolve with the number of steps?

b) Use Euler's mid-point method with 1 and 5 steps and plot the results.

c) Compare the results from Euler's mid-point method with those form Euler's method including the number of evaluations of f in each case. Which method seems to be best?

12.14 In this exercise we are going to solve the differential equation

$$x' = f(t, x) = t^2 + x^3 - x, \quad x(0) = 1 \tag{12.79}$$

numerically with the quadratic Taylor method.

a) Find a formula for $x''(t)$ by differentiating equation 12.79.

b) Use the quadratic Taylor method and your result from a) to find an approximation to $x(2)$ using 1, 2 and, 5 steps. .

c) Write a computer program that implements the quadratic Taylor method and uses it to find an approximation of $x(2)$ with 10, 100 and 1000 steps.

12.15 In this exercise we are going to derive the cubic Taylor method and use it for solving equation (12.79) in exercise 14.

a) Derive a general algorithm for the cubic Taylor method.

b) Find a formula for $x'''(t)$ by differentiating equation 12.79, and find and approximation to $x(2)$ using 1 time step with the cubic Taylor method. Repeat using 2 time steps.

c) How do the results from the cubic Taylor method compare with the results from the quadratic Taylor method obtained in exercise 14?

d) Implement the cubic Taylor method in a program and compute an approximation to $x(2)$ with 10, 100 and 1000 steps.

12.16 When investigating the stability of a numerical method it is common to apply the method to the model equation
$$x' = -\lambda x, \quad x(0) = 1$$

and check for which values of the step length h the solution blows up.

a) Apply Euler's method to the model equation and determine the range of h-values that for which the solution remains bounded.

b) Repeat (a) for Euler's midpoint method.

c) Repeat (a) for the second order Taylor method.

d) Repeat (a) for the fourth order Runge-Kutte method.

12.17 Radon-222 is actually an intermediate decay product of a decay chain from Uranium-238. In this chain there are 16 subsequent decays which takes 238-U into a stable lead isotope (206-Pb). In one part of this chain 214-Pb decays through β-decay to 214-Bi which then decays through another β-decay to 214-Po. The two decays have the respective halflifes of 26.8 minutes and 19.7 minutes.

Suppose that we start with a certain amount of 214-Pb atoms and 214-Bi atoms, we want to determine the amounts of 214-Pb and 214-Bi as functions of time.

a) Phrase the problem as a system of two coupled differential equations.

b) Solve the equations from (a) analytically.

322

c) Suppose that the inital amounts of lead and bismuth are 600 atoms and 10 atoms respectively. Find the solutions for these initial conditions and plot the two functions for the first 1.5 hours.

d) When is the amount of bismuth at its maximum?

e) Compute the number of lead and bismuth atoms after 1 hour with Euler's method. Choose the number of steps to use yourself.

f) Repeat (e), but use the fourth order Runge-Kutta method instead and the same number of steps as in (e).

12.18 Write the following differential equations as systems of first order equations. The unknowns x, y, and z are assumed to be functions of t.

a) $x'' + t^2 x' + 3x = 0$.

c) $y''(t) = 2(e^{2t} - y^2)^{1/2}$.

b) $mx'' = -k_s x - k_d x'$.

d) $2x'' - 5x' + x = 0$ with initial conditions $x(3) = 6$, $x'(3) = -1$.

12.19 Write the following systems of differential equations as systems of first order equations. The unknowns x, y, and z are assumed to be functions of t.

a)
$$y'' = y^2 - x + e^t,$$
$$x'' = y - x^2 - e^t.$$

c)
$$x'' = y'' x + (y')^2 x,$$
$$y'' = -y.$$

b)
$$x'' = 2y - 4t^2 x,$$
$$y'' = -2x - 2tx'.$$

d)
$$x''' = y'' x^2 - 3(y')^2 x,$$
$$y'' = t + x'.$$

12.20 Write the system

$$x'' = t + x + y',$$
$$y''' = x''' + y'',$$

as a system of 5 first order equations. Note that this system is not on the form (12.78) since x''' appears on the right in the second equation. Hint: You may need to differentiate one of the equations.

12.21 Solve the system
$$x'' = 2y - 4t^2 x, \quad x(0) = 1,$$
$$y'' = -2x - 2tx', \quad y(0) = 0,$$

numerically on the interval $[0,2]$. Try both Euler's method and Euler's mid-point method with two time steps and plot the results.

12.22 A block of mass m is attached to a horizontal spring. As long as the displacement x (measured in centimeters) from the equilibrium position of the spring is small, we can model the force as a constant times this displacement, i.e. $F = -kx$, where $k = 0.114$ N/cm is the spring constant. (This is Hooke's law). We assume the motion of the spring to be along the x-axis and the position of the centre of mass of the block at time t to be $x(t)$. We then

323

know that the acceleration is given by $a(t) = x''(t)$. Newton's second law applied to the spring now yields

$$mx''(t) = -kx(t). \tag{12.80}$$

Suppose that the block has mass $m = 0.25\,\text{kg}$ and that the spring starts from rest in a position $5.0\,\text{cm}$ from its equilibrium so $x(0) = 5.0\,\text{cm}$ and $x'(0) = 0.0\,\text{cm/s}$.

    a) Rewrite this second order differential equation (12.80) as a system of two coupled differential equations and solve the system analytically.

    b) Use the second order Runge-Kutta method to solve the set of differential equations in the domain $t \in [0, 1.5]$ seconds with 3 time steps, and plot the analytical and approximate numerical solutions together.

    c) Did your numerical method and the number of steps suffice to give a good approximation?

12.23 This is a continuation of exercise 22, and all the constants given in that problem will be reused here. We now consider the case of a vertical spring and denote the position of the block at time t by $y(t)$. This means that in addition to the spring force, gravity will also influence the problem. If we take the positive y-direction to be up, the force of gravity will be given by

$$F_g = -mg. \tag{12.81}$$

Applying Newton's second law we now obtain the differential equation

$$my''(t) = -ky(t) - mg. \tag{12.82}$$

The equilibrium position of the spring will now be slightly altered, but we assume that $y = 0$ corresponds to the horizontal spring equilibrium position.

    a) What is the new equilibrium position y_0?

    b) We let the spring start from rest $5.0\,\text{cm}$ above the new equilibrium, which means that we have $x(0) = 5.0\text{cm} + y_0$, $x'(0) = 0.0\,\text{cm/s}$. Rewrite the second order differential equation as a system of two first order ones and solve the new set of equations analytically.

    c) Choose a numerical method for solving the equations in the interval $t \in [0, 1.5]$ seconds. Choose a method and the number of time steps that you think should make the results good enough.

    d) Plot your new analytical and numerical solutions and compare with the graph from exercise 22. What are the differences? Did your choice of numerical method work better than the second order Runge-Kutta method in exercise 22?

Part IV

Functions of two variables

CHAPTER 13

Numerical differentiation of functions of two variables

So far, most of the functions we have encountered have only depended on one variable, but both within mathematics and in applications there is often a need for functions of several variables. In this chapter we will deduce methods for numerical differentiation of functions of two variables. The methods are simple extensions of the numerical differentiation methods for functions of one variable.

13.1 Functions of two variables

In this section we will review some basic results on functions of two variables, in particular the definition of partial and directional derivatives. For proofs, the reader is referred to a suitable calculus book.

13.1.1 Basic definitions

Functions of two variables are natural generalisations of functions of one variable that act on pairs of numbers rather than a single number. We assume that you are familiar with their basic properties already, but we repeat the definition and some basic notation.

Definition 13.1 (Function of two variables)**.** *A (scalar) function f of two variables is a rule that to a pair of numbers (x, y) assigns a number $f(x, y)$.*
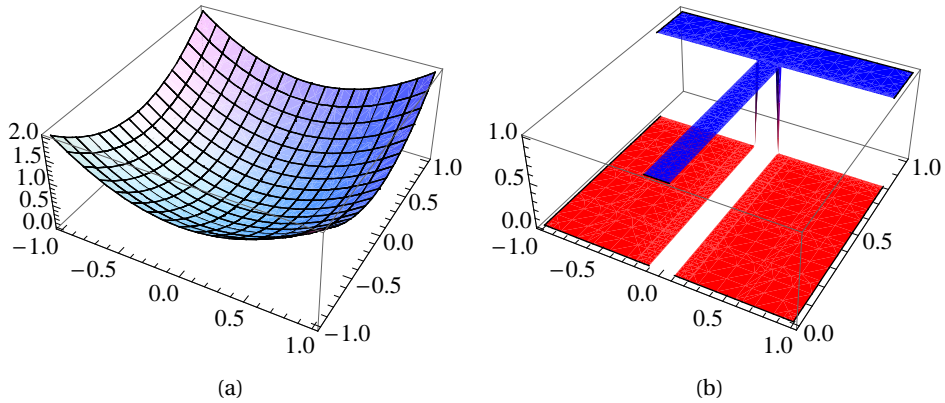
(a)   (b)

Figure 13.1. The plot in (a) shows the function $f(x, y) = x^2 + y^2$ with x and y varying in the interval $[-1, 1]$. The function in (b) is defined by the rule that $f(x, y) = 0$ except in a small area around the y-axis and the line $y = 1$, where the value is $f(x, y) = 1$.

The obvious interpretation is that $f(x, y)$ gives the height above the point in the plane given by (x, y). This interpretation lets us plot functions of two variables, see figure 13.1.

The rule f can be given by a formula like $f(x, y) = x^2 + y^2$, but this is not necessary, we just need to be able to determine $f(x, y)$ from x and y. In figure 13.1 the function in (a) is given by a formula, while the function in (b) is given by the rule

$$f(x, y) = \begin{cases} 1, & \text{if } |x| \le 0.1. \text{ and } 0 \le y \le 1; \\ 1, & \text{if } |y - 1| \le 0.1 \text{ and } -1 \le x \le 1; \\ 0, & \text{otherwise.} \end{cases}$$

We will sometimes use vector notation and refer to (x, y) as the point $\boldsymbol{x}$; then $f(x, y)$ can be written simply as $f(\boldsymbol{x})$. There is also convenient notation for a set of pairs of numbers that are assembled from two intervals.

Notation 13.2. *Let the two sets of numbers $\mathbb{A}$ and $\mathbb{B}$ be given. The set of all pairs of numbers from $\mathbb{A}$ and $\mathbb{B}$ is denoted $\mathbb{A} \times \mathbb{B}$,*

$$\mathbb{A} \times \mathbb{B} = \big\{(a, b) \mid a \in \mathbb{A} \text{ and } b \in \mathbb{B}\big\}.$$

The set $\mathbb{A} \times \mathbb{A}$ is denoted $\mathbb{A}^2$.

The most common set of pairs of numbers is $\mathbb{R}^2$, the set of all pairs of real numbers.

To define differentiation we need the concept of an interior point of a set. This is defined in terms of small discs.

> **Notation 13.3.** *The disc with radius r and centre $\boldsymbol{x} \in \mathbb{R}^2$ is denoted $B(\boldsymbol{x}; r)$. A point $\boldsymbol{x}$ in a subset $\mathbb{A}$ of $\mathbb{R}^2$ is called an* interior point *of $\mathbb{A}$ if there is a real number $\epsilon > 0$ such that the disc $B(\boldsymbol{x}; \epsilon)$ lies completely in $\mathbb{A}$. The disc $B(\boldsymbol{x}; \epsilon)$ is called a* neighbourhood *of $\boldsymbol{x}$.*

More informally, an interior point of $\mathbb{A}$ is a point which is completely surrounded by points from $\mathbb{A}$.

