Numerical Algorithms
and
Digital Representation

Knut Mørken
Department of Mathematics
University of Oslo

August 2022

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 Edu-*

cation (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 foundation 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 ten years. Øyvind Ryan, Andreas Våvang Solbrå, Solveig Bruvoll, and Marit Sandstad have helped directly with recent versions, 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 previous academic home, the Department of Informatics and its chairman, now our dean, Morten Dæhlen, was very supportive of this work by giving me the freedom to extend the Department's teaching, and by extensive support of the CSE-project. It has been a pleasure to work with the Department of Mathematics for more than a decade, 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. Now that I am a part of the Department of Mathematics I am looking forward to assist in forming its future educational profile.

A course like MAT-INF1100 is completely dependent on support from other courses. Tom Lindstrøm has done a tremendous job with the parallel calcu-

lus 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 scientific flavour, and I have benefited from many hours of discussions with both of them. Morten Hjorth-Jensen, Arnt-Inge Vistnes and Anders Malthe-Sørenssen with colleagues 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 people: Morten Hjorth-Jensen and Anders Malthe-Sørenssen from the Physics Department, Hans Petter Langtangen from the Simula Research Lab and the Department of Informatics, Øyvind Ryan from the CMA, Solveig Kristensen and Annik Myhre (Deans of Education at the MN-faculty[1]), Hanne Sølna (Head of the Studies section at the MN-faculty[1]), 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.

Blindern, August 2022

Knut Mørken

[1] The Faculty of Mathematics and Natural Sciences.

Contents

1 Introduction **1**
1.1 A bit of history . . . 1
1.2 Computers and different types of information . . . 3
1.2.1 Text . . . 4
1.2.2 Sound . . . 4
1.2.3 Images . . . 4
1.2.4 Film . . . 4
1.2.5 Geometric form . . . 5
1.2.6 Laws of nature . . . 5
1.2.7 Virtual worlds . . . 5
1.2.8 Summary . . . 6
1.3 Computation by hand and by computer . . . 7
1.4 Algorithms . . . 8
1.4.1 Statements . . . 10
1.4.2 Variables and assignment . . . 11
1.4.3 For-loops . . . 12
1.4.4 If-tests . . . 13
1.4.5 While-loops . . . 14
1.4.6 Print statement . . . 14
1.5 Doing computations on a computer . . . 15
1.5.1 How can computers be used for calculations? . . . 15
1.5.2 What do you need to know? . . . 16
1.5.3 Different computing environments . . . 17

I Numbers 21

2 0 and 1 23
2.1 Robust communication . . . 23
2.2 Why 0 and 1 in computers? . . . 23
2.3 True and False . . . 25
2.3.1 Logical variables and logical operators . . . 26
2.3.2 Combinations of logical operators . . . 29

3 Numbers and Numeral Systems 31
3.1 Terminology and Notation . . . 32
3.2 Natural Numbers in Different Numeral Systems . . . 34
3.2.1 Alternative Numeral Systems . . . 34
3.2.2 Conversion to the Base-β Numeral System . . . 37
3.2.3 Tabular display of the conversion . . . 39
3.2.4 Conversion between base-2 and base-16 . . . 40
3.3 Representation of Fractional Numbers . . . 44
3.3.1 Rational and Irrational Numbers in Base-β . . . 44
3.3.2 Conversion of fractional numbers . . . 46
3.3.3 An Algorithm for Converting Fractional Numbers . . . 48
3.3.4 Conversion between binary and hexadecimal . . . 49
3.3.5 Properties of Fractional Numbers in Base-β . . . 51
3.4 Arithmetic in Base β . . . 56
3.4.1 Addition . . . 56
3.4.2 Subtraction . . . 56
3.4.3 Multiplication . . . 57

4 Computers, Numbers, and Text 61
4.1 Representation of Integers . . . 61
4.1.1 Bits, bytes and numbers . . . 62
4.1.2 Fixed size integers . . . 63
4.1.3 Two's complement . . . 64
4.1.4 Integers in Java . . . 65
4.1.5 Integers in Python . . . 66
4.1.6 Division by zero . . . 67
4.2 Computers and real numbers . . . 68
4.2.1 The challenge of real numbers . . . 69
4.2.2 The normalised form of real numbers . . . 69
4.2.3 32-bit floating-point numbers . . . 72
4.2.4 Special bit combinations . . . 73

4.2.5 64-bit floating-point numbers . . . 73
4.2.6 Floating point numbers in Java . . . 73
4.2.7 Floating point numbers in Python . . . 74
4.3 Representation of letters and other characters . . . 75
4.3.1 The ASCII table . . . 76
4.3.2 ISO latin character sets . . . 79
4.3.3 Unicode . . . 79
4.3.4 UTF-8 . . . 81
4.3.5 UTF-16 . . . 83
4.3.6 UTF-32 . . . 85
4.3.7 Text in Java . . . 85
4.3.8 Text in Python . . . 85
4.4 Representation of general information . . . 90
4.4.1 Text . . . 90
4.4.2 Numbers . . . 91
4.4.3 General information . . . 91
4.4.4 Computer programs . . . 91
4.5 A fundamental principle of computing . . . 92

5 Computer Arithmetic and Round-Off Errors 95
5.1 Integer arithmetic and errors . . . 96
5.2 Floating-point arithmetic and round-off error . . . 96
5.2.1 Truncation and rounding . . . 97
5.2.2 A simplified model for computer arithmetic . . . 98
5.2.3 An algorithm for floating-point addition . . . 99
5.2.4 Observations on round-off errors in addition/subtraction . . . 102
5.2.5 Multiplication and division of floating-point numbers . . . 103
5.2.6 The IEEE standard for floating-point arithmetic . . . 105
5.3 Measuring the error . . . 108
5.3.1 Absolute error . . . 108
5.3.2 Relative error . . . 109
5.3.3 Properties of the relative error . . . 110
5.3.4 Errors in floating-point representation . . . 111
5.4 Rewriting formulas to avoid rounding errors . . . 115

II Sequences of Numbers 119

6 Numerical Simulation of Difference Equations 121

6.1 Why equations? 121
6.2 Difference equations defined 122
6.2.1 Initial conditions 125
6.2.2 Linear difference equations 126
6.2.3 Solving difference equations 126
6.3 Simulating difference equations 128
6.4 Review of the theory for linear equations 132
6.4.1 First-order homogenous equations 132
6.4.2 Second-order homogenous equations 133
6.4.3 Linear, inhomogenous equations 135
6.5 Simulation of difference equations and round-off errors 140
6.5.1 Explanation of example 6.25 142
6.5.2 Estimating the round-off unit 145
6.5.3 Round-off errors and second order, inhomogenous equations 146
6.6 Summary 151

7 Lossless Compression 153
7.1 Introduction 154
7.1.1 Run-length coding 155
7.2 Huffman coding 156
7.2.1 Binary trees 158
7.2.2 Huffman trees 159
7.2.3 The Huffman algorithm 160
7.2.4 Properties of Huffman trees 164
7.3 Probabilities and information entropy 167
7.3.1 Probabilities rather than frequencies 167
7.3.2 Information entropy 168
7.4 Arithmetic coding 171
7.4.1 Arithmetic coding basics 171
7.4.2 An algorithm for arithmetic coding 173
7.4.3 Properties of arithmetic coding 176
7.4.4 A decoding algorithm 179
7.4.5 Arithmetic coding in practice 181
7.5 Lempel-Ziv-Welch algorithm 183
7.6 Lossless compression programs 184
7.6.1 Compress 184
7.6.2 gzip 184

8 Digital Sound **185**
8.1 Sound . . . 185
8.1.1 Loudness: Sound pressure and decibels . . . 186
8.1.2 The pitch of a sound . . . 189
8.1.3 Any function is a sum of sin and cos . . . 191
8.2 Digital sound . . . 194
8.2.1 Sampling . . . 195
8.2.2 Limitations of digital audio: The sampling theorem . . . 195
8.2.3 Reconstructing the original signal . . . 197
8.3 Simple operations on digital sound . . . 199
8.4 More advanced sound processing . . . 204
8.4.1 The Discrete Cosine Transform . . . 204
8.5 Lossy compression of digital sound . . . 205
8.6 Psycho-acoustic models . . . 208
8.7 Digital audio formats . . . 209
8.7.1 Audio sampling — PCM . . . 209
8.7.2 Lossless formats . . . 210
8.7.3 Lossy formats . . . 211

III Functions **215**

9 Polynomial Interpolation **217**
9.1 The Taylor polynomial with remainder . . . 218
9.1.1 The Taylor polynomial . . . 218
9.1.2 The remainder . . . 220
9.2 Interpolation . . . 227
9.2.1 The interpolation problem . . . 227
9.2.2 The Newton form of the interpolating polynomial . . . 229
9.2.3 Evaluating the Newton form . . . 233
9.3 Summary . . . 237

10 Zeros of Functions **239**
10.1 The need for numerical root finding . . . 240
10.1.1 Analysing difference equations . . . 240
10.1.2 Labelling plots . . . 241
10.2 The Bisection method . . . 242
10.2.1 The intermediate value theorem . . . 242
10.2.2 Derivation of the Bisection method . . . 243
10.2.3 Error analysis . . . 246

10.2.4 Revised algorithm . . . 248
10.3 The Secant method . . . 252
10.3.1 Basic idea . . . 252
10.3.2 Testing for convergence . . . 254
10.3.3 Revised algorithm . . . 256
10.3.4 Convergence and convergence order of the Secant method . 257
10.4 Newton's method . . . 259
10.4.1 Basic idea . . . 260
10.4.2 Algorithm . . . 260
10.4.3 Convergence and convergence order . . . 262
10.5 Summary . . . 268

11 Numerical Differentiation **269**
11.1 Newton's difference quotient and its truncation error . . . 270
11.1.1 The basic idea . . . 270
11.1.2 The truncation error . . . 272
11.1.3 An upper bound on the truncation error . . . 274
11.2 The total error in Newton's difference quotient . . . 275
11.2.1 Round-off errors in the function values . . . 275
11.2.2 Round-off errors in the computed derivative . . . 276
11.2.3 An upper bound on the total error . . . 279
11.2.4 Optimal choice of h . . . 280
11.3 A general strategy for constructing differentiation methods . . . 283
11.4 Three differentiation methods . . . 285
11.4.1 A symmetric version of Newton's quotient . . . 285
11.4.2 A four-point differentiation method . . . 288
11.4.3 Numerical approximation of the second derivative . . . 290
11.4.4 Derivation of the method . . . 290

12 Numerical Integration **297**
12.1 What is the integral? . . . 298
12.1.1 Definition of the integral . . . 299
12.1.2 Numerical computation of the integral . . . 300
12.2 The midpoint rule for numerical integration . . . 301
12.2.1 A detailed algorithm . . . 302
12.2.2 The error . . . 304
12.2.3 Estimating the step length . . . 308
12.3 Two other numerical integration methods . . . 310
12.3.1 A strategy for designing integration methods . . . 310
12.3.2 The trapezoidal rule . . . 311

12.3.3 Simpson's rule . . . 313
12.4 Summary . . . 318

13 Numerical Solution of Differential Equations **321**
13.1 What are differential equations? . . . 321
13.1.1 An example from physics . . . 321
13.1.2 General use of differential equations . . . 323
13.1.3 Different types of differential equations . . . 324
13.2 First order differential equations . . . 325
13.2.1 Initial conditions . . . 326
13.2.2 A geometric interpretation of first order differential equations 327
13.2.3 Conditions that guarantee existence of one solution . . . 329
13.2.4 What is a numerical solution of a differential equation? . . . 330
13.3 Euler's method . . . 332
13.3.1 Basic idea and algorithm . . . 332
13.3.2 Geometric interpretation . . . 335
13.3.3 The error in Euler's method . . . 336
13.4 Midpoint Euler and other Runge-Kutta methods . . . 339
13.4.1 Euler's midpoint method . . . 339
13.4.2 The error . . . 342
13.4.3 Runge-Kutta methods . . . 342
13.5 Systems of differential equations . . . 347
13.5.1 Vector notation and existence of solution . . . 347
13.5.2 Numerical methods for systems of first order equations . . . 350
13.5.3 Higher order equations as systems of first order equations . 351
13.6 Final comments . . . 354

IV Functions of two variables **361**

14 Functions of two variables **363**
14.1 Basics . . . 363
14.1.1 Basic definitions . . . 363
14.1.2 Differentiation . . . 365
14.1.3 Vector functions of several variables . . . 367
14.2 Numerical differentiation . . . 368

15 Digital images and image formats **375**
15.1 What is an image? . . . 375
15.1.1 Light . . . 375
15.1.2 Digital output media . . . 376

15.1.3 Digital input media 377
15.1.4 Definition of digital image 377
15.1.5 Images as surfaces 379
15.2 Operations on images 381
15.2.1 Normalising the intensities 382
15.2.2 Extracting the different colours 382
15.2.3 Converting from colour to grey-level 382
15.2.4 Computing the negative image 385
15.2.5 Increasing the contrast 385
15.2.6 Smoothing an image 387
15.2.7 Detecting edges 388
15.2.8 Comparing the first derivatives 393
15.2.9 Second-order derivatives 393
15.3 Image formats 394
15.3.1 Raster graphics and vector graphics 395
15.3.2 Vector graphics formats 397
15.3.3 Raster graphics formats 397

Answers **401**

Solutions **427**

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?

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

$$\begin{aligned} 2x+4y-2z&=2,\\ 3x-6z&=3,\\ 4x-2y+4z&=2. \end{aligned}$$

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

$$\begin{aligned} 2+4z+4y-2z&=2,\\ 4+8z-2y+4z&=2, \end{aligned}$$

or

$$\begin{aligned} 4y+2z&=0,\\ -2y+12z&=-2. \end{aligned}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 using the operator = for assignment and == 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 semi-colon.

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 quantity, 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 $i = 1, 2, \ldots, 100$
    $s = s + i;$

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

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, ..., 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 $i \leq 100$
$s = s + i$;
$i = i + 1$;

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 $i = 1, 2, \ldots, 100$
    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

$$s = \sum_{i=1}^{100} a_i = a_1 + a_2 + \cdots + a_{100},$$

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

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

Here we have added '1' as a superscript to indicate that this is the first time we group terms in the sum together — it is the first time step of the algorithm. The key is that these partial sums, each with two terms, are independent of each other. In other words we may hire 50 people, give them two numbers each, and tell them to add their two numbers.

When everybody is finished, we can repeat this and ask 25 people to add the 50 results,

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

The superscript '2' here does not signify that the number in question should be squared; it simply means that this is the second time step of the algorithm.

At the next time 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 time 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,..., 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 time 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 2020) most powerful computer has 7 630 848 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 16 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 for Section 1.5

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.

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

3. Formulate an algorithm which describes how you multiply 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. This distinction is going to be important in what follows since we are going to discuss logical expressions which may involve tests for equality.

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 $p = \mathtt{T}$. 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 $p(2) = \mathtt{F}$ and $p(4) = \mathtt{T}$. 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 comparison 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 $\wedge$, $\vee$ and $\neg$. Let us consider each of these in turn.

The expression $\neg p$ 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 $p \wedge q$ 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 $p \vee q$ is only F if both p and q are F; otherwise it is T; see column five in table 2.2.

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \oplus q$
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 $\neg$ (not), $\wedge$ (and), $\vee$ (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 ($\vee$) which is T when at least one of p and q are 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 $\neg$, $\wedge$, $\vee$, 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 expressions 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.,*

$$\begin{aligned} p \wedge q = q \wedge p, &\quad (p \wedge q) \wedge r = p \wedge (q \wedge r), \\ p \vee q = q \vee p, &\quad (p \vee q) \vee r = p \vee (q \vee r), \\ p \oplus q = q \oplus p, &\quad (p \oplus q) \oplus r = p \oplus (q \oplus r). \end{aligned}$$

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 for Section 2.3

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 $(p \oplus q) \oplus r = p \oplus (q \oplus r)$.

2. Prove de Morgan's laws.

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.

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.

Computers were originally thought of as computing devices — machines that could do numerical computations quickly. Today most people use computers for surfing the web, reading email or for entertainment, but more than ever they are excellent number crunchers, capable of adding billions of numbers every second. And at the lowest level almost all operations in a computer can be thought of as operations on numbers.

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 $\mathbb{N}_0 = \{0, 1, 2, 3, 4, \dots\}$[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 $-n$, and we denote by $\mathbb{Z}$ the set of natural numbers augmented with all these negative numbers,

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

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. In order to define these terms precisely, we need to name the digits in a 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 whose leading and trailing zeros have been discarded. 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.*

This may look confusing, but a simple example should illuminate things: If $x = 3.14$, we have $d_0 = 3$, $d_{-1} = 1$, and $d_{-2} = 4$, with all other ds equal to zero. The integer part is 3 and the fractional part is 0.14.

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 $a \,/\!/\, b$ is the result obtained by dividing a by b and discarding the*

[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.

remainder (integer division)*. The number* $a\,\%\,b$ *is the remainder when* a *is divided by* b*.*

For example $3\,//\,2 = 1$, $9\,//\,4 = 2$ and $24\,//\,6 = 4$, 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 define the four intervals*

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

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

Exercises for Section 3.1

1. Mark each of the following statements as true or false:

(a). $a//b$ is always bigger than $a\%b$.

(b). (a, b) is a subset of $[a, b]$.

(c). The fractional part of π is 0.14.

(d). The integer part of π is 3.

2. Compare each number below with definition 3.1, and determine the values of the digits d_k, d_{k-1}, $\ldots$, d_0, d_{-1}, $\ldots$.

(a). $x = 10.5$.

(b). $x = 27.1828$.

(c). $x = 100.20$.

(d). 0.0.

3. Compute $a \,//\, b$ and $a \,\%\, b$ in the cases below.

(a). $a = 8$, $b = 3$.

(b). $a = 10$, $b = 2$.

(c). $a = -29$, $b = 7$.

(d). $a = 0$, $b = 1$.

4. Find a formula for computing the number of digits $k = f(x)$ to the left of the decimal point in a number x, see definition 3.1.

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 numeral system.

3.2.1 Alternative Numeral Systems

In the decimal system we 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

$$\begin{aligned}
3761 &= 13652_7 = 1 \times 7^4 + 3 \times 7^3 + 6 \times 7^2 + 5 \times 7^1 + 2 \times 7^0, \\
50 &= 101_7 = 1 \times 7^2 + 0 \times 7^1 + 1 \times 7^0, \\
100 &= 202_7 = 2 \times 7^2 + 0 \times 7^1 + 2 \times 7^0,
\end{aligned}$$

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 1. 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, $\dots$, $n_{\beta-1}$ be β distinct numerals (also called digits) such that n_i denotes the integer i. A natural number representation in base β is an ordered collection of digits $(d_k d_{k-1} \dots d_1 d_0)_\beta$ which is interpreted as the natural number*

$$d_k\beta^k + d_{k-1}\beta^{k-1} + d_{k-2}\beta^{k-2} + \cdots + d_1\beta^1 + d_0\beta^0 \tag{3.1}$$

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

The definition is not quite precise: In (3.1) each digit d_j should be interpreted as the integer represented by the digit, and not just the numeral, see the example below for $\beta = 16$.

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 $10, 11, \dots, \beta - 1$. 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 $n_{10} = \text{a}$, $n_{11} = \text{b}$, $n_{12} = \text{c}$, $n_{13} = \text{d}$, $n_{14} = \text{e}$ and $n_{15} = \text{f}$. This is called the *hexadecimal* numeral system and in this system the number 3761 becomes

$$\text{eb1}_{16} = \text{e} \times 16^2 + \text{b} \times 16^1 + 1 \times 16^0 = 14 \times 256 + 11 \times 16 + 1 = 3761.$$

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 idea: 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

$$3761 = (d_3 d_2 d_1 d_0)_8 = d_3 8^3 + d_2 8^2 + d_1 8 + d_0. \tag{3.2}$$

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 (3.2) 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

$$58 = d_3 8 + d_2 \tag{3.3}$$

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 (3.3) 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 only discuss representation of natural numbers in this section — negative numbers are simply represented by prefixing with $-$, just like for decimal numbers. For example, the decimal number -3761 is represented 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:*

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

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, ..., 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 $a \,//\, \beta$ back in a.

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

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:*

$i = 0$;
while $a > 0$
    $d_i = a \,\%\, \beta$;
    $a = a \,//\, \beta$;
    $i = i + 1$;

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

while $a > 0$
    $d = a \,\%\, \beta$;
    $a = a \,//\, \beta$;
    print d;

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

3.2.3 Tabular display of the conversion

The results produced by Algorithm 3.7 are 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.

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

3.2.4 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 $x = \mathrm{a}_{16}$. 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

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

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

$$\begin{aligned}1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3 = 13_{10} = \mathrm{d}_{16}, \\ 0 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3 = 12_{10} = \mathrm{c}_{16}.\end{aligned}$$

But then we have

$$\begin{aligned}x &= (1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3) + (0 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3)2^4 \\ &= \mathrm{d}_{16} \times 16^0 + 16^1 \times \mathrm{c}_{16} = \mathrm{cd}_{16}.\end{aligned}$$

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

$$1100_2 = \mathrm{c}_{16}, \quad 1101_2 = \mathrm{d}_{16}, \quad 1100\,1101_2 = \mathrm{cd}_{16}.$$

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 four successive binary digits into hexadecimal, starting with the least significant digits.*

Example 3.12. Let us convert the hexadecimal number $3\mathrm{c}5_{16}$ to binary. We have

$$\begin{aligned}5_{16} &= 0101_2, \\ \mathrm{c}_{16} &= 1100_2, \\ 3_{16} &= 0011_2,\end{aligned}$$

which means that $3\mathrm{c}5_{16} = 11\,1100\,0101_2$ 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). Many will perhaps do this by going via decimal representation, but all the conversions can be found in table 3.1.

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.

Exercises for Section 3.2

1. Mark each of the following statements as true or false:

(a). When numbers are represented in base β according to Definition 3.4, the number 1 is always written the same way.

(b). The number 16 can be written in exactly two different ways in base 7.

(c). In base 16, $af < ba$.

2. Convert the following natural numbers to the indicated bases:

(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. Convert to base-8:

(a). 1011001_2

(b). 110111_2

(c). 10101010_2

4. Convert to base-2:

(a). 44_8

(b). 100_8

(c). 327_8

5. Convert to base-16:

(a). 1001101_2

(b). 1100_2

(c). 10100111100100_2

(d). 0.0101100101_2

(e). 0.00000101001_2

(f). 0.111111111_2

The fractional number notation in the last three of these problems will be explained in the start of the next section.

6. Convert to base-2:

(a). $3c_{16}$

(b). 100_{16}

(c). $e51_{16}$

(d). $0.0aa_{16}$

(e). 0.001_{16}

(f). $0.f01_{16}$

7. Conversion of special numbers.

(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 $a = 10_\beta$?

8. Conversion of special numbers.

(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 $a = 1000_\beta$?

3.3 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, ..., $n_{\beta-1}$ be β distinct numerals (also called digits) such that n_i denotes the number i. A fractional representation in base β is a finite or infinite, ordered collection of digits $(0.d_{-1}d_{-2}d_{-3}\dots)_\beta$ which is interpreted as the real number*

$$d_{-1}\beta^{-1} + d_{-2}\beta^{-2} + d_{-3}\beta^{-3} + \cdots \tag{3.4}$$

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

As in the representation of integers, the numerals in (3.4) should be interpreted as the integer they represent.

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 (3.4). 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 $i = -1, -2, \dots$. 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

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

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* (3.4) *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 $0.99999999999999\dots$ (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.

3.3.2 Conversion of fractional numbers

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

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

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

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

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 (3.6)

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

This equation has the same form as (3.5) and can be used to determine d_{-2}. We multiply both sides of (3.7) by 8,

$$\frac{24}{5} = d_{-2} + d_{-3}8^{-1} + d_{-4}8^{-3} + \cdots. \tag{3.8}$$

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 (3.8) and obtain

$$\frac{4}{5} = d_{-3}8^{-1} + d_{-4}8^{-2} + d_{-5}8^{-3} + \cdots. \tag{3.9}$$

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 (3.5), 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 3.8, 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 (3.8), 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 (3.5) come as close to $1/5$ as we wish by including sufficiently many terms.

Set $a = 1/5$. Then equation (3.7) can be written

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

while (3.9) 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. ∎

3.3.3 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 46 to 47 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,

$$\begin{array}{r|cc} 1/5 & 1 & (8/5) \\ 3/5 & 4 & (24/5) \\ 4/5 & 6 & (32/5) \\ 2/5 & 3 & (16/5) \\ 1/5 & 1 & (8/5) \end{array}$$

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

3.3.4 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

$$x = 16^{-1}13 + 16^{-2}7 = 0.\mathrm{d}7_{16}.$$

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 $x = 0.3\mathrm{a}8_{16}$ to binary. From table 3.1 we find

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

which means that

$$0.3\mathrm{a}8_{16} = 0.0011\ 1010\ 1000_2 = 0.0011\ 1010\ 1_2.$$

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

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

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

$$0.1100\ 1001\ 0110\ 1_2 = 0.\mathrm{c}968_{16}.$$

3.3.5 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 $(b*\beta)\,/\!/\,c$. 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

$$b = (b*\beta)\,\%\,c. \tag{3.10}$$

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 $i = -1, -2, \ldots, -k$
  $d_i = (b*\beta)\,/\!/\,c$;
  $b = (b*\beta)\,\%\,c$;

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, ..., $c-1$. 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, ... Since the numerals d_i are computed from b, they must repeat with the same frequency. Note however that there 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 statements 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 7.

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 8.

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

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

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.

Exercises for Section 3.3

1. Mark each of the following statements as true or false:

(a). The number 10_β is greater in base $\beta = 10$ than in base $\beta = 9$.

(b). The number 0.1_β is greater in base $\beta = 10$ than in base $\beta = 9$.

(c). The number 17_β is always prime, regardless of the value of β.

(d). The number

$$\frac{\ln\sqrt{e^\pi}}{\pi}$$

is a rational number.

2. (Mid-way exam 2010) For which base β is it possible to represent the rational number $2/3$ with a finite sequence of digits?

☐ $\beta = 2$

☐ $\beta = 4$

☐ $\beta = 10$

☐ $\beta = 6$

3. 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

4. Convert π to base-9.

5. Special rational numbers.

(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?

6. If $a = b/c$ is a rational number, what is the maximum length of the repeating sequence?

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

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

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

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

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

$$\begin{array}{r} \scriptstyle 8\,8 \\ \not{3}\not{2}1_8 \\ -177_8 \\ \hline =122_8 \end{array}$$

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

$$\begin{array}{l} \quad 312_4 \times 12_4 \\ \hline \;\; 1230_4 \\ \;\; 312_4 \\ \hline 11010_4 \end{array}$$

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 for Section 3.4

1. Find the correct alternative in the following multiple choice exercises.

(a). The equation $7_\beta + 8_\beta = 13_\beta$ is true in which numeral system?

☐ base 10

☐ base 11

☐ base 12

☐ base 13

(b). Which of the following equations is true in base 9?

☐ 4 + 5 = 11
☐ 3 + 3 = 7
☐ 5 + 5 = 11
☐ 11 - 3 = 5

(c). (Mid-way exam 2009) In the numeral system with base $\beta = 8$, the decimal number 40.125 becomes

☐ 40.1_8
☐ 50.3_8
☐ 40.11_8
☐ 50.1_8

2. 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. 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). $3_7 - 11_7$

(g). -5_7

4. 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). $101_2 \cdot 11_2$

5. In this exercise we will consider an alternative method for division by a natural number greater than or equal to 2. The method consists of the following simple algorithm:

1. Write the dividend as a number in the base of the divisor.
2. Carry out the division.
3. Convert the quotient back to base-10

(a). Use the method described above to carry out the following divisions:

I) 49/7

II) 365/8

III) 4720/16

(b). Formulate a general algorithm for this method of division for a given dividend a and a given divisor β.

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 10 yottabytes by 2022.

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

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

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 left-most bit, so -1 would be 1001. Adding numbers should just correspond to addition in the binary system. It turns out that this leads to some problems. Consider first what happens if we compute $-1+1$ using the normal rule of addition. In the computer this would become

$$1001 + 0001 = 1000. \tag{4.1}$$

This result should obviously be treated as 0, so both 1000 and 0000 must represent zero. Consider next the addition

$$0111 + 0001 = 1000$$

which corresponds to adding 1 to the largest positive number. The result is the same as in (4.1), i.e., negative 0, which is definitely troublesome. These two examples show that the naive representation of integers leads to complications.

The actual representation used in most computers avoids this by making 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.

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.

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*

$$x^- = 2^n - |x|, \tag{4.2}$$

where n is the total number of bits in the representation. Numbers are added using the normal rules of arithmetic in the binary numeral system.

Example 4.4. Let us find the representation of $x = -6$ in two's complement from (4.2) 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.

Exercises for Section 4.1

1. Find the correct alternative in the following multiple choice exercises.

(a). Suppose we use the method of Two's complement to store integers with 8 bits of information. What would be the largest positive integer we would be able to store?

☐ 128
☐ 256
☐ 127
☐ 255

(b). Suppose we execute the following code in Java, what would be the output?

```
int a = 1;
int ap = 0;
while (a > ap) {
        a = a + 1;
        ap = ap + 1;
}
System.out.println(a);
```

☐ Nothing, the code would loop forever, or until the machine is out of memory.
☐ a = 2147483647
☐ a = -2147483648
☐ a = 0

(c). What would happen if you were to execute a code similar to the one in b), in Python?

☐ Nothing, the code would loop forever, or until the machine is out of memory.
☐ $a = 2147483647$
☐ $a = -2147483648$
☐ $a = 0$

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?

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?

4.2 Computers and real numbers

Computations with integers are 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.

4.2.1 The challenge of real numbers

To illustrate the challenge, consider the two real numbers

$$\pi = 3.141592653589793238462643383279502884197\ldots,$$
$$10^6\pi = 3.141592653589793238462643383279502884197\ldots \times 10^6.$$

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 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*.

4.2.2 The normalised form 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 nu-

meral 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 (Normalised form of real number). *Let a be a real number different from zero. Then a can be written uniquely as*

$$a = b \times 10^n \tag{4.3}$$

where b is bounded by

$$\frac{1}{10} \leq |b| < 1 \tag{4.4}$$

and n is an integer. This is called the normalised form of a, and the number b is called the significand while n is called the exponent of a. The normalised form of 0 is $0 = 0 \times 10^0$.

Note that the digits of a and b are the same; to arrive at the normalised form in (4.3) we simply multiply a by the power of 10 that brings b into the range given by (4.4).

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

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

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

$$\begin{aligned}
&-0.9999 \times 10^9, \qquad -0.1000 \times 10^{-9}, \\
&\ \ 0.1000 \times 10^{-9}, \qquad\ \ 0.9999 \times 10^9.
\end{aligned}$$

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

Observation 4.6 (Binary normalised 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 normalised form of a, and the number b is called the significand while n is called the exponent of a. The normalised 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.

4.2.3 32-bit floating-point numbers

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 normalised form, the first bit of the significand will always be 1 (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}.

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

4.2.4 Special bit combinations

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`.

4.2.5 64-bit floating-point numbers

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`.

4.2.6 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.

4.2.7 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.

Exercises for Section 4.2

1. Which of the following statements are true and which are false?

(a). Any integer that can be represented in the 32 bit integer format, can also be represented exactly in the 32-bit floating-point format.

(b). The distance between two neighbouring numbers in the IEEE 32-bit floating-point format is always 1.4×10^{-45}.

(c). When you add two sufficiently large floating-point numbers you will get overflow, and the result will be a large negative number.

2. We are going to write e in decimal, normalised form, with 4 digits for the significand. Which of the following is the correct normalised form?

☐ $e \approx 2.718 \times 10^0$

☐ $e \approx 2.7181 \times 10^0$

☐ $e \approx 0.2718 \times 10^1$

☐ $e \approx 0.27181 \times 10^1$

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

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.

5. Write the following numbers (or approximations of them) in normalised form, using both 4 and 8 digits

(a). 4752735

(b). $602214179 * 10^{15}$

(c). 0.00008617343

(d). 9.81

(e). 385252

(f). $e^{10\pi}$

6. Redo exercise 5d, but write the number in binary normalised form.

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.

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

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.*

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 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

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	ˆ	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'.

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.

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.

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 *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 equip-

ment 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 (ˆD) has the description 'End of Transmission' and is often 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 ˆD (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 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

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	Ö	247	f7	ö
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.

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 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 ensures that none of these bytes start with 0. This is accomplished 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 $c = (d_6d_5d_4d_3d_2d_1d_0)_2$ is in the decimal range 0–127 (hexadecimal 00_{16}–$7f_{16}$), it is encoded in one byte as*

$$0d_6d_5d_4d_3d_2d_1d_0. \tag{4.5}$$

2. *If $c = (d_{10}d_9d_8d_7d_6d_5d_4d_3d_2d_1d_0)_2$ is in the decimal range 128–2047 (hexadecimal 80_{16}–$7ff_{16}$) it is encoded as the two-byte binary number*

$$110d_{10}d_9d_8d_7d_6 \quad 10d_5d_4d_3d_2d_1d_0. \tag{4.6}$$

3. *If $c = (d_{15}d_{14}d_{13}d_{12}d_{11}d_{10}d_9d_8d_7d_6d_5d_4d_3d_2d_1d_0)_2$ is in the decimal range 2048–65535 (hexadecimal 800_{16}–$ffff_{16}$) it is encoded as the three-byte binary number*

$$1110d_{15}d_{14}d_{13}d_{12} \quad 10d_{11}d_{10}d_9d_8d_7d_6 \quad 10d_5d_4d_3d_2d_1d_0. \tag{4.7}$$

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

4. *If* $c = (d_{20}d_{19}d_{18}d_{17}d_{16}d_{15}d_{14}d_{13}d_{12}d_{11}d_{10}d_9d_8d_7d_6d_5d_4d_3d_2d_1d_0)_2$ *is in the decimal range 65536–1114111 (hexadecimal* 10000_{16}*–*$10ffff_{16}$*) it is encoded as the four-byte binary number*

$$11110d_{20}d_{19}d_{18} \quad 10d_{17}d_{16}d_{15}d_{14}d_{13}d_{12} \quad 10d_{11}d_{10}d_9d_8d_7d_6 \quad 10d_5d_4d_3d_2d_1d_0. \qquad (4.8)$$

This may seem complicated at first sight, but is in fact quite simple and elegant. We note that 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

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

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 (4.6) 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 $c385_{16}$.

4.3.5 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). *In UTF-16, a Unicode character with code point c is encoded according to the 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 four 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.

Exercises for Section 4.3

1. Which of the following statements are true and which are false?

(a). A text encoded in ASCII will display correctly if it is treated as a text encoded in ISO Latin 1.

(b). A text encoded in ISO Latin 1 will display correctly if it is treated as a text encoded in ASCII.

(c). The UTF-8 code "11101101 10001101 10101100 01110001" corresponds to a sequence of 4 characters.

(d). A text written with Norwegian letters (standard Latin alphabet, with the addition of the letters 'æ', 'ø', and 'å') encoded in UTF-8 will sometimes require more space than if encoded in ISO Latin 1.

2. Find the correct alternative in the following multiple choice exercises.

(a). (Exam 2007) In a text there are three different symbols, encoded with one of the methods for representation of text discussed in this section. It turns out that one symbol is represented with one byte, another with two bytes and the third one with three bytes. Which of the following statements is correct?

☐ The symbols could have been encoded with ASCII

☐ The symbols could have been encoded with UTF-8

☐ The symbols could have been encoded with UTF-16

☐ The symbols could have been encoded with ISO-Latin 1.

(b). (Continuation exam 2007) In which of the following encoding schemes is the Norwegian letter 'ø' encoded with two bytes?

☐ ASCII

☐ UTF-8

☐ ISO Latin 1

☐ UTF-32

(c). (Continuation exam 2011) A text is stored in 4 different files with the encoding schemes ISO Latin 1, UTF-8, UTF-16, UTF-32. Which of the following statements is then true?

☐ The file encoded with ISO-Latin 1 will be the largest.

☐ The file encoded with UTF-8 will be the largest.

☐ The file encoded with UTF-16 will be the largest.

☐ The file encoded with UTF-32 will be the largest.

3. 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. Determine the UTF-16 encodings of the Unicode characters in exercise 3.

5. 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.

6. Encoding your name.

(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).

7. In this exercise you are going to derive an algorithm to find the nth character in a UTF-8 encoded file. We assume that the file is given as an array of 8-bit integers, i.e., each integer is in the range 0–255.

(a). The UTF-8 encoding scheme makes use of a variable bit length (see Fact 4.10). In order to determine whether a byte represents the start of a new character, one must determine in which integer interval the byte lies.

Determine which integer intervals denote a new character and which intervals denote the continuation of a character code.

(b). Derive an algorithm for finding the nth character. Your algorithm may begin as follows:

$counter = 0$
while $counter < n$:
  get a new byte
  if $byte$ **is in** correct interval:
    act accordingly
  else:
    act accordingly

(c). In this exercise you are going to write a Python script for finding the n-th sign in an UTF-8 encoded file. You may use this layout:

```
# Coding: utf-8


# This part of the program creates a list 'bytelist' where each
# element is the value of the corresponding byte in the file.

infile = open('filename', 'rb')
bytelist = []
while True:
        byte = infile.read(1)
        if not byte:
                break
        bytelist.append(ord(byte))



# Your code goes here. The code should run through the byte list
# and match each sequence of bytes to a corresponding character.
# When you reach the n-th character, you should compute the code
# point, store it in the variable 'character' and print it.

character = unichr(character)
print(character)
```

(d). Repeat the exercise for the UTF-16 encoding scheme

8. You are given the following hexadecimal code:

$$41\ 42\ 43\ 44\ 45_{16}.$$

Determine whether it is a text encoded in UTF-8 or UTF-16.

9. You are given the following binary code:

11000101 10010011 11000011 10111000

What does this code translate to if it is a text encoded in

(a). UTF-8?

(b). UTF-16?

10. The following sequence of bytes represent a UTF-8 encoded text, but some of the bytes are wrong. Find which ones they are, and explain why they are wrong:

41 C3 98 41 C3 41 41 C3 98 98 41.

11. The $\log_2$-function is defined by the relation

$$2^{\log_2 x} = x.$$

Use this definition to show that

$$\log_2 x = \frac{\ln x}{\ln 2}.$$

12. In this exercise, we want to create a fixed length encoding scheme for simple Norwegian text. Our set of characters must include the 29 uppercase Norwegian letters, a space character and a period character.

(a). How many bits are needed per character to represent this character set?

(b). Now assume that we also want to add the lowercase letters, the numbers 0-9 and a more complete set of notation characters, so that the total number of characters is 113. How many bits are now needed per character?

(c). How many bits are needed per character in a fixed length encoding scheme with n unique characters? (Hint: The result in exercise 11 might be useful.)

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 just 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 the 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 on page 85.

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 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 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

[5]As we have seen earlier integers are in fact stored in two's complement.

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 for Section 4.5

1. Determine the details of the mp3 file-format by searching the web. A possible starting point is provided by the url

`http://mpgedit.org/mpgedit/mpeg_format/mpeghdr.htm`.

2. 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.).

CHAPTER 5

Computer Arithmetic and 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*.

As we saw in Chapter 4, the standard representation of non-integer numbers in computers 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. The key observation is that subtraction of two almost equal numbers may lead to large round-off error. There is nothing mysterious about this — it is a simple consequence of how arithmetic is performed with floating-point numbers. We will therefore need a quick introduction to floating-point arithmetic. We will also discuss the

two most common ways to measure error, namely *absolute error* and *relative error*.

5.1 Integer arithmetic and errors

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. This 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.

5.2 Floating-point arithmetic and round-off error

Errors in floating-point arithmetic are more subtle than errors in integer arithmetic since, 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. A simple example will illustrate.

Example 5.1. On a typical calculator we compute $x = \sqrt{2}$, then $y = x^2$, and finally $z = y - 2$, i.e., the result should be $z = \left(\sqrt{2}\right)^2 - 2$, which of course is 0. The result reported by the calculator is

$$z = -1.38032020120975 \times 10^{-16}.$$

This is a simple example of round-off error.

The aim of this section is to explain why computations like the one in example 5.1 give this obviously wrong result. To do this we will give a very high-level introduction to computer arithmetic and discuss the potential for errors in

the four elementary arithmetic 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.

5.2.1 Truncation and rounding

Floating point numbers on most computers use binary representation, and we know that in the binary numeral system all real numbers that are fractions on the form a/b, with a an integer and $b = 2^k$ for some positive integer k can be represented exactly (provided a and b are not too large), 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 3.43 are *not* represented exactly. And of course all numbers that cannot be represented exactly in the decimal system cannot be represented exactly in the binary system either. These numbers include fractions like 1/3 and 5/12 as well as all irrational numbers. Even before we start doing arithmetic we therefore have the challenge of finding good approximations to these numbers that cannot be represented exactly within the floating-point model being used. We will distinguish between two different ways to do this, *truncation* and *rounding*.

Definition 5.2 (Truncation). *A number is said to be truncated to m digits when each digit except the m leftmost ones is replaced by* 0.

Example 5.3 (Examples of truncation). The decimal number 0.33333333 truncated to 4 digits is 0.3333, while 128.4 truncated to 2 digits is 120, and 0.67899 truncated to 4 digits is 0.6789.

Note that truncating a positive number a to an integer is equivalent to applying the floor function to a, i.e., if the result is b then

$$b = \lfloor a \rfloor.$$

Truncation is a very simple way to convert any real number to a floating-point number: We just start computing the digits of the number, and stop as soon as we have all the required digits. However, the error will often be much larger than necessary, as in the last example above. *Rounding* is an alternative to truncation that in general will make the error smaller, but is more complicated.

Definition 5.4 (Rounding). *A number is said to be rounded to m digits when it is replaced by the nearest number with the property that all digits beyond position m is* 0.

Example 5.5 (Examples of rounding). When the decimal number 0.33333333 is rounded to 4 digits the result is 0.3333. The result of rounding 128.4 to 2 digits is 130, while 0.67899 rounded to 4 digits is 0.6790.

Rounding is something most of us do regularly when we go shopping, as well as in many other contexts. However, there is one slight ambiguity in the definition of rounding: What to do when the given number is halfway between two m digit numbers. The standard rule taught in school is to *round up* in such situations. For example, we would usually round 1.15 to 2 digits as 1.2, but 1.1 would give the same error. For our purposes this is ok, but from a statistical point of view it is biased since we always round up when in doubt.

5.2.2 A simplified model for computer arithmetic

The details of computer arithmetic are technical, but luckily the essential features of both the arithmetic and the round-off errors are present if we use the same model of floating-point numbers as in section 4.2. Recall that any positive 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 5.6 (Simplified 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^{-5}. 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 7.

In the normalised, positive real number $a = \alpha \times 10^n$, the integer n provides information about the size of a, while α provides the decimal digits of a, as a fractional number in the interval $[0.1, 1)$. In our simplified arithmetic model, the restriction that n should only have 1 decimal digit restricts the *size* of a, while the requirement that α should have at most 4 decimal digits restricts the *precision* of a.

It is easy to realise that even simple operations may lead to numbers that exceed the maximum size imposed by the floating-point model — just consider a multiplication like $10^8 \times 10^7$. This kind of problem is easy to detect and is therefore not serious. The main challenge with floating-point numbers lies in keeping track of the number of correct digits in the significand. If you are presented with a result like 0.4521×10^1 it is reasonable to expect the 4 digits 4521 to be correct. However, it may well happen that only some of the first digits are correct. It may even happen that none of the digits reported in the result are correct. If the computer told us how many of the digits were correct, this would not be so serious, but in most situations you will get no indication that some of the digits are incorrect. Later in this section, and in later chapters, we will identify some simple situations where digits are lost.

Observation 5.7. *The critical part of floating-point operations is the potential loss of correct digits in the significand.*

5.2.3 An algorithm for floating-point addition

In order to understand how round-off errors occur in addition and subtraction, we need to understand the basic algorithm that is used. 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 5.8. *To add two floating-point numbers a and b on a computer, the following steps are performed:*

1. *The number with largest absolute value, say a, 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 5.9 (Standard case). Suppose that $a = 5.645$ and $b = 7.821$. We convert the numbers to normalised 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 normalised form. In exact arithmetic this would yield the result 0.13466×10^2. However, this is not in normalised form since the significand has five digits. We therefore perform rounding, $0.13466 \approx 0.1347$, and get the final result

$$0.1347 \times 10^2.$$

Example 5.9 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 5.10 (One large and one small number). If $a = 42.34$ and $b = 0.0033$ we convert the largest number to normalised 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

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

The error in example 5.10 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.

Example 5.11 (Subtraction of two similar numbers I). Consider next the case where $a = 10.34$ and $b = -10.27$ have opposite signs. We first rewrite the numbers in normalised 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 normalised form and obtain

$$a + b = 0.0007 \times 10^2 = 0.7000 \times 10^{-1}. \tag{5.1}$$

Example 5.11 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 5.12 (Subtraction of two similar numbers II). Suppose that $a = 10/7$ and $b = -1.42$. Conversion to normalised 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 normalised form, the result is

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

while the true result rounded to four correct digits is

$$0.8571 \times 10^{-3}. \tag{5.2}$$

5.2.4 Observations on round-off errors in addition/subtraction

In example 5.12 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 5.11 where we also had to add digits in the last step (5.1), but there we were lucky in that the added digits were correct. In example 5.12 however, there would be no way for a computer to know that the additional digits to be added in (5.2) should be taken from the decimal expansion of 10/7. Note how the bad loss of accuracy in example 5.12 is reflected in the relative error.

One could hope that what happened in example 5.12 is exceptional; after all example 5.11 worked out very well. This is not the case. It is example 5.11 that is exceptional since we happened to add the correct digits to the result in (5.1). This was because the numbers involved could be represented exactly in our decimal model. In example 5.12 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 5.13. *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 5.8.*

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 5.13 is just as problematic when the numbers are represented as normalised binary numbers.

Example 5.10 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 5.14. *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*

$$|\epsilon| < 0.5\beta^{-m}|a|. \tag{5.3}$$

The exact factor to be used on the right in (5.3) 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 that is apparent 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

if $a == \tilde{a}$

since it is rather unlikely that this will be true.

One situation where this problem may occur is in a loop like

```
x = 0.0;
while x ≤ 1.0
    print x
    x = x + 0.1;
```

What happens here 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.

Convince yourself that, for floating-point numbers a and b, the two computations $a+b$ and $b+a$ should give the same result. In other words, addition of floating-point numbers satisfies the commutative law. It is interesting to consider whether other laws also hold when using floating-point arithmetic. Exercise 10 explores this for the distributive and associative laws.

5.2.5 Multiplication and division of floating-point numbers

Multiplication and division of floating-point numbers is straightforward. Perhaps surprisingly, these operations are not susceptible to round-off errors. As for addition, both the procedures for performing the operations, and the effect of round-off error, is most easily illustrated by some examples.

Example 5.15. 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. The result is the floating-point number

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

i.e., the number -159.3.

Let us also consider an example of division.

Example 5.16. We use the same numbers as in example 5.15, 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

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

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 5.17. *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 5.17 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 5.17 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 5.17 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. If the absolute value of a result becomes too large for the floating-point standard, *overflow* occurs. This is indicated by the result receiving the value `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`.