13.1.2 Differentiation

Differentiation generalises to functions of two variables in a simple way: We keep one variable fixed and differentiate the resulting function as a function of one variable.

> **Definition 13.4** (Partial derivatives)**.** *Let f be a function defined on a set $\mathbb{A} \subseteq \mathbb{R}^2$. The* partial derivatives *of f at an interior point $(a, b) \in \mathbb{A}$ are given by*
>
> $$\frac{\partial f}{\partial x}(a, b) = \lim_{h \to 0} \frac{f(a+h, b) - f(a, b)}{h},$$
> $$\frac{\partial f}{\partial y}(a, b) = \lim_{h \to 0} \frac{f(a, b+h) - f(a, b)}{h}.$$

From the definition we see that the partial derivative $\partial f / \partial x$ is obtained by fixing $y = b$ and differentiating the function $g_1(x) = f(x, b)$ at $x = a$. Similarly, the partial derivative with respect to y is obtained by fixing $x = a$ and differentiating the function $g_2(y) = f(a, y)$ at $y = b$.

Geometrically, the partial derivatives give the slope of f at (a, b) in the directions parallel to the two coordinate axes. The *directional derivative* gives the slope in a general direction.

> **Definition 13.5.** *Suppose the function f is defined on the set $\mathbb{A} \subseteq \mathbb{R}^2$ and that $\boldsymbol{a}$ is an interior point of $\mathbb{A}$. The* directional derivative *at $\boldsymbol{a}$ in the direction $\boldsymbol{r}$ is given by the limit*
> $$f'(\boldsymbol{a}; \boldsymbol{r}) = \lim_{h \to 0} \frac{f(\boldsymbol{a} + h\boldsymbol{r}) - f(\boldsymbol{a})}{h},$$
> *provided the limit exists.*

It turns out that for reasonable functions, the directional derivative can be computed in terms of partial derivatives.

Theorem 13.6. *Suppose the function is defined on the set $\mathbb{A} \subseteq \mathbb{R}^2$ and that $\boldsymbol{a}$ is an interior point of $\mathbb{A}$. If the two partial derivatives $\partial f / \partial x$ and $\partial f / \partial y$ exist in a neighbourhood of $\boldsymbol{a}$ and are continuous at $\boldsymbol{a}$, then the directional derivative $f'(\boldsymbol{a}; \boldsymbol{r})$ exists for all directions $\boldsymbol{r} = (r_1, r_2)$ and*

$$f'(\boldsymbol{a}; \boldsymbol{r}) = r_1 \frac{\partial f}{\partial x}(\boldsymbol{a}) + r_2 \frac{\partial f}{\partial y}(\boldsymbol{a}).$$

The conditions in theorem 13.6 are not very strict, but should be kept in mind. In particular you should be on guard when you need to compute directional derivatives near points where the partial derivatives do not exist.

If we consider a function like $f(x, y) = x^3 y + x^2 y^2$, the partial derivatives are $\partial f / \partial x = 3x^2 y + 2xy^2$ and $\partial f / \partial y = x^3 + 2x^2 y$. Each of these can of course be differentiated again,

$$\frac{\partial^2 f}{\partial x^2} = 6xy + 2y^2, \qquad \frac{\partial^2 f}{\partial y \partial x} = \frac{\partial}{\partial y}\left(\frac{\partial f}{\partial x}\right) = 3x^2 + 4xy,$$

$$\frac{\partial^2 f}{\partial y^2} = 2x^2, \qquad \frac{\partial^2 f}{\partial x \partial y} = \frac{\partial}{\partial x}\left(\frac{\partial f}{\partial y}\right) = 3x^2 y + 4xy.$$

We notice that the two mixed derivatives are equal. In general the derivatives

$$\frac{\partial^2 f}{\partial x \partial y}(\boldsymbol{a}), \qquad \frac{\partial^2 f}{\partial y \partial x}(\boldsymbol{a})$$

are equal if they both exist in a neighbourhood of $\boldsymbol{a}$ and are continuous at $\boldsymbol{a}$. All the functions we consider here have mixed derivatives that are equal. We can of course consider partial derivatives of any order.

Notation 13.7 (Higher order derivatives)**.** *The expression*

$$\frac{\partial^{n+m} f}{\partial x^n \partial y^m}$$

denotes the result of differentiating f, first m times with respect to y, and then differentiating the result n times with respect to x.
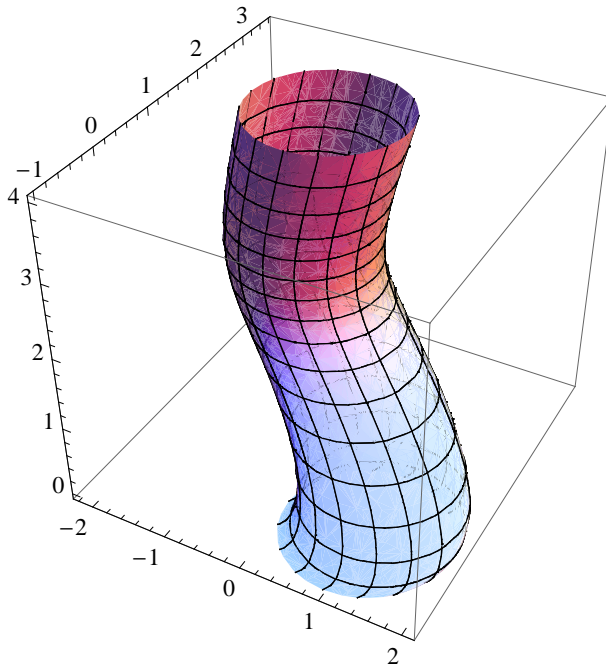
Figure 13.2. An example of a parametric surface.

13.1.3 Vector functions of several variables

The theory of functions of two variables extends nicely to functions of an arbitrary number of variables and functions where the scalar function value is replaced by a vector. We are only going to define these functions, but the whole theory of differentiation works in this more general setting.

Definition 13.8 (General functions)**.** *A function $\boldsymbol{f} : \mathbb{R}^n \mapsto \mathbb{R}^m$ is a rule that to n numbers $\boldsymbol{x} = (x_1, \ldots, x_n)$ assigns m numbers $\boldsymbol{f}(\boldsymbol{x}) = \big(f_1(\boldsymbol{x}), \ldots, f_m(\boldsymbol{x})\big).$*

Apart from the case $n = 2$, $m = 1$ which we considered above, we are interested in the case $n = 2$, $m = 3$.

Definition 13.9. *A function from $\boldsymbol{f} : \mathbb{R}^2 \mapsto \mathbb{R}^3$ is called a* parametric surface.

An example of a parametric surface is shown in figure 13.2. Parametric surfaces can take on almost any shape and are therefore used for representing ge-

ometric form in computer programs for geometric design. These kinds of programs are used for designing cars, aircrafts and other industrial objects as well as the 3D objects and characters in animated movies.

13.2 Numerical differentiation

The reason that we may want to compute derivatives numerically are the same for functions of two variables as for functions of one variable: The function may only be known via some procedure or computer program that can compute function values.

Theorem 13.6 shows that we can compute directional derivatives very easily as long as we can compute partial derivatives. The basic problem in numerical differentiation is therefore to find numerical approximations to the partial derivatives. Since only one variable varies in the definition of a first-order partial derivative, we can actually use the approximations that we obtained for functions of one variable. The simplest approximation is the following.

Proposition 13.10. *Let f be a function defined on a set $\mathbb{A} \subseteq \mathbb{R}^2$ and suppose that the points (a, b), $(a + rh_1, b)$ and $(a, b + rh_2)$ all lie in $\mathbb{A}$ for any $r \in [0, 1]$. Then the two partial derivatives $\partial f / \partial x$ and $\partial f / \partial y$ can be approximated by*

$$\frac{\partial f}{\partial x}(a, b) \approx \frac{f(a + h_1, b) - f(a, b)}{h_1},$$

$$\frac{\partial f}{\partial y}(a, b) \approx \frac{f(a, b + h_2) - f(a, b)}{h_2}.$$

The errors in the two estimates are

$$\frac{\partial f}{\partial x}(a, b) - \frac{f(a + h_1, b) - f(a, b)}{h_1} = \frac{h_1}{2} \frac{\partial^2 f}{\partial x^2}(c_1, b), \tag{13.1}$$

$$\frac{\partial f}{\partial y}(a, b) - \frac{f(a, b + h_2) - f(a, b)}{h_2} = \frac{h_2}{2} \frac{\partial^2 f}{\partial y^2}(a, c_2), \tag{13.2}$$

where c_1 is a number in $(a, a + h_1)$ and c_2 is a number in $(a, a + h_2)$.

Proof. We will just consider the first approximation. For this we define the function $g(x) = f(x, b)$. From 'Setning 9.15' in the Norwegian notes we know that

$$g'(x) = \frac{g(a + h_1) - g(a)}{h_1} + \frac{h_1}{2} g''(c_1)$$

where c_1 is a number in the interval $(a, a + h_1)$. From this the relation (13.1) follows. ∎

The other approximations to the derivatives in chapter 9 of the Norwegian notes lead directly to approximations of partial derivatives that are not mixed. For example we have

$$\frac{\partial f}{\partial x} = \frac{f(a+h,b) - f(a-h,b)}{2h} + \frac{h^2}{6}\frac{\partial^3 f}{\partial x^3}(c,b) \qquad (13.3)$$

where $c \in (a-h, a+h)$. A common approximation of a second derivative is

$$\frac{\partial^2 f}{\partial x^2} \approx \frac{-f(a-h,b) + 2f(a,b) - f(a+h,b)}{h^2},$$

with error bounded by

$$\frac{h^2}{12} \max_{z \in (a-h,a+h)} \left| \frac{\partial^4 f}{\partial x^4}(z,b) \right|,$$

see exercise 9.11 in the Norwegian notes. These approximations of course work equally well for non-mixed derivatives with respect to y.

Approximation of mixed derivatives requires that we use estimates for the derivatives both in the x- and y-directions. This makes it more difficult to keep track of the error. In fact, the easiest way to estimate the error is with the help of Taylor polynomials with remainders for functions of two variables. However, this is beyond the scope of these notes.

Let us consider an example of how an approximation to a mixed derivative can be deduced.