5.2.6 The IEEE standard for floating-point arithmetic

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

[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*.

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.

One should be aware of the fact the IEEE standard is extensive and complicated, and far from all computers and programming languages support the full 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.

Exercises for Section 5.2

1. Mark each of the following statements as true or false.

(a). The number 8.73 truncated to an integer is 9.

(b). The method taught in school of rounding decimal numbers to the nearest integer and rounding 0.5 up to 1 gives the smallest statistical error possible.

(c). Rounding will always give a result which is at least as large as the result of truncation.

2. (Mid-term 2011) Which of the following expressions may give large errors for at least one value of x, when calculated on a machine using floating point arithmetic?

☐ $x^4 + 2$

☐ $x^2 + x^4$

☐ $a = x/(1 + x^2)$

☐ $1/2 + \sin(-x^2)$

3. Rounding and truncation.

(a). Round 1.2386 to 1 digit.

(b). Round 85.001 to 1 digit.

(c). Round 100 to 1 digit.

(d). Round 10.473 to 3 digits.

(e). Truncate 10.473 to 3 digits.

(f). Round 4525 to 3 digits.

4. Try to describe a rounding rule that is symmetric, i.e., it has no statistical bias.

5. The earliest programming languages would provide only the method of truncation for rounding non-integer numbers. This can lead sometimes lead to large errors as 2.000 - 0.001 = 1.999 would be rounded of to 1 if truncated to an integer. Express rounding a number to the nearest integer in terms of truncation.

6. Use the floating-point model defined in this chapter with 4 digits for the significand and 1 digit for the exponent, and use algorithm 5.8 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 $x = 0.1$.

7. Floating-point models for other bases.

(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.

8. 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

9. 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.

10. Rounding and different laws.

(a). A fundamental property of real numbers is given by the distributive law

$$(x+y)z = xz + yz. \tag{5.4}$$

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 (5.4) 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 (b) and (c), but test the associative and commutative laws for multiplication instead.

5.3 Measuring the error

In the previous section we saw that errors usually occur during floating point computations, and we therefore need a way to measure this 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, the *absolute error* and the *relative error*.

5.3.1 Absolute error

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

Definition 5.18 (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.

5.3.2 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 5.19 (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 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.

5.3.3 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 5.20 (Relative error and significant digits). *Let a be a nonzero number and suppose that $\tilde{a}$ is an approximation to a with relative error*

$$r = \frac{|a - \tilde{a}|}{|a|} \approx 10^{-m} \tag{5.5}$$

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

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 $1 \leq \alpha < 10$ 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 (5.5). If we cancel the common factor 10^n and multiply by α, we obtain

$$|\alpha - \tilde{\alpha}| \approx \alpha \times 10^{-m}. \tag{5.6}$$

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} (but note that the fourth digit is only off by one unit). Similarly,

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.

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 stated in Observation 5.20 is just a rule of thumb. If we go back to the 'proof'of observation 5.20, we note that as the left-most digit in a (and α) becomes larger, the right-hand side of (5.6) 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 5.20 is a convenient rule of thumb.

Observation 5.20 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

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

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 property. It can be shown that if $\tilde{a}$ is an approximation to a with small relative error, then a is also an approximation to $\tilde{a}$ with small relative error.

5.3.4 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. From observation 5.20 it is not suprising that the relative error is a good tool for measuring this error.

Lemma 5.21. *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*

$$\frac{|a-\tilde{a}|}{|a|} \leq \frac{1}{2}\beta^{-m+1}. \tag{5.7}$$

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 $(\beta/2)\beta^{-m-1} = \beta^{-m}/2$, see exercise 7. Since we also have $|\alpha| \geq \beta^{-1}$, an argument completely analogous to the one in the decimal case proves (5.7). ∎

Note that lemma 5.21 states what we already know, namely that a and $\tilde{a}$ have roughly $m-1$ or m digits in common.

Exercises for Section 5.3

1. Mark each of the following statements as true or false.

(a). The absolute error is always larger than the relative error.

(b). If the relative error is 0, then the absolute error is also 0.

(c). If $\bar{a}$ and $\tilde{a}$ are two different approximations to a, with relative errors ϵ_1 and ϵ_2, then the relative error of $\bar{a}$ when compared to $\tilde{a}$ is less than or equal to $\epsilon_1 + \epsilon_2$.

(d). If $\bar{a}$ and $\tilde{a}$ are two different approximations to a, with absolute errors ϵ_1 and ϵ_2, then the absolute error of $\bar{a}$ when compared to $\tilde{a}$ is less than or equal to $\epsilon_1 + \epsilon_2$.

2. 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 5.20.

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

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

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

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

3. Compute the relative error of a with regard to ã for the examples in exercise 2, and check whether the two errors are comparable as suggested in the last sentence in section 5.3.3.

4. Compute the relative errors in examples 5.9–5.12, and check whether observation 5.20 is valid in these cases.

5. The Vancouver stock exchange devised a short-lived index (weighted average of the value of a group of stocks). At its inception in 1982, the index was given a value of 1000.000. After 22 months of recomputing the index and truncating to three decimal places at each change in market value, the index stood at 524.881, despite the fact that its 'true' value should have been 1009.811. Find the relative error in the stock exchange value.

6. In the 1991 Gulf War, the Patriot missile defence system failed due to round-off errors. The troubles stemmed from computers performing the tracking calculations. The computer's internal clock generated floating point time values where the unit used was $1/10$ second. In the program used these were converted to seconds, by multiplying the values by 0.1, with the arithmetic carried out in binary.

(a). Show that

$$\frac{1}{10} = 0.0001100110011001100110 0\ldots,$$

where 1100 repeats indefinitely. Explain from this that, when $1/10$ is truncated to the first 23 bits, the result is $0.1 \times (1 - 2^{-20})$, i.e. an absolute error of 0.1×2^{-20}.

The software in the Patriot missile defence system performed the truncation in (a).

(b). Find the relative error when $1/10$ is truncated to the first 23 bits.

(c). The computer continuously converted time to seconds. If i_t is the integer increment of the internal clock (i.e. in tenths of a second), the following naive code was run:

$c = 0.00011001100110011001100_2$
while not ready
    $t = t + i_t * c;$

Find the accumulated error after 1 hour.

(d). Find an alternative algorithm that avoids the accumulation of round-off error.

After the system had been running for 100 hours, an error of 0.3433 seconds had accumulated. The problem was that the Patriot system worked by sending several radar pulses, and the position of the incoming missile was computed by subtracting times of different radar pulses sent towards the missile. For some of these pulses the roundoff error in time had been done, for others not. This led to a wrong calculation of the position of incoming missiles, and the system failed to shoot them down. As a result, an Iraqi Scud missile could not be targeted and was allowed to detonate on a barracks, killing 28 people.

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 result can be evaluated with an alternative formula that is not problematic. In this section we will consider some examples of this.

Example 5.22. Suppose we are going to evaluate the expression

$$\frac{1}{\sqrt{x^2+1}-x} \tag{5.8}$$

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,

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

Even with 64-bit floating-point numbers the square root will therefore be computed as 10^8, so the denominator in (5.8) 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 5.22 may seem very special, but it it is not unusual. Here is another example.

Example 5.23. For most values of x, the expression

$$\frac{1}{\cos^2 x - \sin^2 x} \tag{5.9}$$

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 (5.9) 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.

Exercises for Section 5.4

1. Find the correct alternative in the following multiple choice exercises.

(a). (Mid-term 2011) The number

$$\frac{5-\sqrt{5}}{5+\sqrt{5}} + \frac{\sqrt{5}}{2}$$

is the same as

☐ $\sqrt{5}$
☐ $1+\sqrt{5}$
☐ 1
☐ $3/2$

(b). (Mid-term 2011) Only one of the following statements is true, which one?

☐ Computers will never give round off errors as long as we use positive numbers.
☐ There is no limit to how lagre numbers we can work with on a given computer.
☐ With 64-bit integers we van represent numbers of size up to 2^{65}.
☐ We can represent larger numbers with 64-bit floating point numbers than with 64-bit integers.

2. 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). $\cos^2 x - \sin^2 x$.

3. 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.

4. The binomial coefficient $\binom{n}{i}$ is defined as

$$\binom{n}{i} = \frac{n!}{i!\,(n-i)!} \tag{5.10}$$

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 (5.10) 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 (5.10) 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 (5.10) 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 (5.10) 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 (5.10), 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).

Part II

Sequences of Numbers

CHAPTER 6

Numerical Simulation of Difference Equations

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, or *simulated*, on a computer, and in section 6.3 we will see how this can be done.

In chapter 5 we saw how real numbers can be approximated by floating-point numbers, and how the limitations inherent in the floating-point standard sometimes may cause dramatic errors. In section 6.5 we will see how round-off errors affect the simulation of even the simplest 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.

Exercises for Section 6.1

1. One of the oldest known age puzzles is known as Diophantus' riddle. It comes from the Greek Anthology, a collection of puzzles compiled by Metrodorus of Chios in about 500 AD. The puzzle claims to tell how long Diophantus lived in the form of a riddle engraved on his tombstone:

> God vouchsafed that he should be a boy for the sixth part of his life; when a twelfth was added, his cheeks acquired a beard; He kindled for him the light of marriage after a seventh, and in the fifth year after his marriage He granted him a son. Alas! late-begotten and miserable child, when he had reached the measure of half his father's life, the chill grave took him. After consoling his grief by this science of numbers for four years, he reached the end of his life.

How old were Diophantus and his son at the end of their lives?

6.2 Difference equations defined

The unknown variable in a difference equation is a *sequence of numbers*, rather than just a single number, and the difference equation describes a relation that is required to hold between the terms of the unknown sequence. Difference equations therefore 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, i.e., the relation between the terms of the unknown sequence, is obtained from known principles, and then the equation is solved by a mathematical or numerical method.

Example 6.1. 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 100 000 (in your favourite currency) at time $n = 0$, we have an *initial condition* $x_0 = 100\,000$. This gives the complete model

$$x_{n+1} = 1.05x_n, \quad x_0 = 100\,000. \tag{6.2}$$

This is an example of a *first-order* difference equation with an initial condition. From the information in (6.2) we can compute the values of x_n for $n \geq 0$. If we set $n = 0$, we find

$$x_1 = 1.05x_0 = 1.05 \times 100\,000 = 105\,000.$$

We can then set $n = 1$ and obtain

$$x_2 = 1.05x_1 = 1.05 \times 105\,000 = 110\,250.$$

These computations can clearly be continued for as long as we wish, and in this way we can compute the value of x_n for any positive integer n. For example, we find that $x_{10} \approx 162\,889$.

Example 6.2. Suppose that we withdraw 1 000 from the account every year. If we include this in our model we obtain the equation

$$x_{n+1} = 1.05x_n - 1\,000. \tag{6.3}$$

If we start with the same amount $x_0 = 100\,000$ as above, we now find $x_1 = 104\,000$, $x_2 = 108\,200$, and $x_{10} \approx 150\,312$.

Example 6.3. 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

$$x_{n+1} = 1.05x_n - (1\,000 + 300n). \tag{6.4}$$

In this case we find $x_1 = 104\,000$, $x_2 = 107\,900$, and $x_{10} = 134\,844$.

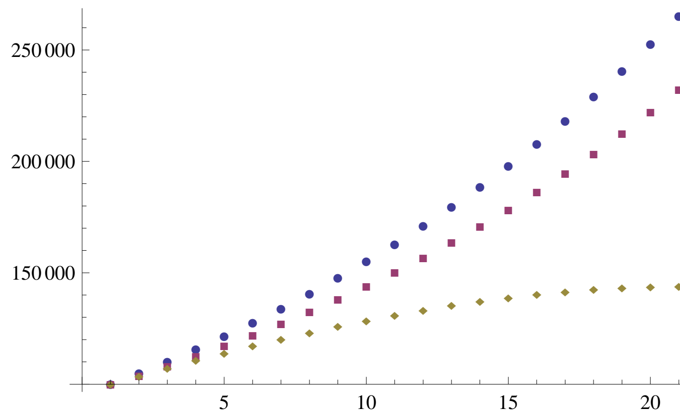


Figure 6.1. The growth of capital according to the models (6.1) (largest growth), (6.3) (middle growth), and (6.4) (smallest growth).

Plots of the development of the capital in the three different cases are 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 gone, and x_{46} will in fact be negative, i.e., money must be borrowed in order to keep up the withdrawals.

After these simple examples, let us define difference equations in general.

Definition 6.4 (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*

$$x_{n+k} = f(n, x_n, x_{n+1}, \dots, x_{n+k-1}), \tag{6.5}$$

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 6.4. In all three cases we have $k = 1$; in the case (6.1) we have $f(t,x) = 1.05x$, in (6.3) we see that $f(t,x) = 1.05x - 1000$, and in (6.4) we have $f(t,x) = 1.05x - (1000 + 300t)$.

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 6.5. 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.

6.2.1 Initial conditions

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), and this makes it easy to successively compute the balance, starting with the first year.

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 6.6 (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*

$$x_0 = a_0, \quad x_1 = a_1, \quad \ldots \quad x_{k-1} = a_{k-1},$$

where $a_0, \ldots, a_{k-1}$ 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

$$x_0 = \cdots = x_{k-1} = 1. \tag{6.7}$$

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.

6.2.2 Linear difference equations

It is convenient to classify difference equations according to their structure, and for many purposes the simplest ones are the *linear* difference equations.

Definition 6.7. *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 $f_0, \ldots, f_{k-1}$ are functions of n. The equation is said to have constant coefficients *if the functions $f_0 \ldots, f_{k-1}$ do not depend on n. It is said to be homogenous if $g(n) = 0$ for all n.*

From this definition we see that 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.

Linear difference equations are important because it is relatively simple to predict and analyse their solutions, as we will see in section 6.4.

6.2.3 Solving difference equations

In examples 6.1–6.3 we saw how easy it is to compute the terms of the sequence determined by a difference equation since the equation itself is quite simply a formula which tells us how one term can be computed from the previous ones. Provided the functions involved are computable and the calculations are done correctly (without round-off errors), we can therefore determine the exact value of any term in the solution sequence in this way. We refer to this as *simulating* the difference equation.

There is another way to solve a difference equation, namely by determining an explicit formula for the solution. For instance, the difference equations in examples 6.1–6.3 turn out to have the solutions that are given by the formulas

$$x_n = 100\,000 \times 1.05^n, \tag{6.8}$$

$$x_n = 80\,000 \times 1.05^n + 20\,000, \tag{6.9}$$

$$x_n = -40\,000 \times 1.05^n + 6\,000n + 140\,000. \tag{6.10}$$

The advantage of these formulas is that we can compute the value of a term immediately, without first computing all the preceding terms. With such formulas

we can also deduce the asymptotic behaviour of the solution. For example we see straightaway from (6.10) that in the situation in example 6.3, all the capital will eventually be used up, since x_n becomes negative for sufficiently large n. Another use of solution formulas like the ones in (6.8)–(6.10) is for predicting the effect of round-off errors on the numerical solutions computed by simulating a difference equation, see section 6.5.

Observation 6.8 (Solving difference equations). *There are two different ways to 'solve' a difference equation:*

1. *By simulating the equation, i.e., by starting with the initial values, and then successively computing the numerical values of the terms of the solution sequence, as in examples 6.1–6.3.*
2. *By finding an explicit formula for the solution sequence, as in* (6.8)–(6.10).

We emphasise that solution by formulas like (6.8)–(6.10) is only possible in some special cases like linear equations with constant coefficients. On the other hand, simulation of the difference equation is possible for very general equations in the form (6.5), the only requirement is that all the functions involved are computable.

We will discuss simulation of difference equations in section 6.3, and then review solution by a formula for linear equations in section 6.4. In section 6.5 we will then use our knowledge of solution formulas to analyse the effects of round-off errors on the simulation of linear difference equations.

Exercises for Section 6.2

1. Find the correct alternative in the following multiple choice exercises.

(a). (Mid-term 2007) Which one of the following difference equations is linear and has constant coefficients?

☐ $x_{n+1} + nx_n = 1$
☐ $x_{n+2} - 4x_{n+1} + x_n^2 = 0$
☐ $x_{n+1} - x_n = 0$
☐ $x_{n+2} + 4\sin(2^n)x_{n+1} + x_n = 1$.

(b). If we have the difference equation $x_{n+1} - x_n = n$, with $x_1 = 1$, what is the value of x_5?

☐ 11
☐ 12
☐ 13
☐ 14

2. Compare with (6.5) and determine the function f for the difference equations below. Also compute the values of $x_2, \dots, x_5$ in each case.

(a). $x_{n+2} = 3x_{n+1} - x_n, \quad x_0 = 2,\ x_1 = 1.$

(b). $x_{n+2} = x_{n+1} + 3x_n, \quad x_0 = 4,\ x_1 = 5.$

(c). $x_{n+2} = 2x_{n+1}x_n, \quad x_0 = 1,\ x_1 = 2.$

(d). $x_{n+1} = -\sqrt{4 - x_n}, \quad x_0 = 0.$

(e). $5x_{n+2} - 3x_{n+1} + x_n = n, \quad x_0 = 0,\ x_1 = 1.$

(f). $x_{n+1}^2 + 5x_n = 1, \quad x_0 = 3.$

3. Which of the following 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). $nx_{n+2} - x_{n+1}e^n + x_n = n^2.$

6.3 Simulating difference equations

In examples 6.1–6.3 above we saw that it was easy to compute the numerical values of the terms in a difference equation. In this section we are going to formalise this as an algorithm. Let us start by doing this for second-order linear equations. These are equations in the form

$$x_{n+2} = g(n) + f_0(n)x_n + f_1(n)x_{n+1}, \quad x_0 = a_0,\ x_1 = a_1, \tag{6.11}$$

where g, f_0 and f_1 are given functions of n, and a_0 and a_1 are given real numbers. Let us consider an example to remind ourselves how the terms are computed.

Example 6.9. We consider the difference equation

$$x_{n+2} = n + 2x_n - 3nx_{n+1}, \quad x_0 = 1,\ x_1 = 2,$$

in other words we have $g(n) = n$, $f_0(n) = 2$, and $f_1(n) = -3n$ in this case. If we set $n = 0$ in the difference equation we can compute x_2 as

$$x_2 = 0 + 2 \times x_0 - 3 \times 0 \times x_1 = 2.$$

We continue and set $n = 1$ which yields

$$x_3 = 1 + 2 \times x_1 - 3 \times 1 \times x_2 = 1 + 4 - 6 = -1.$$

We take one more step and obtain $(n = 2)$

$$x_4 = 2 + 2 \times x_2 - 3 \times 2 \times x_3 = 2 + 4 + 6 = 12.$$

In general, these computations can be phrased as a formal algorithm.

Algorithm 6.10. *Suppose the second-order equation* (6.11) *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 $x_0, x_1, \ldots, x_N$ of the solution:*

$x_0 = a_0$;
$x_1 = a_1$;
for $i = 2, 3, \ldots, N$
    $x_i = g(i-2) + f_0(i-2)x_{i-2} + f_1(i-2)x_{i-1}$;

This algorithm computes all the $N+1$ terms and saves them in the array $\boldsymbol{x} = [x_0, \ldots, x_N]$. 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 6.11. *The following algorithm computes the solution of* (6.11)*, just like algorithm 6.10, but prints each term instead of storing them:*

$x_{pp} = a_0$;
$x_p = a_1$;
for $i = 2, 3, \ldots, N$
    $x = g(i-2) + f_0(i-2)x_{pp} + f_1(i-2)x_p$;
    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 (6.10) and (6.11) can easily be generalised to 3rd or 4th order, or equations of any fixed order, and not only linear equations. The most convenient is to have an algorithm which takes the order of the equation as input.

Algorithm 6.12. *The following algorithm computes and prints the first $N+1$ terms of the solution of the kth-order difference equation*

$$x_{n+k} = f(n, x_n, x_{n+1}, \dots, x_{n+k-1}), \quad n = 0, 1, \dots, N-k \tag{6.12}$$

with initial values $x_0 = a_0$, $x_1 = a_1$, ..., $x_{k-1} = a_{k-1}$. Here f is a given function of $k+1$ variables, and a_0, ..., a_{k-1} are given real numbers.

```
for i = 0, 1, ..., k-1
    z_i = a_i;
    print z_i;
for i = k, k+1, ..., N
    x = f(i-k, z_0, ..., z_{k-1});
    print x;
    for j = 0, ..., k-2
        z_j = z_{j+1};
    z_{k-1} = x;
```

Algorithm 6.12 is similar to algorithm 6.11 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 $\boldsymbol{z} = [z_0, \dots, z_{k-1}]$. The values x_k, x_{k+1}, ..., x_N are computed in the second **for**-loop. By comparison with (6.12) 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 sequence 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 floating-point 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.

Exercises for Section 6.3

1. Write functions in Python which simulate a general difference equation of first and second order, following algorithm 6.12. Denote your functions by `difference_eq_1(f, x0, N)` and `difference_eq_2(f, x0, x1, N)`, where

- `f` is the right hand side of the difference equation,
- `x0` and `x1` are initial values, and
- `N` is the number of iterations.

2. Program algorithm 6.11 and test it on the Fibonacci equation

$$x_{n+2} = x_{n+1} + x_n, \quad x_0 = 0, x_1 = 1.$$

3. Generalise algorithm 6.11 to third order equations and test it on the Fibonacci like equation

$$x_{n+3} = x_{n+2} + x_{n+1} + x_n, \quad x_0 = 0, x_1 = 1, x_2 = 1.$$

4. A close relative of the Fibonacci numbers is called the Lucas numbers, and these are defined by the difference equation

$$L_{n+2} = L_{n+1} + L_n, \quad L_0 = 2, L_1 = 1.$$

Write a program which prints the following information:

(a). The 18th Lucas number.

(b). The first Lucas number greater than 100.

(c). The value of n for the number in (b).

(d). The Lucas number closest to 1000.

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). We will derive explicit formulas for the solutions of such equations, and from this, the behaviour of the solution when n tends to infinity. This will help us to understand how round-off errors influence the results of numerical simulations of difference equations—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

$$x_{n+1} = bx_n, \tag{6.13}$$

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 (6.13) we find

$$x_{n+1} = bx_n = b^2 x_{n-1} = b^3 x_{n-2} = \cdots = b^{n+1} x_0. \tag{6.14}$$

This is the content of the first lemma.

Lemma 6.13. *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 (6.14) 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 6.14. *For $n \geq 0$, the solution of the difference equation $x_{n+1} = bx_n$ with initial condition $x_0 \neq 0$ 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

$$x_{n+2} + b_1 x_{n+1} + b_0 x_n = 0. \tag{6.15}$$

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 (6.15) 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 (6.15). 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 (6.15) 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 6.15. *The solution of the homogenous, second-order difference equation*

$$x_{n+2} + b_1 x_{n+1} + b_0 x_n = 0 \tag{6.16}$$

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* (6.16) *is given by*
$$x_n = C r_1^n + D r_2^n.$$
2. *If the two roots are equal, $r_1 = r_2$, the general solution of* (6.16) *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 has the polar form as $r = \rho e^{i\theta}$, then the general solution of* (6.16) *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 the theorem is not so complicated and can be found in a text on difference equations. A couple of examples will illustrate how this works in practice.

Example 6.16. Let us consider the equation

$$x_{n+2} + 5x_{n+1} - 14x_n = 0, \quad x_0 = 2,\ x_1 = 9.$$

The characteristic equation is $r^2 + 5r - 14 = 0$ which has the two solutions $r_1 = 2$ and $r_2 = 7$. The general solution of the difference equation is therefore

$$x_n = C2^n + D7^n.$$

The two initial conditions lead to the system of two linear equations

$$\begin{aligned} 2 &= x_0 = C + D, \\ 9 &= x_1 = 2C + 7D, \end{aligned}$$

whose solution is $C = 1$ and $D = 1$. The solution that satisfies the initial conditions is therfore

$$x_n = 2^n + 7^n.$$

Example 6.17. The difference equation

$$x_{n+2} - 2x_{n+1} + 2x_n = 0, \quad x_0 = 1, x_1 = 1,$$

has the characteristic equation $r^2 - 2r + 2 = 0$. The two roots are $r_1 = 1 + i$ and $r_2 = 1 - i$. The absolute value of $r = r_1$ is $|r| = \sqrt{1^2 + 1^2} = \sqrt{2}$, while a drawing shows that the argument of r is $\arg r = \pi/4$. The general solution of the difference equation is therefore

$$x_n = \big(\sqrt{2}\big)^n \big(C \cos(n\pi/4) + D \sin(n\pi/4)\big).$$

We determine C and D by enforcing the initial conditions

$$\begin{aligned} 1 &= x_0 = \sqrt{2}^0 (C \cos 0 + D \sin 0) = C, \\ 1 &= x_1 = \sqrt{2}\big(C \cos(\pi/4) + D \sin(\pi/4)\big) = \sqrt{2}\big(C\sqrt{2}/2 + D\sqrt{2}/2\big) = C + D. \end{aligned}$$

From this we see that $C = 1$ and $D = 0$. The final solution is therefore

$$x_n = \big(\sqrt{2}\big)^n \cos(n\pi/4).$$

The theory that is outlined here for homogenous, second order difference equations can be generalised in a straightforward way to linear equations of any order, but this is beyond what we consider here.

6.4.3 Linear, 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 standard text books on difference equations.

Lemma 6.18. *Suppose that* $\{x_n^p\}$ *is a particular solution of the inhomogenous difference equation*

$$x_{n+2} + b_1 x_{n+1} + b_0 x_n = g(n). \tag{6.17}$$

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*

$$x_{n+k} + b_1 x_{n+1} + b_0 x_n = 0.$$

More informally, lemma 6.18 means that we can find the general solution of (6.17) 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 6.19. *One of the solutions of the inhomogenous equation*

$$x_{n+k} + b_1 x_{n+1} + b_0 x_n = g(n)$$

has the same form as $g(n)$.

Some examples will illustrate how this works.

Example 6.20. Consider the equation

$$x_{n+1} - 2x_n = 3. \tag{6.18}$$

Here the right-hand side is constant, so we try with the a particular solution $x_n^p = A$, 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 (6.18). Since the general solution of the homogenous equation $x_{n+1} - 2x_n = 0$ is $x_n^h = C2^n$, the general solution of (6.18) is

$$x_n = x_n^h + x_n^p = C2^n - 3.$$

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 6.21. The technique for find particular solutions also work for equations of order greater than 2. In the third-order equation

$$x_{n+3} - 2x_{n+2} + 4x_{n+1} + x_n = n \tag{6.19}$$

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,

$$\begin{aligned} 4A + 3B &= 0, \\ 4B &= 1. \end{aligned}$$

From these equations we find $B = 1/4$ and $A = -3/16$, so one solution of (6.19) is

$$x_n^p = \frac{n}{4} - \frac{3}{16}.$$

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 6.22. Suppose we have the equation

$$x_{n+1} + 4x_n = n3^n. \tag{6.20}$$

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

$$\begin{aligned} n3^n &= \big(A + B(n+1)\big)3^{n+1} + 4(A + Bn)3^n \\ &= 3^n\Big(3\big(A + B(n+1)\big) + 4A + 4Bn\Big) \\ &= 3^n(7A + 3B + 7Bn). \end{aligned}$$

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,

$$\begin{aligned} 7A + 3B &= 0, \\ 7B &= 1. \end{aligned}$$

This system has the solution $B = 1/7$ and $A = -3/49$, so a particular solution of (6.20) is

$$x_n^p = \Big(\frac{1}{7}n - \frac{3}{49}\Big)3^n.$$

The homogenous equation $x_{n+1} - 4x_n = 0$ has the general solution $x_n^h = C4^n$ so according to lemma 6.18 the general solution of (6.20) is

$$x_n = x_n^h + x_n^p = C4^n + \Big(\frac{1}{7}n - \frac{3}{49}\Big)3^n.$$

The theory in this section shows how we can obtain exact formulas for the solution of a class of difference equations, which can be useful in many situations. For example, this makes it quite simple to determine the behaviour of the solution when the number of time steps goes to infinity. Our main use of this theory will be as a tool to analyse the effects of round-off errors on the solution produced by a numerical simulation of linear, second order difference equations.

Exercises for Section 6.4

1. Mark each of the following statements as true or false.

(a). It is always possible to solve a difference equation numerically, given the function describing the equation and the appropriate number of initial conditions.

(b). Using Algorithm 6.10, i.e. by computing and storing all the values x_i as we solve the difference equation numerically, we can find any number x_i in the solution.

(c). When solving a difference equation numerically, we never need to store more than the two previous terms in order to calculate the next one.

2. Find a unique solution for x_n for the following difference equations:

(a). $x_{n+1} = 3x_n, \quad x_0 = 5/3$

(b). $x_{n+2} = 3x_{n+1} + 2x_n, \quad x_0 = 3,\ x_1 = 4.$

(c). $x_{n+2} = -2x_{n+1} - x_n, \quad x_0 = 1,\ x_1 = 1$

(d). $x_{n+2} = 2x_{n+1} + 3x_n, \quad x_0 = 2,\ x_1 = 1$

3. Find a unique solution for x_n for the following difference equations:

(a). $x_{n+2} - 3x_{n+1} - 4x_n = 2, \quad x_0 = 2,\ x_1 = 4.$

(b). $x_{n+2} - 3x_{n+1} + 2x_n = 2n + 1, \quad x_0 = 1,\ x_1 = 3.$

(c). $2x_{n+2} - 3x_n = 15 \times 2^n, \quad x_0 = 3,\ x_1 = 6.$

(d). $x_{n+1} - x_n = 5n2^n, \quad x_0 = 1,\ x_1 = 5$

4. Find the unique solution of the difference equation described in equation 6.4 with initial condition $x_0 = 100000$, and show that all the capital is indeed lost after 45 years.

5. Remember that the Fibonacci numbers are defined as:

$$F_{n+2} = F_{n+1} + F_n, F_0 = 0, F_1 = 1.$$

Remember also from exercise 6.3.4 that the Lucas numbers are defined as:

$$L_{n+2} = L_{n+1} + L_n, L_0 = 2, L_1 = 1.$$

Use this to prove the following identities:

$$L_n = F_{n+1} + F_{n-1}$$

6.5 Simulation of difference equations and round-off errors

In practice, most difference equations are 'solved' by numerical simulation, because of the simplicity of simulation, and because for most difference equations it is not possible to find an explicit formula for the solution. In chapter 5, we saw that computations on a computer often lead to errors, at least when we use floating-point numbers. Therefore, when a difference equation is simulated via one of the algorithms in section 6.3, we must be prepared for round-off 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. Let us start by stating how we intend to do this analysis.

Idea 6.23. *To study the effect of round-off errors on simulation of difference equations, we focus on a class of equations where exact formulas for the solutions are known. These explicit formulas are then used to explain (and predict) how round-off errors influence the numerical values produced by the simulation.*

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 becomes quite visible from a couple of examples.

Example 6.24. Consider the equation

$$x_{n+2} - \frac{2}{3}x_{n+1} - \frac{1}{3}x_n = 0, \quad x_0 = 1,\ x_1 = 0. \tag{6.21}$$

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

$$\begin{aligned} C + D &= 1, \\ C - D/3 &= 0, \end{aligned}$$

which have the solution $C = 1/4$ and $D = 3/4$. The solution of (6.21) 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.

If we simulate equation (6.21) 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 evaluate this expression. If we simulate the difference equation, we obtain the four approximate values

$$\tilde{x}_{10} = 0.250012701316,$$
$$\tilde{x}_{15} = 0.249999947731,$$
$$\tilde{x}_{20} = 0.250000000215,$$
$$\tilde{x}_{30} = 0.250000000000,$$

(throughout this section we will use $\tilde{x}_n$ to denote a computed version of x_n), which agree with the exact solution to 12 digits. In other words, numerical simulation in this case works very well and produces essentially the same result as the exact formula, even if floating-point numbers are used in the calculations.

Example 6.25. We consider the difference equation

$$x_{n+2} - \frac{19}{3}x_{n+1} + 2x_n = -10, \quad x_0 = 2,\ x_1 = 8/3. \tag{6.22}$$

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

$$x_n = x_n^h + x_n^p = 3 + C3^{-n} + D6^n. \tag{6.23}$$

If we enforce the initial conditions, we end up with the system of equations

$$\begin{aligned} 2 = x_0 &= 3 + C + D, \\ 8/3 = x_1 &= 3 + C/3 + 6D. \end{aligned} \tag{6.24}$$

This may be rewritten as the system

$$\begin{aligned} C + D &= -1, \\ C + 18D &= -1, \end{aligned} \tag{6.25}$$

which has the solution $C = -1$ and $D = 0$. The final solution is therefore

$$x_n = 3 - 3^{-n}, \tag{6.26}$$

which tends to 3 when n tends to infinity.

Let us simulate the equation (6.22) 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

$$\tilde{x}_5 = 2.99588477366,$$
$$\tilde{x}_{10} = 2.99998306646,$$
$$\tilde{x}_{15} = 3.00001192858.$$

These values appear to approach 3 as they should. However, some later values are

$$\begin{aligned} \tilde{x}_{20} &= 3.09329859009, \\ \tilde{x}_{30} &= 5641411.98633, \\ \tilde{x}_{40} &= 3.41114428655 \times 10^{14}, \end{aligned} \tag{6.27}$$

and at least the last two of these are obviously completely wrong!

6.5.1 Explanation of example 6.25

The cause of the problem with the numerical simulations in example 6.25 is round-off errors. In this section we are going to see how the general solution formula (6.23) actually explains our numerical problems.

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, but the second one definitely requires floating-point numbers. Note though that the fraction 8/3 cannot be represented exactly in binary with a finite number of digits, and therefore there will inevitably be round-off error. This means that the initial value 8/3 at x_1 becomes $x_1 = \tilde{a}_1 = 8/3 + \epsilon$, where $\tilde{a}_1$ is the floating-point number closest to 8/3 and ϵ is some small number of magnitude about 10^{-17}.

But it is not only the initial values that are not correct. When the next term is computed from the two previous ones, we use the formula

$$x_{n+2} = -10 + \frac{19}{3} x_{n+1} - 2x_n.$$

The number 10, and the coefficient -2 can be represented exactly. The middle coefficient 19/3, however, cannot be represented exactly by floating-point numbers, and is replaced by the nearest floating-point number $\tilde{c} = 19/3 + \delta$, where δ is a small number of magnitude roughly 10^{-17}.

Observation 6.26. *When the difference equation* (6.22) *is simulated numerically, round-off errors cause the difference equation and initial conditions to be computed by the formula*

$$x_{n+2} = \left(\frac{19}{3} + \delta\right) x_{n+1} - 2x_n - 10, \quad x_0 = 2,\ x_1 = 8/3 + \epsilon, \tag{6.28}$$

corresponding to the difference equation

$$x_{n+2} - \left(\frac{19}{3} + \delta\right) x_{n+1} + 2x_n = -10, \quad x_0 = 2,\ x_1 = 8/3 + \epsilon, \tag{6.29}$$

where ϵ *and* δ *are both small numbers of magnitude roughly* 10^{-17}.

The effect of round-off errors on the general solution

Observation 6.26 means that the actual computations are based on the difference equation (6.29), and not (6.22), but we can still determine an approximate formula for the exact solution that is being computed. We first focus on the general solution, and then adapt this to the initial conditions in the usual way.

The characteristic equation now becomes

$$r^2 - \left(\frac{19}{3} + \delta\right) r + 2 = 0$$

which has the two roots

$$r_1 = \frac{1}{6}\left(19 + 3\delta - \sqrt{289 + 114\delta + 9\delta^2}\right), \quad r_2 = \frac{1}{6}\left(19 + 3\delta + \sqrt{289 + 114\delta + 9\delta^2}\right).$$

The dependence on δ in these formulas is quite complicated, but can be simplified by the help of Taylor-polynomials which we will learn about in chapter 9. Using this technique, it is possible to show that

$$r_1 \approx \frac{1}{3} - \frac{\delta}{17}, \quad r_2 \approx 6 + \frac{18\delta}{17}.$$

This means that the general solution of the homogenous equation is approximately

$$x_n^h \approx C\left(\frac{1}{3} - \frac{\delta}{17}\right)^n + D\left(6 + \frac{18\delta}{17}\right)^n.$$

Since the right-hand side of (6.29) is constant, we try with a particular solution $x_n^p = A$ that is constant. If we insert this in (6.29), we find

$$x_n^p = A = \frac{30}{10 + 3\delta}.$$

This means that the general formula for the solution of the difference equation (6.29) is very close to

$$\frac{30}{10+3\delta} + C\Big(\frac{1}{3} - \frac{\delta}{17}\Big)^n + D\Big(6 + \frac{18\delta}{17}\Big)^n.$$

When δ is of magnitude 10^{-17}, this expression in turn will be very close to

$$3 + C\Big(\frac{1}{3}\Big)^n + D6^n \tag{6.30}$$

for all values of n that we typically encounter in practice. This simplifies the analysis of round-off errors for linear difference equations considerably: We can simply ignore round-off in the coefficients when determining the general solution.

Observation 6.27. *The round-off errors that occur in the coefficients (and the right-hand side) of the difference equation* (6.29) *do not lead to significant errors in the determination of the general solution of* (6.29).

The effect of round-off errors in the initial values

We next consider the effect of round-off errors in the initial values. From what we have just seen, we may assume that the result of the simulation is described by the general formula (6.30). The initial values are

$$x_0 = 2, \quad x_1 = 8/3 + \epsilon,$$

and this allows us to determine C and D in (6.30) via the equations

$$\begin{aligned} 2 = x_0 &= 3 + C + D, \\ 8/3 + \epsilon = x_1 &= 3 + C/3 + 6D. \end{aligned}$$

If we solve these equations we find

$$C = -1 - \frac{3}{17}\epsilon, \quad D = \frac{3}{17}\epsilon. \tag{6.31}$$

This is summarised in the next observation where for simplicity we have introduced the notation $\hat{\epsilon} = 3\epsilon/17$.

Observation 6.28. *The round-off error in the second initial value means that the result of numerical simulation of* (6.28) *corresponds to a solution in the form* (6.23)*, where C and D are given by*

$$C = -1 + \hat{\epsilon}, \quad D = \hat{\epsilon} \tag{6.32}$$

and $\hat{\epsilon}$ is a small number. The sequence generated by the numerical simulation is therefore approximately

$$\tilde{x}_n \approx 3 - (1 - \hat{\epsilon})3^{-n} + \hat{\epsilon}\, 6^n. \tag{6.33}$$

From observation 6.28 it is easy to explain where the values in (6.27) come from. Because of round-off errors, the numbers generated by the formula (6.28) are given by (6.33), where $\hat{\epsilon}$ is a small nonzero number. Even if $\hat{\epsilon}$ is small, the product $\hat{\epsilon}\, 6^n$ will eventually become large, since 6^n grows beyond all bounds when n becomes large.

6.5.2 Estimating the round-off unit

We can use the result of the numerical simulation in example 6.25 to estimate the round-off unit $\hat{\epsilon}$. From (6.27) we have $\hat{x}_{40} \approx 3.4 \times 10^{14}$, and for $n = 40$ we also 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 (6.33) is relevant (the other two terms affect the result in about the 30th digit and beyond). This means that we can find $\hat{\epsilon}$ from the relation

$$3.4 \times 10^{14} \approx \tilde{x}_{40} \approx \hat{\epsilon}\, 6^{40} \approx \hat{\epsilon}\, 1.3 \times 10^{31}.$$

From this we see that $\hat{\epsilon} \approx 2.6 \times 10^{-17}$. This is a reasonable value since we know that $\hat{\epsilon}$ 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 6.29. *When $\hat{\epsilon}$ is nonzero in* (6.33)*, the last term $\hat{\epsilon}\, 6^n$ will eventually dominate the computed solution of the difference equation completely, and the computations will end in overflow.*

It is important to realise that the reason for the values generated by the numerical simulation in (6.27) becoming large is not particularly bad round-off

errors; any round-off error at all would eventually lead to the same kind of behaviour. The general problem is that the difference equation corresponds to a family of solutions given by

$$x_n = 3 + C3^{-n} + D6^n, \tag{6.34}$$

and different initial conditions pick out different solutions (different values of C and D) within this family. The exact solution has $D = 0$. However, for numerical simulation with floating-point numbers it is basically impossible to get D to be exactly 0, so the last term in (6.34) will always dominate the computed solution for large values of n and completely overwhelm the other two terms in the solution.

6.5.3 Round-off errors and second order, inhomogenous equations

The difference equation in example 6.25 is not particularly demanding — we will get the same effect whenever we have a difference equation where the exact solution remains significantly smaller than the part of the general solution corresponding to the largest root of the characteristic equation.

Observation 6.30. *Suppose the difference equation*

$$x_{n+2} + b_1 x_{n+1} + b_0 x_n = g(n)$$

is simulated numerically with floating-point numbers by computing the numbers from the formula

$$x_n = g(n-2) - b_1 x_{n-1} - b_0 x_{n-2}, \qquad \text{for } n = 2, 3, \ldots \tag{6.35}$$

with two starting values $x_0 = a_0$ and $x_1 = a_1$. Let x_n^p be a particular solution, and let r be the root of the characteristic equation,

$$r^2 + b_1 r + b_0 = 0,$$

with largest absolute value. If round-off errors occur during the computations and the solution computed by simulation is denoted by $\{\tilde{x}_n\}_n$, then

$$\lim_{n\to\infty} |\tilde{x}_n| \approx \begin{cases} |r|^n, & \text{if } |r|^n > |x_n^p| \text{ for all sufficiently large } n; \\ |x_n^p|, & \text{if } |r|^n < |x_n^p| \text{ for all sufficiently large } n. \end{cases}$$

In example 6.25, 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 interested in just involves 3^{-n} and 3 we get into trouble since 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 6.31. We consider the second order difference equation

$$x_{n+2} - \frac{11}{6}x_{n+1} + \frac{2}{3}x_n = -n, \quad x_0 = 0,\ x_1 = 9.$$

When this is simulated numerically the values are computed with the fomula

$$x_n = -(n-2) + \frac{11}{6}x_{n-1} - \frac{2}{3}x_{n-2}, \qquad n = 2, 3, \ldots$$

with $x_0 = 0$ and $x_1 = 9$.

The coefficients have been chosen so that the roots of the characteristic equation are $r_1 = 1/2$, $r_2 = 4/3$. To find a particular solution we try with $x_n^p = An + B$. When this is inserted in the equation we find $A = B = 6$, so the general solution is

$$x_n = 6 + 6n + C2^{-n} + D(4/3)^n \tag{6.36}$$

The initial conditions force $C = -6$ and $D = 0$, so the exact solution is

$$x_n = 6 + 6n - 6/2^n. \tag{6.37}$$

In this case the starting values are integers than can be represented exactly. The coefficients in the equation, however, can not be represented exactly. As we noted in observation 6.27 this does not influence the general solution, but it does affect the computation of later values. For example we see that x_2 is given by

$$x_2 = 0 + \frac{11}{6}9 - \frac{2}{3}0 = \frac{33}{2}.$$

Even though the final value can be represented exactly there will be round-off errors in the result since there are round-off errors in the coefficients. This illustrates the point that round-off errors at any point in the simulation has the same effect as round-off errors in the starting values.

This is bound to lead to problems, just as in example 6.25. Because of round-off errors, the coefficient D will not be exactly 0 when the equation is simulated. Instead we will have

$$\tilde{x}_n = 6 + 6n - (6 + \epsilon_1)2^{-n} + \epsilon_2(4/3)^n$$

where both ϵ_1 and ϵ_2 are small numbers of magnitude 10^{-17}. Even if ϵ_2 is small, the term $\epsilon_2(4/3)^n$ will dominate when n becomes large. This is confirmed if we do the simulations. The computed value $\tilde{x}_{200}$ is approximately 24418, while the exact value is -1206, both rounded to the nearest integer.

Exercises for Section 6.5

1. Mark each of the following statements as true or false.

(a). There will always be major round-off errors when we solve second order difference equations numerically.

(b). When solving difference equations numerically, it is impossible to know when we will end up with completely wrong answers due to round-off errors.

2. Find the correct alternative in the following multiple choice exercises.

(a). (Mid-term 2009) We have the difference equation

$$3x_{n+2} + 4x_{n+1} - 4x_n = 0, \quad x_0 = 1, \quad x_1 = 2/3$$

and simulate this with 64-bit floating-point numbers on a computer. For large n, the computed solution $\bar{x}_n$ will give the result

☐ underflow
☐ overflow and then NaN
☐ $(2/3)^n$
☐ overflows with alternating sign

(b). (Mid-term 2009) We have the difference equation

$$3x_{n+1} - x_n/3 = 1, \quad x_1 = 1$$

and simulate this with 64-bit floating point numbers on a computer. For large n, the computed $\bar{x}_n$ solution will then approach

☐ n
☐ 1
☐ 0
☐ 3/8

(c). (Mid-term 2010) We have the difference equation

$$x_{n+1} - x_n/3 = 2, \quad x_0 = 2,$$

and simulate this with 64-bit floating-point numbers on a computer. For all n larger than a certain limit, the computed solution $\bar{x}_n$ will then give the result

☐ 3

☐ 1

☐ 0

☐ $3 - 3^n$

3. In each of the cases, find the exact solution of the difference equation, and describe the behaviour of the simulated solution for large values of n.

(a). $x_{n+1} - \frac{1}{3}x_n = 2, \quad x_0 = 2$

(b). $x_{n+2} - 6x_{n+1} + 12x_n = 1, \quad x_0 = 1/7,\ x_1 = 1/7$

(c). $3x_{n+2} + 4x_{n+1} - 4x_n = 0, \quad x_0 = 1,\ x_1 = 2/3$

4. In this exercise we are going to study the difference equation

$$x_{n+1} - 3x_n = 5^{-n}, \quad x_0 = -5/14. \tag{6.38}$$

(a). Show that the general solution of (6.38) 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 6.38 numerically.

(c). Do the simulation and check that your prediction in (b) is correct.

5. We consider the Fibonacci equation with nonstandard initial values

$$x_{n+2} - x_{n+1} - x_n = 0, \quad x_0 = 1,\ x_1 = (1 - \sqrt{5})/2. \tag{6.39}$$

(a). Show that the general solution of the equation is

$$x_n = C\left(\frac{1+\sqrt{5}}{2}\right)^n + D\left(\frac{1-\sqrt{5}}{2}\right)^n,$$

and that the initial values select the solution

$$x_n = \left(\frac{1-\sqrt{5}}{2}\right)^n.$$

(b). What will happen if you simulate (6.39) on a computer?

(c). Do the simulation and check that your predictions are correct.

6. We have the difference equation

$$x_{n+2} - \frac{2}{5}x_{n+1} + \frac{1}{45}x_n = 0, \quad x_0 = 1,\, x_1 = 1/15. \tag{6.40}$$

(a). Determine the general solution of (6.40) 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.

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.

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 6.25 led to serious problems with round-off errors. By making use of the theory behind linear difference equations with constant coefficients, 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*.