Example 13.11. Let us consider the simplest mixed derivative,

$$\frac{\partial^2 f}{\partial x \partial y}(a,b).$$

If we set

$$g(a) = \frac{\partial f}{\partial y}(a,b) \qquad (13.4)$$

we can use the approximation

$$g'(a) \approx \frac{g(a+h_1) - g(a-h_1)}{2h_1}.$$

If we insert (13.4) in this approximation we obtain

$$\frac{\partial^2 f}{\partial x \partial y}(a,b) \approx \frac{\frac{\partial f}{\partial y}(a+h_1,b) - \frac{\partial f}{\partial y}(a-h_1,b)}{2h_1}. \qquad (13.5)$$

Now we can use the same kind of approximation for the two first-order partial derivatives in (13.5),

$$\frac{\partial f}{\partial y}(a + h_1, b) \approx \frac{f(a + h_1, b + h_2) - f(a + h_1, b - h_2)}{2h_2},$$

$$\frac{\partial f}{\partial y}(a - h_1, b) \approx \frac{f(a - h_1, b + h_2) - f(a - h_1, b - h_2)}{2h_2}.$$

If we insert these expressions in (13.5) we obtain the final approximation

$$\frac{\partial^2 f}{\partial x \partial y}(a, b) \approx$$
$$\frac{f(a + h_1, b + h_2) - f(a + h_1, b - h_2) - f(a - h_1, b + h_2) + f(a - h_1, b - h_2)}{4h_1 h_2}.$$

If we introduce the notation

$$f(a - h_1, b - h_2) = f_{-1,-1}, \qquad f(a + h_1, b - h_2) = f_{1,-1},$$
$$f(a - h_1, b + h_2) = f_{-1,1}, \qquad f(a + h_1, b + h_2) = f_{1,1}, \tag{13.6}$$

we can write the approximation more compactly as

$$\frac{\partial^2 f}{\partial x \partial y}(a, b) \approx \frac{f_{1,1} - f_{1,-1} - f_{-1,1} + f_{-1,-1}}{4h_1 h_2}.$$

These approximations require f to be a 'nice' function. A sufficient condition is that all partial derivatives up to order four are continuous in a disc that contains the rectangle with corners $(a - h_1, b - h_2)$ and $(a + h_1, b + h_2)$. ∎

We record the approximation in example 13.11 in a proposition. We do not have the right tools to estimate the error, but just indicate how it behaves.

Proposition 13.12 (Approximation of a mixed derivative). *Suppose that f has continuous derivatives up to order four in a disc that contains the rectangle with corners $(a - h_1, b - h_2)$ and $(a + h_1, b + h_2)$. Then the mixed second derivative of f can be approximated by*

$$\frac{\partial^2 f}{\partial x \partial y}(a, b) \approx \frac{f_{1,1} - f_{1,-1} - f_{-1,1} + f_{-1,-1}}{4h_1 h_2}, \tag{13.7}$$

where the notation is defined in (13.6). The error is proportional to $h_1^2 h_2^2$.
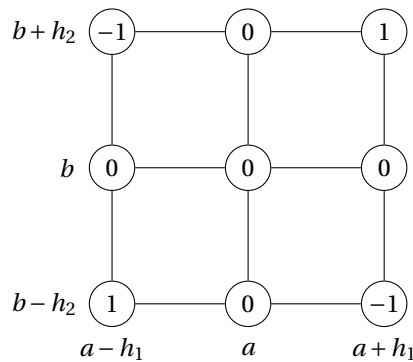
Figure 13.3. The weights involved in computing the mixed second derivative with the approximation in example 13.11. This kind of figure is referred to as the *computational molecule* of the approximation.
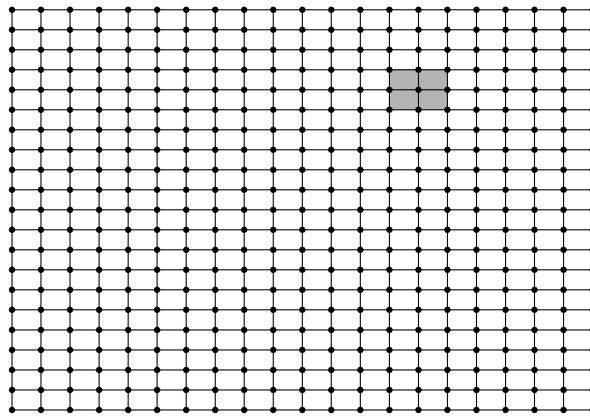

Figure 13.4. Numerical approximations to partial derivatives are often computed at all points of a grid like the one shown here by sliding around the grid a computational molecule like the one in figure 13.3.

Numerical approximations of other mixed partial derivatives can be derived with the same technique as in example 13.11, see exercise 1.

A formula like (13.7) is often visualised with a drawing like the one in figure 13.3 which is called a *computational molecule*. The arguments of the function values involved in the approximation are placed in a rectangular grid together with the corresponding coefficients of the function values. More complicated approximations will usually be based on additional values and involve more complicated coefficients.

Approximations to derivatives are usually computed at many points, and often the points form a rectangular grid as in figure 13.4. The computations can be performed by moving the computational molecule of the approximation across

the grid and computing the approximation at each point, as indicated by the grey area in figure 13.4.

Exercises

13.1 In this exercise we are going to derive approximations to mixed derivatives.

    a) Use the approximation $g'(a) = \big(g(a+h) - g(a-h)\big)/(2h)$ repeatedly as in example 13.11 and deduce the approximation

$$\frac{\partial^3 f}{\partial x^2 \partial y} \approx \frac{f_{2,1} - 2f_{0,1} + f_{-2,1} - f_{2,-1} + 2f_{0,-1} - f_{-2,-1}}{8h_1^2 h_2}.$$

    Hint: Use the approximation in (13.7).

    b) Use the same technique as in (a) and deduce the approximation

$$\frac{\partial^4 f}{\partial x^2 \partial y^2} \approx$$

$$\frac{f_{2,2} - 2f_{0,2} + f_{-2,2} - 2f_{2,0} + 4f_{0,0} - 2f_{-2,0} + f_{2,-2} - 2f_{0,-2} + f_{-2,-2}}{16h_1^2 h_2^2}.$$

    Hint: Use the approximation in (a) as a starting point.

13.2 Determine approximations to the two mixed derivatives

$$\frac{\partial^3 f}{\partial x^2 \partial y}, \quad \frac{\partial^4 f}{\partial x^2 \partial y^2},$$

in exercise 1, but use the approximation $g'(a) = \big(g(a+h) - g(a)\big)/h$ at every stage.

CHAPTER 14

Digital images
and image formats

An important type of digital media is images, and in this chapter we are going to review how images are represented and how they can be manipulated with simple mathematics. This is useful general knowledge for anyone who has a digital camera and a computer, but for many scientists, it is an essential tool. In astrophysics data from both satellites and distant stars and galaxies is collected in the form of images, and information extracted from the images with advanced image processing techniques. Medical imaging makes it possible to gather different kinds of information in the form of images, even from the inside of the body. By analysing these images it is possible to discover tumours and other disorders.

14.1   What is an image?

Before we do computations with images, it is helpful to be clear about what an image really is. Images cannot be perceived unless there is some light present, so we first review superficially what light is.

14.1.1   Light

> **Fact 14.1** (What is light?)**.** *Light is electromagnetic radiation with wavelengths in the range 400–700 nm (1 nm is 10^{-9} m): Violet has wavelength 400 nm and red has wavelength 700 nm. White light contains roughly equal amounts of all wave lengths.*

Other examples of electromagnetic radiation are gamma radiation, ultraviolet and infrared radiation and radio waves, and all electromagnetic radiation travel at the speed of light (3×10^8 m/s). Electromagnetic radiation consists of waves and may be reflected and refracted, just like sound waves (but sound waves are not electromagnetic waves).

We can only see objects that emit light, and there are two ways that this can happen. The object can emit light itself, like a lamp or a computer monitor, or it reflects light that falls on it. An object that reflects light usually absorbs light as well. If we perceive the object as red it means that the object absorbs all light except red, which is reflected. An object that emits light is different; if it is to be perceived as being red it must emit only red light.

14.1.2 Digital output media

Our focus will be on objects that emit light, for example a computer display. A computer monitor consists of a rectangular array of small dots which emit light. In most technologies, each dot is really three smaller dots, and each of these smaller dots emit red, green and blue light. If the amounts of red, green and blue is varied, our brain merges the light from the three small light sources and perceives light of different colours. In this way the colour at each set of three dots can be controlled, and a colour image can be built from the total number of dots.

It is important to realise that it is possible to generate most, but not all, colours by mixing red, green and blue. In addition, different computer monitors use slightly different red, green and blue colours, and unless this is taken into consideration, colours will look different on the two monitors. This also means that some colours that can be displayed on one monitor may not be displayable on a different monitor.

Printers use the same principle of building an image from small dots. On most printers however, the small dots do not consist of smaller dots of different colours. Instead as many as 7–8 different inks (or similar substances) are mixed to the right colour. This makes it possible to produce a wide range of colours, but not all, and the problem of matching a colour from another device like a monitor is at least as difficult as matching different colours across different monitors.

Video projectors builds an image that is projected onto a wall. The final image is therefore a reflected image and it is important that the surface is white so that it reflects all colours equally.

The quality of a device is closely linked to the density of the dots.

Fact 14.2 (Resolution)**.** *The resolution of a medium is the number of dots per inch (dpi). The number of dots per inch for monitors is usually in the range 70–120, while for printers it is in the range 150–4800 dpi. The horizontal and vertical densities may be different. On a monitor the dots are usually referred to as* pixels *(picture elements).*

14.1.3   Digital input media

The two most common ways to acquire digital images is with a digital camera or a scanner. A scanner essentially takes a photo of a document in the form of a rectangular array of (possibly coloured) dots. As for printers, an important measure of quality is the number of dots per inch.

Fact 14.3. *The resolution of a scanner usually varies in the range 75 dpi to 9600 dpi, and the colour is represented with up to 48 bits per dot.*

For digital cameras it does not make sense to measure the resolution in dots per inch, as this depends on how the image is printed (its size). Instead the resolution is measured in the number of dots recorded.

Fact 14.4. *The number of pixels recorded by a digital camera usually varies in the range* 320×240 *to* 6000×4000 *with 24 bits of colour information per pixel. The total number of pixels varies in the range 76 800 to 24 000 000 (0.077 megapixels to 24 megapixels).*

For scanners and cameras it is easy to think that the more dots (pixels), the better the quality. Although there is some truth to this, there are many other factors that influence the quality. The main problem is that the measured colour information is very easily polluted by noise. And of course high resolution also means that the resulting files become very big; an uncompressed 6000×4000 image produces a 72 MB file. The advantage of high resolution is that you can magnify the image considerably and still maintain reasonable quality.

14.1.4   Definition of digital image

We have already talked about digital images, but we have not yet been precise about what it is. From a mathematical point of view, an image is quite simple.

339

(a)	(b)	(c)

Figure 14.1. Different version of the same image; black and white (a), grey-level (b), and colour (c).

Fact 14.5 (Digital image). *A digital image P is a rectangular array of intensity values $\{p_{i,j}\}_{i,j=1}^{m,n}$. For grey-level images, the value $p_{i,j}$ is a single number, while for colour images each $p_{i,j}$ is a vector of three or more values. If the image is recorded in the rgb-model, each $p_{i,j}$ is a vector of three values,*

$$p_{i,j} = (r_{i,j}, g_{i,j}, b_{i,j}),$$

that denote the amount of red, green and blue at the point (i, j).

The value $p_{i,j}$ gives the colour information at the point (i, j). It is important to remember that there are many formats for this. The simplest case is plain black and white images in which case $p_{i,j}$ is either 0 or 1. For grey-level images the intensities are usually integers in the range 0–255. However, we will assume that the intensities vary in the interval $[0,1]$, as this sometimes simplifies the form of some mathematical functions. For colour images there are many different formats, but we will just consider the rgb-format mentioned in the fact box. Usually the three components are given as integers in the range 0–255, but as for grey-level images, we will assume that they are real numbers in the interval $[0, 1]$ (the conversion between the two ranges is straightforward, see section 14.2.3 below). Figure 14.1 shows an image in different formats.

Fact 14.6. *In these notes the intensity values $p_{i,j}$ are assumed to be real numbers in the interval $[0,1]$. For colour images, each of the red, green, and blue*
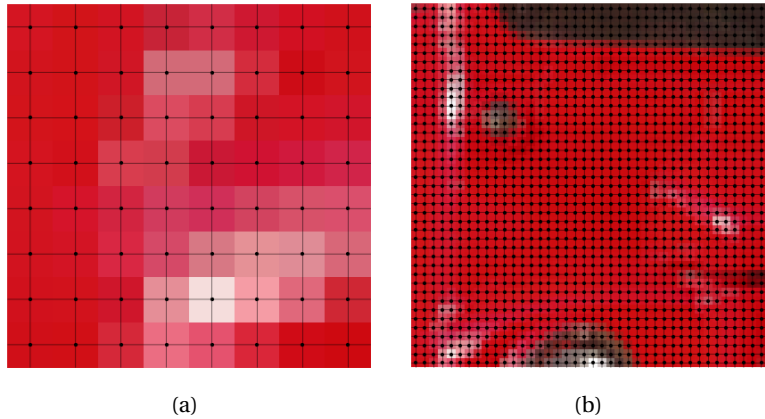
340

| (a) | (b) |

Figure 14.2. Two excerpt of the colour image in figure 14.1. The dots indicate the position of the points (i, j).

intensity values are assumed to be real numbers in $[0, 1]$.

If we magnify a small part of the colour image in figure 14.1, we obtain the image in figure 14.2 (the black lines and dots have been added). A we can see, the pixels have been magnified to big squares. This is a standard representation used by many programs — the actual shape of the pixels will depend on the output medium. Nevertheless, we will consider the pixels to be square, with integer coordinates at their centres, as indicated by the grids in figure 14.2.

Fact 14.7 (Shape of pixel). *The pixels of an image are assumed to be square with sides of length one, with the pixel with value $p_{i,j}$ centred at the point (i, j).*

14.1.5   Images as surfaces

Recall from chapter 13 that a function $f : \mathbb{R}^2 \mapsto \mathbb{R}$ can be visualised as a surface in space. A grey-level image is almost on this form. If we define the set of integer pairs by

$$\mathbb{Z}_{m,n} = \big\{ (i, j) \mid 1 \le i \le m \text{ and } 1 \le j \le n \big\},$$

we can consider a grey-level image as a function $P : \mathbb{Z}_{m,n} \mapsto [0, 1]$. In other words, we may consider an image to be a sampled version of a surface with the intensity value denoting the height above the (x, y)-plane, see figure 14.3.

Figure 14.3. The grey-level image in figure 14.1 plotted as a surface. The height above the (x, y)-plane is given by the intensity value.

> **Fact 14.8** (Grey-level image as a surface). *Let $P = (p)_{i,j=1}^{m,n}$ be a grey-level image. Then P can be considered a sampled version of the piecewise constant surface*
>
> $$F_P : [1/2, m+1/2] \times [1/2, n+1/2] \rightarrow [0, 1]$$
>
> *which has the constant value $p_{i,j}$ in the square (pixel)*
>
> $$[i-1/2, i+1/2] \times [j-1/2, j+1/2]$$
>
> *for $i = 1, \ldots, m$ and $j = 1, \ldots, n$.*

What about a colour image P? Then each $p_{i,j} = (r_{i,j}, g_{i,j}, b_{i,j})$ is a triple of numbers so we have a mapping

$$P : \mathbb{Z}_{m,n} \rightarrow \mathbb{R}^3.$$

If we compare with definition 13.9, we see that this corresponds to a sampled version of a parametric surface if we consider the colour values $(r_{i,j}, g_{i,j}, b_{i,j})$ to be x-, y-, and z-coordinates. This may be useful for computations in certain settings, but visually it does not make much sense, see figure 14.4
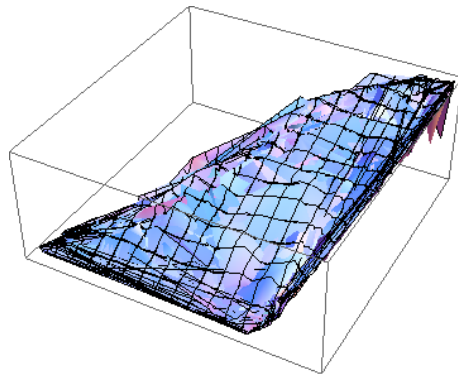
342

Figure 14.4. A colour image viewed as a parametric surface in space.

14.2 Operations on images

When we know that a digital image is a two-dimensional array of numbers, it is quite obvious that we can manipulate the image by performing mathematical operations on the numbers. In this section we will consider some of the simpler operations.

14.2.1 Normalising the intensities

We have assumed that the intensities all lie in the interval $[0, 1]$, but as we noted, many formats in fact use integer values in the range 0–255. And as we perform computations with the intensities, we quickly end up with intensities outside $[0, 1]$ even if we start out with intensities within this interval. We therefore need to be able to *normalise* the intensities. This we can do with the simple linear function in observation 7.23,

$$g(x) = \frac{x - a}{b - a}, \quad a < b,$$

which maps the interval $[a, b]$ to $[0, 1]$. A simple case is mapping $[0, 255]$ to $[0, 1]$ which we accomplish with the scaling $g(x) = x/255$. More generally, we typically perform computations that result in intensities outside the interval $[0, 1]$. We can then compute the minimum and maximum intensities p_{min} and p_{max} and map the interval $[p_{min}, p_{max}]$ back to $[0, 1]$. Several examples of this will be shown below.

(a)   (b)   (c)

Figure 14.5. The red (a), green (b), and blue (c) components of the colour image in figure 14.1.

(a)   (b)   (c)

Figure 14.6. Alternative ways to convert the colour image in figure 14.1 to a grey level image. In (a) each colour triple has been replaced by its maximum, in (b) each colour triple has been replaced by its sum and the result mapped to $[0,1]$, while in (c) each triple has been replaced by its length and the result mapped to $[0,1]$.

14.2.2   Extracting the different colours

If we have a colour image $P = (r_{i,j}, g_{i,j}, b_{i,j})_{i,j=1}^{m,n}$ it is often useful to manipulate the three colour components separately as the three images

$$P_r = (r_{i,j})_{i,j=1}^{m,n}, \quad P_r = (g_{i,j})_{i,j=1}^{m,n}, \quad P_r = (b_{i,j})_{i,j=1}^{m,n}.$$

These are conveniently visualised as grey-level images as in figure 14.5.

14.2.3 Converting from colour to grey-level

If we have a colour image we can convert it to a grey-level image. This means that at each point in the image we have to replace the three colour values (r, g, b) by a single value p that will represent the grey level. If we want the grey-level image to be a reasonable representation of the colour image, the value p should somehow reflect the intensity of the image at the point. There are several ways to do this.

It is not unreasonable to use the largest of the three colour components as a measure of the intensity, i.e, to set $p = \max(r, g, b)$. The result of this can be seen in figure 14.6a.

An alternative is to use the sum of the three values as a measure of the total intensity at the point. This corresponds to setting $p = r + g + b$. Here we have to be a bit careful with a subtle point. We have required each of the r, g and b values to lie in the range $[0, 1]$, but their sum may of course become as large as 3. We also require our grey-level values to lie in the range $[0, 1]$ so after having computed all the sums we must normalise as explained above. The result can be seen in figure 14.6b.

A third possibility is to think of the intensity of (r, g, b) as the length of the colour vector, in analogy with points in space, and set $p = \sqrt{r^2 + g^2 + b^2}$. Again we may end up with values in the range $[0, 3]$ so we have to normalise like we did in the second case. The result is shown in figure 14.6c.

Let us sum this up as an algorithm.

Algorithm 14.9 (Conversion from colour to grey level)**.** *A colour image $P = (r_{i,j}, g_{i,j}, b_{i,j})_{i,j=1}^{m,n}$ can be converted to a grey level image $Q = (q_{i,j})_{i,j=1}^{m,n}$ by one of the following three operations:*

1. *Set $q_{i,j} = \max(r_{i,j}, g_{i,j}, b_{i,j})$ for all i and j.*

2. *(a) Compute $\hat{q}_{i,j} = r_{i,j} + g_{i,j} + b_{i,j}$ for all i and j.*

    (b) Transform all the values to the interval $[0, 1]$ by setting
    $$q_{i,j} = \frac{\hat{q}_{i,j}}{\max_{k,l} \hat{q}_{k,l}}.$$

3. *(a) Compute $\hat{q}_{i,j} = \sqrt{r_{i,j}^2 + g_{i,j}^2 + b_{i,j}^2}$ for all i and j.*

    (b) Transform all the values to the interval $[0, 1]$ by setting
    $$q_{i,j} = \frac{\hat{q}_{i,j}}{\max_{k,l} \hat{q}_{k,l}}.$$
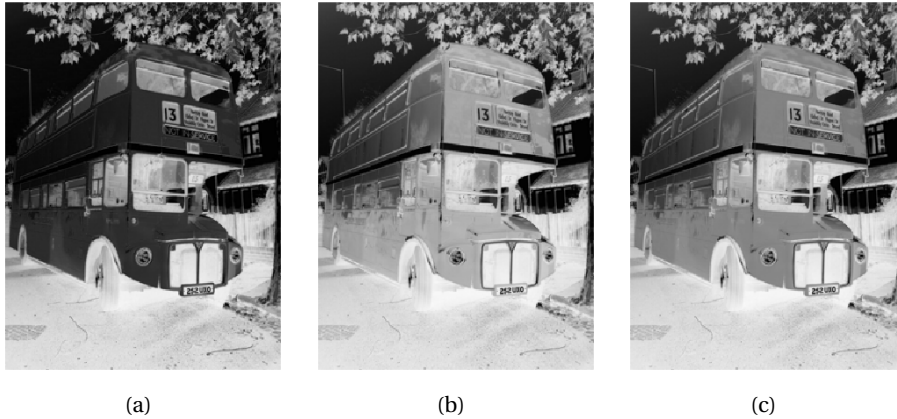

| (a) | (b) | (c) |

Figure 14.7. The negative versions of the corresponding images in figure 14.6.

In practice one of the last two methods are usually preferred, perhaps with a preference for the last method, but the actual choice depends on the application.

14.2.4 Computing the negative image

In film-based photography a negative image was obtained when the film was developed, and then a positive image was created from the negative. We can easily simulate this and compute a negative digital image.

Suppose we have a grey-level image $P = (p_{i,j})_{i,j=1}^{m,n}$ with intensity values in the interval $[0,1]$. Here intensity value 0 corresponds to black and 1 corresponds to white. To obtain the negative image we just have to replace an intensity p by its 'mirror value' $1 - p$.