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.

```
00000000000000000000000000000000000000000
00000000000000000111000000000000000000000
00000000000000000111000000000000000000000
00000000000000000111000000000000000000000
00000000000000000111000000000000000000000
00000000000000000111000000000000000000000
00000000000000000111000000000000000000000
00000000000000000111000000000000000000000
00000000000000000111000000000000000000000
00000000000000000111000000000000000000000
00000000000000000111000000000000000000000
00000000000000000111000000000000000000000
00000000000000000111000000000000000000000
00000000000000000111000000000000000000000
00000000000000000111000000000000000000000
00000000000000000111000000000000000000000
00000000000000000111000000000000000000000
00000000000000000000000000000000000000000
```

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.

Exercises for Section 7.1

1. In general, encoding schemes for different kinds of information, such as text, images and music, usually employ a fixed number of bits for each piece of information (character, pixel or audio sample), no matter how complex the information is. The idea behind lossless compression is that the simpler the structure

of the information we want to store, the fewer bits we need to store it. In this exercise, we are going to check this in some particular cases.

(a). Create the following three text files:

'AAAAAAAAAAAAAAAAAAAAA' (21 As),
'AAAAAAAAAAAAAAAAAAAAB' (20 As followed by one B),
'AAAAAAAAAABAAAAAAAAAA' (10 As followed by one B and 10 more As).

These files will all be the same size. Try to decide how their compressed counterparts will compare in size. Carry out the compression using the program `gzip` (see section 7.6), and check if your assumptions are correct.

(b). On a Windows computer, Open paint, make a large, all-black, image, save it in .bmp format, and compress it with a program like winzip. Then make a few stripes in different colors in the image, save it, compress it and compare the size to the previous 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* $\boldsymbol{x} = \{x_1, x_2, \ldots, x_m\}$. *The symbols are assumed to be taken from an* alphabet *that is denoted* $\mathcal{A} = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$, *and the number of times that the symbol* α_i *occurs in* $\boldsymbol{x}$ *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* $\boldsymbol{x}$ *is stored as the bit-sequence* $\boldsymbol{z}$ *obtained by replacing each character in* $\boldsymbol{x}$ *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 $\boldsymbol{x}$ will just be a string of letters and other characters like $\boldsymbol{x}$ = {h,e, l, l, o, , a, g, a, i, n, .} (the character after 'o' is space, and the last character a period). The alphabet $\mathcal{A}$ 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 $\boldsymbol{x} = \{x_1, x_2, \dots, x_m\}$ 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 $\{c(x_1), c(x_2), \dots, c(x_m)\}$. 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 $\boldsymbol{x}$ have short codes and use long codes for the rare symbols. This leaves us with the problem of knowing the boundary between the codes. The following simple example illustrates this.

Example 7.2. Suppose we have the text $\boldsymbol{x}$ = DBACDBD. We note that the frequencies of the four symbols are $f(A) = 1$, $f(B) = 2$, $f(C) = 1$ and $f(D) = 3$. We assign the shortest codes to the most frequent symbols,

$$c(D) = 0, \quad c(B) = 1, \quad c(C) = 01, \quad c(A) = 10.$$

If we replace the symbols in $\boldsymbol{x}$ by their codes we obtain the compressed text

$$\boldsymbol{z} = 011001010,$$

altogether 9 bits, instead of the 56 bits (7 bytes) required by a standard text representation. However, we now have a major problem, how can we decode and find the original text from this compressed version? Do the first two bits represent the two symbols 'D' and 'B' or is it the symbol 'C'? One way to get round this problem is to have a special code that we can use to separate the symbols. But this is not a good solution as this would take up additional storage space.

Huffman coding uses a clever set of binary codes which makes it impossible to confuse the different symbols in a compressed text even though they may have different lengths.

Fact 7.3 (Huffman coding). *In Huffman coding the most frequent symbols in a text $\boldsymbol{x}$ 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 $\boldsymbol{x}$ are replaced by their codes and the resulting sequence of bits $\boldsymbol{z}$ is the compressed version of $\boldsymbol{x}$.*

This may sound a bit vague, so let us consider another example.

Example 7.4. Consider the same four-symbol text $\boldsymbol{x}$ = DBACDBD as in example 7.2. We now use the codes

$$c(D) = 1, \quad c(B) = 01, \quad c(C) = 001, \quad c(A) = 000. \tag{7.1}$$

We can then store the text as

$$\boldsymbol{z} = 1010000011011, \tag{7.2}$$

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 7.4. 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 $\boldsymbol{z}$. 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 7.3 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 7.5 (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.*

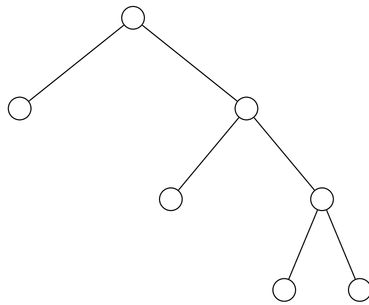


Figure 7.2. An example of a binary tree.

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.

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 7.6. *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.*

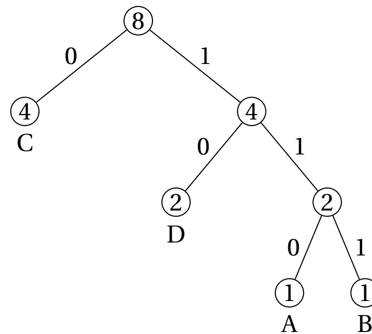


Figure 7.3. A Huffman tree.

Example 7.7. 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 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 7.7 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 7.8 (Huffman algorithm). *Let the text $\boldsymbol{x}$ 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:*

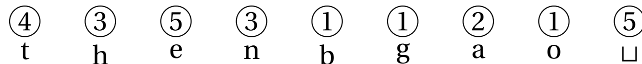
(a) Choose two trees T_0 and T_1 with minimal weights and replace them with a new tree which has T_0 as its left subtree and T_1 as its right subtree.

3. The tree remaining after the previous step is a Huffman tree for the given text $\boldsymbol{x}$.

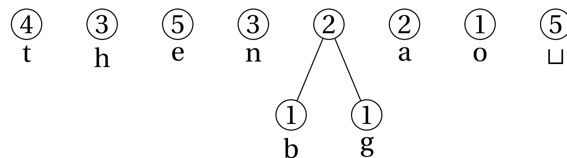
Most of the work in algorithm 7.8 is in step 2, but note that the number of trees is reduced by one each time, so the loop will run at most n times.

The easiest way to get to grips with the algorithm is to try it on a simple example.

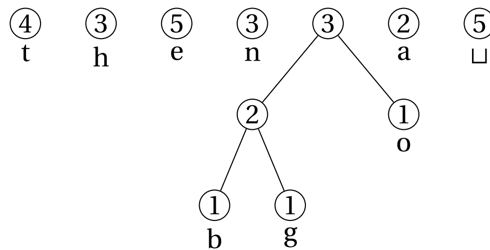
Example 7.9. Let us try out algorithm 7.8 on the text 'then the hen began to eat'. This text consists of 25 characters, including the five spaces. We first determine the frequencies of the different characters by counting. We find the collection of one-node trees



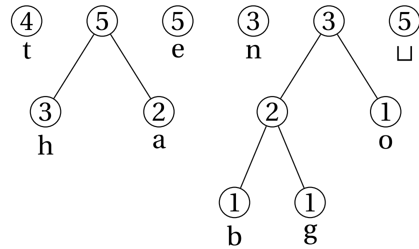
where the last character denotes the space character. Since 'b' and 'g' are two characters with the lowest frequency, we combine them into a tree,



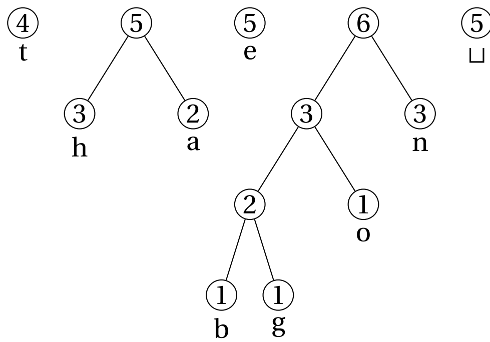
The two trees with the lowest weights are now the character 'o' and the tree we formed in the last step. If we combine these we obtain



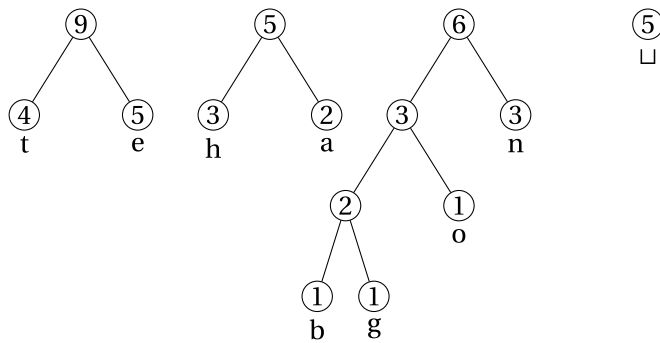
Now we have several choices. We choose to combine 'a' and 'h',



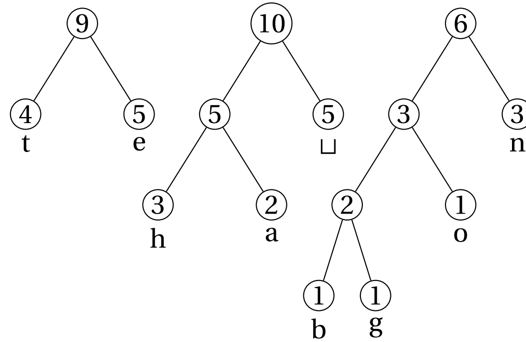
At the next step we combine the two trees with weight 3,



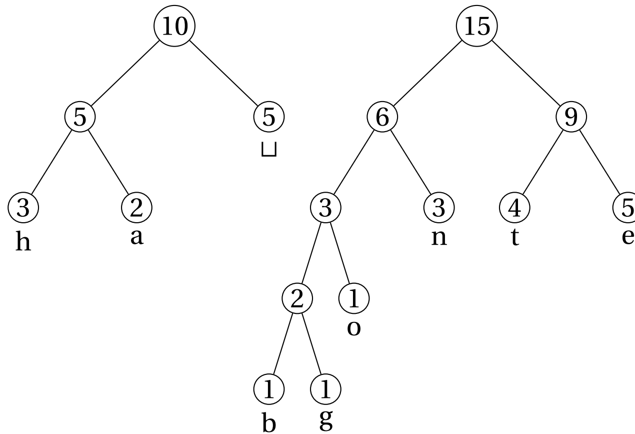
Next we combine the 't' and the 'e',



We now have two trees with weight 5 that must be combined



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

$$
\begin{aligned}
c(h) &= 000, & c(b) &= 10000, & c(n) &= 101,\\
c(a) &= 001, & c(g) &= 10001, & c(t) &= 110,\\
c(\sqcup) &= 01, & c(o) &= 1001, & c(e) &= 111.
\end{aligned}
$$

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.

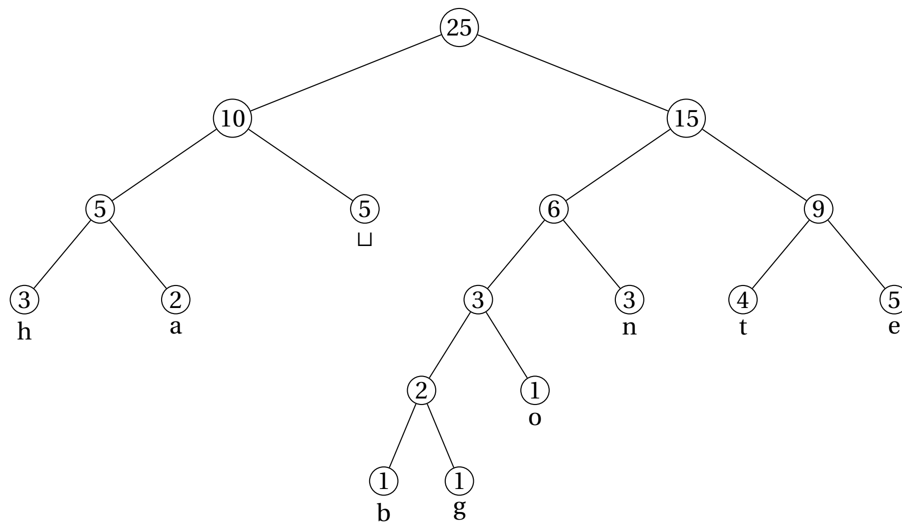


Figure 7.4. The Huffman tree for the text 'then the hen began to eat'.

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 7.3. This is easy to deduce from simple properties of Huffman trees.

Proposition 7.10 (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 7.9. 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 7.11 (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 7.11 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.

Exercises for Section 7.2

1. Mark each of the following statements as true or false.

(a). Huffman coding uses a special code to separate each symbol in a text.

(b). Huffman coding is the most optimal coding scheme based on binary trees.

(c). Because there is no ambiguity in the Huffman algorithm we do not need to store the code for each symbol in a text, only how many times each symbol occurs.

(d). The Huffman algorithm assigns codes to symbols in the order of most frequent symbol to least frequent symbol.

2. Only one of the following statements is true. Which one?

☐ Huffman coding allows you to store information with an absolute minimum amount of bits.

☐ Huffman coding can only be used to store letters, not numbers.

☐ In Huffman coding, the most frequent symbols in a text get the shortest codes.

☐ A node in a binary tree can have anywhere from 2 to 5 subtrees.

3. 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 7.8 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?

4. 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.

5. In this exercise we are going to do Huffman coding for the text given by $\boldsymbol{x} = \{ABACABCA\}$.

(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).

6. 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

$$c(\alpha_1) = 0000000, \quad c(\alpha_2) = 0000001, \quad c(\alpha_3) = 0000010, \quad \ldots \quad c(\alpha_{127}) = 1111110, \quad c(\alpha_{128}) = 1111111.$$

Explain how these codes can be associated with a certain Huffman tree. What are the frequencies used in the construction of the Huffman tree?

7.3 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 7.9. 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 7.9, 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 7.12 (Bits per symbol). *Let $\mathscr{A} = \{\alpha_1, \ldots, \alpha_n\}$ 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 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 7.12 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 7.13 (Shannon's theorem). *Let $\mathscr{A} = \{\alpha_1, \dots, \alpha_n\}$ 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*

$$H = H(p_1, \dots, p_n) = -\sum_{i=1}^{n} p(\alpha_i) \log_2 p(\alpha_i).$$

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 7.14. Let us return to example 7.9 and compute the entropy in this particular case. From the frequencies we obtain the probabilities

$$\begin{aligned} c(t) &= 4/25, & c(n) &= 3/25, & c(a) &= 2/25, \\ c(h) &= 3/25, & c(b) &= 1/25, & c(o) &= 1/25, \\ c(e) &= 1/5, & c(g) &= 1/25, & c(\sqcup) &= 1/5. \end{aligned}$$

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 7.9, 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 $\log_2\bigl(1/p(\alpha_i)\bigr)$. 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 5 in section 7.2.

Exercises for Section 7.3

1. Mark each of the following statements as true or false.

(a). The text "AAAABBBB" has less entropy than “AABAABBBA”.

(b). A text consisting of only one symbol repeated an arbitrary number of times will always have 0 entropy.

(c). In general, long texts will have a higher entropy than short texts.

(d). The entropy of the answer to this question will be more than 2.3.

2. (Exam 2010) The entropy of a text gives the minimum number of bits needed per symbol that the text is coded with. If we use Huffman-coding based on the frequency of the symbols in the text, which of these texts will not achieve a minimal amount of bits per symbol?

☐ AABB

☐ ABCC

☐ ABBB

☐ ABCD

3. Use the relation $2^{\log_2 x} = x$ to derive a formula for $\log_2 x$ in terms of natural logarithms.

4. Find the information entropy of the following famous quotes. The space character is included in the quote.

(a). ‘to be is to do’ — *Socrates*

(b). ‘do be do be do’ — *Sinatra*

(c). ‘scooby dooby doo’ — *Scooby Doo*

5. **(a).** Search the www and find the probabilities of the different letters in the English alphabet.

(b). Based on the probabilities you found in (a), what is the information entropy of an English text?

(c). Try to repeat (a) and (b) for your own language.

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 7.15. 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 7.16 (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.

Example 7.17 (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.

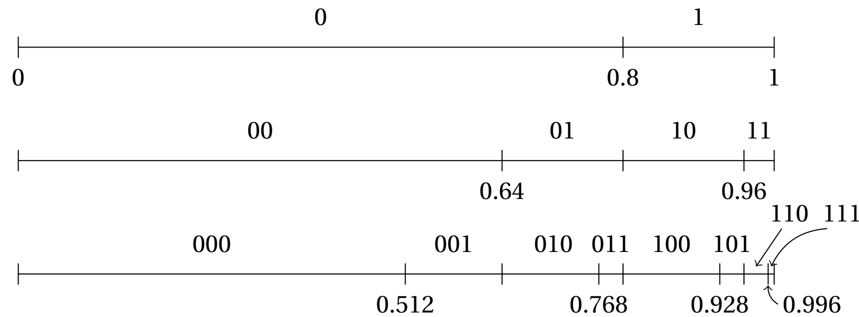


Figure 7.5. The basic principle of arithmetic coding applied to the text in example 7.17.

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 $0 \le i < 2^k$. Such numbers are called *dyadic 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 7.17 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 7.17 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 7.18. *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 $\mathscr{A} = \{\alpha_1, \ldots, \alpha_n\}$, 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 7.19 (Cumulative probability distribution). *Let $\mathscr{A} = \{\alpha_1, \ldots, \alpha_n\}$ be an alphabet where the probability of α_i is $p(\alpha_i)$. The* cumulative probability distribution *F is defined as*

$$F(\alpha_j) = \sum_{i=1}^{j} p(\alpha_i), \quad \text{for } j = 1, 2, \ldots, n.$$

The related function L is defined by $L(\alpha_1) = 0$ and

$$L(\alpha_j) = F(\alpha_j) - p(\alpha_j) = F(\alpha_{j-1}), \quad \text{for } j = 2, 3, \ldots, n.$$

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 $1 \le i \le n$.

The basic idea of arithmetic coding is to split the interval $[0,1)$ into the n subintervals

$$\big[0, F(\alpha_1)\big), \quad \big[F(\alpha_1), F(\alpha_2)\big), \ \dots, \ \big[F(\alpha_{n-2}), F(\alpha_{n-1})\big), \quad \big[F(\alpha_{n-1}), 1\big) \tag{7.6}$$

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

$$\begin{aligned} a_1 &= p(\alpha_1) + p(\alpha_2) + \cdots + p(\alpha_{i-1}) = F(\alpha_{i-1}) = L(\alpha_i) = L(x_1), \\ b_1 &= a_1 + p(\alpha_i) = F(\alpha_i) = F(x_1). \end{aligned}$$

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 7.18, 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 7.20 (Arithmetic coding). *Let the text* $\boldsymbol{x} = \{x_1, \dots, x_m\}$ *be given, with the symbols being taken from an alphabet* $\mathscr{A} = \{\alpha_1, \dots, \alpha_n\}$, *with probabilities* $p(\alpha_i)$ *for* $i = 1, \dots, n$. *Generate a sequence of* m *subintervals of* $[0,1)$:

1. *Set* $[a_0, b_0] = \big[0, 1\big)$.
2. *For* $k = 1, \dots, m$

(a) Define the linear function $g_k(z) = a_{k-1} + z(b_{k-1} - a_{k-1})$.

(b) Set $[a_k, b_k] = \Big[g_k\big(L(x_k)\big), g_k\big(F(x_k)\big)\Big)$.

The arithmetic code of the text $\boldsymbol{x}$ *is the midpoint* $C(\boldsymbol{x})$ *of the interval* $[a_m, b_m)$*, i.e., the number*

$$\frac{a_m + b_m}{2},$$

truncated to

$$\Big\lceil -\log_2\big(p(x_1)p(x_2)\cdots p(x_m)\big)\Big\rceil + 1$$

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 7.21. Suppose we have the text $\boldsymbol{x} = \{ACBBCAABAA\}$ 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,

$$
\begin{aligned}
g_4(z) &= 0.45 + 0.03z, \quad x_4 = B, \quad [a_4, b_4) = \big[g_4(0.5), g_4(0.8)\big) = [0.465, 0.474),\\
g_5(z) &= 0.465 + 0.009z, \quad x_5 = C, \quad [a_5, b_5) = \big[g_5(0.8), g_5(1)\big) = [0.4722, 0.474),\\
g_6(z) &= 0.4722 + 0.0018z, \quad x_6 = A, \quad [a_6, b_6) = \big[g_6(0), g_6(0.5)\big) = [0.4722, 0.4731),\\
g_7(z) &= 0.4722 + 0.0009z, \quad x_7 = A, \quad [a_7, b_7) = \big[g_7(0), g_7(0.5)\big) = [0.4722, 0.47265),\\
g_8(z) &= 0.4722 + 0.00045z, \quad x_8 = B, \quad [a_8, b_8) = \big[g_8(0.5), g_8(0.8)\big) = [0.472425, 0.47256),\\
g_9(z) &= 0.472425 + 0.000135z, \quad x_9 = A,\\
&\qquad [a_9, b_9) = \big[g_9(0), g_9(0.5)\big) = [0.472425, 0.4724925),\\
g_{10}(z) &= 0.472425 + 0.0000675z, \quad x_{10} = A,\\
&\qquad [a_{10}, b_{10}) = \big[g_{10}(0), g_{10}(0.5)\big) = [0.472425, 0.47245875).
\end{aligned}
$$

The midpoint M of this final interval is

$$M = 0.472441875 = 0.011110001111000111111_2,$$

and the arithmetic code is M rounded to

$$\Big\lceil -\log_2\big(p(A)^5 p(B)^3 p(C)^2\big)\Big\rceil + 1 = 16$$

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 7.17 we chose the arithmetic code to be the dyadic number with the smallest denominator within the interval $[a_m, b_m)$. In algorithm 7.20 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 7.22. *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 7.19. 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 7.20,

$$\begin{aligned} b_k - a_k &= g_k\big(F(x_k)\big) - g_k\big(L(x_k)\big) \\ &= \big(F(x_k) - L(x_k)\big)(b_{k-1} - a_{k-1}) \\ &= p(x_k)p(x_1)\cdots p(x_{k-1}). \end{aligned}$$

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,

$$\lambda = \lceil \mu \rceil = \Big\lceil -\log_2\big(p(x_1)\cdots p(x_m)\big)\Big\rceil.$$

This means that $1/2^\lambda$ is smaller than or equal to $b_m - a_m$ since $\lambda \geq \mu$. Consider the collection of dyadic numbers D_λ on the form $j/2^\lambda$ where j is an integer in the range $0 \leq j < 2^\lambda$. 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

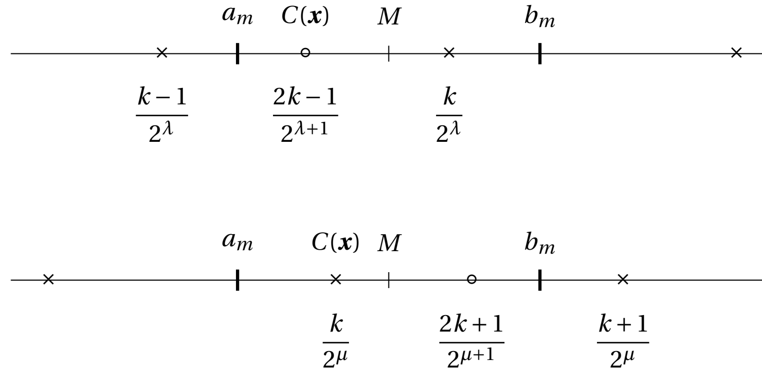


Figure 7.6. The two situations that can occur when determining the number of bits in the arithmetic code.

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 7.17). If we now keep the first λ digits in M we would get $C(\boldsymbol{x}) = k/2^\lambda$. In this case algorithm 7.20 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

$$\left\lceil -\log_2\bigl(p(x_1)p(x_2)\cdots p(x_m)\bigr)\right\rceil + 1.$$

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 $\boldsymbol{x}$. 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

$$\begin{aligned}
\frac{1}{m}\left\lceil -\log_2\bigl(p(x_1)p(x_2)\cdots p(x_m)\bigr)\right\rceil + \frac{1}{m} &\leq -\frac{1}{m}\log_2\bigl(p(x_1)p(x_2)\cdots p(x_m)\bigr) + \frac{2}{m} \\
&= -\frac{1}{m}\sum_{i=1}^{n} f(\alpha_i)\log_2 p(\alpha_i) + \frac{2}{m} \\
&\approx -\sum_{i=1}^{n} p(\alpha_i)\log_2 p(\alpha_i) \\
&= H(p_1,\ldots,p_n).
\end{aligned}$$

In other words, arithmetic coding gives compression rates close to the best possible for long texts.

Corollary 7.23. *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 7.20.

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 7.18.

Observation 7.24. *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 7.24 play a similar role in decoding as the g_ks in algorithm 7.20; 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 7.25. *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 7.17.

Example 7.26 (Decoding of an arithmetic code). Consider the arithmetic code 1001 from example 7.17 together with the two 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 7.24 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 7.20 and 7.25 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 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.

Exercises for Section 7.4

1. Mark each of the following statements as true or false.

(a). 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.

(b). Because computers have limited precision, we can only code very short texts (less than 64 characters) when using arithmetic coding.

(c). If we want to decode an aritmethically coded text, we need to know which probabilities were used as well as the ordering of the symbols.

2. 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).

3. Throughout this exercise we use the four-symbol alphabet $\mathscr{A} = \{A, B, C, D\}$. 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)?

4. We are given the three-symbol alphabet $\mathscr{A} = \{A, B, C\}$ with corresponding probabilities $p(A) = 0.1$, $p(B) = 0.6$ and $p(C) = 0.3$. A text $\boldsymbol{x}$ of length 10 has been encoded by arithmetic coding and the code is 1001101. What is the text $\boldsymbol{x}$?

5. 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

$$\overbrace{AAA \cdots AAA}^{99 \text{ times}} B.$$

6. The two linear functions in observations 7.18 and 7.24 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 7.18 and 7.24.

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 for Section 7.6

1. If it is not already installed, install the program `gzip` on your computer. Read the manual, and experiment with the program by compressing some sample files and observing the amount of compression.

CHAPTER 8

Digital Sound

A major part of the information we receive and perceive every day is in the form of audio. Most of these sounds are transferred directly from the source to our ears, like when we have a face to face conversation with someone or listen to the sounds in a forest or a street. However, a considerable part of the sounds are generated by loudspeakers in various kinds of audio machines like cell phones, digital audio players, home cinemas, radios, television sets and so on. The sounds produced by these machines are either generated from information stored inside, or electromagnetic waves are picked up by an antenna, processed, and then converted to sound. It is this kind of sound we are going to study in this chapter. The sound that is stored inside the machines or picked up by the antennas is usually represented as *digital sound*. This has certain limitations, but at the same time makes it very easy to manipulate and process the sound in a computer. The purpose of this chapter is to give a brief introduction to digital sound representation and processing.

We start by a short discussion of what sound is, which leads us to the conclusion that sound can be conveniently modelled by functions of a real variable in section 8.1. From mathematics it is known that almost any function can be approximated arbitrarily well by a combination of sines and cosines, and we discuss what this means when it is translated to the context of sound. We then go on and discuss digital sound, and simple operations on digital sound in section 8.2. FInally, we consider compression of sound in sections 8.4 and 8.5.

8.1 Sound

What we perceive as sound corresponds to the physical phenomenon of slight variations in air pressure near our ears. Larger variations mean louder sounds,

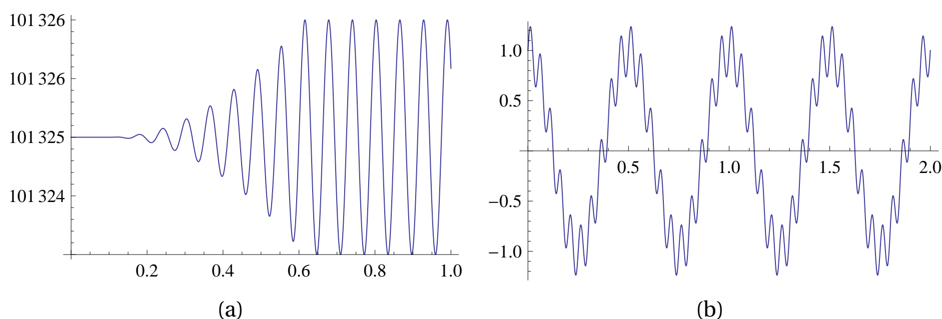


Figure 8.1. Two examples of audio signals.

while faster variations correspond to sounds with a higher pitch. The air pressure varies continuously with time, but at a given point in time it has a precise value. This means that sound can be considered to be a mathematical function. In this section we briefly discuss the basic properties of sound, first the significance of the size of the variations, and then the frequency of the variations. We also consider the important fact that any sound may be considered to be built from very simple basis sounds.

Before we turn to the details, we should be clear about the use of the word *signal* which is often encountered in literature on sound and confuses many.

Observation 8.1. *A sound can be represented by a mathematical function. When a function represents a sound it is often referred to as a* signal.

8.1.1 Loudness: Sound pressure and decibels

An example of a simple sound is shown in figure 8.1a. We observe that the initial air pressure has the value 101 325, and then the pressure starts to vary more and more until it oscillates regularly between the values 101 323 and 101 326. In the area where the air pressure is constant, no sound will be heard, but as the variations increase in size, the sound becomes louder and louder until about time $t = 0.6$ where the size of the oscillations becomes constant. The following summarises some basic facts about air pressure.

Fact 8.2 (Air pressure). *Air pressure is measured by the SI-unit Pa (Pascal) which is equivalent to N/m^2 (force / area). In other words, 1 Pa corresponds to the force exerted on an area of 1 m^2 by the air column above this area. The normal air pressure at sea level is 101 325 Pa.*

Fact 8.2 explains the values on the vertical axis in figure 8.1a: The sound was recorded at the normal air pressure of 101 325 Pa. Once the sound started, the pressure started to vary both below and above this value, and after a short transient phase the pressure varied steadily between 101 324 Pa and 101 326 Pa, which corresponds to variations of size 1 Pa about the fixed value. Everyday sounds typically correspond to variations in air pressure of about 0.002–2 Pa, while a jet engine may cause variations as large as 200 Pa. Short exposure to variations of about 20 Pa may in fact lead to hearing damage. The volcanic eruption at Krakatoa, Indonesia, in 1883, produced a sound wave with variations as large as almost 100 000 Pa, and the explosion could be heard 5000 km away.

When discussing sound, one is usually only interested in the variations in air pressure, so the ambient air pressure is subtracted from the measurement. This corresponds to subtracting 101 325 from the values on the vertical axis in figure 8.1a so that the values vary between -1 and 1. Figure 8.1b shows another sound with a slow, cos-like, variation in air pressure, roughly between -1 and 1. Imposed on this are some smaller and faster variations. This combination of several kinds of vibrations in air pressure is typical for general sounds.

The size of the variations in air pressure is directly related to the loudness of the sound. We have seen that for audible sounds the variations may range from 0.00002 Pa all the way up to 100 000 Pa. This is such a wide range that it is common to measure the loudness of a sound on a logarithmic scale. The following fact box summarises the previous discussion of what a sound is, and introduces the logarithmic decibel scale.

Fact 8.3 (Sound pressure and decibels). *The physical origin of sound is variations in air pressure near the ear. The* sound pressure *of a sound is obtained by subtracting the average air pressure over a suitable time interval from the measured air pressure within the time interval. A square of this difference is then averaged over time, and the sound pressure is the square root of this average.*

It is common to relate a given sound pressure to the smallest sound pressure that can be perceived, as a level on a decibel scale,

$$L_p = 10\log_{10}\left(\frac{p^2}{p_{\text{ref}}^2}\right) = 20\log_{10}\left(\frac{p}{p_{\text{ref}}}\right).$$

Here p is the measured sound pressure while p_{ref} is the sound pressure of a just perceivable sound, usually considered to be 0.00002 Pa.

The square of the sound pressure appears in the definition of L_p since this

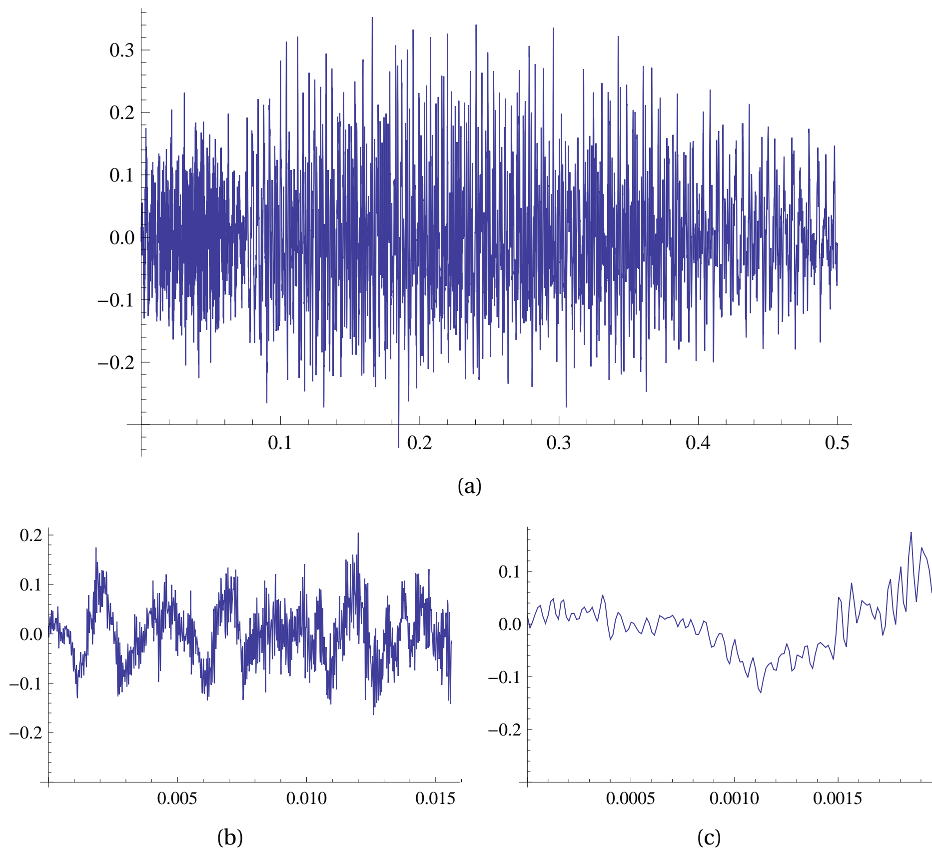


Figure 8.2. Variations in air pressure during parts of a song. Figure (a) shows 0.5 seconds of the song, figure (b) shows just the first 0.015 seconds, and figure (c) shows the first 0.002 seconds.

represents the *power* of the sound which is relevant for what we perceive as loudness.

The sounds in figure 8.1 are synthetic in that they were constructed from mathematical formulas. The sounds in figure 8.2 show the variation in air pressure for a real sound. In (a) there are so many oscillations that it is impossible to see the details, but if we zoom in as in figure (c) we can see that there is a continuous function behind all the ink. It is important to realise that in reality the air pressure varies more than this, even over the short time period in figure 8.2c. However, the measuring equipment was not able to pick up those variations, and it is also doubtful whether we would be able to perceive such rapid variations.

8.1.2 The pitch of a sound

Besides the size of the variations in air pressure, a sound has another important characteristic, namely the frequency (speed) of the variations. For most sounds the frequency of the variations varies with time, but if we are to perceive variations in air pressure as sound, they must fall within a certain range.

Fact 8.4. *For a human with good hearing to perceive variations in air pressure as sound, the number of variations per second must be in the range 20–20 000.*

To make these concepts more precise, we first recall what it means for a function to be periodic.

Definition 8.5. *A real function f is said to be periodic with period τ if*

$$f(t+\tau) = f(t)$$

for all real numbers t.

Note that all the values of a periodic function f with period τ are known if $f(t)$ is known for all t in the interval $[0,\tau)$. The prototypes of periodic functions are the trigonometric ones, and particularly $\sin t$ and $\cos t$ are of interest to us. Since $\sin(t+2\pi) = \sin t$, we see that the period of $\sin t$ is 2π and the same is true for $\cos t$.

There is a simple way to change the period of a periodic function, namely by multiplying the argument by a constant.

Observation 8.6 (Frequency). *If ν is an integer, the function $f(t) = \sin 2\pi\nu t$ is periodic with period $\tau = 1/\nu$. When t varies in the interval $[0,1]$, this function covers a total of ν periods. This is expressed by saying that f has* frequency ν.

Figure 8.3 illustrates observation 8.6. The function in figure (a) is the plain $\sin t$ which covers one period in the interval $[0,2\pi]$. By multiplying the argument by 2π, the period is squeezed into the interval $[0,1]$ so the function $\sin 2\pi t$ has frequency $\nu = 1$. Then, by also multiplying the argument by 2, we push two whole periods into the interval $[0,1]$, so the function $\sin 2\pi 2t$ has frequency $\nu = 2$. In figure (d) the argument has been multiplied by 5 — hence the frequency is 5 and there are five whole periods in the interval $[0,1]$. Note that any function on the form $\sin(2\pi\nu t + a)$ has frequency ν, regardless of the value of a.

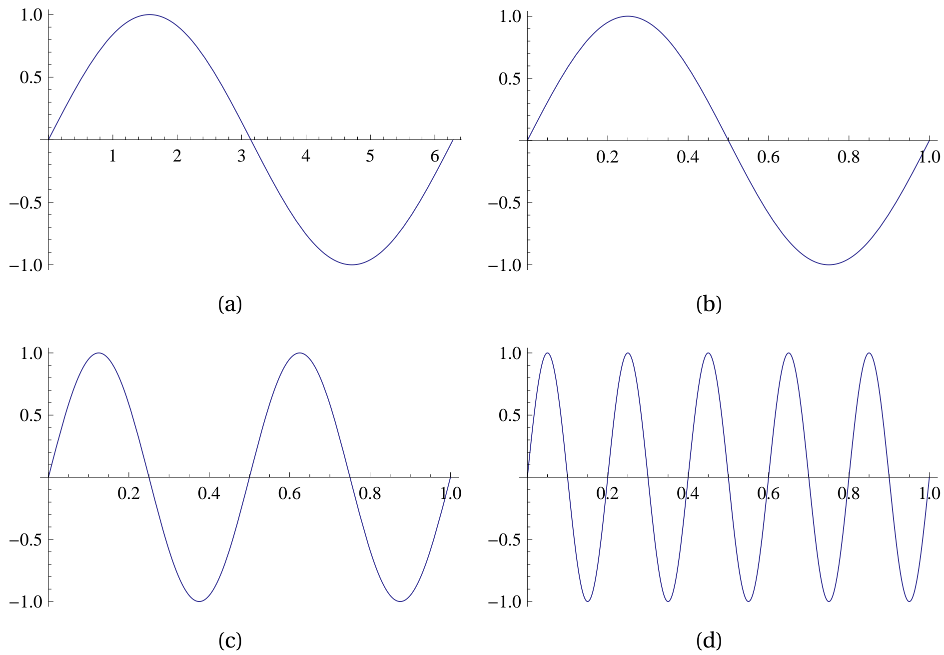


Figure 8.3. Versions of sin with different frequencies. The function in (a) is $\sin t$, the one in (b) is $\sin 2\pi t$, the one in (c) is $\sin 2\pi 2t$, and the one in (d) is $\sin 2\pi 5t$.

Since sound can be modelled by functions, it is reasonable to say that a sound with frequency ν is a trigonometric function with frequency ν.

Definition 8.7. *The function* $\sin 2\pi\nu t$ *represents a pure tone with frequency* ν*. Frequency is measured in Hz (Herz) which is the same as* s^{-1}.

With appropriate software it is easy to generate a sound from a mathematical function; we can 'play' a function. If we play a function like $\sin 2\pi 440t$, we hear a pleasant sound with a very distinct pitch, as expected.

There are many other ways in which a function can oscillate regularly. The function in figure 8.1b for example, definitely oscillates 2 times every second, but it does not have frequency 2 Hz since it is not a pure sin function. Likewise, the two functions in figure 8.4 also oscillate twice every second, but are very different from a smooth, trigonometric function. If we play a function like the one in figure (a), but with 440 periods in a second, we hear a sound with the same pitch as $\sin 2\pi 440t$, but it is definitely not pleasant. The sharp corners

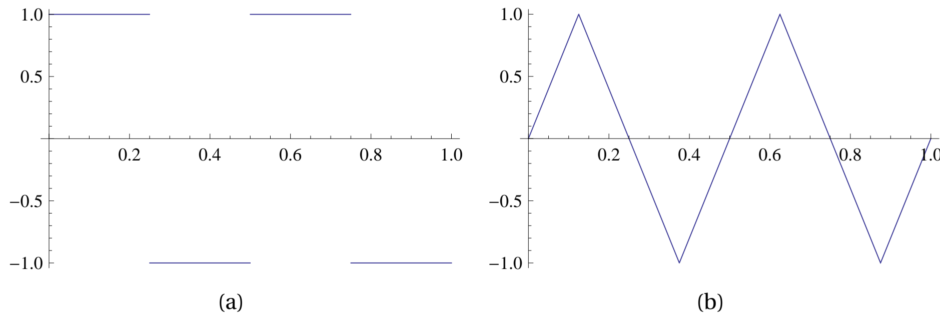


Figure 8.4. Two functions with regular oscillations, but which are not simple, trigonometric functions.

translate into a rather shrieking, piercing sound. The function in figure (b) leads to a smoother sound than the one in (a), but not as smooth as a pure sin sound.

8.1.3 Any function is a sum of sin and cos

A very common tool in mathematics is to approximate general functions by combinations of more standard functions. Perhaps the most well-known example is Taylor series where functions are approximated by combinations of polynomials. In the area of sound it is of more interest to approximate with combinations of trigonometric functions — this is referred to as *Fourier analysis*. The following is an informal version of a very famous theorem.

Theorem 8.8 (Fourier series). *Any reasonable function f can be approximated arbitrarily well on the interval $[0,1]$ by a combination*

$$f(t) \approx a_0 + \sum_{k=1}^{N} (a_k \cos 2\pi k t + b_k \sin 2\pi k t), \tag{8.1}$$

by choosing the integer N sufficiently large. The coefficients $\{a_k\}_{k=0}^{N}$ and $\{b_k\}_{k=1}^{N}$ are given by the formulas

$$a_k = \int_0^1 f(t)\cos(2\pi k t)\,dt, \qquad b_k = \int_0^1 f(t)\sin(2\pi k t)\,dt.$$

The series on the right in (8.1) *is called a Fourier series approximation of f.*

An illustration of the theorem is shown in figure 8.5 where a cubic polynomial is approximated by a Fourier series with $N = 9$. Note that the trigonometric approximation is periodic with period 1, so the approximation becomes poor at

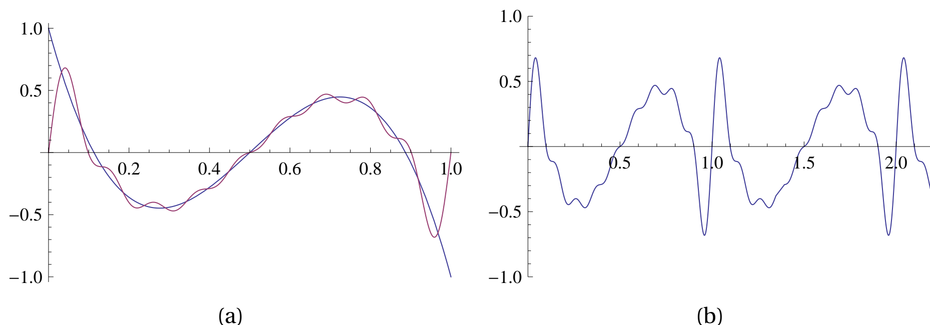


Figure 8.5. Trigonometric approximation of a cubic polynomial on the interval $[0,1]$. In (a) both functions are shown while in (b) the approximation is plotted on the interval $[0,2.2]$.

the ends of the interval since the cubic polynomial is not periodic. The approximation is plotted on a larger interval in figure 8.5b where its periodicity is clearly visible.

Since any sound may be considered to be a function, theorem 8.8 can be translated to a statement about sound. We recognise both trigonometric functions on the right in (8.1) as sounds with pure frequency k. The theorem therefore says that any sound may be approximated arbitrarily well by pure sounds with frequencies $0, 1, 2, \ldots, N$, as long as we choose N sufficiently large.

Observation 8.9 (Decomposition of sound into pure tones). *Any sound f is a sum of pure tones with integer frequencies. The amount of each frequency required to form f is the frequency content of f.*

Observation 8.9 makes it possible to explain more precisely what it means that we only perceive sounds with a frequency in the range 20–20 000.

Fact 8.10. *Humans can only perceive variations in air pressure as sound if the Fourier series of the sound signal contains at least one sufficiently large term with frequency in the range 20–20 000.*

The most basic consequence of observation 8.9 is that gives us an understanding of how any sound can be built from the simple building blocks of sin and cos. But it is also the basis for many operations on sounds. As an example, consider the function in figure 8.6 (a). Even though this function oscillates 5 times regularly between 1 and -1, the discontinuities mean that it is far from

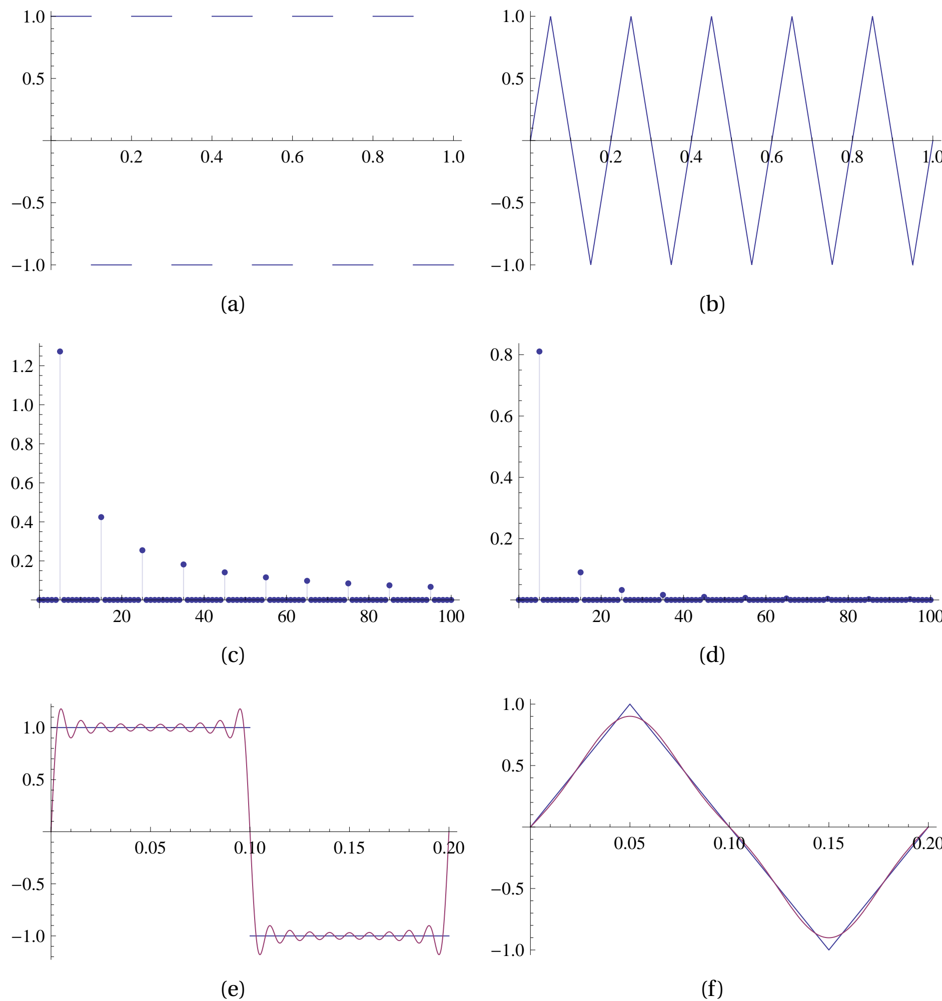


Figure 8.6. Approximations to two periodic functions with Fourier series. Since both functions are antisymmetric, the cos part in (8.1) is zero in both cases (all the a_k are zero). Figure (c) shows $\{a_k\}_{k=0}^{100}$ when f is the function in figure (a), and the plot in (e) shows the resulting approximation (8.1) with $N = 100$. The plots in figures (b), (d), and (e) are similar, except that the approximation in figure (f) corresponds to $N = 20$.

the simple $\sin 2\pi 5t$ which corresponds to a pure tone of frequency 5. If we compute the Fourier coefficients, we find that all the a_k are zero since the function is antisymmetric. The first 100 of the b_k coefficients are shown in figure (c). We note that only $\{b_{10j-5}\}_{j=1}^{10}$ are nonzero, and these decrease in magnitude. Note that the dominant coefficient is b_5, which tells us how much there is of the pure tone $\sin 2\pi 5t$ in the square wave in (a). This is not surprising since the square

wave oscillates 5 times in a second, but the additional nonzero coefficients pollute the pure sound. As we include more and more of these coefficients, we gradually approach the square wave in (a). Figure (e) shows the corresponding approximation of one period of the square wave.