Fact 14.10 (Negative image). *Suppose the grey-level image $P = (p_{i,j})_{i,j=1}^{m,n}$ is given, with intensity values in the interval $[0,1]$. The negative image $Q = (q_{i,j})_{i,j=1}^{m,n}$ has intensity values given by $q_{i,j} = 1 - p_{i,j}$ for all i and j.*

14.2.5 Increasing the contrast

A common problem with images is that the contrast often is not good enough. This typically means that a large proportion of the grey values are concentrated in a rather small subinterval of $[0,1]$. The obvious solution to this problem is to
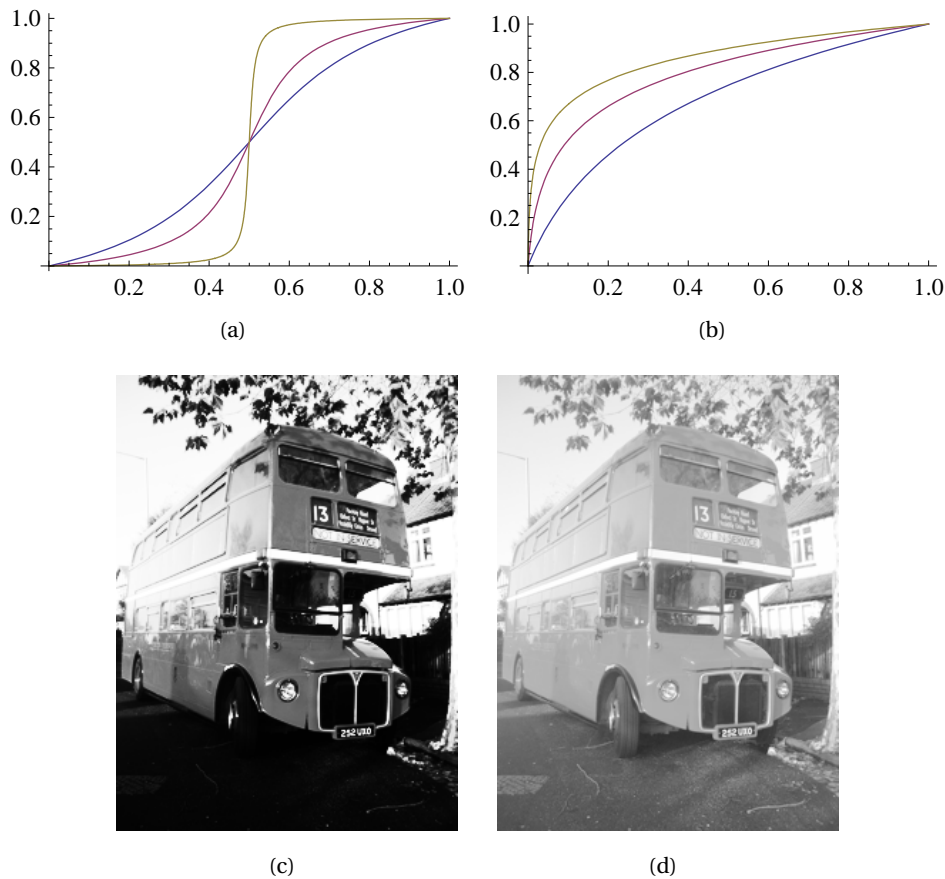
Figure 14.8. The plots in (a) and (b) show some functions that can be used to improve the contrast of an image. In (c) the middle function in (a) has been applied to the intensity values of the image in figure 14.6c, while in (d) the middle function in (b) has been applied to the same image.

somehow spread out the values. This can be accomplished by applying a function f to the intensity values, i.e., new intensity values are computed by the formula

$$\hat{p}_{i,j} = f(p_{i,j})$$

for all i and j. If we choose f so that its derivative is large in the area where many intensity values are concentrated, we obtain the desired effect.

Figure 14.8 shows some examples. The functions in the left plot have quite large derivatives near $x = 0.5$ and will therefore increase the contrast in images with a concentration of intensities with value around 0.5. The functions are all

347

on the form

$$f_n(x) = \frac{\arctan\big(n(x-1/2)\big)}{2\arctan(n/2)} + \frac{1}{2}.$$ (14.1)

For any $n \neq 0$ these functions satisfy the conditions $f_n(0) = 0$ and $f_n(1) = 1$. The three functions in figure 14.8a correspond to $n = 4$, 10, and 100.

Functions of the kind shown in figure 14.8b have a large derivative near $x = 0$ and will therefore increase the contrast in an image with a large proportion of small intensity values, i.e., very dark images. The functions are given by

$$g_\epsilon(x) = \frac{\ln(x+\epsilon) - \ln\epsilon}{\ln(1+\epsilon) - \ln\epsilon},$$ (14.2)

and the ones shown in the plot correspond to $\epsilon = 0.1$, 0.01, and 0.001.

In figure 14.8c the middle function in (a) has been applied to the image in figure 14.6c. Since the image was quite well balanced, this has made the dark areas too dark and the bright areas too bright. In figure 14.8d the function in (b) has been applied to the same image. This has made the image as a whole too bright, but has brought out the details of the road which was very dark in the original.

> **Observation 14.11.** *Suppose a large proportion of the intensity values $p_{i,j}$ of a grey-level image P lie in a subinterval I of $[0,1]$. Then the contrast of the image can be improved by computing new intensities $\hat{p}_{i,j} = f(p_{,j})$ where f is a function with a large derivative in the interval I.*

We will see more examples of how the contrast in an image can be enhanced when we try to detect edges below.

14.2.6 Smoothing an image

When we considered filtering of digital sound in section 4.4.2 of the Norwegian notes, we observed that replacing each sample of a sound by an average of the sample and its neighbours dampened the high frequencies of the sound. We can do a similar operation on images.

Consider the array of numbers given by

$$\frac{1}{16}\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}.$$ (14.3)

We can smooth an image with this array by placing the centre of the array on a pixel, multiplying the pixel and its neighbours by the corresponding weights,

(a)   (b)   (c)

Figure 14.9. The images in (b) and (c) show the effect of smoothing the image in (a).

summing up and dividing by the total sum of the weights. More precisely, we would compute the new pixels by

$$\hat{p}_{i,j} = \frac{1}{16}\big(4p_{i,j} + 2(p_{i,j-1} + p_{i-1,j} + p_{i+1,j} + p_{i,j+1})$$
$$+ p_{i-1,j-1} + p_{i+1,j-1} + p_{i-1,j+1} + p_{i+1,j+1}\big).$$

Since the weights sum to one, the new intensity value $\hat{p}_{i,j}$ is a weighted average of the intensity values on the right. The array of numbers in (14.3) is in fact an example of a computational molecule, see figure 13.3. For simplicity we have omitted the details in the drawing of the computational molecule. We could have used equal weights for all nine pixels, but it seems reasonable that the weight of a pixel should be larger the closer it is to the centre pixel.

As for audio, the values used are taken from Pascal's triangle, since these weights are known to give a very good smoothing effect. A larger filter is given by the array

$$\frac{1}{1024}\begin{pmatrix} 1 & 6 & 15 & 20 & 15 & 6 & 1 \\ 6 & 36 & 90 & 120 & 90 & 36 & 6 \\ 15 & 90 & 225 & 300 & 225 & 90 & 15 \\ 20 & 120 & 300 & 400 & 300 & 120 & 20 \\ 15 & 90 & 225 & 300 & 225 & 90 & 15 \\ 6 & 36 & 90 & 120 & 90 & 36 & 6 \\ 1 & 6 & 15 & 20 & 15 & 6 & 1 \end{pmatrix}. \qquad (14.4)$$

These numbers are taken from row six of Pascal's triangle. More precisely, the value in row k and column l is given by the product $\binom{6}{k}\binom{6}{l}$. The scaling $1/4096$

349

comes from the fact that the sum of all the numbers in the table is $2^{6+6} = 4096$.

The result of applying the two filters in (14.3) and (14.4) to an image is shown in figure 14.9 (b) and (c) respectively. The smoothing effect is clearly visible.

> **Observation 14.12.** *An image P can be smoothed out by replacing the intensity value at each pixel by a weighted average of the intensity at the pixel and the intensity of its neighbours.*

14.2.7 Detecting edges

The final operation on images we are going to consider is edge detection. An edge in an image is characterised by a large change in intensity values over a small distance in the image. For a continuous function this corresponds to a large derivative. An image is only defined at isolated points, so we cannot compute derivatives, but we have a perfect situation for applying numerical differentiation. Since a grey-level image is a scalar function of two variables, the numerical differentiation techniques from section 13.2 can be applied.

Partial derivative in x-direction. Let us first consider computation of the partial derivative $\partial P/\partial x$ at all points in the image. We use the familiar approximation

$$\frac{\partial P}{\partial x}(i, j) = \frac{p_{i+1,j} - p_{i-1,j}}{2}, \tag{14.5}$$

where we have used the convention $h = 1$ which means that the derivative is measured in terms of 'intensity per pixel'. We can run through all the pixels in the image and compute this partial derivative, but have to be careful for $i = 1$ and $i = m$ where the formula refers to non-existing pixels. We will adapt the simple convention of assuming that all pixels outside the image have intensity 0. The result is shown in figure 14.10a.

This image is not very helpful since it is almost completely black. The reason for this is that many of the intensities are in fact negative, and these are just displayed as black. More specifically, the intensities turn out to vary in the interval $[-0.424, 0.418]$. We therefore normalise and map all intensities to $[0, 1]$. The result of this is shown in (b). The predominant colour of this image is an average grey, i.e, an intensity of about 0.5. To get more detail in the image we therefore try to increase the contrast by applying the function f_{50} in equation 13.6 to each intensity value. The result is shown in figure 14.10c which does indeed show more detail.

It is important to understand the colours in these images. We have computed the derivative in the x-direction, and we recall that the computed values varied in the interval $[-0.424, 0.418]$. The negative value corresponds to the
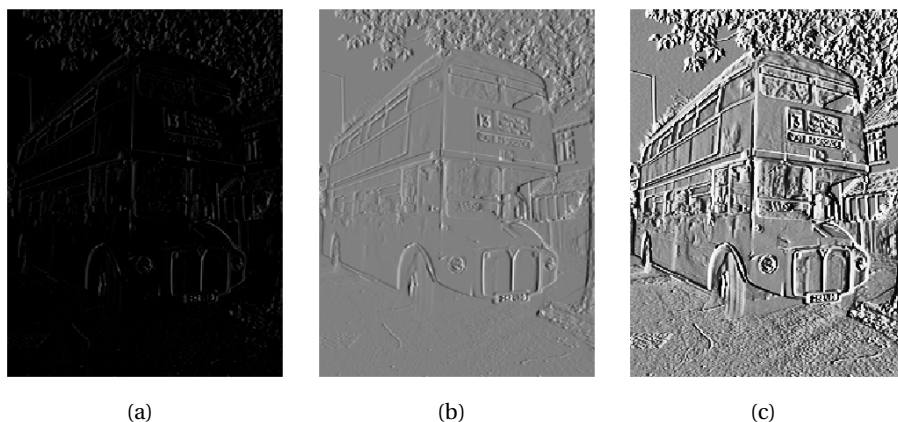
(a)   (b)   (c)

Figure 14.10. The image in (a) shows the partial derivative in the x-direction for the image in 14.6. In (b) the intensities in (a) have been normalised to $[0, 1]$ and in (c) the contrast as been enhanced with the function f_{50}, equation 14.1.

largest average decrease in intensity from a pixel $p_{i-1,j}$ to a pixel $p_{i+1,j}$. The positive value on the other hand corresponds to the largest average increase in intensity. A value of 0 in figure 14.10a corresponds to no change in intensity between the two pixels.

When the values are mapped to the interval $[0, 1]$ in figure 14.10b, the small values are mapped to something close to 0 (almost black), the maximal values are mapped to something close to 1 (almost white), and the values near 0 are mapped to something close to 0.5 (grey). In figure 14.10c these values have just been emphasised even more.

Figure 14.10c tells us that in large parts of the image there is very little variation in the intensity. However, there are some small areas where the intensity changes quite abruptly, and if you look carefully you will notice that in these areas there is typically both black and white pixels close together, like down the vertical front corner of the bus. This will happen when there is a stripe of bright or dark pixels that cut through an area of otherwise quite uniform intensity.

Since we display the derivative as a new image, the denominator is actually not so important as it just corresponds to a constant scaling of all the pixels; when we normalise the intensities to the interval $[0, 1]$ this factor cancels out.

We sum up the computation of the partial derivative by giving its computational molecule.

> **Observation 14.13.** *Let $P = (p_{i,j})_{i,j=1}^{m,n}$ be a given image. The partial derivative $\partial P/\partial x$ of the image can be computed with the computational molecule*
>
> $$\frac{1}{2}\begin{pmatrix} 0 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}.$$

As we remarked above, the factor $1/2$ can usually be ignored. We have included the two rows of 0s just to make it clear how the computational molecule is to be interpreted; it is obviously not necessary to multiply by 0.

Partial derivative in y-direction.   The partial derivative $\partial P/\partial y$ can be computed analogously to $\partial P/\partial x$.

> **Observation 14.14.** *Let $P = (p_{i,j})_{i,j=1}^{m,n}$ be a given image. The partial derivative $\partial P/\partial y$ of the image can be computed with the computational molecule*
>
> $$\frac{1}{2}\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & -1 & 0 \end{pmatrix}.$$

The result is shown in figure 14.12b. The intensities have been normalised and the contrast enhanced by the function f_{50} in (14.1).

The gradient.   The gradient of a scalar function is often used as a measure of the size of the first derivative. The gradient is defined by the vector

$$\nabla P = \left(\frac{\partial P}{\partial x}, \frac{\partial P}{\partial y}\right),$$

so its length is given by

$$|\nabla P| = \sqrt{\left(\frac{\partial P}{\partial x}\right)^2 + \left(\frac{\partial P}{\partial y}\right)^2}.$$

When the two first derivatives have been computed it is a simple matter to compute the gradient vector and its length; the resulting is shown as an image in figure 14.11c.

<div align="center">(a)   (b)   (c)</div>

Figure 14.11. Computing the gradient. The image obtained from the computed gradient is shown in (a) and in (b) the numbers have been normalised. In (c) the contrast has been enhanced with a logarithmic function.

The image of the gradient looks quite different from the images of the two partial derivatives. The reason is that the numbers that represent the length of the gradient are (square roots of) sums of squares of numbers. This means that the parts of the image that have virtually constant intensity (partial derivatives close to 0) are coloured black. In the images of the partial derivatives these values ended up in the middle of the range of intensity values, with a final colour of grey, since there were both positive and negative values.

Figure 14.11a shows the computed values of the gradient. Although it is possible that the length of the gradient could become larger than 1, the maximum value in this case is about 0.876. By normalising the intensities we therefore increase the contrast slightly and obtain the image in figure 14.11b.

To enhance the contrast further we have to do something different from what was done in the other images since we now have a large number of intensities near 0. The solution is to apply a function like the ones shown in figure 14.8b to the intensities. If we use the function $g_{0.01}$ defined in equation (14.2) we obtain the image in figure 14.11c.

14.2.8   Comparing the first derivatives

Figure 14.12 shows the two first-order partial derivatives and the gradient. If we compare the two partial derivatives we see that the x-derivative seems to emphasise vertical edges while the y-derivative seems to emphasise horizontal edges. This is precisely what we must expect. The x-derivative is large when the difference between neighbouring pixels in the x-direction is large, which is the case across a vertical edge. The y-derivative enhances horizontal edges for a
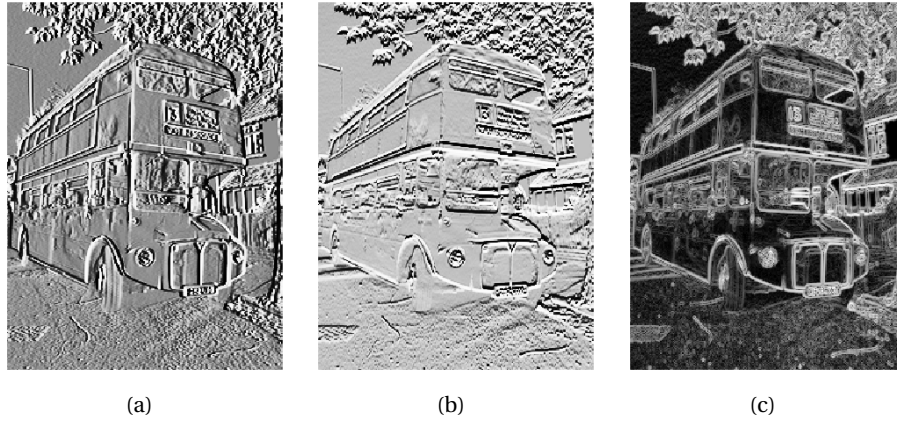
(a)   (b)   (c)

Figure 14.12. The first-order partial derivatives in the x-direction (a) and y-direction (b), and the length of the gradient (c). In all images, the computed numbers have been normalised and the contrast enhanced.

similar reason.

The gradient contains information about both derivatives and therefore emphasises edges in all directions. It also gives a simpler image since the sign of the derivatives has been removed.

14.2.9   Second-order derivatives

To compute the three second order derivatives we apply the corresponding computational molecules which we described in section 13.2.

Observation 14.15 (Second order derivatives of an image)**.** *The second order derivatives of an image P can be computed by applying the computational molecules*

$$\frac{\partial^2 P}{\partial x^2}: \quad \begin{pmatrix} 0 & 0 & 0 \\ -1 & 2 & -1 \\ 0 & 0 & 0 \end{pmatrix},$$

$$\frac{\partial^2 P}{\partial y \partial x}: \quad \frac{1}{4}\begin{pmatrix} -1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & -1 \end{pmatrix},$$

$$\frac{\partial^2 P}{\partial y^2}: \quad \begin{pmatrix} 0 & 1 & 0 \\ 0 & 2 & 0 \\ 0 & -1 & 0 \end{pmatrix}.$$
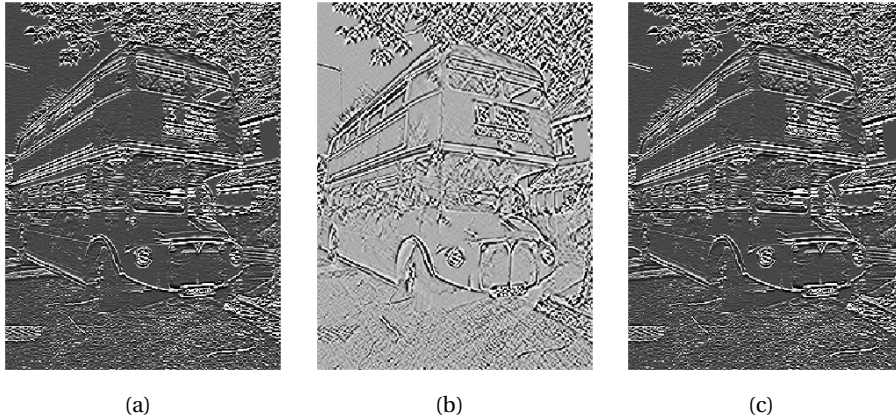

Figure 14.13. The second-order partial derivatives in the x-direction (a) and xy-direction (b), and the y-direction (c). In all images, the computed numbers have been normalised and the contrast enhanced.

With the information in observation 14.15 it is quite easy to compute the second-order derivatives, and the results are shown in figure 14.13. The computed derivatives were first normalised and then the contrast enhanced with the function f_{100} in each image, see equation 14.1.

As for the first derivatives, the xx-derivative seems to emphasise vertical edges and the yy-derivative horizontal edges. However, we also see that the second derivatives are more sensitive to noise in the image (the areas of grey are less uniform). The mixed derivative behaves a bit differently from the other two, and not surprisingly it seems to pick up both horizontal and vertical edges.

14.3   Image formats

Just as there are many audio formats, there are many image formats, and in this section we will give a superficial description of some of them. Before we do this however, we want to distinguish between two important types of graphics representations.

14.3.1   Raster graphics and vector graphics

At the beginning of this chapter we saw that everything that is printed on a computer monitor or by a printer consists of small dots. This is a perfect match for digital images which also consist of a large number of small dots. However, as we magnify an image, the dots in the image become visible as is evident in figure 14.2.
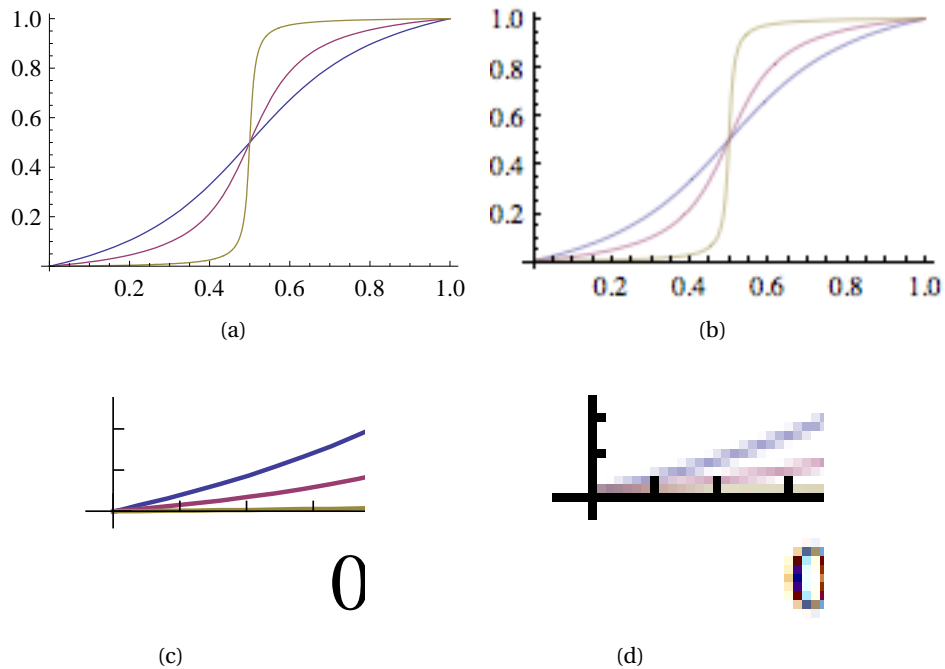
355

Figure 14.14. The difference between vector graphics ((a) and (c)) and raster graphics ((b) and (d)).