Figures 8.6 (b), (d), and (f) show the analogous information for a triangular wave. The function in figure (a) is continuous and therefore the trigonometric functions in (8.1) converge much faster. This can be seen from the size of the coefficients in figure (d), and from the plot of the approximation in figure (f). (Here we have only included two nonzero terms. With more terms, the triangular wave and the approximation become virtually indistinguishable.)

From figure 8.6 we can also see how we can use the Fourier coefficients to analyse or improve the sound. Noise in a sound often correspond to the presence of some high frequencies with large coefficients, and by removing these, we remove the noise. For example, in figure (b), we could set all the coefficients except the first one to zero. This would change the unpleasant square wave to the pure tone $\sin 2\pi 5t$ with the same number of oscillations per second. Another common operation is to dampen the treble of a sound. This can be done quite easily by reducing the size of the coefficients corresponding to high frequencies. Similarly, the bass can be adjusted by changing the coefficients corresponding to the lower frequencies.

Exercises for Section 8.1

1. Mark each of the following statements as true or false.

(a). The function $\sin(1000t)$ has a frequency of approximately 159 Hz.

(b). A constant pressure of 101 325.01 Pa will be precieved as a sound with pressure 0.01 Pa, which corresponds to a sound of around 74 dB, if we use a reference pressure of 0.00002 Pa.

(c). Any sound f is a sum of pure tones with integer frequencies, i.e. functions of the form $\sin(2\pi kt)$ and $\cos(2\pi kt)$, where k is an integer.

8.2 Digital sound

In the previous section we considered some basic properties of sound, but it was all in terms of functions defined for all times in some interval. On computers and various kinds of media players the sound is usually digital, and in this section we are going to see what this means.

8.2.1 Sampling

Digital sound is very simple: The air pressure of a sound is measured a fixed number of times per second, and the measurements are stored as numbers in a file.

Definition 8.11 (Digital sound). *A digital sound consists of an array $\boldsymbol{a}$ of numbers,* the samples, *that correspond to measurements of the air pressure of a sound, recorded at a fixed rate of s,* the sample rate, *measurements per second. If the sound is in stereo there will be two arrays $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$, one for each channel. Measuring the sound is also referred to as* sampling *the sound, or analog to digital (AD) conversion.*

There are many different digital sound formats. A couple of them are described in the following two examples.

Fact 8.12 (CD-format). *The digital sound on a CD has sample rate 44 100, and each measurement is stored as a 16 bit integer.*

Fact 8.13 (GSM-telephone). *The digital sound in GSM mobile telephony has sample rate 8 000, and each measurement is stored as a 13 bit number in a floating-point like format.*

There are many other digital sound formats in use, with sample rates as high as 192 000 and above, using 24 bits and more to store each number.

8.2.2 Limitations of digital audio: The sampling theorem

An example of sampling is illustrated in figure 8.7. When we see the samples on their own in figure (b) it is clear that some information is lost in the sampling process. An important question is therefore how densely we must sample a function in order to not lose too much information.

The difficult functions to sample are those that oscillate quickly, and the challenge is to make sure there are no important features between the samples. By zooming in on a function, we can reduce the extreme situation to something simple. This is illustrated in Figure 8.8. If we consider one period of $\sin 2\pi t$, we see from figure (a) that we need at least two sample points, since one point

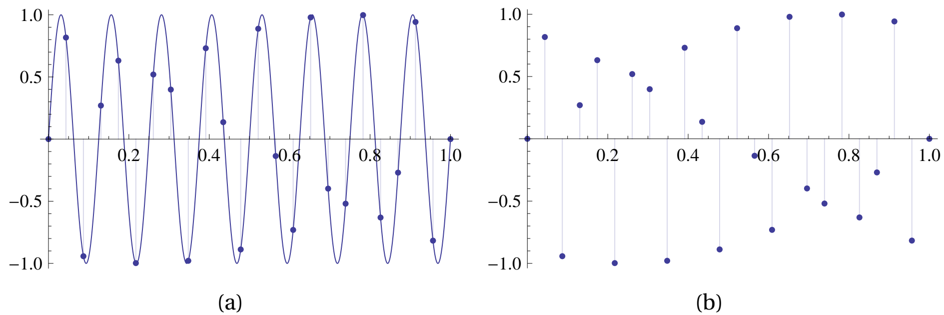


Figure 8.7. An example of sampling. Figure (a) shows how the samples are picked from underlying continuous time function. Figure (b) shows what the samples look like on their own.

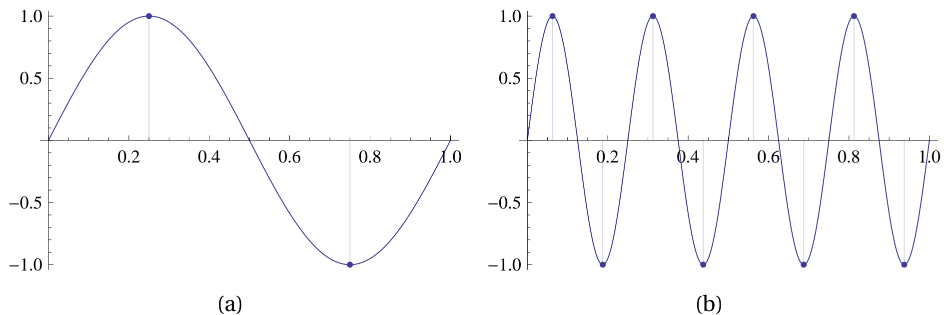


Figure 8.8. Sampling the function $\sin 2\pi t$ with two points, and the function $\sin 2\pi 4t$ with eight points.

would clearly be too little. This translates directly into having at least eight sample points in figure (b) where the function is $\sin 2\pi 4t$ which has four periods in the interval $[0,1]$.

Suppose now that we have a sound (i.e., a function) whose Fourier series contains terms with frequency at most equal to ν. This means that the function in the series that varies most quickly is $\sin 2\pi\nu t$ which requires 2ν sample point per second. This informal observation is the content of an important theorem. We emphasise that the simple argument above is no proof of this theorem; it just shows that it is reasonable.

Theorem 8.14 (Shannon-Nyquist sampling theorem). *A sound that includes frequencies up to ν Hz must be sampled at least 2ν times per second if no information is to be lost.*

The sampling theorem partly explains why the sampling rate on a CD is 44

100. Since the human ear can perceive frequencies up to about 20 000 Hz, the sampling rate must be at least 40 000 to ensure that the highest frequencies are accounted for. The actual sampling rate of 44 100 is well above this limit and ensures that there is some room to smoothly taper off the high frequencies from 20 000 Hz.

8.2.3 Reconstructing the original signal

Before we consider some simple operations on digital sound, we need to discuss a basic challenge: Sound which is going to be played back through an audio system must be defined for continuous time. In other words, we must fill in all the values of the air pressure between two sample points. There is obviously no unique way to do this since there are infinitely many paths for a graph to follow between to given points.

Fact 8.15 (Reconstruction of digital audio). *Before a digital sound can be played through an audio system, the gaps between the sample points must be filled by some mathematical function. This process is referred to as digital to analog (DA) conversion.*

Figure 8.9 illustrates two ways to reconstruct an analog audio signal from a digital one. In the top four figures, the points have been sampled from the function $\sin 2\pi 4t$, while in the lower two figures the samples are taken from $\cos 2\pi 4t$. In the first column, neighbouring sample points have been connected by straight lines which results in a piecewise linear function that passes through (interpolates) the sample points. This works very well if the sample points are close together relative to the frequency of the oscillations, as in figure 8.9a. When the samples are further apart, as in (c) and (e), the discontinuities in the derivative become visible, and we know that this may be heard as noise in the reconstructed signal.

In the second column, the gap between two sample points has been filled with a cubic polynomial, and neighbouring cubic polynomials have been joined smoothly together so that the total function is continuous and has continuous first and second derivative. We see that this works much better and produces a smooth result that is very similar to the original trigonometric signal.

Figure 8.9 illustrates the general principle: If the sampling rate is high, quite simple reconstruction techniques will be sufficient, while if the sampling rate is low, more sophisticated methods for reconstruction will be necessary.

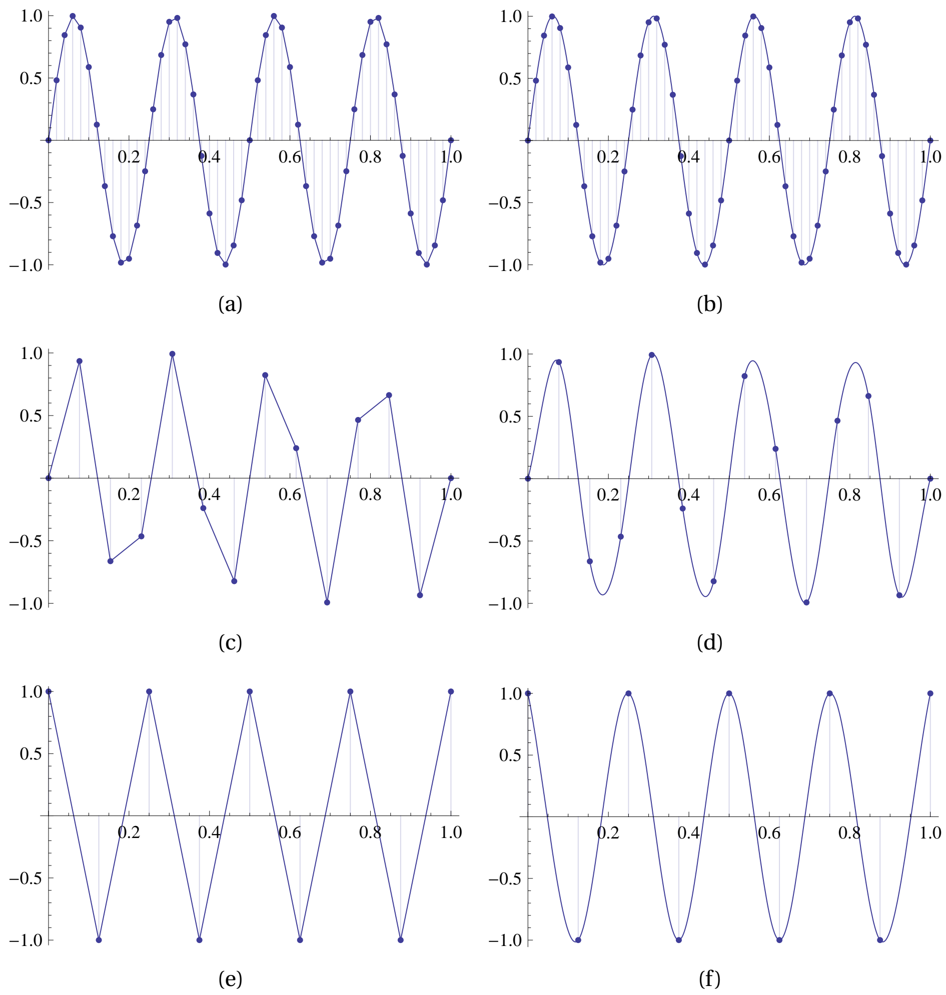


Figure 8.9. Reconstruction of sampled data.

Exercises for Section 8.2

1. Mark each of the following statements as true or false.

(a). If we sample the function $\sin(2\pi 800t)$ with 1000 samples per second it may be perfectly reconstructed.

(b). The CD-format has a high enough sample rate to reconstruct any sound that is perceivable by the human ear.

2. (Exam 2009) A program generates a digital sound by measuring the sound 22050 times per second (in one channel, i.e. not stereo), and each measurement is stored as a 32 bit integer. For each minute of music, this gives a total of

☐ 1 323 000 bytes

☐ 5 292 000 bytes

☐ 2 646 000 bytes

☐ 42 336 000 bytes

8.3 Simple operations on digital sound

So far we have discussed what digital sound is, the limitations in sampling, and how the information missing in sampled information may be reconstructed. It is now time to see how digital sound can be processed and manipulated.

Recall that a digital sound is just an array of sample values $\boldsymbol{a} = (a_i)_{i=0}^N$ together with the sample rate s. Performing operations on the sound therefore amounts to doing the appropriate computations with the sample values and the sample rate.

The most basic operation we can perform on a sound is simply playing it, and if we are working with sound we need a mechanism for doing this.

Playing a sound. Simple operations and computations with sound can be done in any programming environment, but in order to play the sound, it is necessary to use an environment that includes a command like `play(a, s)` (the ocmmand may of course have some other name; it is the functionality that is important). This will simply play the array of samples $\boldsymbol{a}$ using the sample rate s. If no `play`-function is available, you may still be able to play the result of your computations if there is support for saving the sound in some standard format like mp3. The resulting file can then be played by the standard audio player on your computer.

The `play`-function is just a software interface to the sound card in your computer. It basically sends the array of sample values and the sample rate to the sound card which uses some method for reconstructing the sound to an analog sound signal. This analog signal is then sent to the loudspeakers and we hear the sound.

Fact 8.16. *The basic command in a programming environment that handles sound is a command*

```
play(a, s)
```

which takes as input an array of sample values $\boldsymbol{a}$ *and a sample rate* s*, and plays the corresponding sound through the computer's loudspeakers.*

Changing the sample rate. We can easily play back a sound with a different sample rate than the standard one. If we have a sound $(\boldsymbol{a}, s)$ and we play it with the command `play(a, 2s)`, the sound card will assume that the time distance between neighbouring samples is half the time distance in the original. The result is that the sound takes half as long, and the frequency of all tones is doubled. For voices the result is a characteristic Donald Duck-like sound.

Conversely, the sound can be played with half the sample rate as in the command `play(a, s/2)`. Then the length of the sound is doubled and all frequencies are halved. This results in low pitch, roaring voices.

Fact 8.17. *A digital sound* $(\boldsymbol{a}, s)$ *can be played back with a double or half sample rate with the commands*

```
play(a, 2s)
play(a, s/2)
```

Playing the sound backwards. At times a popular game as been to play music backwards to try and find secret messages. In the old days of analog music on vinyl this was not so easy, but with digital sound it is quite simple; we just need to reverse the samples. To do this we just loop through the array and put the last samples first.

Fact 8.18. *Let* $\boldsymbol{a} = \{a_i\}_{i=0}^N$ *be the samples of a digital sound. Then the samples* $\boldsymbol{b} = \{b_i\}_{i=0}^N$ *of the reverse sound are given by*

$$b_i = a_{N-i}, \quad \text{for } i = 0, 1, \ldots N.$$

Adding noise. To remove noise from recorded sound can be very challenging, but adding noise is simple. There are many kinds of noise, but one kind is easily obtained by adding random numbers to the samples of a sound.

Fact 8.19. *Let* $\boldsymbol{a}$ *be the samples of a digital sound, normalised so that each sample is a real number in the interval* $[-1,1]$*. A new sound* $\boldsymbol{b}$ *with noise added can be obtained by adding a random number to each sample,*

$$b_i = a_i + c\,\texttt{random()}$$

where `random()` *is a function that gives a random number in the interval* $[-1,1]$*, and* c *is a constant (usually smaller than 1) that dampens the noise.*

This will produce a general hissing noise similar to the noise you hear on the radio when the reception is bad. The factor c is important, if it is too large the noise will simply drown the signal $\boldsymbol{b}$.

Adding echo. An echo is a copy of the sound that is delayed and softer than the original sound. We observe that the sample that comes m seconds before sample i has index $i - ms$ where s is the sample rate. This also makes sense even if m is not an integer so we can use this to produce delays that are less than one second. The one complication with this is that the number ms may not be an integer. We can get round this by rounding ms to the nearest integer which corresponds to adjusting the echo slightly.

Fact 8.20. *Let* $(\boldsymbol{a}, s)$ *be a digital sound. Then the sound* $\boldsymbol{b}$ *with samples given by*

$$b_i = \begin{cases} a_i, & \text{for } i = 0, 1, \ldots, d-1; \\ a_i + c a_{i-d}, & \text{for } i = d, d+1, \ldots, N; \end{cases}$$

will include a echo of the original sound. Here $d = \texttt{round}(ms)$ *is the integer closest to* ms*, and* c *is a constant which is usually smaller than* 1.

As in the case of noise it is important to dampen the part that is added to the original sound, otherwise the echo will be too loud. Note also that the formula that creates the echo does not work at the beginning of the signal, so there we just copy a_i to b_i.

Reducing the treble. The treble in a sound is generated by the fast oscillations (high frequencies) in the signal. If we want to reduce the treble we have to adjust the sample values in a way that reduces those fast oscillations. A general way of reducing variations in a sequence of numbers is to replace one number by the

average of itself and its neighbours, and this is easily done with a digital sound signal. If we let the new sound signal be $\boldsymbol{b} = (b_i)_{i=0}^N$ we can compute it as

$$b_i = \begin{cases} a_i, & \text{for } i = 0; \\ (a_{i-1} + a_i + a_{i+1})/3, & \text{for } 0 < i < N; \\ a_i, & \text{for } i = N. \end{cases}$$

This kind of operation is often referred to as *filtering* the sound, and the sequence $\{1/3, 1/3, 1/3\}$ is referred to as a *filter*.

It is reasonable to let the middle sample a_i count more than the neighbours in the average, so an alternative is to compute the average as

$$b_i = \begin{cases} a_i, & \text{for } i = 0; \\ (a_{i-1} + 2a_i + a_{i+1})/4, & \text{for } 0 < i < N; \\ a_i, & \text{for } i = N. \end{cases} \tag{8.2}$$

We can also take averages of more numbers. We note that the coefficients used in (8.2) are taken from row 2 in Pascal's triangle. If we pick coefficients from row 4 instead, the computations become

$$b_i = \begin{cases} a_i, & \text{for } i = 0,\, 1; \\ (a_{i-2} + 4a_{i-1} + 6a_i + 4a_{i+1} + a_{i+2})/16, & \text{for } 1 < i < N-1; \\ a_i, & \text{for } i = N-1,\, N. \end{cases} \tag{8.3}$$

We have not developed the tools needed to analyse the quality of filters, but it turns out that picking coefficients from a row in Pascal's triangle works very well, and better the longer the filter is.

Observation 8.21. *Let $\boldsymbol{a}$ be the samples of a digital sound, and let $\{c_i\}_{i=0}^{2k}$ be the numbers in row $2k$ of Pascal's triangle. Then the sound with samples $\boldsymbol{b}$ given by*

$$b_i = \begin{cases} a_i, & \text{for } i = 0, 1, \ldots, k-1; \\ \left(\sum_{j=0}^{2k} c_j a_{i+j-k}\right)\Big/ 2^k, & \text{for } 1 < i < N-1; \\ a_i, & \text{for } i = N-k+1, N-k+2, \ldots, N. \end{cases} \tag{8.4}$$

has reduced treble compared with the sound given by the samples $\boldsymbol{a}$.

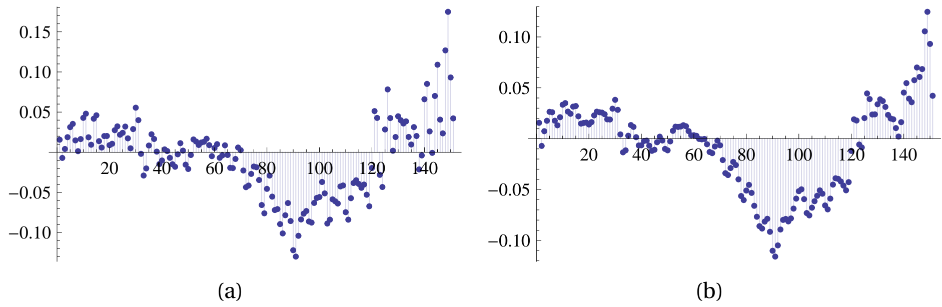


Figure 8.10. Reducing the treble. Figure (a) shows the original sound signal, while the plot in (b) shows the result of applying the filter from row 4 of Pascal's triangle.

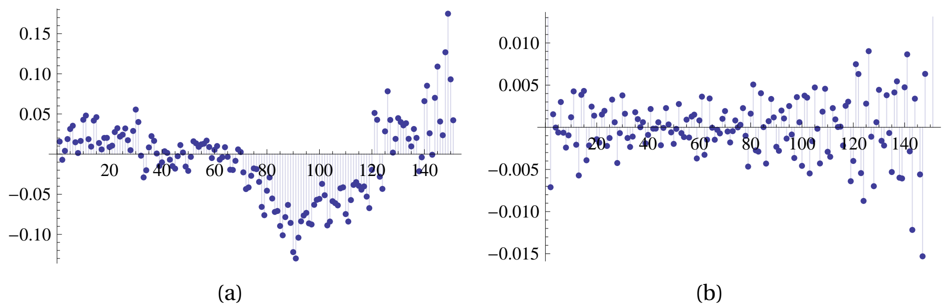


Figure 8.11. Reducing the bass. Figure (a) shows the original sound signal, while the plot in (b) shows the result of applying the filter in (8.5).

An example of the result of the averaging is shown in figure 8.10. Figure (a) shows a real sound sampled at CD-quality (44 100 samples per second). Figure (b) shows the result of applying the averaging process in (8.3). We see that the oscillations have been reduced, and if we play the sound it has considerably less treble.

Reducing the bass. Another common option in an audio system is reducing the bass. This corresponds to reducing the low frequencies in the sound, or equivalently, the slow variations in the sample values. It turns out that this can be accomplished by simply changing the sign of the coefficients used for reducing the treble. We can for instance change the filter described in (8.3) to

$$b_i = \begin{cases} a_i, & \text{for } i = 0, 1; \\ (a_{i-2} - 4a_{i-1} + 6a_i - 4a_{i+1} + a_{i+2})/16, & \text{for } 1 < i < N-1; \\ a_i, & \text{for } i = N-1, N. \end{cases} \tag{8.5}$$

An example is shown in figure 8.11. The original signal is shown in figure (a) and the result in figure (b). We observe that the samples in (b) oscillate much more than the samples in (a). If we play the sound in (b), it is quite obvious that the bass has disappeared almost completely.

Observation 8.22. *Let* $\boldsymbol{a}$ *be the samples of a digital sound, and let* $\{c_i\}_{i=0}^{2k}$ *be the numbers in row* $2k$ *of Pascal's triangle. Then the sound with samples* $\boldsymbol{b}$ *given by*

$$b_i = \begin{cases} a_i, & \text{for } i = 0, 1, \dots, k-1; \\ \left(\sum_{j=0}^{2k} (-1)^{k-j} c_j a_{i+j-k}\right) \Big/ 2^k, & \text{for } 1 < i < N-1; \\ a_i, & \text{for } i = N-k+1, N-k+2, \dots, N. \end{cases} \tag{8.6}$$

has reduced bass compared to the sound given by the samples $\boldsymbol{b}$.

8.4 More advanced sound processing

The operations on digital sound described in section 8.3 are simple and can be performed directly on the sample values. We saw in section 8.1.3 that a sound defined for continuous time could be decomposed into different frequency components, see theorem 8.8. The same can be done for digital sound with a digital version of the Fourier decomposition. When the sound has been decomposed into frequency components, the bass and treble can be adjusted by adjusting the corresponding frequencies. This is part of the field of *signal processing*.

8.4.1 The Discrete Cosine Transform

In Fourier analysis a sound is decomposed into sines and cosines. For digital sound a close relative, the Discrete Cosine Transform (DCT) is often used instead. This just decomposes the digital signal into cosines with different frequencies. The DCT is particularly popular for processing the sound before compression, so we will consider it briefly here.

Definition 8.23 (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 the sequence* $\boldsymbol{v}$ *whose terms are given by*

$$v_s = \frac{1}{\sqrt{n}} \sum_{r=0}^{n-1} u_r \cos\left(\frac{(2r+1)s\pi}{2n}\right), \quad \text{for } s = 0, \dots, n-1. \tag{8.7}$$

With the DCT we compute the sequence $\boldsymbol{v}$. It turns out that we can get back to the $\boldsymbol{u}$ sequence by computations that are very similar to the DCT. This is called the inverse DCT.

Theorem 8.24 (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* (8.7). *Then $\boldsymbol{u}$ can be recovered from $\boldsymbol{v}$ via the formula*

$$u_r = \frac{1}{\sqrt{n}}\left(v_0 + 2\sum_{s=1}^{n-1} v_s \cos\left(\frac{(2r+1)s\pi}{2n}\right)\right), \quad \text{for } r = 0, \ldots, n-1. \tag{8.8}$$

The two formulas (8.7) and (8.8) allow us to switch back and forth between two different representations of the digital sound. The sequence $\boldsymbol{u}$ is often referred to as representation in the *time domain*, while the sequence $\boldsymbol{v}$ is referred to as representation in the *frequency domain*. There are fast algorithms for performing these operations, so switching between the two representations is very fast.

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 8.12. 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 (in this particular case where $N = 5$), 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.

8.5 Lossy compression of digital sound

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-

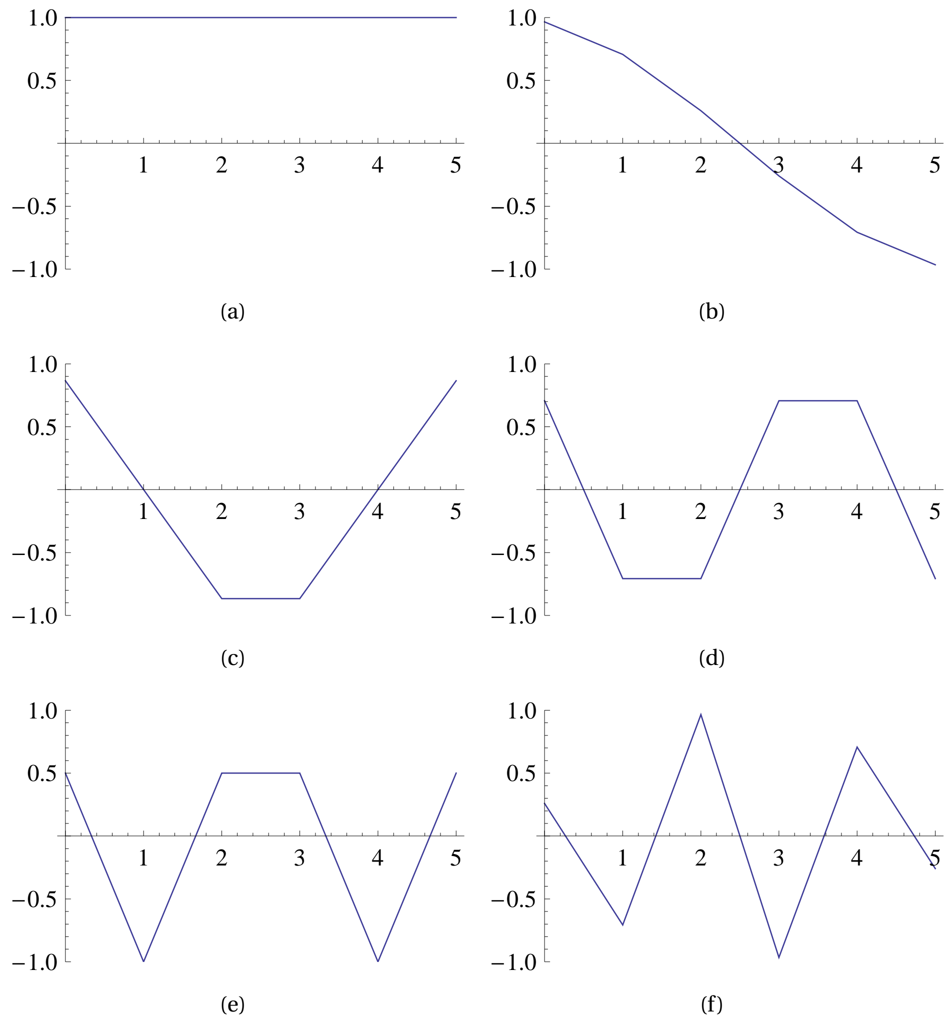


Figure 8.12. The 6 different versions of the cos function used in DCT for $n = 6$. The plots show piecewise linear functions, but this is just to make the plots more readable: Only the values at the integers $0, \ldots, 5$ are used.

values will be virtually 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. When the sound is to be played back, we first convert the sdjusted DCT-values to the time domain with the inverse DCT as given in theorem 8.24.

Example 8.25. Let us test a naive compression strategy based on the above idea.

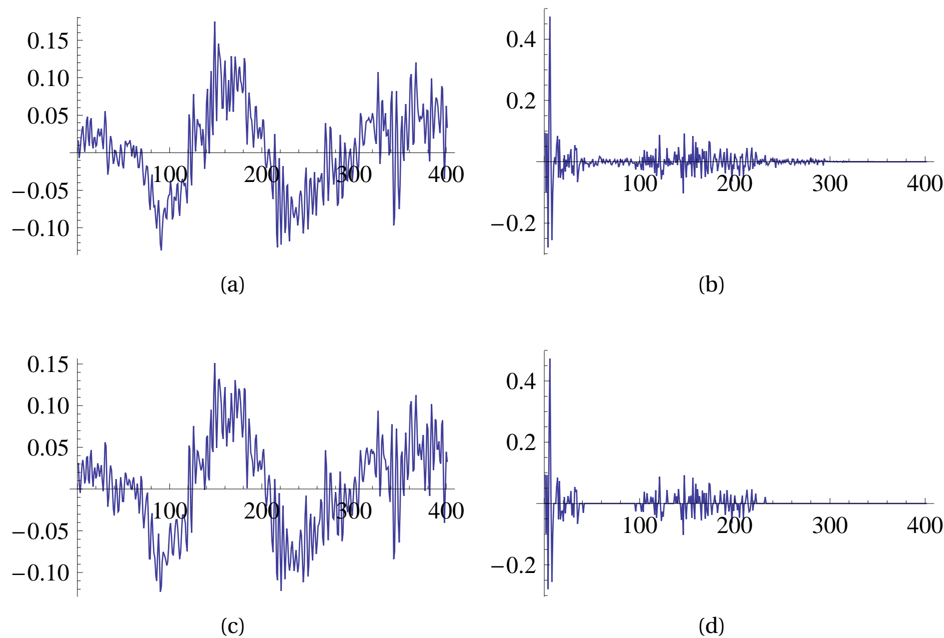


Figure 8.13. The signal in (a) is a small part of a song. The plot in (b) shows the DCT of the signal. In (d), all values of the DCT that are smaller than 0.02 in absolute value have been set to 0, a total of 309 values. In (c) the signal has been reconstructed from these perturbed values of the DCT. Note that all signals are discrete; the values have been connected by straight lines to make it easier to interpret the plots.

The plots in figure 8.13 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 8.24; 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).

We test this compression strategy on a data set that consists of 300 001 points. We compute the DCT and set all values smaller than a suitable tolerance to 0. 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. 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.

The approach to compression that we have outlined in the above example is essentially what is used in practice. The difference is that commercial software

does everything in a more sophisticated way and thereby gets better compression rates.

Fact 8.26 (Basic idea behind audio compression). *Suppose a digital audio signal* $\boldsymbol{u}$ *is given. To compress* $\boldsymbol{u}$*, perform the following steps:*

1. *Rewrite the signal* $\boldsymbol{u}$ *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 adjusted frequency information with some lossless coding method.*

All the lossy compression strategies used in the commercial formats that we review below, use the strategy in fact 8.26. 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.

8.6 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 code 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.

8.7 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 digital audio formats, both lossy and lossless ones.

8.7.1 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 significand. 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 rea-

sonably 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 sample. 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.

8.7.2 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 com-

monly 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.

8.7.3 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 kb/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 to MP3 and AAC: It uses the MDCT to transform groups of samples to the frequency domain, it then applies a psycho-acoustic 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 complicated 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 functions. 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, trigonometric 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 information 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 increasing 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.

The basic idea of Taylor approximation is to construct an approximating polynomial to a function f from the values of f and its first derivatives at a single point. A natural alternative is to construct a polynomial approximation from a selection of distinct function values of f in a suitable interval; 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 The Taylor polynomial with remainder

A discussion of Taylor polynomials involves two parts: The Taylor polynomial itself, and the error, the remainder, committed in approximating a function by a polynomial. Let us consider each of these in turn.

9.1.1 The Taylor polynomial

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 $T_n(f;a)$ (sometimes shortened to $T_n(x)$) 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 (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.

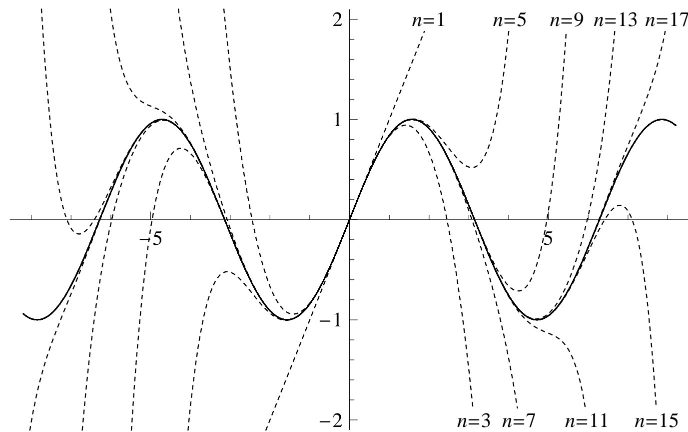


Figure 9.1. The Taylor polynomials of $\sin x$ (around $a = 0$) for degrees 1 to 17.

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 an ever larger interval.

Formula (9.2) is a classical result of calculus which is proved in most calculus books. Note however that the polynomial in (9.2) is written in non-standard form.

Observation 9.3. *In the derivation of the Taylor polynomial, the manipulations simplify if polynomials of degree n are written as*

$$p_n(x) = c_0 + c_1(x-a) + c_2(x-a)^2 + \cdots + c_n(x-a)^n.$$

This is an important observation: It is wise to adapt the form of the polynomial to the problem that is to be solved. We will see another example of this when we discuss interpolation below.

The elementary exponential and trigonometric functions have very simple and important Taylor polynomials.

Example 9.4 (The Taylor polynomial of e^x). The function $f(x) = e^x$ has the nice property that $f^{(n)}(x) = e^x$ for all integers $n \geq 0$. The Taylor polynomial about $a = 0$ is therefore very simple since $f^{(n)}(0) = 1$ for all n. The general term in the Taylor polynomial then becomes

$$\frac{(x-a)^k f^{(k)}(a)}{k!} = \frac{x^k}{k!}.$$

This means that the Taylor polynomial of degree n about $a = 0$ for $f(x) = e^x$ is given by

$$T_n(x) = 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!}.$$

For the exponential function the Taylor polynomials will be very good approximations for large values of n. More specifically, it can be shown that for any value of x, the difference between $T_n(x)$ and e^x can be made as small as we wish if we just let n be big enough. This is often expressed by writing

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \cdots.$$

It turns out that the Taylor polynomials of the trigonometric functions $\sin x$ and $\cos x$ converge in a similar way. In exercise 6 these three Taylor polynomials are linked together via a classical formula.

9.1.2 The remainder

The Taylor polynomial $T_n(f)$ 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 9.5. *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*

$$R_n(f;a)(x) = \frac{1}{n!}\int_a^x f^{(n+1)}(t)(x-t)^n \, dt. \tag{9.3}$$

The remainder may also be written as

$$R_n(f;a)(x) = \frac{(x-a)^{n+1}}{(n+1)!} f^{(n+1)}(\xi), \tag{9.4}$$

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 9.6. 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 (9.4), 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}.

We want to find the smallest n such that

$$\left|\frac{x^{n+1}}{(n+1)!}f^{(n+1)}(\xi)\right| \le 10^{-5}, \tag{9.5}$$

where the function $f(x)=\sin x$ 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

$$\left|f^{(n+1)}(\xi)\right| \le 1. \tag{9.6}$$

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 be much more difficult. If we insert the estimate (9.6) on the left in (9.5), 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 (9.5) will also be satisfied. Since $x\in[-1,1]$ we know that $|x|\le 1$ so this last inequality will

be satisfied if

$$\frac{1}{(n+1)!} \leq 10^{-5}. \tag{9.7}$$

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 (9.7) 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 check the approximation at $x = 1$, we find $p_8(1) \approx 0.8414682$. Comparing with the exact value $\sin 1 \approx 0.8414710$, we find that the error is roughly 2.73×10^{-6}, which is close to the upper bound $1/9!$ 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 (9.4) will be most useful for us, and for easy reference we record the complete Taylor expansion in a corollary.

Corollary 9.7. *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,*

$$f(x) = f(a) + (x-a)f'(a) + \cdots + \frac{(x-a)^n}{n!} f^{(n)}(a) + \frac{(x-a)^{n+1}}{(n+1)!} f^{(n+1)}(\xi_x), \tag{9.8}$$

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 (9.8) lets us control the error in the Taylor approximation. It turns out that the error behaves quite differently for different functions.

Example 9.8 (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

$$e(x) = \frac{(x-a)^{n+1}}{(n+1)!} f^{(n+1)}(\xi). \tag{9.9}$$

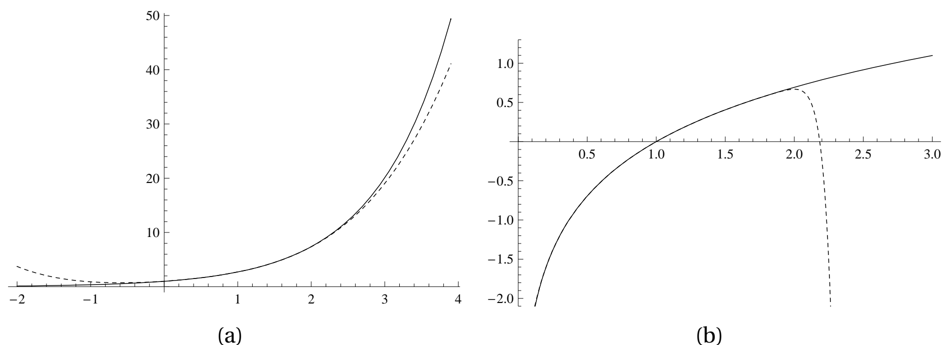


Figure 9.2. In (a) the Taylor polynomial of degree 4 about the point $a = 1$ for the function $f(x) = e^x$ is shown. Figure (b) shows the Taylor polynomial of degree 20 for the function $f(x) = \log x$, also about the point $a = 1$.

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 what the value of b is.

Example 9.9 (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 9.8.

Example 9.10 (Taylor polynomials for $f(x) = \ln x$). The plot in figure 9.2 shows the logarithm 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.

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

$$f^{(k)}(x) = (-1)^{k+1}(k-1)!\,x^{-k}, \quad k \geq 1. \tag{9.10}$$

Since $a = 1$, this means that the general term in the Taylor polynomial is

$$\frac{(x-1)^k}{k!} f^{(k)}(1) = (-1)^{k+1}\frac{(x-1)^k}{k}.$$

The Taylor expansion (9.8) therefore 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 therefore $\xi^{-n-1} < 1$. The error is then 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 (9.3) 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 $c > 1$. 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).

Exercises for Section 9.1

1. Mark each of the following statements as true or false.

(a). A function can have an infinite number of Taylor polynomials of a given order n.

(b). The Taylor polynomial of a sum of functions $f(x)+g(x)$ of degree n is equal to the sum of the Taylor polynomials of $f(x)$ and $g(x)$, i.e. $T_n(f+g;a)(x) = T_n(f;a)(x) + T_n(g;a)(x)$.

(c). The Taylor polynomial of a product of functions $f(x)g(x)$ of degree n is equal to the product of the Taylor polynomials of $f(x)$ and $g(x)$, i.e. $T_n(fg;a)(x) = T_n(f;a)(x)T_n(g;a)(x)$.

2. Find the correct alternative in the following multiple choice exercises.

(a). (Mid-term 2008) Suppose we compute the Taylor polynomial of degree n about the point $a=0$ for the function $f(x)=\cos(x)$; what can we then say about the remainder $R_n(x)$?

☐ For every x the remainder will increase when n increases.

☐ For any real number, the remainder will approach 0 when n tends to ∞.

☐ The remainder is 0 everywhere.

☐ The remainder will tend to 0 for $x \in [-\pi,\pi]$, but not for other values of x.

(b). (Mid-term 2010) You are going to approximate the function $f(x)=e^x$ with a Taylor polynomial of degree n on the interval $[0,1]$, expanded about $a=0$. It turns out that the error is bounded by

$$\frac{3x^{n+1}}{(n+1)!}.$$

What is the lowest degree n that causes the error to be smaller than 0.01 for all x in the interval $[0,1]$?

☐ $n=1$

☐ $n=3$

☐ $n=4$

☐ $n=5$

(c). (Mid-term 2011) For which value of c will the Taylor-polynomial of degree 3 around $a = 0$ for the function $f(x) = \sin(x) - 2x/(c + x^2)$ equal to $x^3/3$?

☐ $c = 1$

☐ $c = 0$

☐ $c = -1$

☐ $c = 2$

(d). (Mid-term 2008) What is the Taylor polynomial of degree 2 about $a = 0$ for the function $f(x) = x^3$?

☐ x^3

☐ x^2

☐ 0

☐ $1 + 3x + 6x^2$

(e). (Mid-term 2008) What is the Taylor polynomial of degree 2 about $a = 1$ for the function $f(x) = x^3$?

☐ x^2

☐ 0

☐ $1 + 3x + 3x^2$

☐ $1 - 3x + 3x^2$

3. In this exercise we are going to see that the calculations simplify if we adapt the form of a polynomial to the problem to be solved. The function f is a given function to be approximated by a quadratic polynomial near $x = a$, and it is assumed that f can be differentiated twice at a.

(a). Assume that the quadratic Taylor polynomial is on the form $p(x) = b_0 + b_1 x + b_2 x^2$, and determine the unknown coefficients from the three conditions $p(a) = f(a)$, $p'(a) = f'(a)$, $p''(a) = f''(a)$.

(b). Repeat (a), but write the unknown polynomial in the form $p(x) = b_0 + b_1(x - a) + b_2(x - a)^2$.

4. Find the second order Taylor approximation of the following functions at the given point a.

(a). $f(x) = x^3$, $a = 1$

(b). $f(x) = 12x^2 + 3x + 1$, $a = 0$

(c). $f(x) = 2^x$, $a = 0$

5. In many contexts, the approximation $\sin x \approx x$ is often used.

(a). Explain why this approximation is reasonable.

(b). Estimate the error in the approximation for x in the interval $[0, 0.1]$

6. The Taylor polynomials of e^x, $\cos x$ and $\sin x$ expanded around zero are

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + \frac{x^6}{720} + \frac{x^7}{5040} + \cdots$$
$$\cos x = 1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} + \cdots$$
$$\sin x = x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + \cdots$$

Use the Taylor polynomial for e^x to formally calculate the Taylor polynomial of the complex exponential e^{ix}. Compare the result with the Taylor polynomials above, and explain why Euler's formula $e^{ix} = \cos x + i \sin x$ is reasonable.

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 The interpolation problem

Taylor polynomials generalise the tangent. In a similar way, interpolating polynomials are generalisations of the secant, see figure 9.3.

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. A general parabola has three free coefficients, and we should therefore expect to be able to force a parabola through three arbitrary points. More generally, suppose we have $n+1$ distinct numbers $\{x_i\}_{i=0}^n$ scattered throughout an interval $[a, b]$ where f is defined. Since a general polynomial of degree n has $n+1$ coefficients that can be chosen freely it is natural to try and find a polynomial of degree n with the same values as f at the numbers $\{x_i\}_{i=0}^n$.

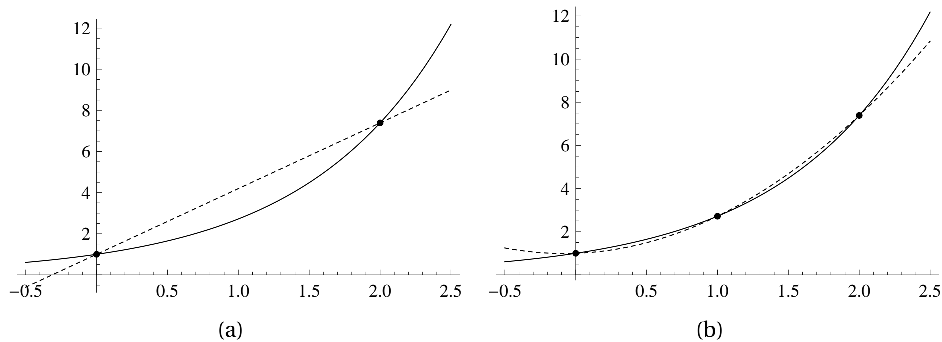


Figure 9.3. Interpolation of e^x at two points with a secant (a), and at three points with a parabola (b).

Problem 9.11 (Polynomial interpolation). *Let f be a given function defined in an interval $[a,b]$, and let $\{x_i\}_{i=0}^n$ be $n+1$ distinct numbers in $[a,b]$. The polynomial interpolation problem is to find a polynomial $p_n = P(f; x_0, \dots, x_n)$ of degree n that matches f at each x_i,*

$$p_n(x_i) = f(x_i), \quad \textit{for } i = 0, 1, \dots, n. \tag{9.11}$$

The numbers $\{x_i\}_{i=0}^n$ are called interpolation points*, the conditions* (9.11) *are called the* interpolation conditions*, and the polynomial $p_n = P(f; x_0, \dots, x_n)$ is called a polynomial* interpolant.

The notation $P(f; x_0, \dots, x_n)$ for a 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 just denote the interpolant by p_n when no confusion is possible.