In addition to images, text and various kinds of line art (drawings) are also displayed on monitors and printed by printers, and must therefore be represented in terms of dots. There is a big difference though, in how these kinds of graphical images are stored. As an example, consider the plots in figure 14.14. In figure (c), the plot in (a) has been magnified, without any dots becoming visible. In (d), the plot in (b) has been magnified, and here the dots have become clearly visible. The difference is that while the plots in (b)-(d) are represented as an image with a certain number of dots, the plots in (a)-(d) are represented in terms of mathematical primitives like lines and curves — this is usually referred to as a *vector representation* or *vector graphics*. The advantage of vector graphics is that the actual dots to be used are not determined until the figure is to be drawn. This means that in figure (c) the dots which are drawn were not determined until the magnification was known. On the other hand, the plot in (b) was saved as an image with a fixed number of dots, just like the pictures of the bus earlier in the chapter. So when this image is magnified, the only possibility is to magnify the dots themselves, which inevitably produces a grainy picture like the one in(d).

In vector graphics formats all elements of a drawing are represented in terms of mathematical primitives. This includes all lines and curves as well as text. A
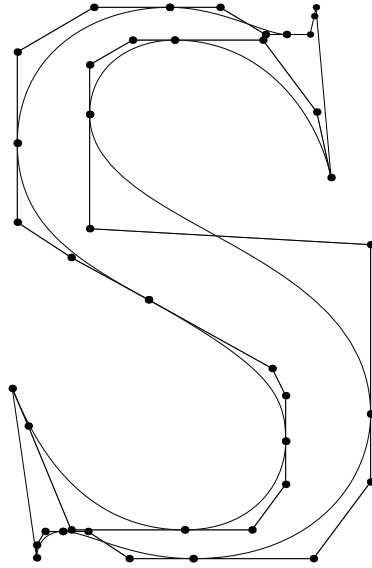
Figure 14.15. The character 'S' in the font Times Roman. The dots are parameters that control the shape of the curves.

line is typically represented by its two endpoints and its width. Curved shapes are either represented in terms of short connected line segments or smoothly connected polynomial curve segments. Whenever a drawing on a monitor or printer is requested, the actual dots to be printed are determined from the mathematical representation. In particular this applies to fonts (the graphical shapes of characters) which are usually represented in terms of quadratic or cubic polynomial curves (so-called *Bezier curves*), see figure 14.15 for an example.

Fact 14.16. *In vector graphics a graphical image is represented in terms of mathematical primitives like lines and curves, and can be magnified without any loss in quality. In raster graphics, a graphical image is represented as a digital image, i,.e., in terms of pixels. As the image is magnified, the pixels become visible and the quality of the image deteriorates.*

14.3.2 Vector graphics formats

The two most common vector graphics formats are Postscript and PDF which are formats for representing two-dimensional graphics. There are also standards for three-dimensional graphics, but these are not as universally accepted.

Postscript. *Postscript* is a programming language developed by Adobe Systems in the early 1980s. Its principal application is representation of page images, i.e., information that may be displayed on a monitor or printed by a printer. The basic primitives in Postscript are straight line segments and cubic polynomial curves which are often joined (smoothly) together to form more complex shapes. Postscript fonts consist of Postscript programs which define the outlines of the shapes of all the characters in the font. Whenever a Postscript program needs to print something, software is required that can translate from the mathematical Postscript representation to the actual raster representation to be use on the output device. This software is referred to as a Postscript *interpreter* or *driver*. Postscript files are standard text files so the program that produces a page can be inspected (and edited) in a standard editor. A disadvantage of this is that Postscript files are coded inefficiently and require a lot of storage space. Postscript files have extension `.eps` or `.ps`.

Since many pages contain images, Postscript also has support for including raster graphics within a page.

PDF. *Portable Document Format* is a standard for representing page images that was also developed by Adobe. In contrast to Postscript, which may require external information like font libraries to display a page correctly, a PDF-file contains all the necessary information within itself. It supports the same mathematical primitives as Postscript, but codes the information in a compact format. Since a page may contain images, it is also possible to store a digital image in PDF-format. PDF-files may be locked so that they cannot be changed. PDF is in wide-spread use across computer platforms and is a preferred format for exchanging documents. PDF-files have extension `.pdf`.

14.3.3 Raster graphics formats

There are many formats for representing digital images. We have already mentioned Postscript and PDF; here we will mention a few more which are pure image formats (no support for vector graphics).

Before we describe the formats we need to understand a technical detail about representation of colour. As we have already seen, in most colour images the colour of a pixel is represented in terms of the amount of red, green and blue or (r, g, b). Each of these numbers is usually represented by eight bits and can take integer values in the range 0–255. In other words, the colour information at each pixel requires three bytes. When colour images and monitors became commonly available in the 1980s, the file size for a 24-bit image file was very large compared to the size of hard drives and available computer memory.

Instead of storing all 24 bits of colour information it was therefore common to create a table of 256 colours with which a given image could be represented quite well. Instead of storing the 24 bits, one could just store the table at the beginning of the file, and at each pixel, the eight bits corresponding to the correct entry in the table. This is usually referred to as eight-bit colour and the table is called a *look-up* table or *palette*. For large photographs, 256 colours is far from sufficient to obtain reasonable colour reproduction.

Images may contain a large amount of data and have great potential for both lossless and lossy compression. For lossy compression, strategies similar to the ones used for audio compression are used. This means that the data are transformed by a DCT or wavelet transform (these transforms generalise easily to images), small values are set to zero and the resulting data coded with a lossless coding algorithm.

Like audio formats, image formats usually contain information like resolution, time when the image was recorded and similar information at the beginning of the file.

GIF. *Graphics Interchange Format* was introduced in 1987 as a compact representation of colour images. It uses a palette of at most 256 colours sampled from the 24-bit colour model, as explained above. This means that it is unsuitable for colour images with continuous colour tones, but it works quite well for smaller images with large areas of constant colour, like logos and buttons on web pages. Gif-files are losslessly coded with a variant of the Lempel-Ziv-Welch algorithm. The extension of GIF-files is `.gif`.

TIFF. *Tagged Image File Format* is a flexible image format that may contain multiple images of different types in the same file via so-called 'tags'. TIFF supports lossless image compression via Lempel-Ziv-Welch compression, but may also contain JPEG-compressed images (see below). TIFF was originally developed as a format for scanned documents and supports images with one-bit pixel values (black and white). It also supports advanced data formats like more than eight bits per colour component. TIFF-files have extension `.tiff`.

JPEG. *Joint Photographic Experts Group* is an image format that was approved as an international standard in 1994. JPEG is usually lossy, but may also be lossless and has become a popular format for image representation on the Internet. The standard defines both the algorithms for encoding and decoding and the storage format. JPEG divides the image into 8×8 blocks and transforms each block with a Discrete Cosine Transform. These values corresponding to higher

frequencies (rapid variations in colour) are then set to 0 unless they are quite large, as this is not noticed much by human perception. The perturbed DCT values are then coded by a variation of Huffman coding. JPEG may also use arithmetic coding, but this increases both the encoding and decoding times, with only about 5 % improvement in the compression ratio. The compression level in JPEG images is selected by the user and may result in conspicuous artefacts if set too high. JPEG is especially prone to artefacts in areas where the intensity changes quickly from pixel to pixel. The extension of a JPEG-file is `.jpg` or `.jpeg`.

PNG. *Portable Network Graphics* is a lossless image format that was published in 1996. PNG was not designed for professional use, but rather for transferring images on the Internet, and only supports grey-level images and rgb images (also palette based colour images). PNG was created to avoid a patent on the LZW-algorithm used in GIF, and also GIF's limitation to eight bit colour information. For efficient coding PNG may (this is an option) predict the value of a pixel from the value of previous pixels, and subtract the predicted value from the actual value. It can then code these error values using a lossless coding method called DEFLATE which uses a combination of the LZ77 algorithm and Huffman coding. This is similar to the algorithm used in lossless audio formats like Apple Lossless and FLAC. The extension of PNG-files is `.png`.

JPEG 2000. This lossy (can also be used as lossless) image format was developed by the Joint Photographic Experts Group and published in 2000. JPEG 2000 transforms the image data with a wavelet transform rather than a DCT. After significant processing of the wavelet coefficients, the final coding uses a version of arithmetic coding. At the cost of increased encoding and decoding times, JPEG 2000 leads to as much as 20 % improvement in compression ratios for medium compression rates, possibly more for high or low compression rates. The artefacts are less visible than in JPEG and appear at higher compression rates. Although a number of components in JPEG 2000 are patented, the patent holders have agreed that the core software should be available free of charge, and JPEG 2000 is part of most Linux distributions. However, there appear to be some further, rather obscure, patents that have not been licensed, and this may be the reason why JPEG 2000 is not used more. The extension of JPEG 2000 files is `.jp2`.

APPENDIX

Answers to some exercises

Chapter 1

1.1   **a)**

$s1 := 0; s2 := 0;$
for $k := 1, 2, \ldots, n$
    if $a_k > 0$
        $s1 := s1 + a_k;$
    else
        $s2 := s2 + a_k;$
$s2 := -s2;$

Note that we could also replace the statement in the **else**-branch by $s2 := s2 - a_k$ and leave out the last statement.

b) We introduce two new variables pos and neg which count the number of positive and negative elements, respectively.

$s1 := 0; pos := 0;$
$s2 := 0; neg := 0;$
for $k := 1, 2, \ldots, n$
    if $a_k > 0$
        $s1 := s1 + a_k;$
        $pos := pos + 1;$
    else
        $s2 := s2 + a_k;$
        $neg := neg + 1;$
$s2 := -s2;$

1.2 We represent the three-digit numbers by their decimal numerals which are integers in the range 0–9. The numerals of the number $x = 431$ for example, is represented by $x_1 = 1$, $x_2 = 3$ and $x_3 = 4$. Adding two arbitrary such numbers x and y produces a sum z which can be computed by the algorithm

361

if $x_1 + y_1 < 10$
    $z_1 := x_1 + y_1;$
else
    $x_2 := x_2 + 1;$
    $z_1 := x_1 + y_1 - 10;$
if $x_2 + y_2 < 10$
    $z_2 := x_2 + y_2;$
else
    $x_3 := x_3 + 1;$
    $z_2 := x_2 + y_2 - 10;$
if $x_3 + y_3 < 10$
    $z_3 := x_3 + y_3;$
else
    $z_4 := 1;$
    $z_3 := x_3 + y_3 - 10;$

1.3 We use the same representation as in the solution for exercise 2. Multiplication of two three-digit numbers x and y can then be performed by the formulas

$product1 := x_1 * y1 + 10 * x_1 * y_2 + 100 * x_1 * y_3;$
$product2 := 10 * x_2 * y1 + 100 * x_2 * y_2 + 1000 * x_2 * y_3;$
$product3 := 100 * x_3 * y1 + 1000 * x_3 * y_2 + 10000 * x_3 * y_3;$
$product := product1 + product2 + product3;$

Chapter 2

2.1 The truth table is

p	q	r	$p \oplus q$	$(p \oplus q) \oplus r$	$q \oplus r$	$p \oplus (q \oplus r)$
F	F	F	F	F	F	F
F	F	T	F	T	T	T
F	T	F	T	T	T	T
F	T	T	T	F	F	F
T	F	F	T	T	F	T
T	F	T	T	F	T	F
T	T	F	F	F	T	F
T	T	T	F	T	F	T