In many situations the function f may not be known, just its function values at the points $\{x_i\}_{i=0}^n$, as in the following example.

Example 9.12. Suppose we want to find a polynomial that passes through the three points $(0,1)$, $(1,3)$, and $(2,2)$. In other words, we want to find a polynomial p such that

$$p(0) = 1, \quad p(1) = 3, \quad p(2) = 2. \tag{9.12}$$

Since there are three points it is natural to use a quadratic polynomial, i.e., we assume that $p(x) = c_0 + c_1 x + c_2 x^2$. If we insert this in the conditions (9.12) we

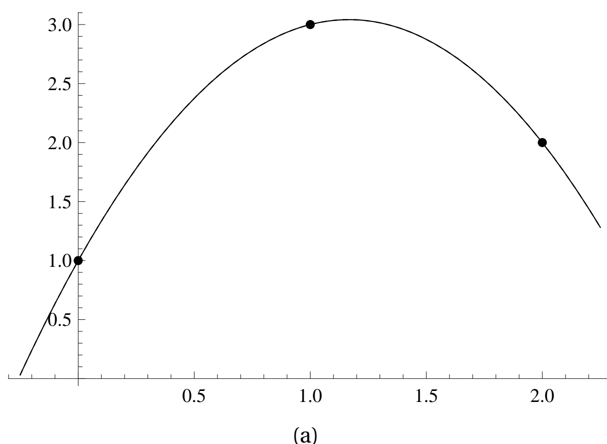


Figure 9.4. Three interpolation points and the corresponding quadratic interpolating polynomial.

obtain the three equations

$$\begin{aligned} 1 &= p(0) = c_0, \\ 3 &= p(1) = c_0 + c_1 + c_2, \\ 2 &= p(2) = c_0 + 2c_1 + 4c_2. \end{aligned}$$

We solve these and find $c_0 = 1$, $c_1 = 7/2$, and $c_2 = -3/2$, so p is given by

$$p(x) = 1 + \frac{7}{2}x - \frac{3}{2}x^2.$$

A plot of this polynomial and the interpolation points is shown in figure 9.4.

There are at least four questions raised by problem 9.11: Is there a polynomial of degree n that satisfies the interpolation conditions (9.11)? How many such polynomials are there? How can we find one such polynomial? What is a convenient way to write an interpolating polynomial?

9.2.2 The Newton form of the interpolating polynomial

We start by considering the last of the four questions above. We have already seen that by writing polynomials in a particular form, the computations of the Taylor polynomial simplified. This is also the case for interpolating polynomials.

Definition 9.13 (Newton form). *Let* $\{x_i\}_{i=0}^n$ *be* $n+1$ *distinct real numbers. The Newton form of a polynomial of degree* n *is an expression in the form*

$$p_n(x) = c_0 + c_1(x-x_0) + c_2(x-x_0)(x-x_1) + \cdots \\ + c_n(x-x_0)(x-x_1)\cdots(x-x_{n-1}). \quad (9.13)$$

The advantage of the Newton form will become evident when we consider some examples.

Example 9.14 (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 have to choose $c_0 = f(x_0)$,

$$p_0(x) = f(x_0).$$

Example 9.15 (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 $f(x_0) = p_1(x_0) = c_0$, while interpolation at x_1 yields

$$f(x_1) = p_1(x_1) = f(x_0) + c_1(x_1 - x_0).$$

Together this means that

$$c_0 = f(x_0), \quad c_1 = \frac{f(x_1) - f(x_0)}{x_1 - x_0}. \quad (9.14)$$

We note that c_0 has the same value as in the case $n=0$.

Example 9.16 (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 leads to 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 $n=1$ so c_0 and c_1 must be given by (9.14) also when $n=2$. 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

$$c_2 = \frac{\frac{f(x_2)-f(x_1)}{x_2-x_1} - \frac{f(x_1)-f(x_0)}{x_1-x_0}}{x_2 - x_0}. \tag{9.15}$$

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 $\big(x_0, f(x_0)\big)$, $\big(x_1, f(x_1)\big)$, ..., $\big(x_k, f(x_k)\big)$. This becomes clear if we write down all the equations,

$$\begin{aligned}
f(x_0) &= c_0,\\
f(x_1) &= c_0 + c_1(x_1 - x_0),\\
f(x_2) &= c_0 + c_1(x_2 - x_0) + c_2(x_2 - x_0)(x_2 - x_1),\\
&\ \ \vdots\\
f(x_k) &= c_0 + c_1(x_k - x_0) + c_2(x_k - x_0)(x_k - x_1) + \cdots\\
&\qquad\qquad + c_{k-1}(x_k - x_0)\cdots(x_k - x_{k-2}) + c_k(x_k - x_0)\cdots(x_k - x_{k-1}).
\end{aligned} \tag{9.16}$$

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 $\big(x_0, f(x_0)\big)$, $\big(x_1, f(x_1)\big)$, ..., $\big(x_k, f(x_k)\big)$, so the following theorem is immediate.

Theorem 9.17. *Let f be a given function and x_0, ..., x_n given and distinct interpolation points. There is a unique polynomial of degree n which interpolates f at these points. If the interpolating polynomial is expressed in Newton form,*

$$p_n(x) = c_0 + c_1(x - x_0) + \cdots + c_n(x - x_0)(x - x_1)\cdots(x - x_{n-1}), \tag{9.17}$$

then c_k depends only on $\big(x_0, f(x_0)\big)$, $\big(x_1, f(x_1)\big)$, ..., $\big(x_k, f(x_k)\big)$ which is indicated by the notation

$$c_k = f[x_0, \dots, x_k] \tag{9.18}$$

for $k = 0, 1, \dots, n$. The interpolating polynomials p_n and p_{n-1} are related by

$$p_n(x) = p_{n-1}(x) + f[x_0, \dots, x_n](x - x_0)\cdots(x - x_{n-1}).$$

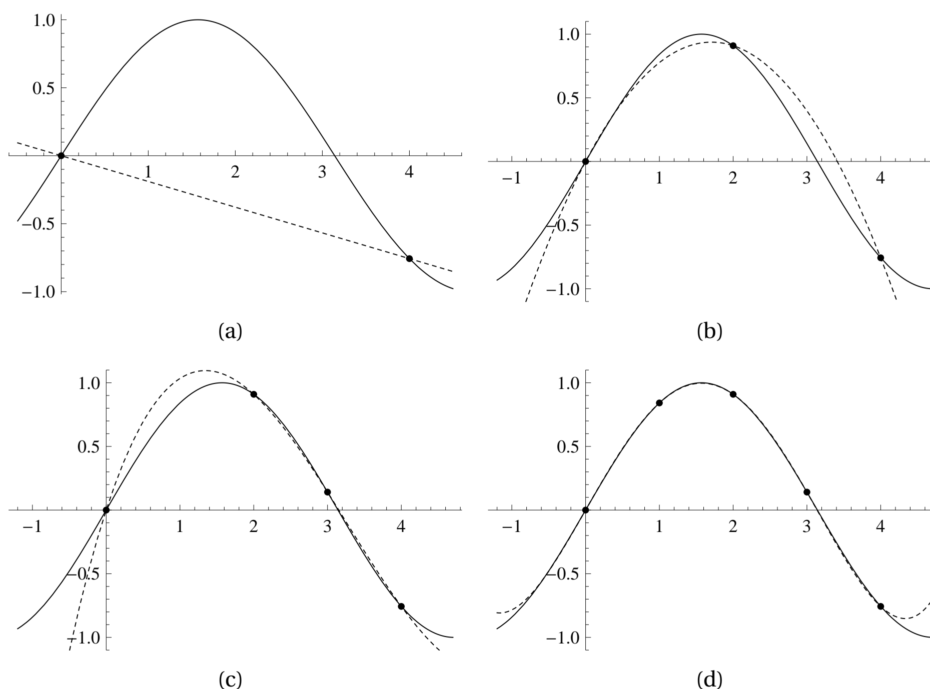


Figure 9.5. Interpolation of $\sin x$ with a line (a), a parabola (b), a cubic (c), and a quartic polynomial (d).

Proof. Most of this theorem is a direct consequence of writing the interpolating polynomial in Newton form, which becomes

$$p_n(x) = f[x_0] + f[x_0,x_1](x-x_0) + \cdots + f[x_0,\ldots,x_n](x-x_0)\cdots(x-x_{n-1}) \quad (9.19)$$

when we write the coeefficients as in (9.18). The coefficients can be computed, one by one, from the equations (9.16), starting with c_0. The uniqueness follows since there is no choice in solving the equations (9.16); there is one and only one solution. ∎

Some examples of interpolation are shown in figure 9.5. Note how the quality of the approximation improves with increasing degree.

Theorem 9.17 answers the questions raised above: Problem 9.11 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 more concrete example will illustrate the procedure further.

Example 9.18. 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. In this case the Newton

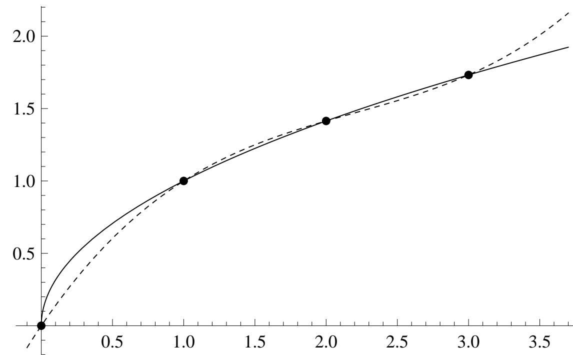


Figure 9.6. The function $f(x) = \sqrt{x}$ (solid) and its cubic interpolant at the four points 0, 1, 2, and 3 (dashed).

form is given by

$$p_3(x) = c_0 + c_1 x + c_2 x(x-1) + c_3 x(x-1)(x-2).$$

The interpolation conditions become

$$\begin{aligned}
0 &= c_0, \\
1 &= c_0 + c_1, \\
\sqrt{2} &= c_0 + 2c_1 + 2c_2, \\
\sqrt{3} &= c_0 + 3c_1 + 6c_2 + 6c_3.
\end{aligned}$$

Not surprisingly, the equations are triangular and we find

$$c_0 = 0, \quad c_1 = 1, \quad c_2 = -(1 - \sqrt{2}/2), \quad c_3 = (3 + \sqrt{3} - 3\sqrt{2})/6$$

Figure 9.6 shows a plot of this interpolant.

We emphasise that the Newton form is just one way to write the interpolating polynomial — there are many alternatives. One of these is the *Lagrange form* which is discussed in exercise 3 below.

9.2.3 Evaluating the Newton form

Interpolation is a technique that is sometimes implemented in a computer, so it is helpful to consider the details of how to compute the value $p_n(x)$ of a polynomial represented in the Newton form. Let us consider the cubic Newton form as a specific example,

$$p_3(x) = f_0 + f_1(x - x_0) + f_2(x - x_0)(x - x_1) + f_3(x - x_0)(x - x_1)(x - x_2). \qquad (9.20)$$

Given a number x, there is an elegant algorithm for computing the value $p_3(x)$ which is based on rewriting (9.20) slightly as

$$p_3(x) = f_0 + (x - x_0)\Big(f_1 + (x - x_1)\big(f_2 + (x - x_2) f_3\big)\Big). \tag{9.21}$$

To compute $p_3(x)$ we start from the inner-most parenthesis and then repeatedly multiply and add,

$$\begin{aligned} s_3 &= f_3, \\ s_2 &= (x - x_2)s_3 + f_2, \\ s_1 &= (x - x_1)s_2 + f_1, \\ s_0 &= (x - x_0)s_1 + f_0. \end{aligned}$$

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 variable.

Algorithm 9.19 (Horner's rule). *Let $x_0, \ldots, x_n$ be given numbers, and let $(f_k)_{k=0}^n$ be the coefficients of the polynomial*

$$p_n(x) = f_0 + f_1(x - x_0) + \cdots + f_n(x - x_0)\cdots(x - x_{n-1}). \tag{9.22}$$

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 $p_n(x)$.

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 9.7 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.

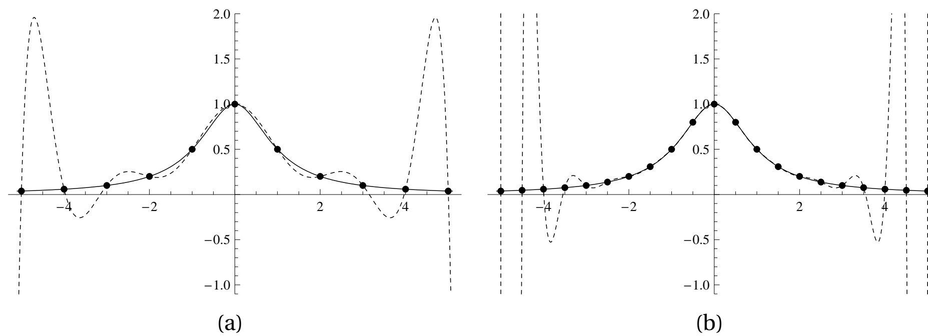


Figure 9.7. Interpolation of the function $f(x) = 1/(1+x^2)$ on the interval $[-5,5]$ with polynomials of degree 10 in (a), and degree 20 in (b). The points are uniformly distributed in the interval in each case.

Exercises for Section 9.2

1. Mark each of the following statements as true or false.

(a). The interpolating polynomial of a sum of functions $f(x)+g(x)$ of degree n is equal to the sum of the interpolating polynomials of $f(x)$ and $g(x)$

(b). The interpolating polynomial of a product of functions $f(x)g(x)$ of degree n is equal to the product of the interpolating polynomials of $f(x)$ and $g(x)$.

(c). There are several polynomials of degree n that interpolates a given function f at $n+1$ points.

2. (Mid-term 2009) We interpolate the function $f(x) = x^2$ with a polynomial p_3 of degree 3, at the points 0,1,2 and 3. What will the value of $p_3(4)$ be, i.e., the value of $p_3(x)$ at $x = 4$?

☐ 16

☐ 0

☐ 8

☐ 4

3. The data

x	0	1	3	4
$f(x)$	1	0	2	1

are given.

(a). Write the cubic interpolating polynomial in the form

$$p_3(x) = c_0(x-1)(x-3)(x-4)+c_1x(x-3)(x-4)+c_2x(x-1)(x-4)+c_3x(x-1)(x-3),$$

and determine the coefficients from the interpolation conditions. This is called the *Lagrange form* of the interpolating polynomial.

(b). Determine the Newton form of the interpolating polynomial.

(c). Verify that the solutions in (a) and (b) are the same.

4. In this exercise we are going to consider an alternative proof that the interpolating polynomial is unique.

(a). Suppose that there are two quadratic polynomials p_1 and p_2 that interpolate a function f at the three points x_0, x_1 and x_2. Consider the difference $p = p_2 - p_1$. What is the value of p at the interpolation points?

(b). Use the observation in (a) to prove that p_1 and p_2 must be the same polynomial.

(c). Generalise the results in (a) and (b) to polynomials of degree n.

5. Interpolation of a data set.

(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 $x_0, \ldots, x_n$ 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)?

6. Suppose we have the data

$$(0, y_0), \quad (1, y_1), \quad (2, y_2), \quad (3, y_3) \tag{9.23}$$

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.

(a). Determine the straight line p_1 that interpolates the two middle points in (9.23), 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 (9.23) 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?

9.3 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 different ways to write polynomials, and a problem may simplify considerably by adapting the form of the polynomial to the problem at hand.

CHAPTER 10

Zeros of Functions

An important part of the mathematics 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 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

$$\frac{\sqrt{\sqrt[3]{81-\sqrt{5793}}+\sqrt[3]{81+\sqrt{5793}}}}{2\sqrt[6]{2}\sqrt[3]{3}} \\ + \frac{1}{2}\sqrt{\frac{1}{3}\left(-\sqrt[3]{\frac{3}{2}\left(81-\sqrt{5793}\right)}-\sqrt[3]{\frac{3}{2}\left(81+\sqrt{5793}\right)}\right.} \\ \overline{\left. + \frac{18\sqrt[6]{2}\sqrt[3]{3}}{\sqrt{\sqrt[3]{81-\sqrt{5793}}+\sqrt[3]{81+\sqrt{5793}}}}\right)}$$

(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 -0.099900298805472842029, 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

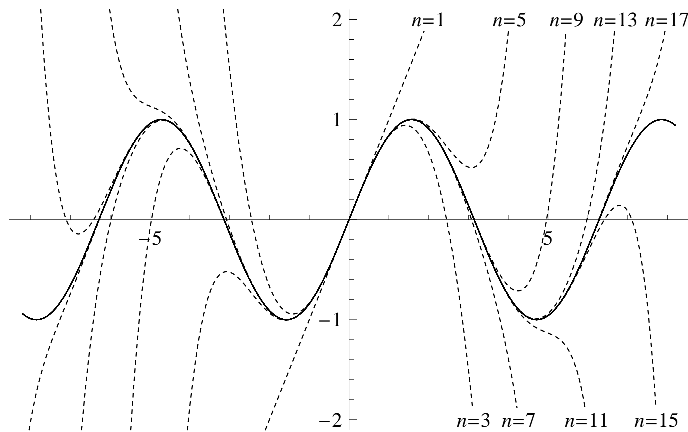


Figure 10.1. Plot with automatically placed labels.

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

$$x - \frac{x^3}{6} + \frac{x^5}{720} - \frac{x^7}{5040} + \frac{x^9}{362880} - 2 = 0.$$

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 $\big(x_k, (-1)^{k+1}2\big)$. 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.

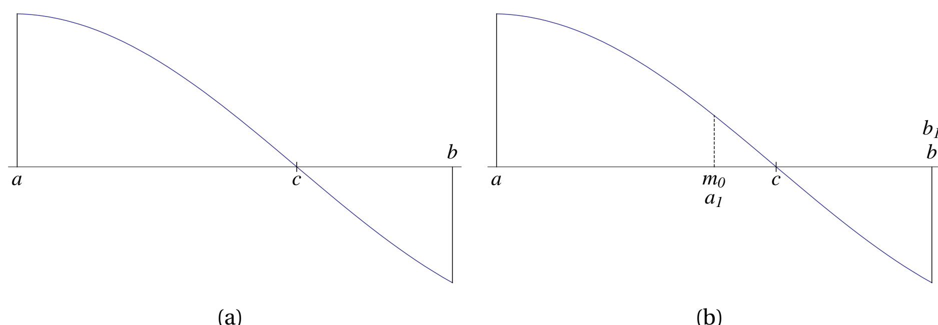


Figure 10.2. Illustration of the mean value theorem (a), and the first step of the Bisection method (b).

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 the 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 it can be found. However, based on the theorem it is easy to devise a method for finding good approximations to the zero.

Initially, 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 $[a,b]$, and with the property

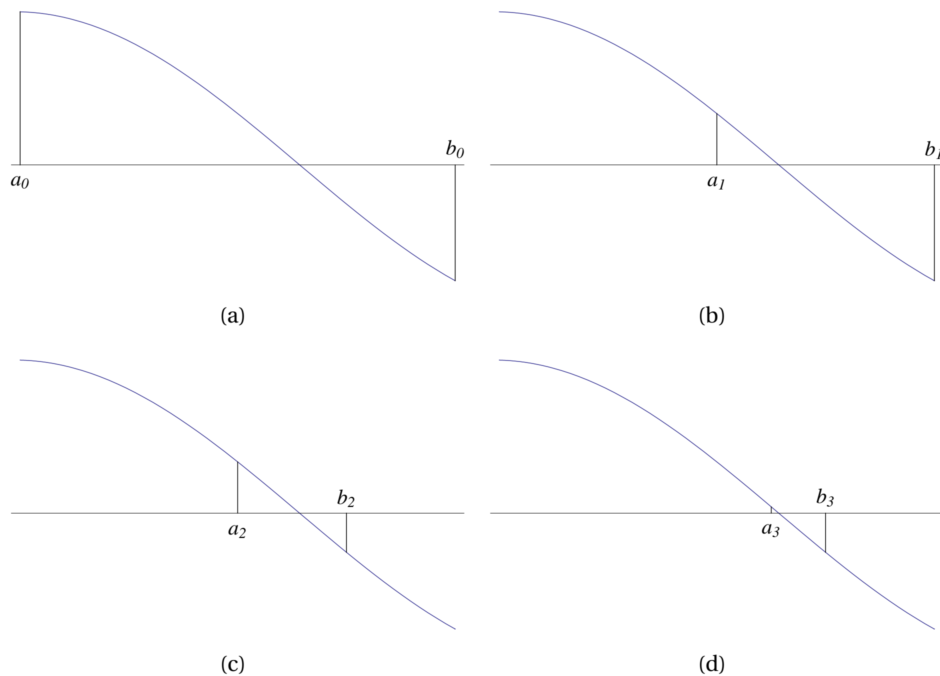


Figure 10.3. The first four steps of the bisection algorithm.

that 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 $i = 1, 2, \dots, N$
  $m_{i-1} = (a_{i-1} + b_{i-1})/2;$
  if $f(m_{i-1}) == 0$
    $a_i = b_i = m_{i-1};$
  if $f(a_{i-1})f(m_{i-1}) < 0$

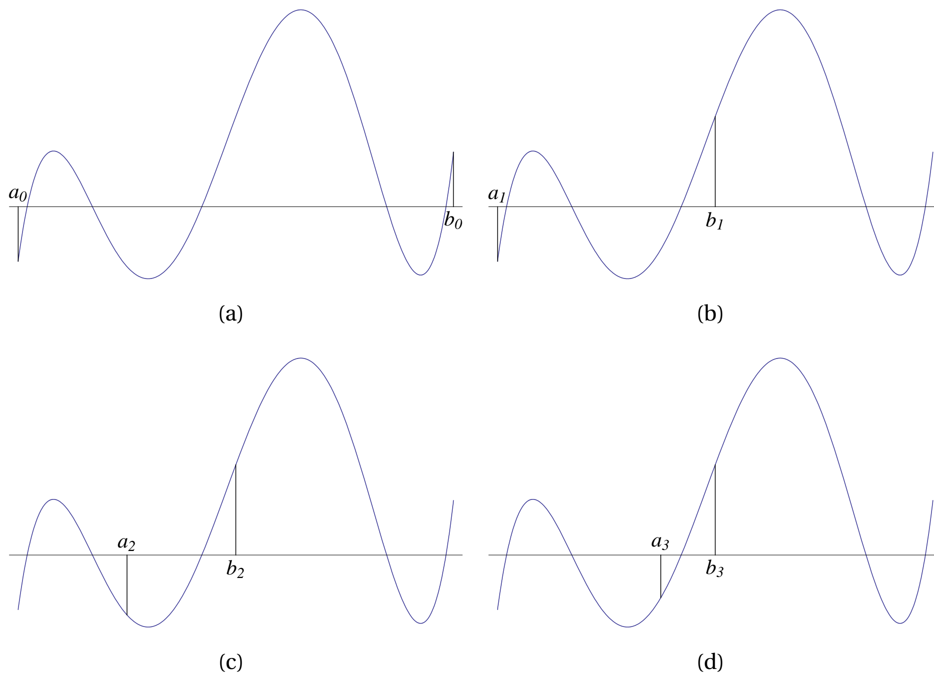


Figure 10.4. The first four steps of the bisection algorithm for a function with five zeros in the initial interval.

$a_i = a_{i-1}$;
$b_i = m_{i-1}$;
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 10.2 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 bounded by*

$$|c-m| \le \frac{b-a}{2}.$$

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*

$$|c-m_N| \le \frac{b-a}{2^{N+1}}. \tag{10.2}$$

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,

$$|c-m_N| \le \frac{b-a}{2^{N+1}} \le \epsilon.$$

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.

Observation 10.5. *Suppose that f has only one zero in $[a,b]$. If the number of bisections in the Bisection method is at least*

$$N \geq \frac{\ln(b-a) - \ln \epsilon}{\ln 2} - 1 \tag{10.3}$$

the error will be at most ϵ.

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 less 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 10.2 we find

$$\begin{aligned}
m_0 &= 1.50000000000, \\
m_1 &= 1.25000000000, \\
m_2 &= 1.37500000000, \\
&\vdots \\
m_{33} &= 1.41421356233.
\end{aligned}$$

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 advance knowledge of the zero c, or the approximation m_i, which is hardly possible.

Recall from observation 5.20 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 numbers.*

10.2.4 Revised algorithm

If we look back on algorithm 10.2, 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 magnitude of the numbers involved is very different from 1.

Instead, we 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 ≤ N and abserr > ϵ|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 more advanced software development.

Exercises for Section 10.2

1. Mark each of the following statements as true or false.

(a). The error bound in the bisection method is reduced by a factor 2 for each iteration.

(b). When using the bisection method, at a given iteration, we use the left endpoint as an approximation to the zero point.

(c). When using the bisection method, we may sometimes find that there may be a zero on both sides of the midpoint.

(d). In cases where there is more than zero in an interval, the bisection method will find all the zeros.

(e). If there is exactly one zero in the starting interval $[a,b]$, the bisection method will always converge to this zero.

2. Find the correct alternative in the following multiple choice exercises.

(a). (Mid-term 2006) We are trying to find the zeros of the function $f(x) = (x-3)(x^2-3x+2)$ using the bisection method. We start with the interval $[a,b] = [0,3.5]$, perform 1000 iterations and let x denote the last estimate for the zero. What will the result be?

☐ x close to $\sqrt{2}$
☐ No convergence
☐ x close to 2
☐ x close to 1

(b). We use the bisection method to find a zero of the function $f(x) = \cos(x)$ on the interval $[0,10]$, where x is given in radians. Then the approximated solution will converge to

☐ $\pi/2$

☐ $3\pi/2$

☐ $5\pi/2$

☐ The method will not converge

(c). (Mid-term 2005) We define a relative of the bisection method for solving the equation $f(x) = 0$, which we call the trisection method. Instead of dividing the interval into two equal parts each time, we divide it into three equal parts, and choose the subinterval where f has opposite signs at the ends. If this occurs for several subintervals we choose the subinterval which is furthest to the right on the real line. After the final iteration, the midpoint of the final interval is chosen as our estimate. We start with the interval $[0,1]$ and know that f is continuous and only has one root in this interval, but we do not know where the root is. Which is the smallest number of iterations that we need to use to be certain that the trisection method gives an absolute error less than 10^{-12}?

☐ 11

☐ 41

☐ 18

☐ 25

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). Run the Bisection method to compute an approximation to the root with the number of iterations that you found in (b). How does the actual error compare with ten correct digits?

(d). Make sure you are using 64 bit floating-point numbers and do 60 iterations. Verfiy that the error does not improve after about 54 iterations.

4. Repeat exercise 3, but use the function $f(x) = x^2 - 2$ with a suitable starting interval that contains the root $\sqrt{2}$. The first 20 digits of this root are

$$\sqrt{2} \approx 1.4142135623730950488.$$

5. Apply the Bisection method to the function $\sin x$ on the interval $[-1, 20]$ sufficiently many times to see which root is selected by the method in this case.

6. In this exercise we are going see how well the approximation (10.4) of the relative error works in practice. We use the function $f(x) = x^2 - 2$ and the root $\sqrt{2}$ for the tests.

> **(a).** Start with the interval $[1, 1.5]$ and perform 10 steps with the Bisection method. Compute the relative error in each step by using the approximation (10.4).
>
> **(b).** Compute the relative errors in the steps in (a) by instead using the approximation $\sqrt{2} \approx 1.414213562$ for the root. How do the approximations of the relative error from (a) compare with this?

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 $\big(a, f(a)\big)$ and $\big(b, f(b)\big)$ as an approximation to the zero instead of the midpoint, as shown in figure 10.5b. Recall that the secant is the same as the linear interpolant to f at the points a and b, see section 9.2.

Idea 10.10 (Secant idea). *Let f be a continuous function, let a and b be two points in its domain, and let*

$$s(x) = f(a) + \frac{f(b) - f(a)}{b - a}(x - a)$$

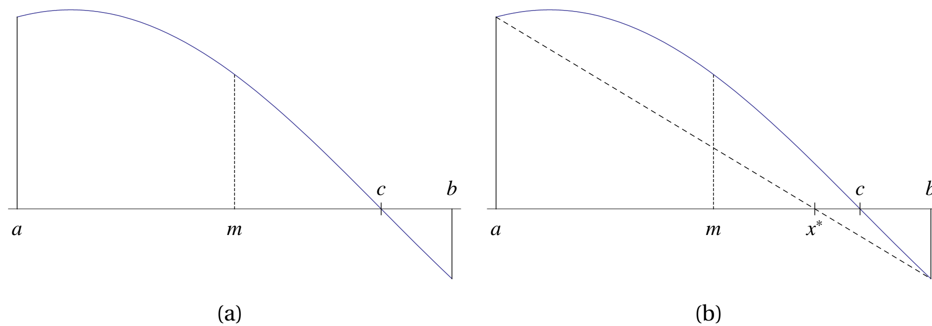


Figure 10.5. An example of the first step with the Bisection method (a), and the alternative approximation to the zero provided by the secant (b).

be the secant between the two points $\big(a, f(a)\big)$ *and* $\big(b, f(b)\big)$. *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 formula (10.5) 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 shown 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*

$$x_i = x_{i-1} - \frac{x_{i-1} - x_{i-2}}{f(x_{i-1}) - f(x_{i-2})} f(x_{i-1}), \quad i = 2, 3, \ldots, N, \tag{10.6}$$

will in certain situations converge to a zero of f.

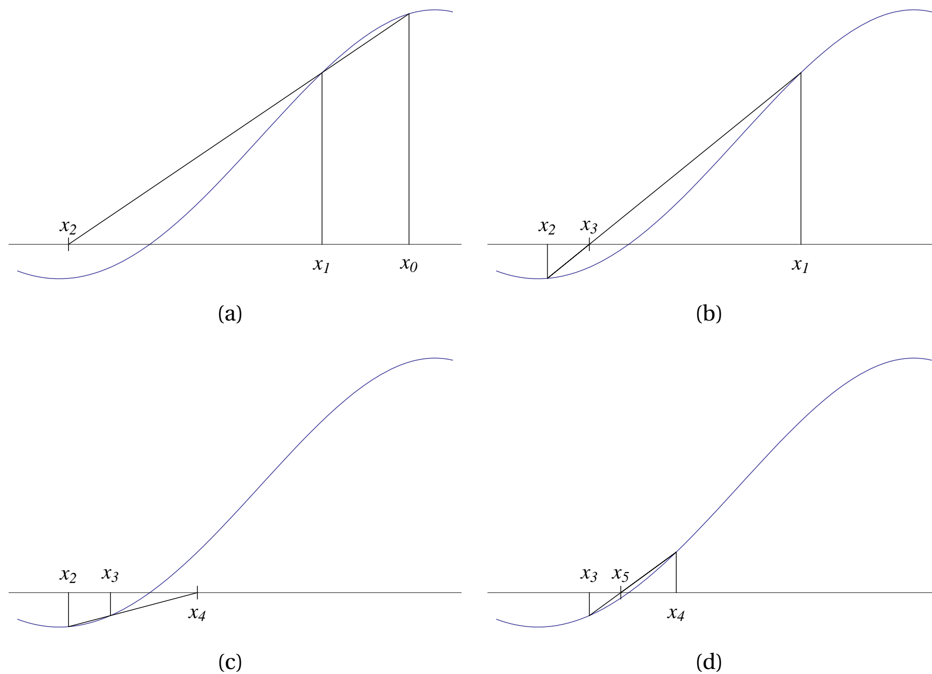


Figure 10.6. An example of the first four steps of the Secant method.

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.

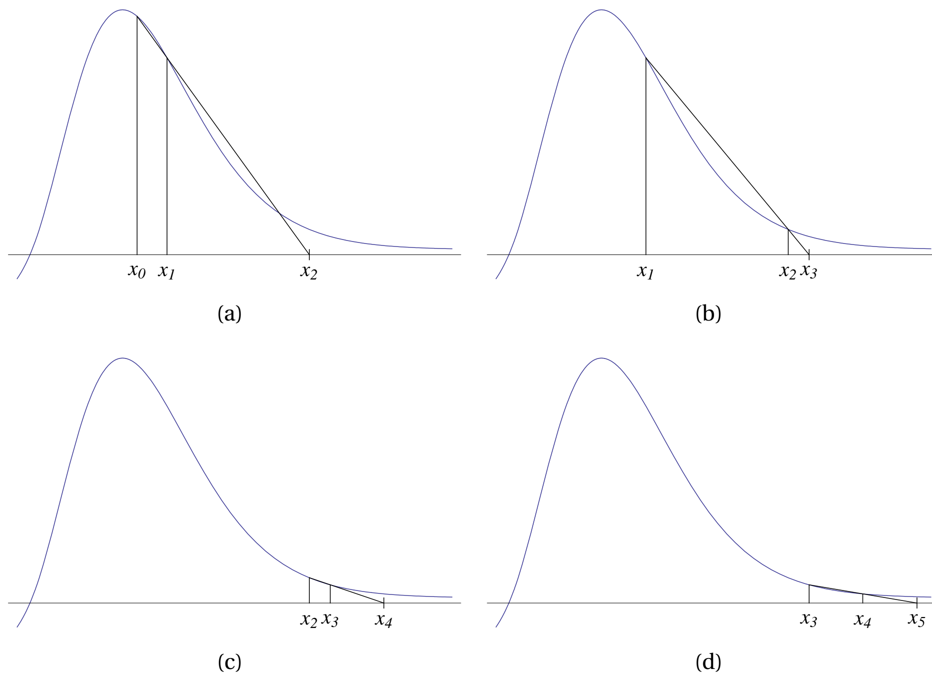


Figure 10.7. An example where the Secant method fails.

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 $|x_n - x_{n-1}| \leq \epsilon$.

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 $|x_n - x_{n-1}| \leq \epsilon |x_n|$. 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* 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*

$$|x_n - x_{n-1}| \leq \epsilon |x_n|$$

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 $i \leq N$ **and** $abserr \geq \epsilon|z|$
  $z = xp - f(xp)(xp - xpp)\big/\big(f(xp) - f(xpp)\big)$;
  $abserr = |z - xp|$;
  $xpp = xp$;
  $xp = z$;
  $i = i + 1$;

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 $|z| \geq \epsilon|z|$ 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 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 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*

$$|e_n| \leq K|e_{n-1}|^r, \quad n = 2, 3, \dots, \tag{10.7}$$

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. (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

$$\begin{aligned}
e_{k+1} &\lesssim e_k^r = 10^{-r} \approx 10^{-1.618} \approx 2.41 \times 10^{-2}, \\
e_{k+2} &\lesssim e_{k+1}^r \lesssim e_k^{r^2} = 10^{-r^2} \approx 2.41 \times 10^{-3}, \\
e_{k+3} &\lesssim 5.81 \times 10^{-5}, \\
e_{k+4} &\lesssim 1.40 \times 10^{-7}, \\
e_{k+5} &\lesssim 8.15 \times 10^{-12}, \\
e_{k+6} &\lesssim 1.43 \times 10^{-18}.
\end{aligned}$$

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.

Observation 10.15. *When the Secant method converges to a zero c with $f'(c) \neq 0$, the number of correct digits increases by about 62 % per iteration.*

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

$$\begin{aligned}
x_2 &\approx 1.42857142857142857, & e_2 &\approx 1.4 \times 10^{-2}, \\
x_3 &\approx 1.41463414634146341, & e_3 &\approx 4.2 \times 10^{-4}, \\
x_4 &\approx 1.41421568627450980, & e_4 &\approx 2.1 \times 10^{-6}, \\
x_5 &\approx 1.41421356268886964, & e_5 &\approx 3.2 \times 10^{-10}, \\
x_6 &\approx 1.41421356237309529, & e_6 &\approx 2.4 \times 10^{-16}.
\end{aligned}$$

This confirms the claim in observation 10.15.

Exercises for Section 10.3

1. Mark each of the following statements as true or false.

(a). If we use the secant method on a function that has exactly one zero, the method will always converge.

(b). When the Secant method converges to a zero c with $f'(c) \neq 0$, the number of correct digits increases by about a factor of 1.62 per iteration.

2. (Exam 2010) You are to use the secant method to find the zero of $x^3 - 2$ and start with the initial values $x_0 = -2$ and $x_1 = 2$. After one step, what is the approximate zero x^*?

☐ $x^* = -0.2$

☐ $x^* = 0$

☐ $x^* = 0.33$

☐ $x^* = 0.5$

3. For each of the following values of c, find a function f so that $f(c) = 0$. Then use the Secant method with this f to determine an approximation to c with 2 correct digits by hand, and run the secant method to compute an approximation to c with 15 correct digits.

(a). $c = \sqrt{3}$.

(b). $c = 2^{1/12}$.

(c). $c = e$, where $e = 2.71828\cdots$ is the base for natural logarithms.

4. Sketch the graphs of some functions and find an example where the Secant method will diverge.

5. In this exercise we are going to test the Secant method on the function $f(x) = (x-1)^3$ with the starting values $x_0 = 0.5$ and $x_1 = 1.2$.

(a). Perform 7 iterations with the Secant method, and compute the relative error at each iteration.

(b). How many correct digits are gained in each of the iterations, and how does this compare with observation 10.15? Explain your answer.

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 secant as an approximation to the zero of f. In Newton's method we use the tangent of f instead, i.e., the first-order Taylor polynomial of f at a given point.

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.

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,

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}, \quad n = 1, 2, \ldots \tag{10.9}$$

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 (nonlinear) 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 criterion is also exactly the same.

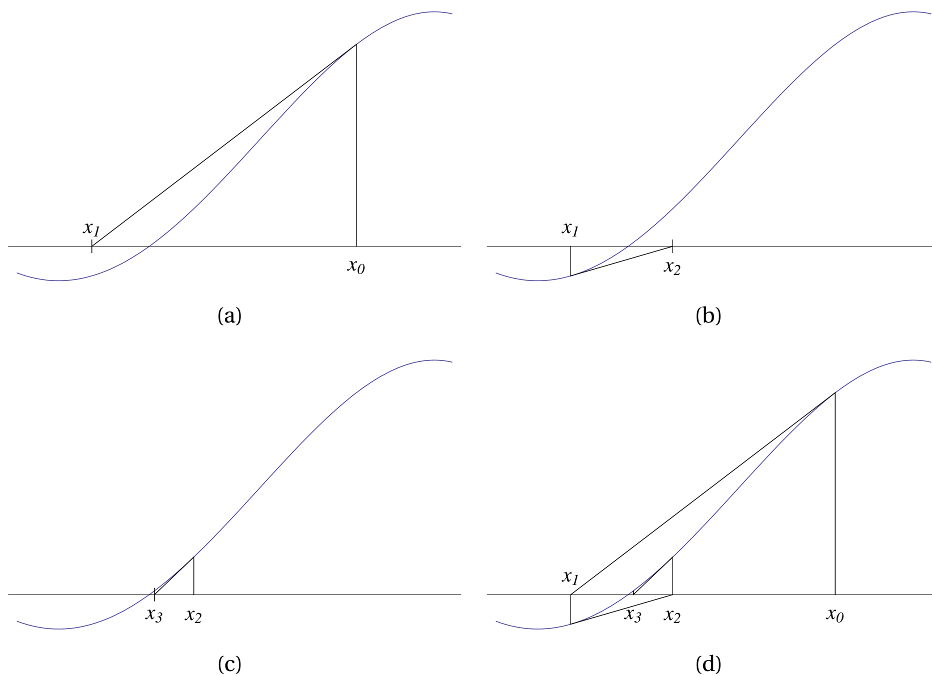


Figure 10.8. An example of the first three steps of Newton's method (a)–(c). The plot in shows a standard way of illustrating all three steps in one figure.

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

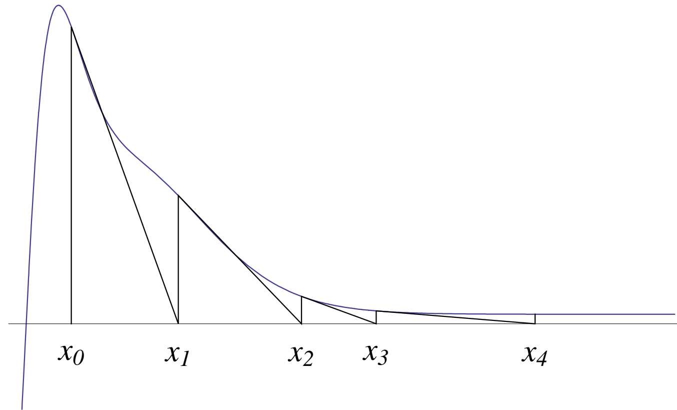


Figure 10.9. An example where Newton's method fails to converge because of a bad starting value.

that we may in some 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 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*

$$e_{n+1} = \frac{f''(\xi_n)}{2f'(x_n)} e_n^2, \tag{10.10}$$

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

$$e_{n+1} = e_n^2 \frac{f''(\xi_n)}{2 f'(x_n)},$$

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*

$$|e_{n+1}| \leq K |e_n|^2, \quad n = 1, 2, \ldots \tag{10.12}$$

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 $f'(c) \neq 0$. 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

$$
\begin{aligned}
e_{k+1} &\approx e_k^2 \approx 10^{-2},\\
e_{k+2} &\approx e_{k+1}^2 \approx 10^{-4},\\
e_{k+3} &\approx e_{k+2}^2 \approx 10^{-8},\\
e_{k+4} &\approx e_{k+3}^2 \approx 10^{-16}.
\end{aligned}
$$

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 $f'(c) \neq 0$, the number of correct digits roughly 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

$$
\begin{aligned}
x_1 &\approx 1.43823529411764706, & e_2 &\approx 2.3 \times 10^{-1},\\
x_2 &\approx 1.41441417057620594, & e_3 &\approx 2.4 \times 10^{-2},\\
x_3 &\approx 1.41421357659935635, & e_4 &\approx 2.0 \times 10^{-4},\\
x_4 &\approx 1.41421356237309512, & e_5 &\approx 1.4 \times 10^{-8},\\
x_5 &\approx 1.41421356237309505, & e_6 &\approx 7.2 \times 10^{-17}.
\end{aligned}
$$

We see that although we only use one starting value, which is further from the root than the 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.

Exercises for Section 10.4

1. Mark each of the following statements as true or false.

(a). If both the secant method and Newton's method converges, Newton's method will in general converge faster.

(b). Newton's method needs two initial values

2. Find the correct alternative in the following multiple choice exercises.

(a). (Mid-term 2007) We are discussing methods for finding solutions of the equation $f(x) = 0$, where f is a continuous function on the interval $[a, b]$.

☐ If $f(x)$ is a polynomial of degree 4 or higher, the zeros can only be found using numerical techniques.

☐ The bisection method gives a solution only if there is exactly one zero in $[a, b]$.

☐ The secant method can only be used when $f(x)$ has different signs at $x = a$ and $x = b$.

☐ If it works, Newton's method will converge faster than the bisection method.

(b). (Continuation exam 2010) We use Newton's method to find an approximation to the positive solution of $x^2 = 3$, with starting value $x_0 = 1$. Then x_2 is given by

☐ $x_2 = 1$

☐ $x_2 = 2$

☐ $x_2 = 9/4$

☐ $x_2 = 7/4$

(c). (Mid-term 2004) We apply Newton's method x_{n+1} to the function $f(x) = x^2 - A$ where A is a positive, real number. If we denote the error by $e_n = x_n - \sqrt{A}$, we have

☐ $e_{n+1} = \frac{e_n}{2x_n}$

☐ $e_{n+1} = \frac{e_n^2}{2x_n}$

☐ $e_{n+1} = \frac{e_n^2}{x_n^2}$

☐ $e_{n+1} = \frac{e_n e_{n-1}}{x_n}$

(d). (Exam 2008) We have a function $f(x)$ and we are going to find a numerical approximation to the solution of the equation $f(x) = 0$. Then:

☐ The secant method demands that $f'(x)$ is known.

☐ The secant method will usually converge faster than Newton's method.

☐ Newton's method will converge for all functions f.

☐ Newton's method will usually converge faster than the bisection method.

3. Perform 7 iterations with Newton's method with the function $f(x) = 1 - \ln x$ which has the root $x = e$, starting with $x_0 = 3$. How many correct digits are there in the final approximation?

4. 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 $x = \pi \approx 3.1415926535897932385$.

> **(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?

5. Repeat exercise 4 with the equation $(x - 10/3)^5 = 0$. Why do you think the error behaves differently than in exercise 4 for two of the methods? Hint: Take a careful look at the conditions in theorems 10.14 and 10.20.

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
>
> $$e_{n+1} = \frac{e_n^2}{2x_n} \tag{10.13}$$
>
> 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.

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.

8. Suppose that you are working with a function f where both f, f' and f'' are continuous 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

$$e_{n+1} = \frac{1}{2}\frac{f''(\xi_n)}{f'(x_n)}e_n^2 \quad \text{for } n \geq 0, \tag{10.14}$$

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).

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 even for a zero c where $f(c) = f'(c) = 0$, 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 subdivided 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.

CHAPTER 11

Numerical Differentiation

Differentiation is a basic mathematical operation with a wide range of applications in many areas of science. It is therefore important to have good methods to compute and manipulate derivatives. You probably learnt the basic rules of differentiation in school — symbolic methods suitable for pencil-and-paper calculations. Such methods are of limited value on computers since the most common programming environments do not have support for symbolic computations.

Another complication is the fact that in many practical applications a function is only known at a few isolated 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. When the position is known at all times (as a mathematical function), we can find the speed by symbolic differentiation. But when the position is only known at isolated times, this is not possible.

The solution is to use approximate methods of differentiation. In our context, these are going to be numerical methods. We are going to focus on the simplest method, including its limitations and error. In fact numerical differentiation is a good example for learning about both the advantages and the limitations of numerical methods.

We are also going to present a general strategy for deriving more advanced numerical differentiation 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.

The basic strategy for deriving numerical differentiation methods is to evaluate a function at a few points, find the polynomial that interpolates the function at these points, and use the derivative of this polynomial as an approximation to

the derivative of the function. This technique also allows us to keep track of the so-called *truncation error*, the mathematical error committed by differentiating the polynomial instead of the function itself. In addition to the truncation error, there are also *round-off* errors, which are unavoidable when we use floating-point numbers to perform calculations with real numbers.

The general idea of the chapter is to discuss the simplest method for numerical differentiation in detail in section 11.1, including a detailed error analysis. We will then describe the general strategy for deriving numerical differentiation methods in section 11.3 and illustrate how it works with some examples. In principle, the error analysis can be carried out in the same way for the more general methods, but the details are tedious so we just state the results.

Note that the methods for numerical integration in Chapter 12 are derived and analysed in much the same way as the differentiation methods in this chapter.

11.1 Newton's difference quotient and its truncation error

We start by introducing the simplest method for numerical differentiation, derive its error, and its sensitivity to round-off errors. The procedure used here for deriving the method and analysing the error can be extended quite simply to other methods.

Let us first explain what we mean by numerical differentiation.

Problem 11.1 (Numerical differentiation). *Let f be a given function that is 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 function values of f.*

A typical example is that f is given by a computer program (more specifically a function, procedure or method, depending on your 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 the derivative of a function, we must be clear about how it is defined. The standard definition of $f'(a)$ is by a limit process,

$$f'(a) = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h}. \tag{11.1}$$

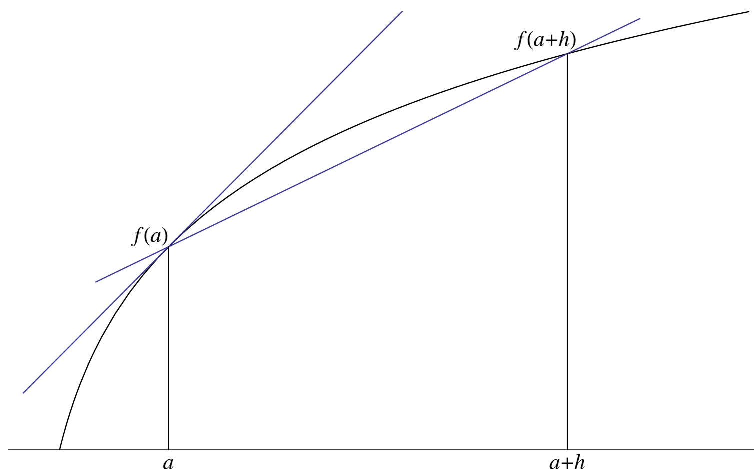


Figure 11.1. The secant of a function based at a and $a+h$, as well as the tangent at a.

This means that we compute the average change of f over a small interval around a and then check whether this average change tends to a limit when the width of the interval goes to zero.

In the following we will assume that the limit exists, in other words that f is differentiable at $x = a$. From the definition (11.1) we immediately have a natural approximation of $f'(a)$: We simply pick a positive number h and use the approximation

$$f'(a) \approx \frac{f(a+h) - f(a)}{h}. \tag{11.2}$$

Recall that the straight line p_1 that interpolates f at a and $a+h$ (the secant based at these points) is given by

$$p_1(x) = f(a) + \frac{f(a+h) - f(a)}{h}(x-a).$$

The derivative of this secant is exactly the right-hand side in (11.2) and corresponds to the secant's slope. The approximation (11.2) therefore corresponds to approximating f by the secant based at a and $a+h$, and using its slope as an approximation to the slope of f at a, see figure 11.1.

The tangent to f at a has the same slope as f at a, so we may also obtain the approximation (11.2) by considering the secant based at a and $a+h$ as an approximation to the tangent at a, see again figure 11.1.

Observation 11.2 (Newton's difference quotient). *The derivative of f at a can be approximated by*

$$f'(a) \approx \frac{f(a+h) - f(a)}{h}. \tag{11.3}$$

This approximation is referred to as Newton's difference quotient or just Newton's quotient.

An alternative to the approximation (11.3) is the left-sided version

$$f'(a) \approx \frac{f(a) - f(a-h)}{h}.$$

Not surprisingly, this approximation behaves similarly, and the analysis is also completely analogous to that of the more common right-sided version.

In later sections, we will derive several formulas like (11.2). Which formula to use in a particular situation, and exactly how to apply it, will have to be decided in each case.

11.1.2 The truncation error

We know that the formula $f'(a) \approx \big(f(a+h) - f(a)\big)/h$ is just an approximation to the derivative of f at a. Whenever we use approximations it is important to try to understand how the error behaves; in this case how the error depends on the function f and the number h. For this it is instructive to consider a simple example.

Example 11.3. Let us test the approximation (11.3) for the function $f(x) = \sin x$ at $a = 0.5$ (using 64-bit floating-point numbers). In this case we know that the exact derivative is $f'(x) = \cos x$ so $f'(a) \approx 0.8775825619$ with 10 correct digits. This makes it is easy to check the accuracy of the numerical method. We try with a few values of h and find

h	$\big(f(a+h) - f(a)\big)/h$	$E(f;a,h)$
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 $E(f;a,h) = f'(a) - \big(f(a+h) - f(a)\big)/h$. We observe that the approximation improves with decreasing h, as expected. More precisely, when h is reduced by a factor of 10, the error is reduced by the same factor.

We can analyse the error in numerical differentiation by making use of Taylor polynomials with remainders. We start by doing a linear Taylor expansion of $f(a+h)$ about $x = a$ which results in the relation

$$f(a+h) = f(a) + hf'(a) + \frac{h^2}{2} f''(\xi_h), \tag{11.4}$$

where ξ_h lies in the interval $(a, a+h)$. This formula may be rearranged to give an expression for the error,

$$f'(a) - \frac{f(a+h) - f(a)}{h} = -\frac{h}{2} f''(\xi_h). \tag{11.5}$$

This is often referred to as the *truncation error* of the approximation.

Example 11.4. Let us check that the error formula (11.5) agrees with the numerical values in example 11.3. We have $f''(x) = -\sin x$, so the right-hand side in (11.5) becomes

$$E(\sin; 0.5, h) = \frac{h}{2} \sin \xi_h,$$

where $\xi_h \in (0.5, 0.5+h)$. We do not know the exact value of ξ_h, but for the values of h in question, we know that $\sin x$ is monotone on this interval. 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 we see that the right end point of the interval is the maximum value of the right-hand side in (11.5).

When h is reduced by a factor of 10, the number $h/2$ is reduced by the same factor, while ξ_h is restricted to an interval whose width is also reduced by a factor of 10. As h becomes even smaller, the number 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

$$\frac{10^{-n}}{2} \sin 0.5 \approx \frac{0.2397}{10^n},$$

which is in close agreement with the numbers computed in example 11.3.

The observation at the end of example 11.4 is true in general: If f'' is continuous, then ξ_h will approach a when h goes to zero. But even for small, positive

values of h, the error in using the approximation $f''(\xi_h) \approx f''(a)$ is usually acceptable. This is the case since we are almost always only interested in knowing the approximate 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 when using Newton's quotient to approximate $f'(a)$ is given approximately by*

$$\left| f'(a) - \frac{f(a+h)-f(a)}{h} \right| \approx \frac{h}{2} \left| f''(a) \right|. \tag{11.6}$$

11.1.3 An upper bound on the truncation error

For practical purposes, the approximation (11.6) is usually sufficient. But let us also take the time to present a more precise argument. We will use a technique from chapter 9 and derive an upper bound on the truncation error.

We go back to (11.5) and start by taking absolute values,

$$\left| f'(a) - \frac{f(a+h)-f(a)}{h} \right| = \frac{h}{2} \left| f''(\xi_h) \right|.$$

We know that ξ_h is a number in the interval $(a, a+h)$, so it is natural to replace $|f''(\xi_h)|$ by its maximum in this interval. Here we must be a bit careful since this maximum does not always exist. But recall from the Extreme value theorem that if a function is continuous, then it always attains its maximum on any closed and bounded interval. It is therefore natural to include the end points of the interval $(a, a+h)$ and take the maximum over $[a, a+h]$. This leads to the following proposition.

Proposition 11.6. *Suppose 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*

$$E(f;a,h) = \left| f'(a) - \frac{f(a+h)-f(a)}{h} \right| \le \frac{h}{2} \max_{x\in[a,a+h]} \left| f''(x) \right|. \tag{11.7}$$

Exercises for Section 11.1

1. Write a program that computes the Newton difference quotient for the function $f(x) = e^x$ at $a = 1$ using $h = 10^{-k}$ for $k = 1, 2, \ldots, 14$. Also compute the error. Does the approximation and its error behave as predicted by Proposition 11.6?

11.2 The total error in Newton's difference quotient

In the previous section we only 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 using small values of h we have to perform the one critical operation $f(a+h) - f(a)$, i.e., subtraction of two almost equal numbers, which we know from chapter 5 may lead to large round-off errors. Let us continue the calculations in example 11.3 and see what happens if we use smaller values of h.

Example 11.7. 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 for 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}

This shows very clearly that something quite dramatic happens. Ultimately, when we come to $h = 10^{-17}$, the derivative is computed as zero.

11.2.1 Round-off errors in the function values

Let us see if we can explain what happened in example 11.7. We will go through the explanation for a general function, but keep the concrete example in mind.

The function value $f(a)$ will usually not be representable exactly in the computer and will therefore be replaced by the nearest floating-point number which we denote $\overline{f(a)}$. We then know from lemma 5.21 that the relative error in this approximation will be bounded by 5×2^{-53} since floating-point numbers are represented in binary ($\beta = 2$) with 53 bits for the significand ($m = 53$). In other words,

if we set

$$\epsilon_1 = \frac{\overline{f(a)} - f(a)}{f(a)}, \tag{11.8}$$

we have

$$|\epsilon_1| \le 5 \times 2^{-53} \approx 6 \times 10^{-16}. \tag{11.9}$$

This means that $|\epsilon_1|$ is the relative error, while ϵ_1 itself is the signed relative error.

Note that ϵ_1 will depend both on a and f, and in practice, there will usually be better upper bounds on ϵ_1 than the one in (11.9). In the following we will denote the least upper bound by ϵ^*.

Notation 11.8. *The maximum relative error that occurs when real numbers are represented by floating-point numbers, and there is no underflow or overflow, is denoted by ϵ^*.*

We will see later in this chapter that a reasonable estimate for ϵ^* is $\epsilon^* \approx 7 \times 10^{-17}$. We note that equation (11.8) may be rewritten in a form that will be more convenient for us.

Observation 11.9. *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*

$$\overline{f(a)} = f(a)(1+\epsilon_1) \tag{11.10}$$

where $|\epsilon_1| \le \epsilon^*$, and ϵ_1 depends on both a and f.

The computation of $f(a+h)$ is of course also affected by round-off error, so in total we have

$$\overline{f(a)} = f(a)(1+\epsilon_1), \quad \overline{f(a+h)} = f(a+h)(1+\epsilon_2), \tag{11.11}$$

where $|\epsilon_i| \le \epsilon^*$ 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 an apparently random way.

11.2.2 Round-off errors in the computed derivative

The next step is to see how these errors affect the computed approximation of $f'(a)$. Recall from example 5.12 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

$$\frac{\overline{f(a+h)}-\overline{f(a)}}{h},$$

and ignore the error in the division by h. If we insert the expressions (11.11), and also make use of equation (11.5), we obtain

$$\begin{aligned} f'(a)-\frac{\overline{f(a+h)}-\overline{f(a)}}{h} &= f'(a)-\frac{f(a+h)-f(a)}{h}-\frac{f(a+h)\epsilon_2-f(a)\epsilon_1}{h} \\ &= -\frac{h}{2}f''(\xi_h)-\frac{f(a+h)\epsilon_2-f(a)\epsilon_1}{h}, \end{aligned} \tag{11.12}$$

where $\xi_h \in (a, a+h)$. This shows that the total error in the computed approximation to the derivative consists of two parts: The truncation error that we derived in the previous section, plus the last term on the right in (11.12), which is due to the round-off when real numbers are replaced by floating-point numbers. The truncation error is proportional to h and therefore tends to 0 when h tends to 0. The error due to round-off however, is proportional to $1/h$ and therefore becomes large when h tends to 0.

In observation 11.5 we obtained an approximate expression for the truncation error, for small values of h, by replacing ξ_h by a. When h is small we may also assume that $f(a+h) \approx f(a)$ so (11.12) leads to the approximate error estimate

$$f'(a)-\frac{\overline{f(a+h)}-\overline{f(a)}}{h} \approx -\frac{h}{2}f''(a)-\frac{\epsilon_2-\epsilon_1}{h}f(a). \tag{11.13}$$

The most uncertain term in (11.13) is the difference $\epsilon_2-\epsilon_1$. Since we do not even know the signs of the two numbers ϵ_1 and ϵ_2, we cannot estimate this difference accurately. But we do know that both numbers represent relative errors in floating-point numbers, so the magnitude of each is about 10^{-17}. If they are of opposite signs, this magnitude may be doubled, so we replace the difference $\epsilon_2-\epsilon_1$ by $2\tilde{\epsilon}(h)$ to emphasise the dependence on h. The error (11.13) then becomes

$$f'(a)-\frac{\overline{f(a+h)}-\overline{f(a)}}{h} \approx -\frac{h}{2}f''(a)-\frac{2\tilde{\epsilon}(h)}{h}f(a). \tag{11.14}$$

Let us check if this agrees with the computations in examples 11.3 and 11.7.

Example 11.10. For large values of h the first term on the right in (11.14) 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 11.7 can be modelled when h becomes smaller.

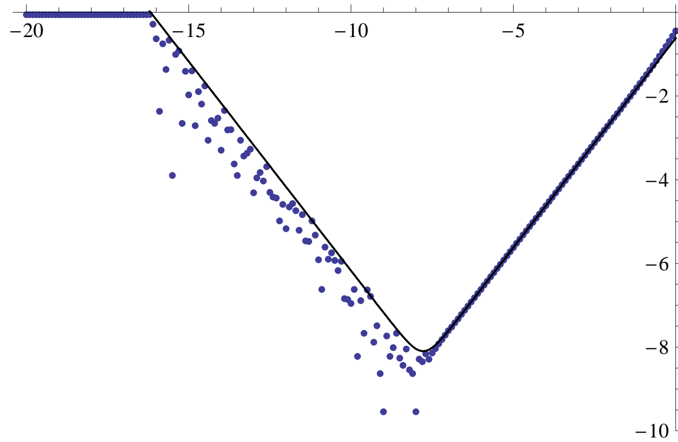


Figure 11.2. Numerical approximation of the derivative of $f(x) = \sin x$ at $x = 0.5$ using Newton's quotient, see proposition 11.6. The plot is a $\log_{10}$-$\log_{10}$ plot which shows the logarithm to base 10 of the absolute value of the total error as a function of the logarithm to base 10 of h, 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 solid line is a plot of the error estimate $g(h)$ given by (11.15).

To investigate this, we denote the left-hand side of (11.14) by $E(f;a,h)$ and solve for $\tilde{\epsilon}(h)$,

$$\tilde{\epsilon}(h) \approx -\frac{h}{2f(a)}\Big(E(f;a,h) + \frac{h}{2}f''(a)\Big).$$

From example 11.7 we have corresponding values of h and $E(f;a,h)$ which allow us to estimate $\tilde{\epsilon}(h)$ (recall that $f(x) = \sin x$ and $a = 0.5$ in this example). If we do this we can augment the table on page 275 with an additional column

h	$\big(f(a+h)-f(a)\big)/h$	$E(f;a,h)$	$\tilde{\epsilon}(h)$
10^{-7}	0. 8775825372	2.5×10^{-8}	-7.6×10^{-17}
10^{-8}	0.8775825622	-2.9×10^{-10}	2.8×10^{-17}
10^{-9}	0.8775825622	-2.9×10^{-10}	5.5×10^{-19}
10^{-11}	0.8775813409	1.2×10^{-6}	-1.3×10^{-17}
10^{-14}	0.8770761895	5.1×10^{-4}	-5.3×10^{-18}
10^{-15}	0.8881784197	-1.1×10^{-2}	1.1×10^{-17}
10^{-16}	1.110223025	-2.3×10^{-1}	2.4×10^{-17}
10^{-17}	0.000000000	8.8×10^{-1}	-9.2×10^{-18}

We observe that all these values are considerably smaller than the upper limit 6×10^{-16} in (11.9). (Note that in order to compute $\tilde{\epsilon}(h)$ correctly for $h = 10^{-7}$, you need to use the more accurate value 2.4695×10^{-8} for the error in this case.)

Figure 11.2 shows plots of the error. The numerical approximation has been computed for the values $h = 10^{-z}$, for $z = 0, \ldots, 20$ in steps of 1/10, and the

absolute value of the total error plotted in a log-log plot. The errors are shown as isolated dots, and the function

$$g(h) = \frac{h}{2}\sin 0.5 + \epsilon \frac{2}{h}\sin 0.5 \tag{11.15}$$

with $\epsilon = 7 \times 10^{-17}$ is shown as a solid graph. This corresponds to adding the absolute value of the truncation error and the round-off error, even in the case where they have opposite signs. It appears that the choice of ϵ makes $g(h)$ a reasonable upper bound on the error so we may consider this to be a decent estimate of ϵ^*.

The estimates (11.13) and (11.14) give the approximate error with sign. In general, it is more convenient to consider the absolute value of the error. Starting with (11.13), we then have

$$\begin{aligned}
\left| f'(a) - \frac{\overline{f(a+h)} - \overline{f(a)}}{h} \right| &\approx \left| -\frac{h}{2} f''(a) - \frac{\epsilon_2 - \epsilon_1}{h} f(a) \right| \\
&\leq \frac{h}{2}|f''(a)| + \frac{|\epsilon_2 - \epsilon_1|}{h}|f(a)| \\
&\leq \frac{h}{2}|f''(a)| + \frac{|\epsilon_2| + |\epsilon_1|}{h}|f(a)| \\
&\leq \frac{h}{2}|f''(a)| + \frac{2\epsilon(h)}{h}|f(a)|
\end{aligned}$$

where we used the triangle inequality in the first and second inequality, and $\epsilon(h)$ is the largest of the two numbers $|\epsilon_1|$ and $|\epsilon_2|$.

Observation 11.11. *Suppose that f and its first two derivatives are continuous near a. When the derivative of f at a is approximated by Newton's difference quotient* (11.3)*, the error in the computed approximation is roughly bounded by*

$$\left| f'(a) - \frac{\overline{f(a+h)} - \overline{f(a)}}{h} \right| \lesssim \frac{h}{2}\left|f''(a)\right| + \frac{2\epsilon(h)}{h}\left|f(a)\right|, \tag{11.16}$$

where $\epsilon(h)$ is the largest of the relative errors in $\overline{f(a)}$ and $\overline{f(a+h)}$, and the notation $\alpha \lesssim \beta$ indicates that α is approximately smaller than β.

11.2.3 An upper bound on the total error

The $\lesssim$ notation is vague mathematically, so we include a more precise error estimate.

Theorem 11.12. *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 bounded by

$$\left|f'(a)-\frac{\overline{f(a+h)}-\overline{f(a)}}{h}\right|\le\frac{h}{2}M_1+\frac{2\epsilon^*}{h}M_2,\tag{11.17}$$

where

$$M_1=\max_{x\in[a,a+h]}\left|f''(x)\right|,\quad M_2=\max_{x\in[a,a+h]}\left|f(x)\right|.$$

Proof. To get to (11.17) we start with (11.12), take absolute values, and use the triangle inequality a number of times. We also replace $\left|f''(\xi_h)\right|$ by its maximum on the interval $[a,a+h]$, and we replace $f(a)$ and $f(a+h)$ by their common maximum on $[a,a+h]$. The details are:

$$\begin{aligned}\left|f'(a)-\frac{\overline{f(a+h)}-\overline{f(a)}}{h}\right|&=\left|\frac{h}{2}f''(\xi_h)-\frac{f(a+h)\epsilon_2-f(a)\epsilon_1}{h}\right|\\&\le\frac{h}{2}|f''(\xi_h)|+\frac{|f(a+h)\epsilon_2-f(a)\epsilon_1|}{h}\\&\le\frac{h}{2}|f''(\xi_h)|+\frac{|f(a+h)||\epsilon_2|+|f(a)||\epsilon_1|}{h}\\&\le\frac{h}{2}M_1+\frac{M_2|\epsilon_2|+M_2|\epsilon_1|}{h}\\&=\frac{h}{2}M_1+\frac{|\epsilon_2|+|\epsilon_1|}{h}M_2\\&\le\frac{h}{2}M_1+\frac{2\epsilon^*}{h}M_2.\quad\blacksquare\end{aligned}\tag{11.18}$$

11.2.4 Optimal choice of h

Figure 11.2 indicates that there is an optimal value of h which minimises the total error. We can find a decent estimate for this h by minimising the upper bound in one of the error estimates (11.16) or (11.17). In practice it is easiest to use (11.16) since the two numbers M_1 and M_2 in (11.17) depend on h (although we could insert some upper bound which is independent of h).

The right-hand side of (11.16) contains the term $\epsilon(h)$ whose exact dependence on h is very uncertain. We therefore replace $\epsilon(h)$ by the upper bound ϵ^*.

This gives us the error estimate

$$e(h) = \frac{h}{2}\left|f''(a)\right| + \frac{2\epsilon^*}{h}\left|f(a)\right|. \tag{11.19}$$

To find the value of h which minimises this expression, we differentiate with respect to h and set the derivative to zero. We find

$$e'(h) = \frac{\left|f''(a)\right|}{2} - \frac{2\epsilon^*}{h^2}\left|f(a)\right|.$$

If we solve the equation $e'(h) = 0$, we obtain the approximate optimal value.

Lemma 11.13. *Let f be a function with continuous derivatives up to order 2. If the derivative of f at a is approximated as in proposition 11.6, then the value of h which minimises the total error (truncation error + round-off error) is approximately*

$$h^* \approx 2\frac{\sqrt{\epsilon^*\left|f(a)\right|}}{\sqrt{\left|f''(a)\right|}}.$$

It is easy to see that the optimal value of h is the value that balances the two terms in (11.19), i.e., the truncation error and the round-off error are equal.

Example 11.14. Based on example 11.7, we saw above that a good value of ϵ^* is 7×10^{-17}. Let us check what the optimal value of h is in this case. We have $f(x) = \sin x$ and $a = 0.5$ so

$$h^* = 2\sqrt{\epsilon} = 2\sqrt{7\times 10^{-17}} \approx 1.7\times 10^{-8}.$$

For this value of h we find

$$\frac{\sin(0.5+h^*) - \sin 0.5}{h^*} = 0.877582555644682,$$

and the error in this case is about 6.2×10^{-9}. It turns out that roughly all h in the interval $[3.2\times 10^{-9}, 2\times 10^{-8}]$ give an error of about the same magnitude which shows that the determination of h^* is quite robust.

Exercises for Section 11.2

1. Mark each of the following statements as true or false.

(a). When we use the approximation $f'(a) \approx (f(a+h) - f(a))/h$ on a computer, we can always obtain higher accuracy by choosing a smaller value for h.

(b). If we increase the number of bits for storing floating-point numbers (e.g. 128-bit precision), we can obtain better numerical approximations to derivatives.

(c). We are using Newton's difference quotient method to approximate the derivative of the function $f(x) = e^x$ at the point $x = 1$ with a step value of $h = 0.1$ (with 64-bit precission). If we change the step length to $h = 0.01$ then the error will be reduced by approximately a factor of 10.

(d). The approximation $f'(a) \approx (f(a+h)-f(a))/h$ will give the exact answer (ignoring numerical round-off errors) if the function f is linear.

(e). Since we cannot know exactly how well the values of $f(a+h)$ and $f(a)$ are represented on a computer, it is difficult to estimate accurately what the error will be in numerical differentiation.

2. Here we will again consider the Newton difference quotient approximation to the derivative $f'(a)$, i.e.

$$f'(a) \approx \frac{f(a+h) - f(a)}{h}.$$

Find the correct alternative in the following multiple choice exercises.

(a). (Exam 2010) Assume that $f(x) = \cos(x)$. The absolute error for any $h > 0$ is bounded by (we do not take round off errors into account)

☐ $h^2/2$
☐ $h^2 \cos(1)$
☐ $h \cos(a)/4$
☐ $h/2$

(b). (Exam 2008) If we are using floating point numbers the total error is bounded by (in the two last alternatives ϵ^* depends on the type of floating point numbers used):

☐ $\frac{h^2}{2}\max_{x\in[a,a+h]}\left|f''(x)\right|$
☐ $\frac{h^3}{6}\max_{x\in[a,a+h]}\left|f'''(x)\right|$
☐ $\frac{h^2}{6}\max_{x\in[a,a+h]}\left|f'''(x)\right|+\frac{6\epsilon^*}{h^3}\max_{x\in[a,a+h]}\left|f(x)\right|$
☐ $\frac{h}{2}\max_{x\in[a,a+h]}\left|f''(x)\right|+\frac{2\epsilon^*}{h}\max_{x\in[a,a+h]}\left|f(x)\right|$

3. In this exercise we are going to numerically compute the derivative of $f(x) = e^x$ at $a = 1$ using Newton's quotient as described in observation 11.2. The exact derivative to 20 digits is

$$f'(1) \approx 2.7182818284590452354.$$

(a). Compute the approximation $\big(f(1+h)-f(1)\big)/h$ to $f'(1)$. Start with $h = 10^{-4}$, and then gradually reduce h. Also compute the error, and determine an h that gives close to minimal error.

(b). Determine the optimal h as described in Lemma 11.13 and compare with the value you found in (a).

4. When deriving the truncation error given by (11.7) it is not obvious what the degree of the Taylor polynomial in (11.4) should be. In this exercise you are going to try and increase and reduce the degree of the Taylor polynomial and see what happens.

(a). Redo the Taylor expansion in (11.4), but use the Taylor polynomial of degree 2. From this try and derive an error formula similar to (11.5).

(b). Repeat (a), but use a Taylor polynomial of degree 0, i.e., just a constant.

(c). Why can you conclude that the linear Taylor polynomial and the error term in (11.5) is the best?

11.3 A general strategy for constructing differentiation methods

Before we continue, let us sum up the derivation and analysis of the Newton's difference quotient in section 11.1, since this is standard for all differentiation methods.

The first step is 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 derived 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 in choosing the correct degree of the Taylor polynomial, see exercise 11.1.4.

The next task is to estimate the total error, including the 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 11.9. 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 estimates can then be simplified to something like (11.16) or (11.17). As a final step, the optimal value of h can be found by minimising the total error.

Procedure 11.15. *The following is a general procedure for deriving numerical methods for differentiation:*

1. *Interpolate the function f by a polynomial p at suitable points.*
2. *Approximate the derivative of f by the derivative 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.*
4. *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.*

Exercises for Section 11.3

1. Determine an approximation to the derivative $f'(a)$ using the function values $f(a)$, $f(a+h)$ and $f(a+2h)$ by interpolating f by a quadratic polynomial p_2

at the three points a, $a+h$, and $a+2h$, and then using $f'(a) \approx p_2'(a)$.

11.4 Three differentiation methods

In this section we use the procedure in section 11.3 to derive three different differentiation methods. All the methods can be analysed in exactly the same way as with Newton's difference quotient, but here we just give the final results.

11.4.1 A symmetric version of Newton's quotient

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 start by writing p_2 in Newton form,

$$p_2(x) = c_0 + c_1(x-(a-h)) + c_2(x-(a-h))(x-a). \tag{11.20}$$

We differentiate and find

$$p_2'(x) = c_1 + c_2(2x-2a+h).$$

Setting $x=a$ yields

$$p_2'(a) = c_1 + c_2 h. \tag{11.21}$$

If we do the algebra we find

$$c_1 = \frac{f(a)-f(a-h)}{h}, \quad c_2 = \frac{f(a+h)-2f(a)+f(a-h)}{2h^2}.$$

When this inserted into (11.21) we end up with

$$p_2'(a) = \frac{f(a+h)-f(a-h)}{2h}.$$

We note that the approximation to the derivative given by $p_2'(a)$ agrees with the slope of the secant based at $a-h$ and $a+h$.

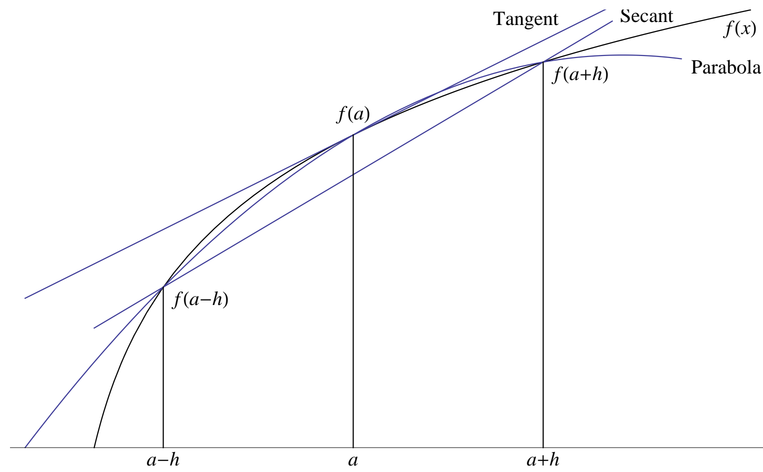


Figure 11.3. The secant of a function based at $a-h$ and $a+h$, as well as the tangent at a.

Lemma 11.16 (Symmetric Newton's quotient). *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*

$$f'(a) \approx p_2'(a) = \frac{f(a+h) - f(a-h)}{2h}, \tag{11.22}$$

and agrees with the slope of the secant based at $a-h$ and $a+h$. The total error is bounded by

$$\left| f'(a) - \frac{\overline{f(a+h)} - \overline{f(a-h)}}{2h} \right| \le \frac{h^2}{6} M_1 + \frac{\epsilon^*}{h} M_2, \tag{11.23}$$

where

$$M_1 = \max_{x \in [a-h, a+h]} \left| f'''(x) \right|, \quad M_2 = \max_{x \in [a-h, a+h]} \left| f(x) \right|, \tag{11.24}$$

and the optimal value of h is

$$h^* = \frac{\sqrt[3]{3\epsilon^* M_2}}{\sqrt[3]{M_1}} \approx \frac{\sqrt[3]{3\epsilon^* \left| f(a) \right|}}{\sqrt[3]{\left| f'''(a) \right|}}. \tag{11.25}$$

The symmetric Newton's quotient is illustrated in figure 11.3. The derivative of f at a is given by the slope of the tangent, while the approximation defined by $p_2'(a)$ is given by the slope of tangent of the parabola at a (which is the same as the slope of the secant of f based at $a-h$ and $a+h$). Let us test this approximation on the function $f(x) = \sin x$ at $a = 0.5$ so we can compare with the original Newton's quotient that we discussed in section 11.1.

Example 11.17. We test the approximation (11.22) with the same values of h as in examples 11.3 and 11.7. Recall that $f'(0.5) \approx 0.8775825619$ with 10 correct digits. The results are

h	$\bigl(f(a+h)-f(a-h)\bigr)/(2h)$	$E(f;a,h)$
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 11.7, the errors are generally smaller for the same value of h. 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 starts to increase. It therefore seems like the truncation error is smaller than for the original method based on Newton's quotient, but as before, 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 asymmetric Newton's quotient.

A plot of how the error behaves in the symmetric Newton's quotient, together with the estimate of the error on the right is shown in figure 11.4.

At the end of section 11.2.4 we saw that a reasonable value for ϵ^* was $\epsilon^* = 7 \times 10^{-17}$. The optimal value of h in example 11.17, where $f(x) = \sin x$ and $a = 0.5$, 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}.

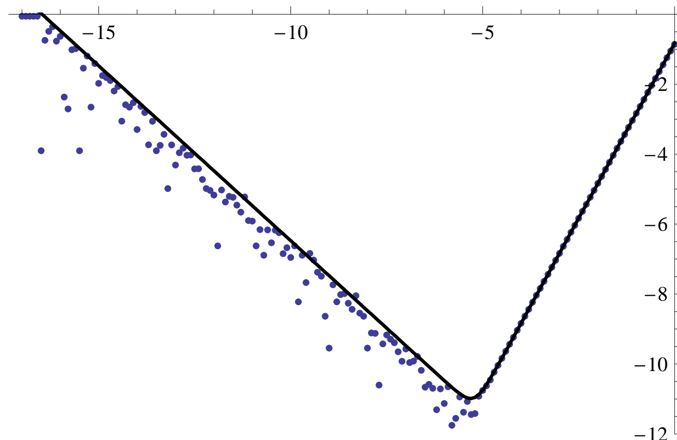


Figure 11.4. Log-log plot of the error in the approximation to the derivative of $f(x) = \sin x$ at $x = 1/2$ for values of h in the interval $[0, 10^{-17}]$, using the symmetric Newton's quotient in theorem **??**. The solid graph represents the right-hand side of (**??**) with $\epsilon^* = 7 \times 10^{-17}$, as a function of h.

11.4.2 A four-point differentiation method

The asymmetric and symmetric Newton's quotients are the two most commonly used methods for approximating derivatives. Whenever possible, one would prefer the symmetric version whose truncation error is proportional to h^2. This means that the error goes to 0 more quickly than for the asymmetric version, as was clearly evident in examples 11.3 and 11.17. In this section we derive another method for which the truncation error is proportional to h^4. This also illustrates the procedure 11.15 in a more complicated situation.

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 simpler Newton's quotient.

We want better accuracy than the symmetric Newton's quotient 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

$$p_3(x) = c_0 + c_1(x - x_0) + c_2(x - x_0)(x - x_1) + c_3(x - x_0)(x - x_1)(x - x_2),$$

and its derivative is

$$p_3'(x) = c_1 + c_2(2x - x_0 - x_1) \\ + c_3\big((x-x_1)(x-x_2) + (x-x_0)(x-x_2) + (x-x_0)(x-x_1)\big).$$

If we evaluate this expression at $x = a$ and simplify (this is quite a bit of work), we find that the resulting approximation to $f'(a)$ is

$$f'(a) \approx p_3'(a) = \frac{f(a-2h) - 8f(a-h) + 8f(a+h) - f(a+2h)}{12h}. \tag{11.26}$$

We can estimate the truncation error and the round-off error, just like we did for the Newton quotient, but everything becomes a bit more involved. The final result is summarised in the following observation (we could of course also derive a more formal upper bound on the error, similar to (11.17) and (11.23).)

Observation 11.18. *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

$$\left| f'(a) - \frac{\overline{f(a-2h)} - 8\overline{f(a-h)} + 8\overline{f(a+h)} - \overline{f(a+2h)}}{12h} \right| \lesssim \\ \frac{h^4}{18}\left|f^{(v)}(a)\right| + \frac{3\epsilon^*}{h}\left|f(a)\right|. \tag{11.27}$$

The optimal value of h is given by

$$h^* = \frac{\sqrt[5]{27\epsilon^*\left|f(a)\right|}}{\sqrt[5]{2\left|f^{(v)}(a)\right|}}. \tag{11.28}$$

A plot of the error in the approximation for the $\sin x$ example that we used for the previous approximations is shown in figure 11.5. For this example ($f(x) = \sin x$ and $a = 0.5$) the optimal value of h is $h^* \approx 8.8\times10^{-4}$. The actual error is then roughly 10^{-14}.

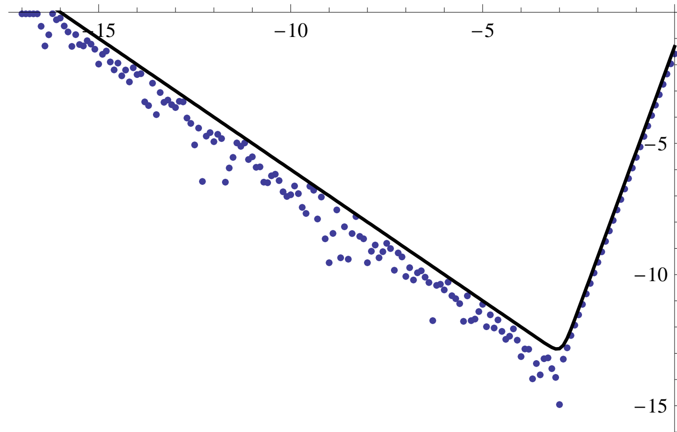


Figure 11.5. Log-log plot of the error in the approximation to the derivative of $f(x) = \sin x$ at $x = 1/2$, using the method in observation 11.18, with h in the interval $[0, 10^{-17}]$. The function plotted is the right-hand side of (11.27) with $\epsilon^* = 7 \times 10^{-17}$.

11.4.3 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.

11.4.4 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 11.4 and it is given in equation (11.20). The second derivative of p_2 is constant, and the approximation of $f''(a)$ is

$$f''(a) \approx p_2''(a) = \frac{f(a+h) - 2f(a) + f(a-h)}{h^2}.$$

Once again the error can be derived in the same way as we did for Newton's quotient. Everything is summed up in the following theorem.

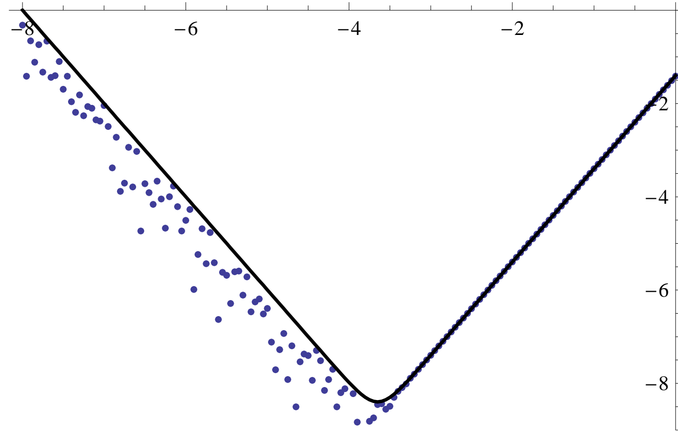


Figure 11.6. Log-log plot of the error in the approximation to the derivative of $f(x) = \sin x$ at $x = 1/2$ for h in the interval $[0, 10^{-8}]$, using the method in theorem 11.19. The function plotted is the right-hand side of (11.29) with $\epsilon^* = 7 \times 10^{-17}$.

Theorem 11.19. *Suppose f and its first three derivatives are continuous near a, and that $f''(a)$ is approximated by*

$$f''(a) \approx \frac{f(a+h) - 2f(a) + f(a-h)}{h^2}.$$

Then the total error (truncation error + round-off error) in the computed approximation is bounded by

$$\left| f''(a) - \frac{\overline{f(a+h)} - 2\overline{f(a)} + \overline{f(a-h)}}{h^2} \right| \le \frac{h^2}{12} M_1 + \frac{3\epsilon^*}{h^2} M_2, \tag{11.29}$$

where

$$M_1 = \max_{x \in [a-h, a+h]} \left| f^{(iv)}(x) \right|, \quad M_2 = \max_{x \in [a-h, a+h]} \left| f(x) \right|.$$

The optimal value of h is given by

$$h^* = \frac{\sqrt[4]{36\epsilon^* \left| f(a) \right|}}{\sqrt[4]{\left| f^{(iv)}(a) \right|}}. \tag{11.30}$$

Figure 11.6 shows the errors in the approximation to the second derivative given in theorem 11.19 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 (11.29) which describes an approximate upper bound on the error as a 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.

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}.

Exercises for Section 11.4

1. Mark each of the following statements as true or false.

(a). If we ignore round-off errors, the symmetric Newton's quotient is exact for polynomials of degree 2 or lower.

(b). Even though the symmetric Newton differentiation scheme provides better accuracy, there is a trade-off since it is much more computationally demanding (i.e. it requires many more calculations) than the non-symmetric method.

2. In this exercise we are going to check the symmetric Newton's quotient and numerically compute the derivative of $f(x) = e^x$ at $a = 1$, see exercise 11.2.3. Recall that the exact derivative with 20 correct digits is

$$f'(1) \approx 2.7182818284590452354.$$

(a). Compute the approximation $\big(f(1+h) - f(1-h)\big)/(2h)$ to $f'(1)$. Start with $h = 10^{-3}$, and then gradually reduce h. Also compute the error, and determine an h that gives close to minimal error.

(b). Determine the optimal h given by (11.25) and compare it with the value you found in (a).

3. Determine $f'(a)$ numerically using the two asymmetric Newton's quotients

$$f_r(a) = \frac{f(a+h) - f(a)}{h}, \qquad f_l(a) = \frac{f(a) - f(a-h)}{h}$$

as well as the symmetric Newton's quotient. Also compute and compare the relative errors in each case.

(a). $f(x) = x^2$; $a = 2$; $h = 0.01$.

(b). $f(x) = \sin x$; $a = \pi/3$; $h = 0.1$.

(c). $f(x) = \sin x$; $a = \pi/3$; $h = 0.001$.

(d). $f(x) = \sin x$; $a = \pi/3$; $h = 0.00001$.

(e). $f(x) = 2^x$; $a = 1$; $h = 0.0001$.

(f). $f(x) = x\cos x$; $a = \pi/3$; $h = 0.0001$.

4. In this exercise we are going to relate the symmetric Newton quotient to the two asymmetric Newton quotients.

(a). Show that the approximation to $f'(a)$ given by the symmetric Newton's quotient is the average of the two asymmetric quotients

$$f_r(a) = \frac{f(a+h) - f(a)}{h}, \qquad f_l(a) = \frac{f(a) - f(a-h)}{h}.$$

(b). Sketch the graph of the function

$$f(x) = \frac{-x^2 + 10x - 5}{4}$$

on the interval $[0,6]$ together with the three secants associated with the three approximations to the derivative in (a) (use $a = 3$ and $h = 2$). Can you from this judge which approximation is best?

(c). Determine the three difference quotients in (a) numerically for the function $f(x)$ using $a = 3$ and $h_1 = 0.1$ and $h_2 = 0.001$. What are the relative errors?

(d). Show that the symmetric Newton's quotient at $x = a$ for a quadratic function $f(x) = ax^2 + bx + c$ is equal to the derivative $f'(a)$.

5. Use the symmetric Newton's quotient and determine an approximation to the derivative $f'(a)$ in each case below. Use the values of h given by $h = 10^{-k}$ $k = 4, 5, \ldots, 12$ and compare the relative errors. Which of these values of h gives the smallest error? Compare with the optimal h predicted by (11.25).

(a). The function $f(x) = 1/(1 + \cos(x^2))$ at the point $a = \pi/4$.

(b). The function $f(x) = x^3 + x + 1$ at the point $a = 0$.

6. Find the correct alternative in the following multiple choice exercises.

(a). (Exam 2009) We use the expression $(f(h) - 2f(0) + f(h))/h^2$ to calculate approximations to $f''(0)$ (we do the calculations exact, without round off errors). Then the result will always be correct if $f(x)$ is

☐ a trigonometric function

☐ a logarithmic function

☐ a polynomial of degree 4

☐ a polynomial of degree 3

(b). (Exam 2007) We approximate the second derivative of the function $f(x)$ at $x = 0$, by the approximation

$$D_2 f(0) = \frac{f(h) - 2f(0) + f(-h)}{h^2}$$

We assume that f is differentiable an infinite number of times, and we do not take round off errors into account. Then the error

$$\left| f''(0) - D_2 f(0) \right|$$

is bounded by

☐ $\frac{h^2}{12} \max_{x\in[-h,h]} \left| f''(x) \right|$

☐ $\frac{h^2}{48} \max_{x\in[-h,h]} \left| f^{(4)}(x) \right|$

☐ $\frac{h}{4} \max_{x\in[-h,h]} \left| f''(x) \right|$

☐ $\frac{h^2}{12} \max_{x\in[-h,h]} \left| f^{(4)}(x) \right|$

7. We use our standard example $f(x) = e^x$ and $a = 1$ to check the 3-point approximation to the second derivative given in Theorem 11.19. For comparison recall that the exact second derivative to 20 digits is

$$f''(1) \approx 2.7182818284590452354.$$

(a). Compute the approximation $\big(f(a-h) - 2f(a) + f(a+h)\big)/h^2$ to $f''(1)$. Start with $h = 10^{-3}$, and then gradually reduce h. Also compute the actual error, and determine an h that gives close to minimal error.

(b). Determine the optimal value of h given by (11.30) and compare with the value you determined in (a).

8. This exercise illustrates a different approach to designing numerical differentiation methods.

(a). Suppose 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. Do you recognise the method?

(d). Show that the method in (c) is exact for all cubic polynomials.

9. Previously we saw the that the Newton difference quotient could be applied reduce bass in digital sound. What will happen to the sound if we instead apply the numerical approximation of the second derivative to the sound samples?

10. Mark each of the following statements as true or false.

(a). The 4-point method with a step length of $h = 0.2$ will usually have a smaller error than the symmetric Newton's quotient method with $h = 0.1$.

(b). If we ignore round-off, the 4-point method is exact for all polynomials.

11. In this exercise we are going to check the 4-point method and numerically compute the derivative of $f(x) = e^x$ at $a = 1$. For comparison, the exact derivative to 20 digits is

$$f'(1) \approx 2.7182818284590452354.$$

(a). Compute the approximation

$$\frac{f(a-2h) - 8f(a-h) + 8f(a+h) - f(a+2h)}{12h}$$

to $f'(1)$. Start with $h = 10^{-3}$, and then gradually reduce h. Also compute the error, and determine an h that gives close to minimal error.

(b). Determine the optimal h given by (11.28) and compare with the experimental value you found in (a).

CHAPTER 12

Numerical Integration

Numerical differentiation methods compute approximations to the derivative of a function from known values of the function. Numerical integration uses the same information to compute numerical approximations to the integral of the function. An important use of both types of methods is estimation of derivatives and integrals for functions that are only known at isolated points, as is the case with for example measurement data. An important difference between differentiation and integration is that for most functions it is not possible to determine the integral via symbolic methods, but we can still compute numerical approximations to virtually any definite integral. Numerical integration methods are therefore more useful than numerical differentiation methods, and are essential in many practical situations.

We use the same general strategy for deriving numerical integration methods as we did for numerical differentiation methods: We find the polynomial that interpolates the function at some suitable points, and use the integral of the polynomial as an approximation to the function. This means that the truncation error can be analysed in basically the same way as for numerical differentiation. However, when it comes to round-off error, integration behaves differently from differentiation: Numerical integration is very *insensitive* to round-off errors, so we will ignore round-off in our analysis.

The mathematical definition of the integral is basically via a numerical integration method, and we therefore start by reviewing this definition. We then derive the simplest numerical integration method, and see how its error can be analysed. We then derive two other methods that are more accurate, but for these we just indicate how the error analysis can be done.

We emphasise that the general procedure for deriving both numerical differ-

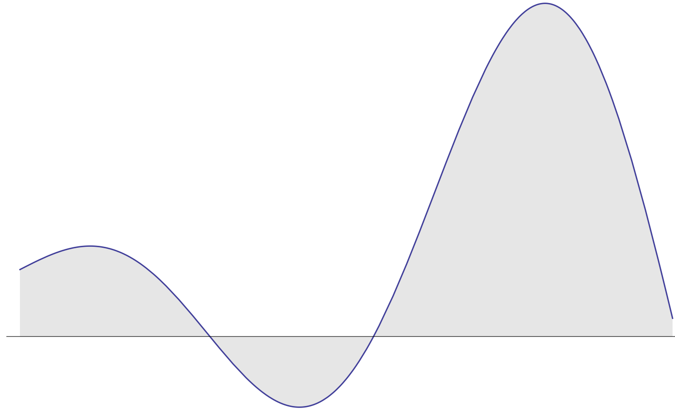


Figure 12.1. The area under the graph of a function.

entiation and integration methods with error analyses is the same with the exception that round-off errors are not of much interest for the integration methods.

12.1 What is the integral?

The integral of a function f over the interval $[a,b]$ is simply the area under the graph of f between a and b, with positive values of f contributing positively to the area, and negative values contributing negatively, see figure 12.1. The standard notation for this integral is

$$\int_a^b f(x)\,dx.$$

The integral has a close link with the derivative, but this is peripheral to the discussion in this chapter. Instead, we will consider three different methods for estimating the integral numerically. Before we continue, we need to define a term which will be used repeatedly in our description of integration.