2.2 Solution by truth table for $\neg(p \wedge q) = \neg(p \vee q)$

p	q	$p \wedge q$	$\neg p$	$\neg q$	$\neg(p \wedge q)$	$(\neg p) \vee (\neg q)$
F	F	F	T	T	T	T
F	T	F	T	F	T	T
T	F	F	F	T	T	T
T	T	T	F	F	F	F

Solution by truth table for $\neg(p \vee q) = \neg(p \wedge q)$

p	q	$p \vee q$	$\neg p$	$\neg q$	$\neg(p \vee q)$	$(\neg p) \wedge (\neg q)$
F	F	F	T	T	T	T
F	T	T	T	F	F	F
T	F	T	F	T	F	F
T	T	T	F	F	F	F

2.3 No answer given.

2.4 No answer given.

Chapter 3

3.1   **a)** 220

      b) 32

      c) 10001

      d) 1022634

      e) 123456

      f) $7e$

3.2   **a)** 0.01

      b) 0.102120102120102120...

      c) 0.01

      d) 0.001111111...

      e) 0.7

      f) 0.6060606...

      g) $0.e$

      h) 0.24

      i) 0.343

3.3 $\pi_9 \approx 3.12_9$

3.4   **a)** 131

      b) 67

      c) 252

3.5   **a)** 100100

      b) 1000000

      c) 11010111

3.6   **a)** $4d$

      b) c

      c) $29e4$

      d) 0.594

      e) 0.052

      f) $0.ff8$

3.7   **a)** 111100

      b) 100000000

      c) 111001010001

      d) 0.000010101010

      e) 0.000000000001

f) 0.111100000001

3.8   **a)** $7 = 10_7$, $37 = 10_{37}$ and $4 = 10_4$

      b) $\beta = 13, \beta = 100$

3.9   **a)** $400 = 100_{20}$, $4 = 100_2$ and $278 = 100_{17}$

      b) $\beta = 5, \beta = 29$

3.10 No answer given.

3.11 $c - 1$

3.12 No answer given.

3.13 No answer given.

3.14   **a)** 4_7

      b) 13_6

      c) 10001_2

      d) 1100_3

      e) 103_5

      f) $4_5 = 4_7$

3.15   **a)** 3_8

      b) 11_2

      c) 174_8

      d) 112_3

      e) 24_5

      f) -5_7

3.16   **a)** 1100_2

      b) 10010_2

      c) 1210_3

      d) 141_5

      e) 13620_8

      f) 10220_3

      g) 1111_2

Chapter 4

4.1 Largest integer: 01111111111111111111111111111111.

Smallest integer: 10000000000000000000000000000000.

4.2 No answer given.

4.3 No answer given.

4.4 No answer given.

4.5 No answer given.

4.6 Answers to some of the questions:

    a) 0.4752735×10^7

    b) $0.602214179 \times 10^{24}$

    c) 0.8617343×10^{-4}.

4.7 $0.10011100111101011010\ldots \times 2^4$

4.8   **a)** $01011010_2 = 5a_{16}$

    b) $1100001110110101_2 = c3b5_{16}$

    c) $1100111110111000_2 = cfb8_{16}$

    d) $111010001011110010110111_2 = e8bcb7_{16}$

4.9   **a)** $0000000001011010_2 = 005a_{16}$

    b) $0000000011110101_2 = 00f5_{16}$

    c) $0000001111111000_2 = 03f8_{16}$

    d) $1000111100110111_2 = 8f37_{16}$

4.10   **a)** æ: Ã¦, ø: Ã¸ å: Ã¥

    b) Nothing or error message; these codes are not valid UTF-8 codes

    c) æ: NULæ, ø: NULø, å: NULå; each character is preceded (or followed for LE) by a trailing null character, this has no visible impact on the displayed text. The opposite again yields illegitimate UTF-16 encodings (too short).

    d) The conversion from UTF-8 to UTF-16 yields the following Hangul symbols:

    æ: 쎦, ø: 쎸, å: 쎥

    The conversion from UTF-16 to UTF-8 yields illegitimate codes, though there will be an allowed null character preceding (or following for LE) each prohibited letter.

Chapter 5

5.1 Relative errors:

    a) $r = 0.0006$

    b) $r \approx 0.0183$

    c) $r \approx 2.7 \times 10^{-4}$

    d) $r \approx 0.94$

5.2   **a)** 0.1647×10^2

    b) 0.1228×10^2

    c) 0.4100×10^{-1}

    d) 0.6000×10^{-1}

    e) -0.5000×10^{-2}

5.3   **a)** Normalised number in base β: A nonzero number a is written as

$$a = \alpha \times \beta^n$$

    where $\beta^{-1} \leq |\alpha| < 1$.

    b) In any numeral system we have three cases to consider when defining rounding rules. Note also that it is sufficient to define rounding for two-digit fractional numbers.

    In the octal numeral system the three rules are:

        1. A number $(0.d_1 d_2)_8$ is rounded to $0.d_1$ if the digit d_2 is 0, 1, 2 or 3.

        2. If $d_1 < 7$ and d_2 is 4, 5, 6, or 7, then $(0.d_1 d_2)_8$ is rounded to $0.\tilde{d}_1$ where $\tilde{d}_1 = d_1 + 1$.

        3. A number $(0.7d_2)_8$ is rounded to 1.0 if d_2 is 4, 5, 6, or 7.

    c) No answer given.

5.4 One possible program:

```
n := 1;
while 1.0 + 2^-n > 1.0
    n := n + 1;
print n;
```

5.5 No answer given.

5.6 No answer given.

5.7 No answers given

5.8   **a)** No answer given.

b) The fomula $\ln x^2 - \ln(x^2 + x)$ is problematic for large values of x since then the two logarithms will become almost equal and we get cancellation. Using properties of the logarithm, the expression can be rewritten as

$$\ln x^2 - \ln(x^2 + x) = \ln\left(\frac{x^2}{x^2 + x}\right) = \ln\left(\frac{x}{x+1}\right)$$

which will not cause problems with cancellation.

c) No answer given.

5.9 No answer given.

Chapter 6

6.1   **a)**   Linear.

         b)   Nonlinear.

         c)   Nonlinear.

         d)   Linear.

6.2   No answer given.

6.3   No answer given.

6.4   No answers given.

6.5   No answers given.

6.6   **a)** $x_n = C15^{-n} + D3^{-n}$.

         b) $x_n = 15^{-n}$.

         c) $n \approx 24$.

         d)   No answer given.
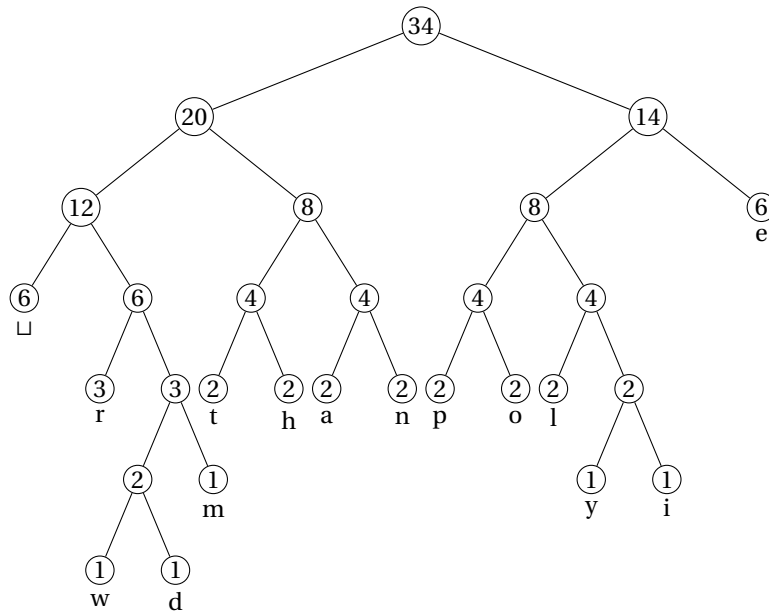
6.7   No answer given.

Figure 16. The Huffman tree for the text 'there are many people in the world'.

Chapter 7

7.1 Huffman coding of the text "There are many peopole in the world". (In this soultion we will treat the capital t in "There" as if it were not capitalized)

a)

$$f(t) = 2, \qquad f(a) = 2, \qquad f(o) = 2,$$
$$f(h) = 2, \qquad f(m) = 1, \qquad f(l) = 2,$$
$$f(e) = 6, \qquad f(n) = 2, \qquad f(i) = 1,$$
$$f(r) = 3, \qquad f(y) = 1, \qquad f(w) = 1,$$
$$f(\sqcup) = 6, \qquad f(p) = 2, \qquad f(d) = 1.$$

b) An example of a Huffman tree for this text can be seen in figure 16:

c) The Huffman coding for the text "there are many people in the world" is then:

0100 0101 11 0010 11 000 0110 0010 11 000

00111 0110 0111 10110 000 1000 11 1001 1000

1010 11 000 10111 0111 000 0100 0101 11 000

001100 1001 0010 1010 001101

The entropy is:

$$H = 3.6325 \qquad\qquad (.6)$$

which means an optimal coding of the text would use 3.6325 bits per symbol. There are 34 symbols so the minimum coding would consist of 15 bytes and 4 bits. The Huffman coding above gave 15 bytes and 5 bits of information, so this coding is very good.

7.2 **a)** Use ternary trees instead of binary ones. (Each tree has either zero or three sub-trees/children).

b) Use n-nary trees. (Each tree has either zero or n subtrees/children)

7.3 **a)**
$$f(A) = 4,$$
$$f(B) = 2,$$
$$f(C) = 2,$$

One of the four possible Huffman codings are:

0 10 0 11 0 10 11 0

The entropy is
$$H = 1.5 \qquad\qquad (.7)$$

This gives an optimal coding with 12 bits for 8 symbols, which is just what the Huffman coding gave.

b) Dividing all the frequensies by 2 and interchanging A with C in the four trees in a) gives the four trees for this problem.The four sets of codes are the same (with A interchanged by C) and so is the entropy so the situation is still optimal.

7.4 Frequensies used are all 1.

7.5 **a)**
$$f(A) = 9, \qquad p(A) = 0.1,$$
$$f(B) = 1, \qquad p(B) = 0.9,$$

b) 5 bits

c) 01110

7.6 **a)** $H = 2$

b) 2 bits per symbol

c) 2m + 1 bits $\frac{2m+1}{m} \approx 2$ bits per symbol

d)

00 10 11 01 00 10

371

e)

00 10 11 01 00 10 1

7.7

BCBBCBBBCB

7.8

01 01 11 10 00

7.9

$$f(x) = c + (y - a)\frac{d - c}{b - a} \tag{.8}$$

Chapter 8

8.1

15.1 No solutions given

15.2
$$\frac{\partial^3 f}{\partial x^2 \partial y} \approx \frac{f_{2,1} - f_{2,0} - 2f_{1,1} + 2f_{1,0} + f_{0,1} - f_{0,0}}{h_1^2 h_2}.$$

$$\frac{\partial^4 f}{\partial x^2 \partial y^2} \approx$$

$$\frac{f_{2,2} - 2f_{2,1} + f_{2,0} - 2f_{1,2} + 4f_{1,1} - 2f_{1,0} + f_{0,2} - 2f_{0,1} + f_{0,0}}{h_1^2 h_2^2}.$$

Chapter 16