Definition 12.1 (Partition). *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 $i = 1, \ldots, n$.*

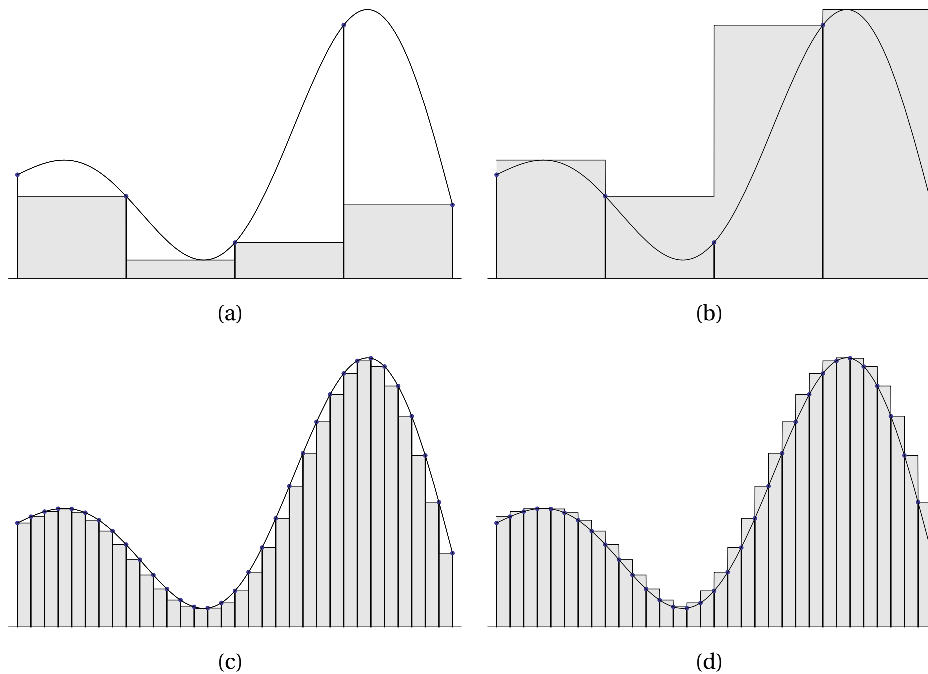


Figure 12.2. The definition of the integral via inscribed and circumsribed step functions.

12.1.1 Definition of the integral

The traditional definition of the integral is based on a numerical approximation to the area. We pick a partition $\{x_i\}_{i=0}^n$ 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 can then compute two obvious approximations to the integral by approximating f by two different functions which are both assumed to be constant on each interval $[x_{i-1},x_i]$: The first has the constant value m_i and the other the value M_i. We then sum up the areas under each of the two step functions an end up with the two approximations

$$\underline{I} = \sum_{i=1}^{n} m_i(x_i - x_{i-1}), \quad \overline{I} = \sum_{i=1}^{n} M_i(x_i - x_{i-1}), \tag{12.1}$$

to the total area. In general, the first of these is too small, the other too large.

To define the integral, we consider larger partitions (smaller step lengths) 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.

Definition 12.2 (Integral). *Let f be a function defined on the interval $[a,b]$, and let $\{x_i\}_{i=0}^{n}$ be a partition of $[a,b]$. Let m_i and M_i denote the minimum and maximum values of f over the interval $[x_{i-1},x_i]$, respectively, assuming they exist. Consider the two numbers $\underline{I}$ and $\overline{I}$ defined in* (12.1). *If* $\sup \underline{I}$ *and* $\inf \overline{I}$ *both exist and are equal, where the* sup *and* inf *are taken over all possible partitions of $[a,b]$, the function f is said to be* integrable*, and the integral of f over $[a,b]$ is defined by*

$$I = \int_a^b f(x)\,dx = \sup \underline{I} = \inf \overline{I}.$$

This process is illustrated in figure 12.2 where we see how the piecewise constant approximations become better when the rectangles become narrower.

12.1.2 Numerical computation of the integral

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 12.2, 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 property that is very useful when it comes to numerical computation.

Theorem 12.3. *Suppose that f is integrable on the interval $[a,b]$, let $\{x_i\}_{i=0}^{n}$ be a partition of $[a,b]$, and let t_i be a number in $[x_{i-1},x_i]$ for $i=1, \dots, n$. Then the sum*

$$\tilde{I} = \sum_{i=1}^{n} f(t_i)(x_i - x_{i-1}) \tag{12.2}$$

will converge to the integral when the distance between all neighbouring x_is tends to zero.

Theorem 12.3 allows us to construct practical, numerical methods for computing the integral. We pick a partition of $[a,b]$, choose t_i equal to x_{i-1} or x_i,

and compute the sum (12.2). It turns out that an even better choice is the more symmetric $t_i = (x_i + x_{i-1})/2$ which leads to the approximation

$$I \approx \sum_{i=1}^{n} f\big((x_i + x_{i-1})/2\big)(x_i - x_{i-1}). \tag{12.3}$$

This is the so-called *midpoint rule* which we will study in the next section.

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 for deriving more advanced numerical integration methods, and this works as long as f can be approximated well by polynomials on each subinterval.

Exercises for Section 12.1

1. Mark each of the following statements as true or false.

(a). Numerical integration methods are usually constructed by dividing the interval of integration into many subintervals and using some sort of approximation to the area under the function on each subinterval.

2. In this exercise we are going to study the definition of the integral for the function $f(x) = e^x$ on the interval $[0, 1]$.

(a). Determine lower and upper sums for a uniform partition consisting of 10 subintervals.

(b). Determine the absolute and relative errors of the sums in (a) compared to the exact value $e - 1 = 1.718281828$ of the integral.

(c). Write a program for calculating the lower and upper sums in this example. How many subintervals are needed to achieve an absolute error less than 3×10^{-3}?

12.2 The midpoint rule for numerical integration

We have already introduced the midpoint rule (12.3) 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.

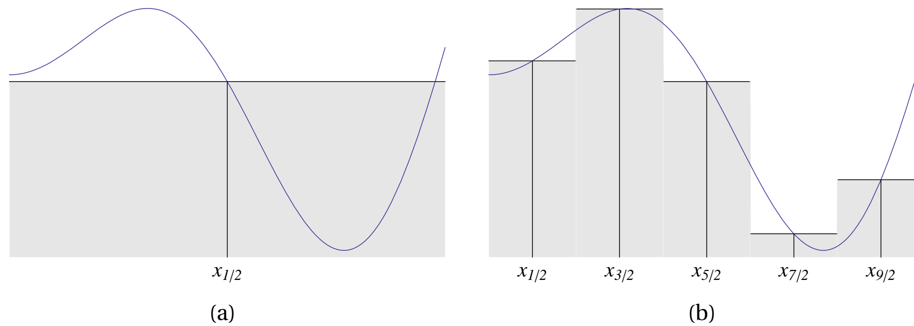


Figure 12.3. The midpoint rule with one subinterval (a) and five subintervals (b).

Algorithm 12.4. *Let f be 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*

$$\int_a^b f(x)\,dx \approx I_{mid}(h) = h\sum_{i=1}^{n} f(x_{i-1/2}), \tag{12.4}$$

where

$$x_{i-1/2} = (x_{i-1}+x_i)/2 = a + (i-1/2)h.$$

This may seem like a strangely formulated algorithm, but all there is to do is to compute the sum on the right in (12.4). The method is illustrated in figure 12.3 in the cases where we have 1 and 5 subintervals.

12.2.1 A detailed algorithm

Algorithm 12.4 describes the midpoint rule, 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. We did this when we discussed algorithms for finding roots of equations in chapter 10, and we can do exactly the same here: We compute the integral for decreasing step lengths, and stop the computations when the difference between two successive approximations is less than the tolerance. More precisely, we choose an initial step length h_0 and compute the approximations

$$I_{\text{mid}}(h_0), I_{\text{mid}}(h_1), \ldots, I_{\text{mid}}(h_k), \ldots,$$

where $h_k = h_0/2^k$. Suppose $I_{\text{mid}}(h_k)$ is our latest approximation. Then we estimate the relative error by the number

$$\frac{|I_{\text{mid}}(h_k) - I_{\text{mid}}(h_{k-1})|}{|I_{\text{mid}}(h_k)|},$$

and stop the computations if this is smaller than ϵ. To avoid potential division by zero, we use the test

$$|I_{\text{mid}}(h_k) - I_{\text{mid}}(h_{k-1})| \le \epsilon |I_{\text{mid}}(h_k)|.$$

As always, we should also limit the number of approximations that are computed, so we count the number of times we divide the subintervals, and stop when we reach a predefined limit which we call M.

Algorithm 12.5. *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 $k := 1, 2, \ldots, n$
    $I := I + f(x)$;
    $x := x + h$;
$j := 1$;
$I := h * I$;
$abserr := |I|$;
while $j < M$ **and** $abserr > \epsilon * |I|$
    $j := j + 1$;
    $Ip := I$;
    $n := 2n$; $h := (b-a)/n$;
    $I := 0$; $x := a + h/2$;
    for $k := 1, 2, \ldots, n$
        $I := I + f(x)$;
        $x := x + h$;
    $I := h * I$;
    $abserr := |I - Ip|$;

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 two times we reach the while loop (the first time we reach the while loop we will always get past the condition). We store the previous approximation in Ip and use this to estimate the error in the next iteration.

In the coming sections we will describe two other methods for numerical integration. These can be implemented in algorithms similar to Algorithm 12.5. In fact, the only difference will be how the actual approximation to the integral is computed.

Example 12.6. Let us try the midpoint rule 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	$I_{\text{mid}}(h)$	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

$$\int_0^1 f(x)\,dx - I_{\text{mid}}(h).$$

Note that each time the step length is halved, the error seems to be reduced by a factor of 4.

12.2.2 The error

Algorithm 12.5 determines a numerical approximation to the integral, and even estimates the error. However, we must remember that the error that is computed

is not always reliable, so we should try and understand the error better. We do this in two steps. First we analyse the error in the situation where we use a very simple partition with only one subinterval, the so-called *local error*. Then we use this result to obtain an estimate of the error in the general case — this is often referred to as the *global error*.

Local error analysis

Suppose we use the midpoint rule with just one subinterval. We want to study the error

$$\int_a^b f(x)\,dx - f\big(a_{1/2}\big)(b-a), \quad a_{1/2} = (a+b)/2. \tag{12.5}$$

Once again, a Taylor polynomial with remainder helps us out. We expand $f(x)$ about the midpoint $a_{1/2}$ and obtain,

$$f(x) = f(a_{1/2}) + (x-a_{1/2})f'(a_{1/2}) + \frac{(x-a_{1/2})^2}{2}f''(\xi),$$

where ξ is a number in the interval $(a_{1/2}, x)$ that depends on x. Next, we integrate the Taylor expansion and obtain

$$\begin{aligned}\int_a^b f(x)\,dx &= \int_a^b \Big(f(a_{1/2}) + (x-a_{1/2})f'(a_{1/2}) + \frac{(x-a_{1/2})^2}{2}f''(\xi)\Big)\,dx\\ &= f(a_{1/2})(b-a) + \frac{f'(a_{1/2})}{2}\big[(x-a_{1/2})^2\big]_a^b + \frac{1}{2}\int_a^b (x-a_{1/2})^2 f''(\xi)\,dx\\ &= f(a_{1/2})(b-a) + \frac{1}{2}\int_a^b (x-a_{1/2})^2 f''(\xi)\,dx,\end{aligned} \tag{12.6}$$

since the middle term is zero. This leads to an expression for the error,

$$\left|\int_a^b f(x)\,dx - f(a_{1/2})(b-a)\right| = \frac{1}{2}\left|\int_a^b (x-a_{1/2})^2 f''(\xi)\,dx\right|. \tag{12.7}$$

Let us simplify the right-hand side of this expression and explain afterwords. We have

$$
\begin{aligned}
\frac{1}{2}\left|\int_a^b (x-a_{1/2})^2 f''(\xi)\,dx\right| &\leq \frac{1}{2}\int_a^b \left|(x-a_{1/2})^2 f''(\xi)\right|\,dx \\
&= \frac{1}{2}\int_a^b (x-a_{1/2})^2 \left|f''(\xi)\right|\,dx \\
&\leq \frac{M}{2}\int_a^b (x-a_{1/2})^2\,dx \\
&= \frac{M}{2}\frac{1}{3}\left[(x-a_{1/2})^3\right]_a^b \\
&= \frac{M}{6}\big((b-a_{1/2})^3-(a-a_{1/2})^3\big) \\
&= \frac{M}{24}(b-a)^3,
\end{aligned}
\tag{12.8}
$$

where $M=\max_{x\in[a,b]}|f''(x)|$. 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 $(x-a_{1/2})^2$ is never negative. The next inequality follows because we replace $\left|f''(\xi)\right|$ with its maximum on the interval $[a,b]$. The next step is just the evaluation of the integral of $(x-a_{1/2})^2$, and the last equality follows since $(b-a_{1/2})^3=-(a-a_{1/2})^3=(b-a)^3/8$. This proves the following lemma.

Lemma 12.7. *Let f be a continuous function whose first two derivatives are continuous on the interval $[a,b]$. The error in the midpoint rule, with only one interval, is bounded by*

$$\left|\int_a^b f(x)\,dx - f\big(a_{1/2}\big)(b-a)\right| \leq \frac{M}{24}(b-a)^3,$$

where $M=\max_{x\in[a,b]}\left|f''(x)\right|$ and $a_{1/2}=(a+b)/2$.

Before we continue, let us sum up the procedure that led up to lemma 12.7 without focusing on the details: Start with the error (12.5) and replace $f(x)$ by its linear Taylor polynomial with remainder. When we integrate the Taylor polynomial, the linear term becomes zero, and we are left with (12.7). At this point we use some standard techniques that give us the final inequality.

The importance of lemma 12.7 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 rule will be reduced by a factor of 8.

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.

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 just analysed,

$$I = \int_a^b f(x)\,dx = \sum_{i=1}^{n} \int_{x_{i-1}}^{x_i} f(x)\,dx \approx \sum_{i=1}^{n} f(x_{i-1/2})h.$$

The total error is then

$$I - I_{\text{mid}} = \sum_{i=1}^{n} \left(\int_{x_{i-1}}^{x_i} f(x)\,dx - f(x_{i-1/2})h \right).$$

We note that the expression inside the parenthesis is just the local error on the interval $[x_{i-1}, x_i]$. We therefore have

$$\begin{aligned} |I - I_{\text{mid}}| &= \left| \sum_{i=1}^{n} \left(\int_{x_{i-1}}^{x_i} f(x)\,dx - f(x_{i-1/2})h \right) \right| \\ &\le \sum_{i=1}^{n} \left| \int_{x_{i-1}}^{x_i} f(x)\,dx - f(x_{i-1/2})h \right| \\ &\le \sum_{i=1}^{n} \frac{h^3}{24} M_i \end{aligned} \tag{12.9}$$

where M_i is the maximum of $\left|f''(x)\right|$ on the interval $[x_{i-1}, x_i]$. The first of these inequalities is just the triangle inequality, while the second inequality follows from lemma 12.7. To simplify the expression (12.9), we extend the maximum on $[x_{i-1}, x_i]$ to all of $[a,b]$. This cannot make the maximum smaller, so for all i we have

$$M_i = \max_{x\in[x_{i-1},x_i]} \left|f''(x)\right| \le \max_{x\in[a,b]} \left|f''(x)\right| = M.$$

Now we can simplify (12.9) further,

$$\sum_{i=1}^{n} \frac{h^3}{24} M_i \le \sum_{i=1}^{n} \frac{h^3}{24} M = \frac{h^3}{24} nM. \tag{12.10}$$

Here, we need one final little observation. Recall that $h = (b-a)/n$, so $hn = b-a$. If we insert this in (12.10), we obtain our main error estimate.

Theorem 12.8. *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

$$|I - I_{mid}| \le (b-a)\frac{h^2}{24} \max_{x\in[a,b]} \left|f''(x)\right|, \tag{12.11}$$

where $x_{i-1/2} = a + (i-1/2)h$.

This confirms the error behaviour that we saw in example 12.6: 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 error in the midpoint rule 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.

12.2.3 Estimating the step length

The error estimate (12.11) 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

$$(b-a)\frac{h^2}{24} \max_{x\in[a,b]} \left|f''(x)\right| \le \epsilon$$

which we can easily solve for h,

$$h \le \sqrt{\frac{24\epsilon}{(b-a)M}}, \qquad M = \max_{x\in[a,b]} \left|f''(x)\right|.$$

This is not quite as simple as it may look since we will have to estimate M, the maximum value of the second derivative, over the whole interval of integration $[a,b]$. This can be difficult, but in some cases it is certainly possible, see exercise 4.

Exercises for Section 12.2

1. Mark each of the following statements as true or false.

(a). When we use the midpoint rule for numerical integration, round-off errors due to subtraction of two similar numbers is a major source of errors.

(b). The midpoint rule gives the exact result for polynomials of degree 1.

(c). The midpoint rule gives the exact result for polynomials of degree 2.

(d). The global error in the midpoint method is one order lower than the local error.

(e). When we decrease the step length h in the midpoint rule by a factor of 3, the error is reduced by roughly a factor of 9.

2. We use the midpoint rule to approximate the integral

$$\int_0^1 x^2\,dx$$

using the midpoint rule with 2 subintervals. What is the result?

☐ 5/16

☐ 1/4

☐ 4/9

☐ 2/5

3. Calculate an approximation to the integral

$$\int_0^{\pi/2} \frac{\sin x}{1+x^2}\,dx = 0.526978557614\ldots$$

with the midpoint rule. Split the interval into 6 subintervals.

4. In this exercise, if you cannot program, use the midpoint algorithm with 10 subintervals, check the error, and skip (b).

(a). Test the midpoint rule with 10 subintervals 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 12.5 slightly and print the absolute error.)

5. Repeat the previous exercise, but compute the integral

$$\int_2^6 \ln x \, dx = \ln(11664) - 4.$$

6. Redo the local error analysis for the midpoint rule, but replace both $f(x)$ and $f(a_{1/2})$ by linear Taylor polynomials with remainders about the left end point a. What happens to the error estimate?

12.3 Two other numerical integration methods

There are many different numerical integration techniques, and in this section we will consider two of the simplest ones. But first of all we show how the strategy in Section 11.3 for designing numerical differentiation methods can be adapted to designing numerical integration methods.

12.3.1 A strategy for designing integration methods

Procedure 12.9. *The following is a general procedure for deriving numerical methods for computing*

$$\int_a^b f(x)\, dx.$$

1. *Split the interval $[a,b]$ into n subintervals*

$$a = x_0 < x_1 < x_2 < \cdots < x_{n-1} < x_n = b.$$

2. *Interpolate the function f over the subinterval $I_i = [x_i, x_{i+1}]$ by a polynomial p_k of degree k at $k+1$ suitable points in I_i.*

3. *Approximate the integral of f over I_i by the integral of p_k over I_i. This approximation can be expressed in terms of the values of f at the interpolation points. Add the integrals over each subinterval to get an approximation to the integral of f over $[a,b]$.*

4. *Derive an estimate for the local error in one subinterval using Taylor expansions with remainders, and obtain an estimate of the global error by adding the local errors.*

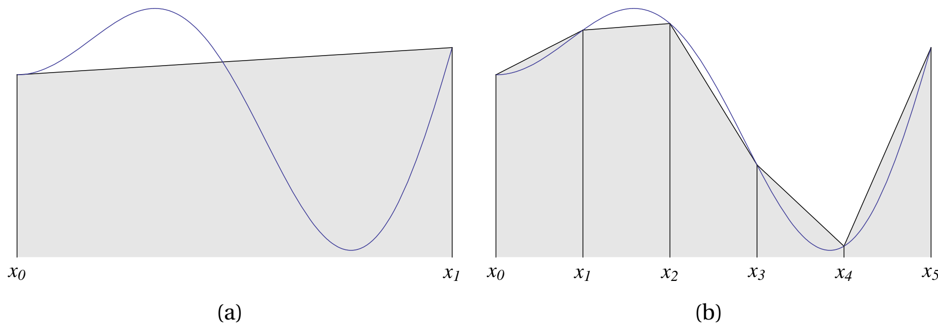


Figure 12.4. The trapezoidal rule with one subinterval (a) and five subintervals (b).

12.3.2 The trapezoidal rule

The midpoint rule is based on a very simple polynomial approximation to the function f to be integrated on each subinterval; we simply use a constant approximation that interpolates 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 12.4a. The approximation to the integral is the area of the trapezoidal polygon under the secant so we have

$$\int_a^b f(x)\,dx \approx \frac{f(a)+f(b)}{2}(b-a). \tag{12.12}$$

To get good accuracy, we will have to split $[a,b]$ into subintervals with a partition and use the trapezoidal approximation on each subinterval, as in figure 12.4b. If we have a uniform partition $\{x_i\}_{i=0}^n$ with step length h, we get the approximation

$$\int_a^b f(x)\,dx = \sum_{i=1}^n \int_{x_{i-1}}^{x_i} f(x)\,dx \approx \sum_{i=1}^n \frac{f(x_{i-1})+f(x_i)}{2}h. \tag{12.13}$$

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 (12.13). This means that if we implement this formula directly we do a lot of unnecessary work. By taking this into consideration and estimating the local and global error in a similar way as we did for the midpoint rule we obtain the following result.

Theorem 12.10 (Trapezoidal 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*

$$\int_a^b f(x)\,dx \approx I_{trap}(h) = h\left(\frac{f(a)+f(b)}{2} + \sum_{i=1}^{n-1} f(x_i)\right). \tag{12.14}$$

The error in this approximation is bounded by

$$\left|I - I_{trap}\right| \le (b-a)\frac{h^2}{12}\max_{x\in[a,b]}\left|f''(x)\right|. \tag{12.15}$$

Once we have the formula (12.14), we can easily derive an algorithm similar to algorithm 12.5. In fact the two algorithms are identical except for the part that calculates the approximations to the integral, so we will not discuss this further.

Example 12.11. We test the trapezoidal rule on the same example as the midpoint rule,

$$\int_0^1 \cos x\,dx = \sin 1 \approx 0.8414709848.$$

As in example 12.6 we split the interval into 2^k subintervals, for $k = 1, 2, \ldots, 10$. The resulting approximations are

h	$I_{\text{trap}}(h)$	Error
0.500000	0.82386686	1.8×10^{-2}
0.250000	0.83708375	4.4×10^{-3}
0.125000	0.84037503	1.1×10^{-3}
0.062500	0.84119705	2.7×10^{-4}
0.031250	0.84140250	6.8×10^{-5}
0.015625	0.84145386	1.7×10^{-5}
0.007813	0.84146670	4.3×10^{-6}
0.003906	0.84146991	1.1×10^{-6}
0.001953	0.84147072	2.7×10^{-7}
0.000977	0.84147092	6.7×10^{-8}

where the error is defined by

$$\int_0^1 f(x)\,dx - I_{\text{trap}}(h).$$

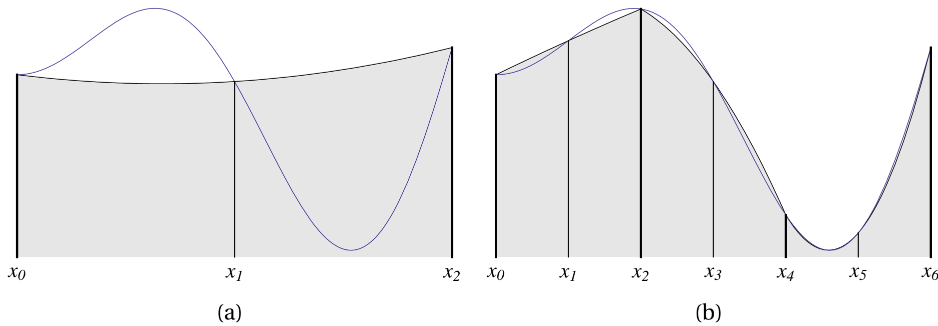


Figure 12.5. Simpson's rule with one subinterval (a) and three subintervals (b).

We note that each time the step length is halved, the error is reduced by a factor of 4, just as for the midpoint rule. But we also note that even though we now use two function values in each subinterval to estimate the integral, the error is actually twice as big as it was for the midpoint rule. This can be seen from the error estimate (12.15) since the constant $1/12$ is twice the size of the constant $1/24$ in (12.11).

12.3.3 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, we omit it here.

Theorem 12.12. *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*

$$\int_a^b f(x)\,dx \approx I_{Simp}(h) = \frac{h}{3}\left(f(a) + f(b) + 2\sum_{i=1}^{n-1} f(x_{2i}) + 4\sum_{i=1}^{n} f(x_{2i-1})\right).$$

The error is bounded by

$$\left|E(f)\right| \le (b-a)\frac{h^4}{180}\max_{x\in[a,b]} |(x)|. \tag{12.16}$$

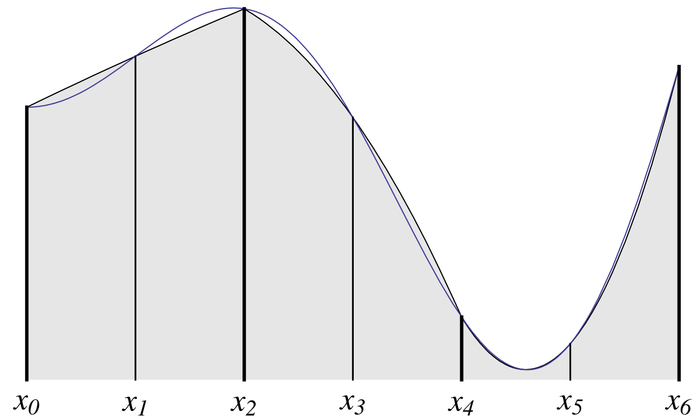


Figure 12.6. Simpson's rule with three subintervals.

In practice, we will usually divide the interval $[a, b]$ into smaller subintervals and use Simpson's rule on each subinterval, see figure 12.5b. Note though that Simpson's rule is not quite like the other numerical integration techniques we have studied when it comes to splitting the interval into smaller pieces: 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 12.6. 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.

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.

Example 12.13. Let us test Simpson's rule on the same example as the midpoint rule and the trapezoidal rule,

$$\int_0^1 \cos x \, dx = \sin 1 \approx 0.8414709848.$$

As in example 12.6, we split the interval into 2^k subintervals, for $k = 1, 2, \dots, 10$.

The result is

h	$I_{\text{Simp}}(h)$	Error
0.250000	0.84148938	-1.8×10^{-5}
0.125000	0.84147213	-1.1×10^{-6}
0.062500	0.84147106	-7.1×10^{-8}
0.031250	0.84147099	-4.5×10^{-9}
0.015625	0.84147099	-2.7×10^{-10}
0.007813	0.84147098	-1.7×10^{-11}
0.003906	0.84147098	-1.1×10^{-12}
0.001953	0.84147098	-6.8×10^{-14}
0.000977	0.84147098	-4.3×10^{-15}
0.000488	0.84147098	-2.2×10^{-16}

where the error is defined by

$$\int_0^1 f(x)\,dx - I_{\text{Simp}}(h).$$

When we compare this table with examples 12.6 and 12.11, we note that the error is now much smaller. We also note that each time the step length is halved, the error is reduced by a factor of 16. In other words, by introducing one more function evaluation in each subinterval, we have obtained a method with much better accuracy. This will be quite evident when we analyse the error below.

The estimate (12.16) explains the behaviour we noticed in example 12.13: Because of the factor h^4, the error is reduced by a factor $2^4 = 16$ when h is halved, and for this reason, Simpson's rule is a very popular method for numerical integration.

Exercises for Section 12.3

1. Mark each of the following statements as true or false.

(a). The trapezoidal rule is usually more accurate than the midpoint rule.

(b). Because every point of measurement in the trapezoidal rule is used in two different subintervals, we must evaluate the function we want to integrate twice at every point.

2. We use the trapezoidal rule to approximate the integral

$$\int_0^1 x^2\,dx$$

using the trapezoidal rule with 2 subintervals. What is the result?

☐ 1/2

☐ 3/8

☐ 5/9

☐ 3/5

3. Calculate an approximation to the integral

$$\int_0^{\pi/2} \frac{\sin x}{1+x^2}\,dx = 0.526978557614\ldots$$

with the trapezoidal rule. Split the interval into 6 subintervals.

4. In this exercise, if you cannot program, use the trapezoidal rule manually with 10 subintervals, check the error, and skip the second part of (b).

(a). Test the trapezoidal rule with 10 subintervals 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 the midpoint rule and check what the actual error is for this value of h. You may have to adjust the midpoint rule function slightly and print the absolute error.

5. When h is halved in the trapezoidal rule, 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 trapezoidal rule with step length $h/2$ that makes it easy to avoid recomputing the function values that were computed on the previous level.

6. Mark each of the following statements as true or false.

(a). Simpson's rule requires that we use an odd number of measurement points.

(b). Simpson's rule is exact for polynomials of degree 3 or lower.

7. Find the correct alternative in the following multiple choice exercises.

(a). (Exam 2010) Which of the integration method (trapezoidal, midpoint and Simpson's) will be most accurate for a polynomial of degree 1?

☐ Just the trapezoidal rule.

☐ Just Simpson's rule.

☐ Just the midpoint rule.

☐ All will be equally accurate.

(b). (Continuation exam 2009) We use Simpson's method to calculate approximations to $\int_a^b f(x)\,dx$ (We do not take round off errors into account). Then the result will always be correct if $f(x)$ is

☐ a trigonometric function.

☐ a logarithmic function.

☐ a polynomial of degree 2.

☐ on the form $g(x)/h(x)$ where f and g are polynomials of degree 2.

(c). (Exam 2008) The midpoint rule evaluates the integral of f on the interval $[a,b]$ by the approximation

$$\int_a^b f(x)\,dx \approx (b-a)f((a+b)/2).$$

We do not take round off errors into account.

☐ The midpoint rule is more accurate than Simpson's rule.

☐ The midpoint rule and the trapezoidal rule always give the exact same error.

☐ The midpoint rule only gives 0 error if $f(x) = c$ for som arbitrary constant c.

☐ The midpoint rule gives 0 error if $f(x)$ is an arbitrary straight line in the x,y-plane.

8. Calculate an approximation to the integral

$$\int_0^{\pi/2} \frac{\sin x}{1+x^2}\,dx = 0.526978557614\ldots$$

with Simpson's rule. Split the interval into 6 subintervals.

9. **(a).** How many function evaluations do you need to calculate the integral

$$\int_0^1 \frac{dx}{1+2x}$$

with the trapezoidal rule to make sure that the error is smaller than 10^{-10}.

(b). How many function evaluations are necessary to achieve the same accuracy with the midpoint rule?

(c). How many function evaluations are necessary to achieve the same accuracy with Simpson's rule?

10. In this exercise, if you cannot program, use Simpson's rule manually with 10 subintervals, check the error, and skip the second part of (b).

(a). Test Simpson's rule with 10 subintervals 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 12.5 slightly and print the absolute error.)

11. **(a).** Verify that Simpson's rule is exact when $f(x) = x^i$ for $i = 0, 1, 2, 3$.

(b). Use (a) to show that Simpson's rule is exact for any cubic polynomial.

(c). Could you reach the same conclusion as in (b) by just considering the error estimate (12.16)?

12. We want to design a numerical integration method

$$\int_a^b f(x) \, dx \approx w_1 f(a) + w_2 f(a_{1/2}) + w_3 f(b).$$

Determine the unknown coefficients w_1, w_2, and w_3 by demanding that the integration method should be exact for the three polynomials $f(x) = x^i$ for $i = 0$, 1, 2. Do you recognise the method?

12.4 Summary

In this chapter we have derived three methods for numerical integration. All these methods and their error analyses may seem rather overwhelming, but they all follow a common thread:

Procedure 12.14. *The following is a general procedure for deriving numerical methods for integration of a function f over the interval $[a, b]$:*

1. *Interpolate the function f by a polynomial p at suitable points.*
2. *Approximate the integral of f by the integral of p. This makes it possible to express the approximation to the integral in terms of function values of f.*
3. *Derive an estimate for the local error by expanding the function values in Taylor series with remainders about the midpoint $a_{1/2} = (a+b)/2$.*
4. *Derive an estimate for the global error by using the technique leading up to theorem 12.8.*

CHAPTER 13

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 itself to its derivative(s).

In this chapter we start by discussing what differential equations are. Our discussion emphasises the simplest ones, the so-called first order equations, which only involve the unknown function and its first derivative. We consider how first order equations 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. In fact, it turns out that even a system of higher order equations can be rewritten as a system of first order equations.

13.1 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.

13.1.1 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,

the distance can be described by a function of time. The divided difference

$$\frac{f(t+\Delta t)-f(t)}{\Delta t} \tag{13.1}$$

then measures the average speed during the time interval from t to $t+\Delta t$. If we take the limit in (13.1) as Δt approaches zero, we obtain the speed $v(t)$ at time t,

$$v(t) = \lim_{\Delta t \to 0} \frac{f(t+\Delta t)-f(t)}{\Delta t}. \tag{13.2}$$

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,

$$a(t) = \lim_{\Delta t \to 0} \frac{v(t+\Delta t)-v(t)}{\Delta t}. \tag{13.3}$$

If we compare the above definitions of speed and acceleration with the definition of the derivative, we notice straightaway that

$$v(t) = f'(t), \qquad a(t) = v'(t) = f''(t). \tag{13.4}$$

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

$$F = ma \tag{13.5}$$

where the proportionality factor m is the mass of the object.

As a simple example of how Newton's law is applied, we 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 (13.4) that $a(t) = v'(t)$ so we have the equation

$$mg - cv(t)^2 = mv'(t)$$

which would usually be shortened and rearranged as

$$mv' = mg - cv^2. \tag{13.6}$$

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 thereby determine the speed $v(t)$.

13.1.2 General use of differential equations

The simple example above illustrates how differential equations are typically used in a variety of contexts:

Procedure 13.1 (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 is obtained.*
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 formula for the unknown function, but this is rare. For a wide range of equations though, it is possible to compute good approximations to the solution via numerical algorithms, and this is the main topic of this chapter.

13.1.3 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 (13.6); this is called a *first order differential equation*. If the equation involves higher derivatives up to 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 in 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 denote 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*.

Exercises for Section 13.1

1. Mark each of the following statements as true or false.

(a). The differential equation $x'(t) + t^2 x(t) = t$ is linear.

(b). The differential equation $x'(t) + tx(t)^2 = t$ is linear.

(c). The differential equation $x'(t) + tx(t)x'(t) = t$ is linear.

2. Newton's law of cooling says that *the rate of heat loss of a body is proportional to the difference in temperatures between the body and the surroundings.* Assuming that you have a cup of coffee placed in a room, with a room temperature of 20 degrees Centrigrade. What would be the appropriate differential equation to model the temperature of the cup?

☐ $T' = T - 20$

☐ $T' = 20T$

☐ $T' = k(20 - T)$

☐ $T' = 20 - 20T$

3. Which of the following differential equations are linear?

(a). $x'' + t^2 x' + x = \sin t$.

(b). $x''' + (\cos t)x' = x^2$.

(c). $x'x = 1$.

(d). $x' = 1/(1+x^2)$.

(e). $x' = x/(1+t^2)$.

13.2 First order differential equations

A first order differential equation is an equation in 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 13.2. 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 $x(t) = t^2 + C$, respectively, 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 explicit formulas for the solutions. In contrast to this, the last two equations do not have solutions given by simple formulas, but we shall see that there are simple numerical methods that allow us to compute good approximations to the solutions.

The situation described in example 13.2 is similar to what we had for nonlinear 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 in 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 a method has been derived we may plug in almost any function f.

13.2.1 Initial conditions

When we solve differential equations numerically we need a bit more information than just the differential equation itself. If we look back on example 13.2, 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 13.3. 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 $x(t) = 2e^{-2}e^{2t} = 2e^{2(t-1)}$.

The general initial condition is $x(a) = x_0$. This leads to $x_0 = x(a) = Ce^{2a}$ or $C = x_0e^{-2a}$. The solution is therefore

$$x(t) = x_0e^{2(t-a)}.$$

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 (13.6) 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 (13.6) 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 downwards 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 13.4 (First order differential equation). *A first order differential equation is an equation in 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, when*

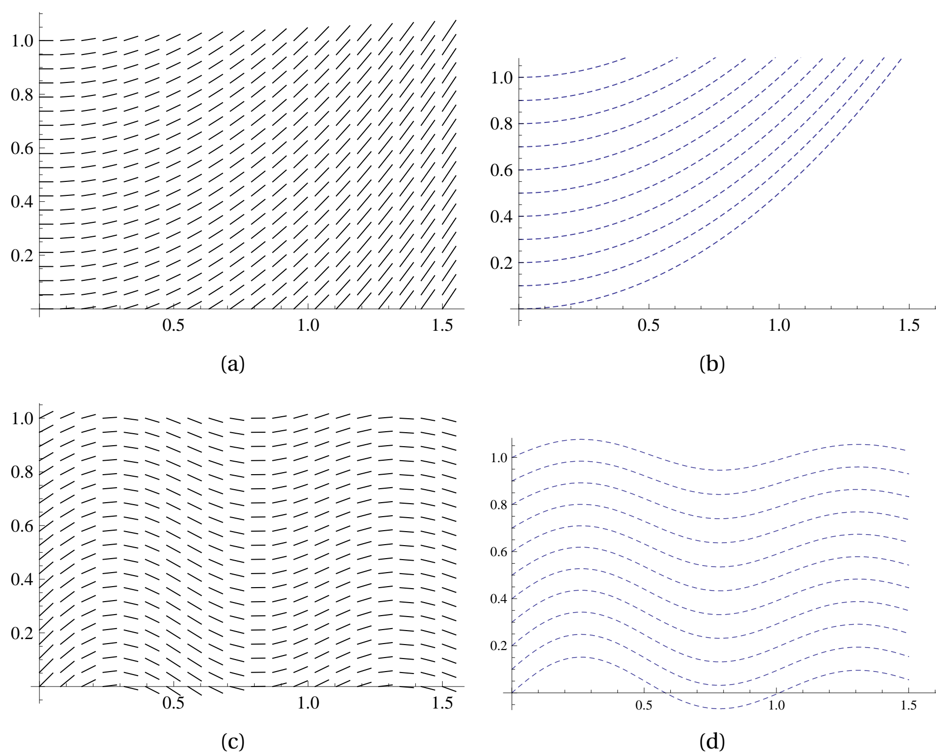


Figure 13.1. Illustration of the geometric interpretation of differential equations. Figure (a) shows 400 tangents generated by the equation $x' = t$, and figure (b) the 11 solution curves corresponding to the initial conditions $x(0) = i/10$ for $i = 0, 1, \ldots, 10$. Figures (c) and (d) show the same information for the differential equation $x' = \cos 6t / \big(1 + t + x^2\big)$.

f is sufficiently smooth, 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

$$x' = f(t, x), \quad x(a) = x_0. \tag{13.7}$$

It is equations of this kind that we will be studying in most of the chapter, with special emphasis on deriving numerical solution algorithms.

13.2.2 A geometric interpretation of first order differential equations

The differential equation in (13.7) 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. A couple of examples will help illustrate this.

Example 13.5. Consider the differential equation

$$x' = f(t,x) = t.$$

This equation describes a family of functions whose tangents have slope t at any point (x,t). At the point $(t,x) = (0,0)$, for example, the slope is given by

$$x'(0) = f(0,0) = 0,$$

i.e., the tangent is horizontal. Similarly, at the point $(t,x) = (0.5,1)$, the slope of the tangent is given by

$$x'(0.5) = f(0.5,1) = 0.5$$

which means that the tangent forms an angle of $\arctan 0.5 \approx 26.6°$ with the t-axis.

In this way, we can compute the tangent direction at any point (x,t) in the plane. Figure 13.1 shows 400 of those tangent directions at a regular grid of points in the rectangle described by $t \in [0,1.5]$ and $x \in [0,1]$ (the length of each tangent is not significant). Note that for this equation all tangents corresponding to the same value of t are parallel. Figure 13.1b shows the actual solutions of the differential equation for the 11 initial values $x(0) = i/10$ for $i = 0, 1, \ldots, 10$.

Since $f(t,x) = t$ is independent of x in this case, the equation can be solved by integration. We find

$$x(t) = \frac{1}{2}t^2 + C,$$

where the constant C corresponds to the initial condition. In other words, we recognise the solutions in (b) as parabolas, and the tangents in (a) as the tangents of these parabolas.

Example 13.6. A more complicated example is provided by the equation

$$x' = f(t,x) = \frac{\cos 6t}{1 + t + x^2}. \tag{13.8}$$

Figure 13.1c shows tangents of the solutions of this equation at a regular grid of 400 points, just like in example 13.5. We clearly perceive a family of wave-like functions, and this becomes clearer in figure 13.1d. The 11 functions in this figure represent solutions of the (13.8), each corresponding to one of the initial conditions $x(0) = i/10$ for $i = 0, \ldots, 10$.

Plots like the ones in figure 13.1a and c are called *slope fields*, and are a common way to visualise a differential equation without solving it.

Observation 13.7 (Geometric 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. A plot of the tangent directions of the solutions of a differential equation is called a* slope field.

It may be tempting to connect neighbouring arrows in a slope field and use this as an approximation to a solution of the differential equation. This is the essence of *Euler's method* which we will study in section 13.3.

13.2.3 Conditions that guarantee existence of one solution

The class of differential equations described by (13.7) is quite general since we have not placed any restrictions on the function f, and this may lead to problems. Consider for example the equation

$$x' = \sqrt{1-x^2}. \tag{13.9}$$

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 like this. 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 always 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 was the case with equation (13.9) 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, so we will not discuss the details of such theorems here, but just informally note the following fact.

Fact 13.8. *The differential equation*

$$x' = f(t,x), \quad x(a) = x_0$$

has a solution for all t near a provided the function f is nicely behaved near the starting point (a, x_0).

The term 'nice' in fact 13.8 typically means that f should be well defined, and both f and its first derivatives should be continuous. When we solve differential equations numerically, it is easy to come up with examples where the solution breaks down because of violations of the condition of 'nice-ness'.

13.2.4 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 $t_1, t_2, t_3, \ldots, t_n$ in an interval $[a, b]$, where $a = t_0 < t_1 < t_2 < t_3 < \cdots < t_n = b$.

Fact 13.9 (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 $x_0, x_1, \ldots, x_n$ such that x_k is an approximation to the true solution $x(t_k)$ at t_k. For $k > 0$, the approximation x_k is computed from one or more of the previous approximations $x_{k-1}, x_{k-2}, \ldots, x_0$. A continuous approximation is obtained by connecting neighbouring points by straight lines.

Exercises for Section 13.2

1. Find the correct alternative in the following multiple choice exercises.

(a). (Continuation Exam 2009) We have the differential equation $y' + ry = -r^2x$ with initial value $y(0) = 1$, where r is an arbitrary real number. The solution is given by

☐ e^{rx}

☐ $1 - r^2x$

☐ $1 + rx$

☐ $1 - rx$

(b). (Exam 2010) We are to solve differential equations numerically. For three of the equations below we may encounter major problems if we choose unfortunate starting values for x and t. Which equation will never give such problems?

☐ $x'x = 1$

☐ $x' = e^t + 2$

☐ $x' = t + \ln x$

☐ $x' = t/(x-2)$

2. 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). $x(-\pi/2) = 3$.

3. What features of the following differential equations could cause problems if you try to solve them?

(a). $x' = t/(1-x)$.

(b). $x' = x/(1-t)$.

(c). $x' = \ln x$.

(d). $x'x = 1$.

(e). $x' = \arcsin x$.

(f). $x' = \sqrt{1-x^2}$.

13.3 Euler's method

Methods for finding analytical solutions of differential equations often 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 13.7.

13.3.1 Basic idea and algorithm

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 $(t_k, x_k)_{k=0}^n$ to the solution, where $t_k = a + kh$.

The initial condition provides us with a point on the true solution, so (t_0, x_0) is also the natural starting point for the approximation. To obtain an approximation to the solution at t_1, we compute the slope of the tangent at (t_0, x_0) as $x'_0 = f(t_0, x_0)$. This gives us the tangent $T_0(t) = x_0 + (t - t_0)x'_0$ to the solution at t_0. As the approximation x_1 at t_1 we use the value of the tangent T_0 which is given by

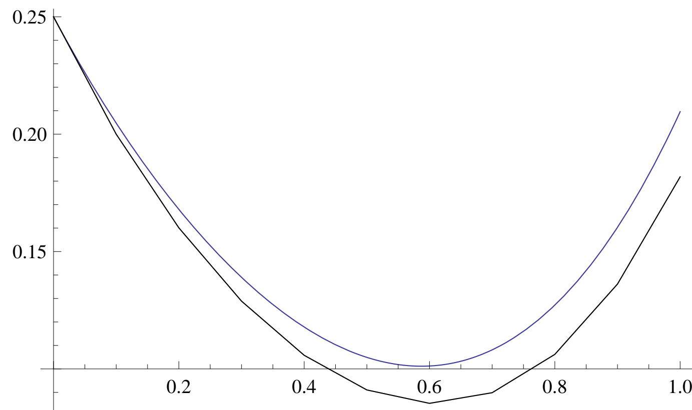
$$x_1 = T_0(t_1) = x_0 + hx'_0 = x_0 + hf(t_0, x_0).$$

This gives us the next approximate solution point (t_1, x_1). To advance to the next point (t_2, x_2), we move along the tangent to the exact solution that passes through (t_1, x_1). The derivative at this point is $x'_1 = f(t_1, x_1)$ and so the tangent is

$$T_1(t) = x_1 + (t - t_1)x'_1 = x_1 + (t - t_1)f(t_1, x_1).$$

The approximate solution at t_2 is therefore

$$x_2 = x_1 + hf(t_1, x_1).$$



(a)

Figure 13.2. Solution of the differential equation $x' = t^3 - 2x$ with initial condition $x(0) = 0.25$ using Euler's method with step length $h = 0.1$. The top function is the exact solution.

If we continue in the same way, 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 crucial 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 13.10. *In Euler's method, an approximate solution (t_k, x_k) is advanced to (t_{k+1}, x_{k+1}) by following the tangent*

$$T_k(t) = x_k + (t - t_k)x'_k = x_k + (t - t_k)f(t_k, x_k)$$

at (t_k, x_k) from t_k to $t_{k+1} = t_k + h$. This results in the approximation

$$x_{k+1} = x_k + hf(t_k, x_k) \tag{13.10}$$

to $x(t_{k+1})$.

Idea 13.10 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. An example will illustrate how this works in practice.

Example 13.11. We consider the differential equation

$$x' = t^3 - 2x, \quad x(0) = 0.25. \tag{13.11}$$

Suppose we want to compute an approximation to the solution at the points $t_1 = 0.1$, $t_2 = 0.2$, ..., $t_{10} = 1$, i.e., the points $t_k = kh$ for $k = 1, 2, \ldots, 10$, with $h = 0.1$.

We start with the initial point $(t_0, x_0) = (0, 0.25)$ and note that $x_0' = x'(0) = 0^3 - 2x(0) = -0.5$. The tangent $T_0(t)$ to the solution at $t = 0$ is therefore given by

$$T_0(t) = x(0) + tx'(0) = 0.25 - 0.5t.$$

To advance the approximate solution to $t = 0.1$, we just follow this tangent,

$$x(0.1) \approx x_1 = T_0(0.1) = 0.25 - 0.5 \times 0.1 = 0.2.$$

At $(t_1, x_1) = (0.1, 0.2)$ the derivative is $x_1' = f(t_1, x_1) = t_1^3 - 2x_1 = 0.001 - 0.4 = -0.399$, so the tangent at t_1 is

$$T_1(t) = x_1 + (t - t_1)x_1' = x_1 + (t - t_1)f(t_1, x_1) = 0.2 - (t - 0.1)0.399.$$

The approximation at t_2 is therefore

$$x(0.2) \approx x_2 = T_1(0.2) = x_1 + hf(t_1, x_1) = 0.2 - 0.1 \times 0.399 = 0.1601.$$

If we continue in the same way, we find (we only print the first 4 decimals)

$$\begin{aligned} x_3 &= 0.1289, & x_4 &= 0.1058, & x_5 &= 0.0910, & x_6 &= 0.0853, \\ x_7 &= 0.0899, & x_8 &= 0.1062, & x_9 &= 0.1362, & x_{10} &= 0.1818. \end{aligned}$$

This is illustrated in figure 13.2 where the computed points are connected by straight line segments.

From the description above and example 13.11 it is easy to derive a more formal algorithm.

Algorithm 13.12 (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*

> $h = (b - a)/n;$
> $t_0 = a;$
> 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 $k = 0, 1, \ldots, n$.

13.3.2 Geometric interpretation

Recall that a differential equation without an initial condition in general has a whole family of solutions, with each particular solution corresponding to a specific initial condition. With this in mind we can give a geometric interpretation of Euler's method. This is easiest by referring to a figure like figure 13.3 which shows the behaviour of Euler's method for the general equation

$$x' = f(t,x), \quad x(a) = x_0,$$

for which

$$f(t,x) = \frac{\cos 6t}{1+t+x^2}, \quad x(0) = 0.$$

The plot in figure 13.3a shows both the approximate solution (dots connected by straight line segments)and the exact solution, but the figure in (b) illustrates better how the approximation is obtained. We start off by following the tangent T_0 at the initial condition $(0,0)$. This takes us to a point (t_1, x_1) that is slightly above the graph of the true solution. There is a solution curve that passes through this second point which corresponds to the original differential equation, but with a different initial condition,

$$x' = f(t,x), \quad x(t_1) = x_1.$$

The solution curve given by this equation has a tangent at t_1, and this is the line we follow to get from (t_1, x_1) to (t_2, x_2). This takes us to another solution curve given by the equation

$$x' = f(t,x), \quad x(t_2) = x_2.$$

Euler's method continues in this way, by jumping from solution curve to solution curve.

Observation 13.13. *Euler's method may be interpreted as stepping between different solution curves of the equation $x' = f(t,x)$. At time t_k, the tangent T_k to the solution curve given by*

$$x' = f(t,x), \quad x(t_k) = x_k$$

is followed to the point (t_{k+1}, x_{k+1}), which is a point on the solution curve given by

$$x' = f(t,x), \quad x(t_{k+1}) = x_{k+1}.$$

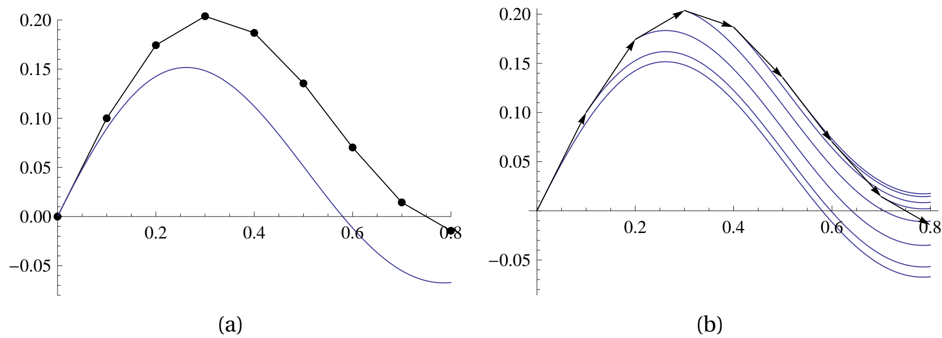


Figure 13.3. The plot in (a) shows the approximation produced by Euler's method to the solution of the differential equation $x' = \cos 6t/(1+t+x^2)$ with initial condition $x(0) = 0$ (smooth graph). The plot in (b) shows the same solution augmented with the solution curves that pass through the points produced by Euler's method.

13.3.3 The error in Euler's method

As for any numerical method that computes an approximate solution, it is important to have an understanding of the limitations of the method, especially its error. As usual, the main tool is Taylor polynomials with remainders.

Definition 13.14 (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*

$$|\epsilon_k| \leq O(h^p),$$

i.e., if

$$|\epsilon_k| \leq Ch^p,$$

for some constant C that is independent of h.

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 13.15. *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 since Euler's method is based on approximation by the tangent, i.e., a first order Taylor polynomial which has a quadratic error term. The explanation is that although the error at one step is quadratic, the error accumulates so that the global order becomes 1.

Exercises for Section 13.3

1. Mark each of the following statements as true or false.

(a). Euler's method gives the values of the exact solution at all the points x_0, $x_1, \ldots, x_n$ if the differential equation is linear.

(b). In Euler's method it is assumed that the solution is a straight line between each calculated point.

2. We have the differential equation $x' = \sqrt{1-x^2}$, $x(0) = 0$ and want to approximate the value of $x(0.1)$ by using a single step with Euler's method. What will the approximated value be?

☐ $x(0.1) = 1/10$

☐ $x(0.1) = 1$

☐ $1/2$

☐ $1/4$

3. Use Euler's method with three steps with $h = 0.1$ on your calculator to compute approximate solutions of the following differential equations:

(a). $x' = t + x, \quad x(0) = 1.$

(b). $x' = \cos x, \quad x(0) = 0.$

(c). $x' = t/(1+x^2), \quad x(0) = 1.$

(d). $x' = 1/x, \quad x(1) = 1.$

(e). $x' = \sqrt{1-x^2}, \quad x(0) = 0.$

4. Write a program that implements Euler's method for first order differential equations in 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 on the equation $x' = x$ with $x(0) = 1$ on the interval $[0,1]$. Plot the exact solution $x(t) = e^t$ alongside the approximation and experiment with different values of n.

5. Suppose we have the differential equation

$$x' = f(t,x), \quad x(b) = x_0,$$

and we seek a solution on the interval $[a,b]$ where $a < b$. Adjust algorithm 13.12 so that it works in this alternative setting where the initial value is at the right end of the interval.

6. Recall that a common approximation to the derivative of x is given by

$$x'(t) \approx \frac{x(t+h) - x(t)}{h}.$$

Derive Euler's method by rewriting this and making use of the differential equation $x'(t) = f\big(t, x(t)\big)$.

7. Mark each of the following statements as true or false.

(a). The order of the global error in Euler's method is one lower than the order of the local error.

(b). Round-off is a major source of errors when we use Euler's method to solve differential equations numerically.

(c). When we decrease the step length h in Euler's method from 0.2 to 0.1, the local error will be reduced by a factor of roughly 4.

8. Suppose we perform one step with Euler's method for the differential equation

$$x' = \sin x, \quad x(0) = 1.$$

Find an upper bound for the absolute error.

9. In this exercise we are going to solve the differential equation

$$x' = f(t,x) = t^2 + x^3 - x, \quad x(0) = 1 \tag{13.12}$$

numerically by using a quadratic Taylor polynomial to advance the approximation form one point to the next.

(a). Find a formula for $x''(t)$ by differentiating equation 13.12.

(b). Use the quadratic Taylor method and your result from (a) to find an approximation to $x(1)$ using 1, 2 and, 5 steps.

(c). Program the quadratic Taylor method and use the program to find an approximation of $x(1)$ with 10, 100 and 1000 steps.

13.4 Midpoint Euler and other Runge-Kutta methods

Euler's method can be generalised by using a higher degree Taylor polynomial instead of the tangent in the basic approximation step, see exercise 9. This will naturally result in more accurate methods, but requires differentiation of the differential equation. In this section we describe some methods of higher order than Euler's method that do not require such differentiation. 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 $[t_k, t_{k+1}]$.

13.4.1 Euler's midpoint method

The first method we consider is a simple improvement of Euler's method. If we look at the plots in figure 13.3, 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 13.4.

Idea 13.16 (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*

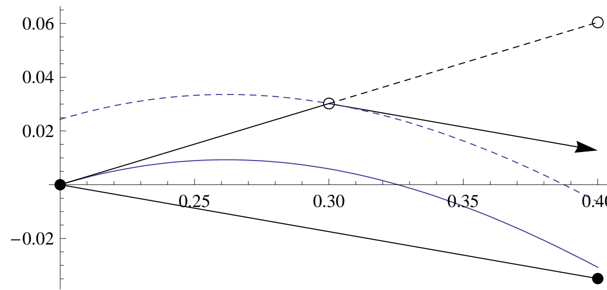


Figure 13.4. The figure illustrates the first step of the midpoint Euler method, starting at $x = 0.2$ and with step length $h = 0.2$. We start by following the tangent at the starting point ($x = 0.2$) to the midpoint ($x = 0.3$). Here we determine the slope of the solution curve that passes through 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.

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 $f(t_k + h/2, x_{k+1/2})$,

$$x_{k+1} = x_k + hf(t_k + h/2, x_{k+1/2}). \tag{13.13}$$

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 13.17 (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*

$h = (b - a)/n$;
$t_0 = a$;

for $k = 0, 1, \dots, n-1$
$x_{k+1/2} = x_k + hf(t_k, x_k)/2$;

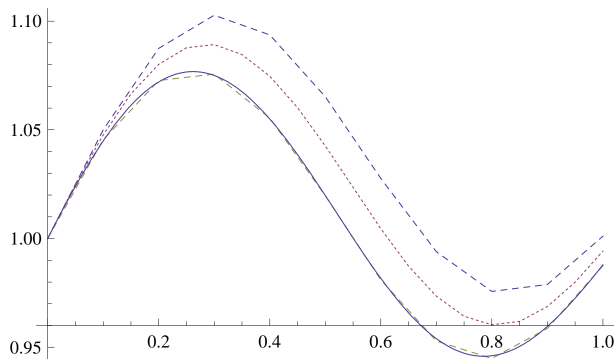


Figure 13.5. Comparison of Euler's method and Euler's midpoint method for the differential equation $x' = \cos(6t)/(1+t+x^2)$ with initial condition $x(0) = 1$ with step length $h = 0.1$. The solid curve is the exact solution and the two approximate solutions are dashed. The dotted curve in the middle 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.

> $x_{k+1} = x_k + hf(t_k + h/2, x_{k+1/2})$;
> $t_{k+1} = a + (k+1)h$;
>
> *After these steps the value x_k will be an approximation to the solution $x(t_k)$ of the differential equation at t_k, for each $k = 0, 1, \ldots, n$.*

As an alternative viewpoint, let us recall the two approximations for numerical differentiation given by

$$x'(t) \approx \frac{x(t+h) - x(t)}{h},$$
$$x'(t+h/2) \approx \frac{x(t+h) - x(t)}{h}.$$

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 13.5 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 13.5.

13.4.2 The error

The error in Euler's midpoint method can be analysed with the help of Taylor expansions. In this case, we first do a Taylor expansion with respect to t, and then another Taylor expansion with respect to x. The analysis shows that the extra evaluation of f at the midpoint improves the error estimate from $O(h^2)$ (for Euler's method) to $O(h^3)$, i.e., the same as the error for the quadratic Taylor method. As for the Taylor methods, the global error is one order lower.

Theorem 13.18. *Euler's midpoint method is of order 2, i.e., the global error is proportional to h^2.*

13.4.3 Runge-Kutta methods

Runge-Kutta methods are generalisations of 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

$$x_{k+1} = x_k + h\big(\lambda_1 f(t_k, x_k) + \lambda_2 f(t_k + r_1 h, x_k + r_2 h f(t_k, x_k)\big), \tag{13.14}$$

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 the relation (13.14) mimics a Taylor method of the highest possible order. 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 13.19 (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*

$$x_{k+1} = x_k + h\Big((1-\lambda) f(t_k, x_k) + \lambda f\Big(t_k + \frac{h}{2\lambda}, x_k + \frac{hf(t_k, x_k)}{2\lambda}\Big)\Big), \qquad (13.15)$$

is 2nd order accurate for any nonzero value of the parameter λ, provided f and its derivatives up to order two are continuous and bounded for $t \in [a,b]$ and $x \in \mathbb{R}$.

The strategy of the proof of theorem 13.19 is similar to the error analysis for Euler's method, but quite technical.

Note that 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 method 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 13.20 (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*

$$\left.\begin{aligned}
k_0 &= f(t_k, x_k),\\
k_1 &= f(t_k + h/2, x_k + hk_0/2),\\
k_2 &= f(t_k + h/2, x_k + hk_1/2),\\
k_3 &= f(t_k + h, x_k + hk_2),\\
x_{k+1} &= x_k + \frac{h}{6}(k_0 + 2k_1 + 2k_2 + k_3),
\end{aligned}\right\}\quad k = 0, 1, \dots, n$$

is 4th order accurate provided the derivatives of f up to order four are continuous and bounded for $t \in [a,b]$ and $x \in \mathbb{R}$.

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 per 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 per evaluation.

Exercises for Section 13.4

1. Find the correct alternative in the following multiple choice exercises.

(a). (Continuation exam 2009)

☐ When solving differential equations numerically, round-off errors are never a problem.

☐ When doing numerical differentiation, round off errors are never a problem.

☐ When solving differential equations numerically, Euler's method is usually less accurate than the 4th order Runge-Kutta method.

☐ When doing numerical integration, the trapezoidal rule is usually more accurate than Simson's rule.

(b). (Exam 2009)

☐ When solving differential equations numerically, Euler's method is usually more accurate than Euler's midpoint method.

☐ When solving differential equations numerically, Taylor's method of third order is usually more accurate than Euler's method.

☐ When solving differential equations numerically, Euler's method is usually more accurate than Taylor's method of second order.

☐ When numerical integration, the trapezoidal rule is usually more accurate than Simpson's rule.

(c). (Continuation exam 2007)

☐ The bisection method is a method for solving differential equations numerically.

☐ Round-off errors never create problems when solving differential equations numerically.
☐ Difference equations is a special case of differential equations.
☐ The 4th order Runge Kutta method is more accurate than Euler's method.

2. Consider the first order differential equation

$$x' = x, \quad x(0) = 1.$$

(a). Estimate $x(1)$ by using one step with Euler's method.

(b). Estimate $x(1)$ by using one step with Euler's midpoint method.

(c). Estimate $x(1)$ by using one step with the Runge Kutta fourth order method.

(d). Estimate $x(1)$ by using two steps with the Runge Kutta fourth order method.

(e). 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?

(f). Do the estimates seem to converge?

(g). Solve the equation analytically and explain your numerical results.

3. 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 midpoint method with 1 and 5 steps and plot the results.

(c). Compare the results from Euler's midpoint method with those form Euler's method including the number of evaluations of f in each case. Which method seems to be best?

4. 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 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.

5. 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?

6. In this problem we are going to use Euler's method to solve the differential equation you found in exercise 5 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 6. What are the relative errors at the computed points?

13.5 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 often necessary to model a problem, and sometimes 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.

13.5.1 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 13.21. At the beginning of this chapter we saw that a vertically falling object subject to gravitation and friction can be modelled by the differential equation

$$v' = g - \frac{c}{m} v^2, \tag{13.16}$$

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 (13.16) 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

$$v_1' = -\frac{c}{m} v_1^2, \qquad v_1(0) = v_{0_x}, \tag{13.17}$$

$$v_2' = \frac{c}{m} v_2^2 - g, \quad v_2(0) = v_{0_y}, \tag{13.18}$$

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 (13.17)–(13.18) 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 13.21 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.

Example 13.22. Consider the three equations with initial conditions

$$x' = xy + \cos z, \qquad x(0) = x_0, \tag{13.19}$$

$$y' = 2 - t^2 + z^2 y, \quad y(0) = y_0, \tag{13.20}$$

$$z' = \sin t - x + y, \quad z(0) = z_0. \tag{13.21}$$

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 (13.19)–(13.21) simply as

$$\boldsymbol{x}' = \boldsymbol{f}(t, \boldsymbol{x}), \quad \boldsymbol{x}(0) = \boldsymbol{x}_0.$$

Examples 13.21–13.22 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 13.23. *A system of M first order differential equations in M unknowns with corresponding initial conditions is given by a vector relation in the form*

$$\boldsymbol{x}' = \boldsymbol{f}(t, \boldsymbol{x}), \quad \boldsymbol{x}(a) = \boldsymbol{x}_0. \tag{13.22}$$

Here $\boldsymbol{x} = \boldsymbol{x}(t) = \big(x_1(t), \ldots, \boldsymbol{x}_M(t)\big)$ 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 $\boldsymbol{x} = (x_1, \ldots, x_M)$, i.e.,

$$\boldsymbol{f}(t, \boldsymbol{x}) = \big(f_1(t, \boldsymbol{x}), \ldots, f_M(t, \boldsymbol{x})\big),$$

while $\boldsymbol{x}_0 = (x_{1,0}, \ldots, x_{M,0})$ is a vector in $\mathbb{R}^M$ of initial values. The notation $\boldsymbol{x}'$ denotes the vector of derivatives of the components of $\boldsymbol{x}$ with respect to t,

$$\boldsymbol{x}' = \boldsymbol{x}'(t) = \big(x_1'(t), \ldots, x_M'(t)\big).$$

It may be helpful to write out the vector equation (13.22) in detail,

$$\begin{aligned}
x_1' &= f_1(t, \boldsymbol{x}) = f_1(t, x_1, \ldots, x_M), \qquad x_1(0) = x_{1,0} \\
&\vdots \\
x_M' &= f_M(t, \boldsymbol{x}) = f_M(t, x_1, \ldots, x_M), \quad x_M(0) = x_{M,0}.
\end{aligned}$$

We see that both the examples above fit into this setting, with $M = 2$ for example 13.21 and $M = 3$ for example 13.22.

Before we start considering numerical solutions of systems of differential equations, we need to know that solutions exist.

Theorem 13.24. *The system of equations*

$$\boldsymbol{x}' = \boldsymbol{f}(t, \boldsymbol{x}), \quad \boldsymbol{x}(a) = \boldsymbol{x}_0$$

has a solution near the initial value $(a, \boldsymbol{x}_0)$ provided all the component functions are reasonably well-behaved near this point.

13.5.2 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 13.25 (Euler's method for a system). We consider the equations in example 13.22,

$$\boldsymbol{x}' = \boldsymbol{f}(t,\boldsymbol{x}), \quad \boldsymbol{x}(a) = \boldsymbol{x}_0,$$

where

$$\begin{aligned}\boldsymbol{f}(t,\boldsymbol{x}) &= \big(f_1(t,x_1,x_2,x_3), f_2(t,x_1,x_2,x_3), f_3(t,x_1,x_2,x_3)\big)\\ &= (x_1x_2+\cos x_3, 2-t^2+x_3^2x_2, \sin t - x_1 + x_2).\end{aligned}$$

Euler's method is easily generalised to vector equations as

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + h\boldsymbol{f}(t_k,\boldsymbol{x}_k), \quad k = 0, 1, \ldots, n-1. \tag{13.23}$$

If we write out the three components explicitly, this becomes

$$\left.\begin{aligned}x_1^{k+1} &= x_1^k + hf_1(t_k,x_1^k,x_2^k,x_3^k) = x_1^k + h\big(x_1^kx_2^k + \cos x_3^k\big),\\ x_2^{k+1} &= x_2^k + hf_2(t_k,x_1^k,x_2^k,x_3^k) = x_2^k + h\big(2 - t_k^2 + (x_3^k)^2x_2^k\big),\\ x_3^{k+1} &= x_3^k + hf_3(t_k,x_1^k,x_2^k,x_3^k) = x_3^k + h\big(\sin t_k - x_1^k + x_2^k\big),\end{aligned}\right\} \tag{13.24}$$

for $k = 0, 1, \ldots, n-1$, with the starting values (a, 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 (13.23), since it nicely hides the details in (13.24).

Example 13.25 illustrates Euler's method for a system of equations, and the other methods we have discussed earlier in the chapter also generalise to systems of equations in a straightforward way.

Observation 13.26 (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

$$\begin{aligned}
\boldsymbol{k}_0 &= \boldsymbol{f}(t_k, \boldsymbol{x}_k),\\
\boldsymbol{k}_1 &= \boldsymbol{f}(t_k + h/2, \boldsymbol{x}_k + h\boldsymbol{k}_0/2),\\
\boldsymbol{k}_2 &= \boldsymbol{f}(t_k + h/2, \boldsymbol{x}_k + h\boldsymbol{k}_1/2),\\
\boldsymbol{k}_3 &= \boldsymbol{f}(t_k + h, \boldsymbol{x}_k + h\boldsymbol{k}_2),\\
\boldsymbol{x}_{k+1} &= \boldsymbol{x}_k + \frac{h}{6}(\boldsymbol{k}_0 + 2\boldsymbol{k}_1 + 2\boldsymbol{k}_2 + \boldsymbol{k}_3).
\end{aligned}$$

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 explicitly, 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.

13.5.3 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 13.27. Consider the second order equation

$$x'' = t^2 + \sin(x + x'), \quad x(0) = 1, \quad x'(0) = 0. \tag{13.25}$$

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 (13.25) can be written as the system

$$x_1' = x_2, \qquad x_1(0) = 1, \tag{13.26}$$

$$x_2' = t^2 + \sin(x_1 + x_2), \quad x_2(0) = 0. \tag{13.27}$$

In other words, equation (13.25) can be written as the system (13.26)–(13.27). We also see that this system can be expressed as the single equation in (13.25), so the two equations (13.26)–(13.27) and the single equation (13.25) are in fact equivalent in the sense that a solution of one automatically gives a solution of the other.

The technique used in example 13.27 works in general—a pth order equation can be rewritten as a system of p first order equations.

Theorem 13.28. *The pth order differential equation*

$$x^{(p)} = g\big(t, x, x', \dots, x^{(p-1)}\big) \tag{13.28}$$

with initial conditions

$$x(a) = d_0,\ x'(a) = d_1, \dots, x^{(p-2)}(a) = d_{p-2},\ x^{(p-1)}(a) = d_{p-1} \tag{13.29}$$

is equivalent to the system of p equations in the p unknown functions x_1, x_2, $\dots$, x_p,

$$\begin{aligned} x_1' &= x_2, & x_1(a) &= d_0,\\ x_2' &= x_3, & x_2(a) &= d_1,\\ &\ \vdots \\ x_{p-1}' &= x_p, & x_{p-1}(a) &= d_{p-2},\\ x_p' &= g(t, x_1, x_2, \dots, x_{p-1}), & x_p(a) &= d_{p-1}, \end{aligned} \tag{13.30}$$

in the sense that the component solution $x_1(t)$ of (13.30) *agrees with the solution $x(t)$ of* (13.28)–(13.29).

Proof. The idea of the proof is just like in example 13.27. From the first $p-1$ relations in (13.30) we see that

$$x_2 = x_1', \quad x_3 = x_2' = x_1'', \quad \dots, \quad x_p = x_{p-1}' = x_{p-2}'' = \cdots = x_1^{(p-1)}.$$

If we insert this in the last equation in (13.30) we obtain a pth order equation for x_1 that is equivalent to (13.28). In addition, the initial values in (13.30) translate into initial values for x_1 that are equivalent to (13.29), so x_1 must solve (13.28)–(13.29). Conversely, if x is a solution of (13.28)–(13.29), it is easy to see that the functions

$$x_1 = x, \quad x_2 = x', \quad x_3 = x'', \quad \dots, \quad x_{p-1} = x^{(p-2)}, \quad x_p = x^{(p-1)}$$

solve the system (13.30). ∎

Theorem 13.28 shows that if we can solve systems of differential equations we can also solve single equations of order higher than one. It turns out that we even handle systems of higher order equations in this way.

Example 13.29 (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

$$\begin{aligned} x_1' &= x_2, && x_1(0) = 1, \\ x_2' &= t + x_2 + y_2, && x_2(0) = 2, \\ y_1' &= y_2, && y_1(0) = -1, \\ y_2' &= y_3, && y_2(0) = 1, \\ y_3' &= x_2 y_3 + x_1, && y_3(0) = 2. \end{aligned}$$

Example 13.29 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,

$$x_i^{(p_i)} = g_i\Big(t, x_1, x_1', \ldots, x_1^{(p_1-1)}, \ldots, x_r, x_r', \ldots, x_r^{(p_r-1)}\Big), \quad i = 1, \ldots, r. \tag{13.31}$$

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 2).

To write the system (13.31) as a system of first order equations, we just follow the same strategy as in example 13.29: 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 (13.31) can then be replaced by the p_i first order equations

$$\begin{aligned}
x'_{i,1} &= x_{i,2},\\
x'_{i,2} &= x_{i,3},\\
&\vdots\\
x'_{i,p_i-1} &= x_{i,p_i},\\
x'_{i,p_i} &= g_i\big(t, x_{1,1}, \ldots, x_{1,p_1}, \ldots, x_{r,1}, \ldots, x_{r,p_r}\big)
\end{aligned}$$

for $i = 1, \ldots, r$. We emphasise that the general procedure is exactly the same as the one used in example 13.29, it is just that the notation becomes rather heavy in the general case.

We record the conclusion in a non-technical theorem.

Theorem 13.30. *A system of differential equations can always be written as a system of first order equations.*

Exercises for Section 13.5

1. (Continuation exam 2009) The solution $x(t)$ of the differential equation $x'' + \sin(tx') - x^2 = e^t$ is equal to the solution $x_1(t)$ of the system of two equations

☐ $x'_1 = x_1, \quad x'_2 = e^t - \sin(tx_2) + x_1^2$

☐ $x'_1 = x_2, \quad x'_2 = e^t - \sin(tx_1) + x_2^2$

☐ $x'_1 = x_2, \quad x'_2 = e^t - \sin(tx_2) + x_1^2$

☐ $x'_1 = x_2, \quad x'_2 = e^t - \sin(tx_1) + x_1^2$

13.6 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 Euler's method, and this kind of analysis can be extended to the 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 for Section 13.6

1. 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).

$$y'' = y^2 - x + e^t,$$
$$x'' = y - x^2 - e^t.$$

(b).

$$x'' = 2y - 4t^2 x,$$
$$y'' = -2x - 2tx'.$$

(c).

$$x'' = y'' x + (y')^2 x,$$
$$y'' = -y.$$

(d).

$$x''' = y'' x^2 - 3(y')^2 x,$$
$$y'' = t + x'.$$

2. 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 (13.31) since x''' appears on the right in the second equation. Hint: You may need to differentiate one of the equations.

3. 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). $x'' + t^2 x' + 3x = 0$.

(b). $mx'' = -k_s x - k_d x'$.

(c). $y''(t) = 2(e^{2t} - y^2)^{1/2}$.

(d). $2x'' - 5x' + x = 0$ with initial conditions $x(3) = 6$, $x'(3) = -1$.

4. Solve the system

$$\begin{aligned} x'' &= 2y - \sin\big(4t^2 x\big), & x(0) = 1,\ x'(0) = 2, \\ y'' &= -2x - \frac{1}{2t^2(x')^2 + 3}, & y(0) = 1,\ y'(0) = 0, \end{aligned}$$

numerically on the interval $[0,2]$. Try both Euler's method and Euler's midpoint method with two time steps and plot the results.

5. This exercise is based on example 13.21 in which we modelled the movement of a ball thrown through air with the equations

$$\begin{aligned} v_1' &= -\frac{c}{m} v_1^2, & v_1(0) = v_{0_x}, \\ v_2' &= \frac{c}{m} v_2^2 - g, & v_2(0) = v_{0_y}, \end{aligned}$$

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 horizontal 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

$$x(0) = y(0) = 0,\ x'(0) = 200\ \text{km/h and}\ y'(0) = 300\ \text{km/h}.$$

Use the constants

$$a = 1.9 * 10^{-4}\frac{\text{Nh}^2}{\text{km}^3\text{kg}}, \quad g_0 = 3.98 * 10^8\frac{(\text{km})^2\text{m}}{\text{s}^2}, \quad c_0 = 0.19\frac{\text{Nh}^2}{\text{km}^2\text{kg}}.$$

The units are not so important, but mean that distances can be measured in km and speeds in km/h.

6. 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.

(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).

7. 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 know that the acceleration is given by $a(t) = x''(t)$. Newton's second law applied to the spring now yields

$$mx''(t) = -kx(t). \tag{13.32}$$

Suppose that the block has mass $m = 0.25$ kg and that the spring starts from rest in a position 5.0 cm from its equilibrium so $x(0) = 5.0$ cm and $x'(0) = 0.0$ cm/s.

(a). Rewrite this second order differential equation (13.32) 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?

8. This is a continuation of exercise 7, 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

$$F_g = -mg. \tag{13.33}$$

Applying Newton's second law we now obtain the differential equation

$$my''(t) = -ky(t) - mg. \tag{13.34}$$

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 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 7. What are the differences? Did your choice of numerical method work better than the second order Runge-Kutta method in exercise 7?

Part IV

Functions of two variables

CHAPTER 14

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.

14.1 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.

14.1.1 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 14.1 (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 $f(x, y)$.*

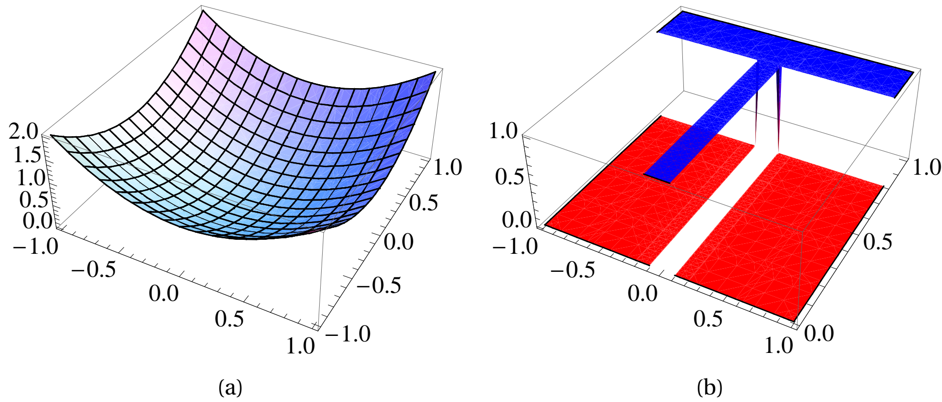


Figure 14.1. The plot in (a) shows the function $f(x, y) = x^2 + y^2$ with x and y varying in the interval $[-1, 1]$. The function in (b) is defined by the rule that $f(x, y) = 0$ except in a small area around the y-axis and the line $y = 1$, where the value is $f(x, y) = 1$.

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 14.1.

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 14.1 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 14.2. *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 $\mathbb{A} \times \mathbb{B}$,*

$$\mathbb{A} \times \mathbb{B} = \{(a, b) \mid a \in \mathbb{A} \text{ and } b \in \mathbb{B}\}.$$

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 14.3. *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 $\mathbb{A}$.

14.1.2 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 14.4 (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 14.5. *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 14.6. *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 14.6 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 $f(x,y) = x^3y + x^2y^2$, the partial derivatives are $\partial f/\partial x = 3x^2y + 2xy^2$ and $\partial f/\partial y = x^3 + 2x^2y$. Each of these can of course be differentiated again,

$$\frac{\partial^2 f}{\partial x^2} = 6xy + 2y^2, \qquad \frac{\partial^2 f}{\partial y \partial x} = \frac{\partial}{\partial y}\left(\frac{\partial f}{\partial x}\right) = 3x^2 + 4xy,$$
$$\frac{\partial^2 f}{\partial y^2} = 2x^2, \qquad \frac{\partial^2 f}{\partial x \partial y} = \frac{\partial}{\partial x}\left(\frac{\partial f}{\partial y}\right) = 3x^2y + 4xy.$$

We notice that the two mixed derivatives are equal. In general the derivatives

$$\frac{\partial^2 f}{\partial x \partial y}(\boldsymbol{a}), \quad \frac{\partial^2 f}{\partial y \partial x}(\boldsymbol{a})$$

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 14.7 (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.

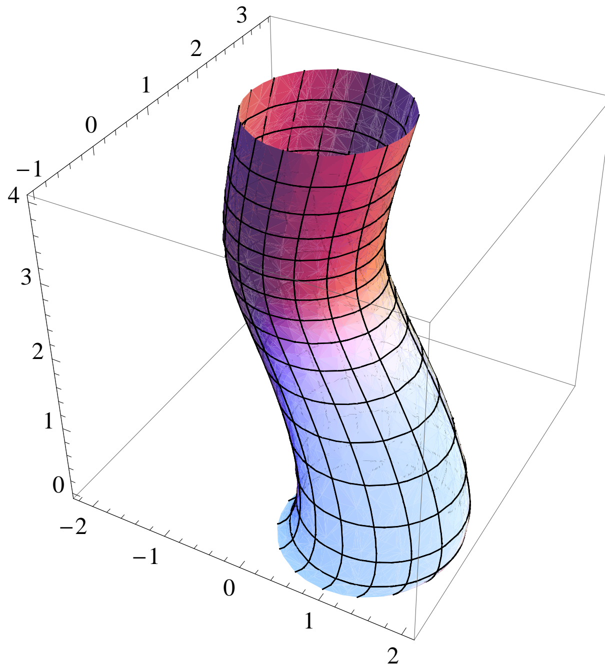


Figure 14.2. An example of a parametric surface.

14.1.3 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 14.8 (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 14.9. *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 14.2. 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.

Exercises for Section 14.1

1. Mark each of the following statements as true or false.

(a). For most well-behaved functions, we have that

$$\frac{\partial^2 f(x,y)}{\partial x \partial y} = \frac{\partial^2 f(x,y)}{\partial y \partial x}.$$

(b). The function $\mathbf{f}(x,y) = (x+y, x-y)$ is scalar.

14.2 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 14.6 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 14.10. *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

$$\frac{\partial f}{\partial x}(a,b) - \frac{f(a+h_1,b)-f(a,b)}{h_1} = \frac{h_1}{2}\frac{\partial^2 f}{\partial x^2}(c_1,b), \tag{14.1}$$

$$\frac{\partial f}{\partial y}(a,b) - \frac{f(a,b+h_2)-f(a,b)}{h_2} = \frac{h_2}{2}\frac{\partial^2 f}{\partial y^2}(a,c_2), \tag{14.2}$$

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 (14.1) 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

$$\frac{\partial f}{\partial x} = \frac{f(a+h,b)-f(a-h,b)}{2h} + \frac{h^2}{6}\frac{\partial^3 f}{\partial x^3}(c,b) \tag{14.3}$$

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 14.11. Let us consider the simplest mixed derivative,

$$\frac{\partial^2 f}{\partial x \partial y}(a,b).$$

If we set

$$g(a) = \frac{\partial f}{\partial y}(a,b) \tag{14.4}$$

we can use the approximation

$$g'(a) \approx \frac{g(a+h_1) - g(a-h_1)}{2h_1}.$$

If we insert (14.4) in this approximation we obtain

$$\frac{\partial^2 f}{\partial x \partial y}(a,b) \approx \frac{\dfrac{\partial f}{\partial y}(a+h_1,b) - \dfrac{\partial f}{\partial y}(a-h_1,b)}{2h_1}. \tag{14.5}$$

Now we can use the same kind of approximation for the two first-order partial derivatives in (14.5),

$$\begin{aligned}
\frac{\partial f}{\partial y}(a+h_1,b) &\approx \frac{f(a+h_1,b+h_2) - f(a+h_1,b-h_2)}{2h_2},\\
\frac{\partial f}{\partial y}(a-h_1,b) &\approx \frac{f(a-h_1,b+h_2) - f(a-h_1,b-h_2)}{2h_2}.
\end{aligned}$$

If we insert these expressions in (14.5) we obtain the final approximation

$$\frac{\partial^2 f}{\partial x \partial y}(a,b) \approx \frac{f(a+h_1,b+h_2) - f(a+h_1,b-h_2) - f(a-h_1,b+h_2) + f(a-h_1,b-h_2)}{4h_1h_2}.$$

If we introduce the notation

$$\begin{aligned}
f(a-h_1,b-h_2) &= f_{-1,-1}, & f(a+h_1,b-h_2) &= f_{1,-1},\\
f(a-h_1,b+h_2) &= f_{-1,1}, & f(a+h_1,b+h_2) &= f_{1,1},
\end{aligned} \tag{14.6}$$

we can write the approximation more compactly as

$$\frac{\partial^2 f}{\partial x \partial y}(a,b) \approx \frac{f_{1,1} - f_{1,-1} - f_{-1,1} + f_{-1,-1}}{4h_1h_2}.$$

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 $(a+h_1, b+h_2)$.

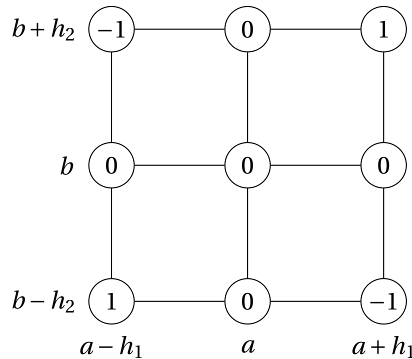


Figure 14.3. The weights involved in computing the mixed second derivative with the approximation in example 14.11. This kind of figure is referred to as the *computational molecule* of the approximation.

We record the approximation in example 14.11 in a proposition. We do not have the right tools to estimate the error, but just indicate how it behaves.

Proposition 14.12 (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*

$$\frac{\partial^2 f}{\partial x \partial y}(a,b) \approx \frac{f_{1,1} - f_{1,-1} - f_{-1,1} + f_{-1,-1}}{4h_1 h_2}, \tag{14.7}$$

where the notation is defined in (14.6)*. The error is proportional to $h_1^2 h_2^2$.*

Numerical approximations of other mixed partial derivatives can be derived with the same technique as in example 14.11, see exercise 2.

A formula like (14.7) is often visualised with a drawing like the one in figure 14.3 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 14.4. 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 14.4.

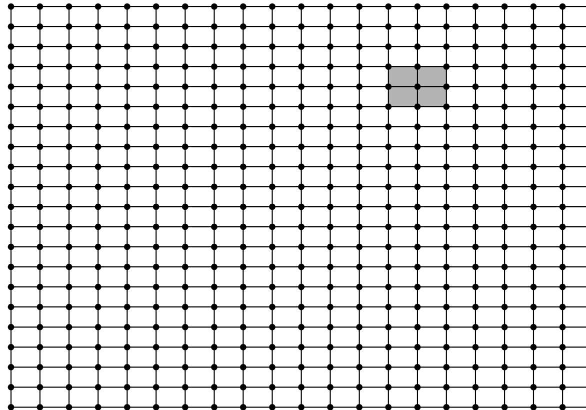


Figure 14.4. Numerical approximations to partial derivatives are often computed at all points of a grid like the one shown here by sliding around the grid a computational molecule like the one in figure 14.3.

Exercises for Section 14.2

1. Mark each of the following statements as true or false.

(a). All the methods we have for numerical integration in one dimension can be used to find partial derivatives along a particular axis for scalar functions of two variables.

2. 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 14.11 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 (14.7).

(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.

3. 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 2, but use the approximation $g'(a) = \big(g(a+h) - g(a)\big)/h$ at every stage.

CHAPTER 15

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.

15.1 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.

15.1.1 Light

Fact 15.1 (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.

15.1.2 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 15.2 (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).*

15.1.3 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 15.3. *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 15.4. *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.

15.1.4 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.

(a) (b) (c)

Figure 15.1. Different version of the same image; black and white (a), grey-level (b), and colour (c).

Fact 15.5 (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 15.2.3 below). Figure 15.1 shows an image in different formats.

Fact 15.6. *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*

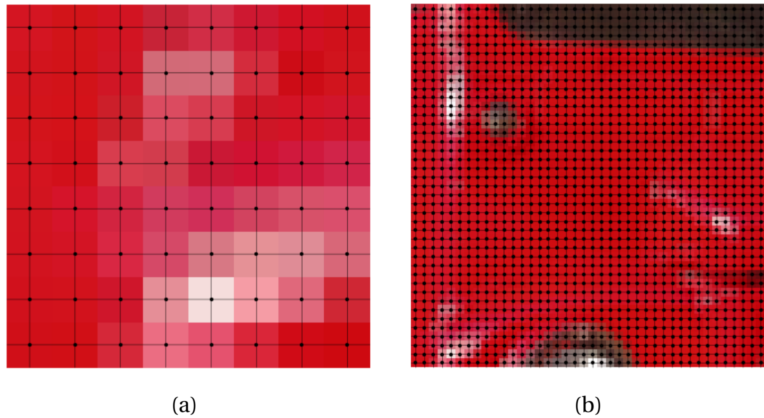


Figure 15.2. Two excerpt of the colour image in figure 15.1. The dots indicate the position of the points (i, j).

intensity values are assumed to be real numbers in $[0,1]$.

If we magnify a small part of the colour image in figure 15.1, we obtain the image in figure 15.2 (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 15.2.

Fact 15.7 (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).

15.1.5 Images as surfaces

Recall from chapter 14 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

$$\mathbb{Z}_{m,n} = \{(i, j) \mid 1 \leq i \leq m \text{ and } 1 \leq j \leq n\},$$

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 15.3.

Figure 15.3. The grey-level image in figure 15.1 plotted as a surface. The height above the (x, y)-plane is given by the intensity value.

Fact 15.8 (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*

$$F_P : [1/2, m + 1/2] \times [1/2, n + 1/2] \mapsto [0, 1]$$

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

$$P : \mathbb{Z}_{m,n} \mapsto \mathbb{R}^3.$$

If we compare with definition 14.9, 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 15.4

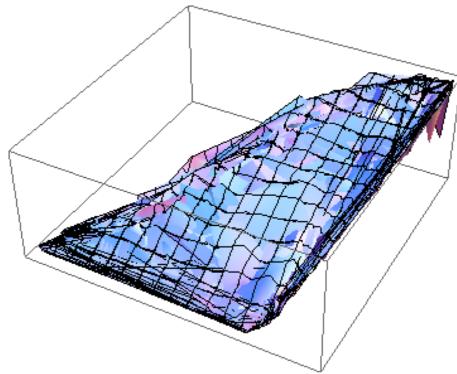


Figure 15.4. A colour image viewed as a parametric surface in space.

Exercises for Section 15.1

1. Find the correct alternative in the following multiple choice exercises.

(a). (Continuation exam 2009) A program generates digital video where every frame contains 800×600 points and there is 25 frames per second. For every second of video this gives

☐ 64 000 000 bytes

☐ 144 000 000 bytes

☐ 36 000 000 bytes

☐ 12 000 000 bytes

(b). ☐ The three base colors used in color images on computers are usually red, yellow and blue.

☐ An image of $2\,000\,000 \times 2\,000\,000$ pixels is said to be 2 Megapixels large.

☐ Electromagnetic radiation with wavelength 0.5 mm is in the range of visible light.

☐ The three base colours used in color images on computers are usually red, green and blue.

15.2 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.

15.2.1 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 7.24,

$$g(x) = \frac{x-a}{b-a}, \quad a < b,$$

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.

15.2.2 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 15.5.

15.2.3 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 15.6a.

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

(a) (b) (c)

Figure 15.5. The red (a), green (b), and blue (c) components of the colour image in figure 15.1.

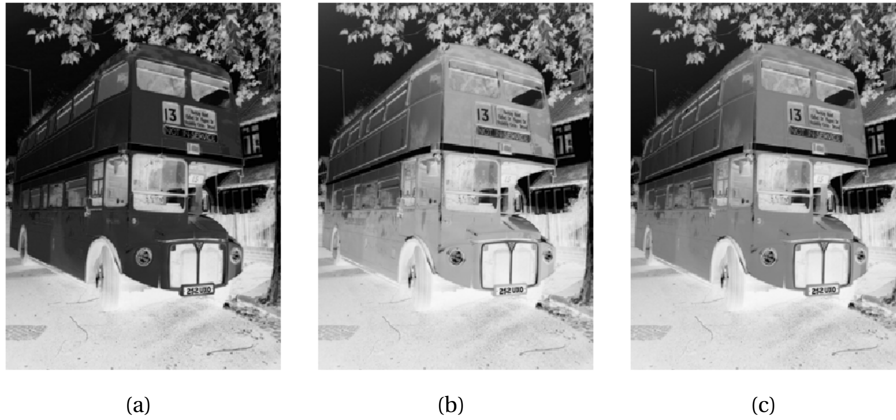
(a) (b) (c)

Figure 15.6. Alternative ways to convert the colour image in figure 15.1 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 $[0,1]$.

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 15.6b.

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 15.6c.

Let us sum this up as an algorithm.



(a) (b) (c)

Figure 15.7. The negative versions of the corresponding images in figure 15.6.

Algorithm 15.9 (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*

$$q_{i,j} = \frac{\hat{q}_{i,j}}{\max_{k,l} \hat{q}_{k,l}}.$$

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.

15.2.4 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 15.10 (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.*

15.2.5 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 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.

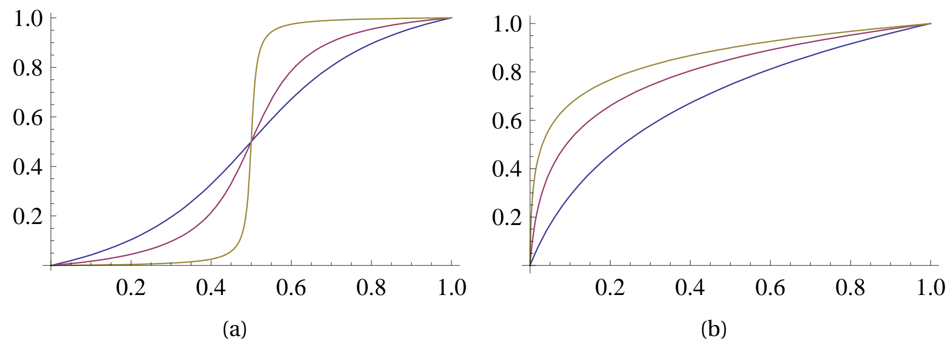
Figure 15.8 shows some examples. The functions in the left plot have quite large derivatives near $x = 0.5$ and will therefore increase the contrast in images with a concentration of intensities with value around 0.5. The functions are all on the form

$$f_n(x) = \frac{\arctan\big(n(x-1/2)\big)}{2\arctan(n/2)} + \frac{1}{2}. \tag{15.1}$$

For any $n \neq 0$ these functions satisfy the conditions $f_n(0) = 0$ and $f_n(1) = 1$. The three functions in figure 15.8a correspond to $n = 4$, 10, and 100.

Functions of the kind shown in figure 15.8b 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

$$g_\epsilon(x) = \frac{\ln(x+\epsilon) - \ln\epsilon}{\ln(1+\epsilon) - \ln\epsilon}, \tag{15.2}$$



(c)

(d)

Figure 15.8. 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 15.6c, while in (d) the middle function in (b) has been applied to the same image.

and the ones shown in the plot correspond to $\epsilon = 0.1$, 0.01, and 0.001.

In figure 15.8c the middle function in (a) has been applied to the image in figure 15.6c. Since the image was quite well balanced, this has made the dark areas too dark and the bright areas too bright. In figure 15.8d 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 15.11. *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.

15.2.6 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

$$\frac{1}{16}\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}. \tag{15.3}$$

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, 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 (15.3) is in fact an example of a computational molecule, see figure 14.3. 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

(a) (b) (c)

Figure 15.9. The images in (b) and (c) show the effect of smoothing the image in (a).

by the array

$$\frac{1}{1024}\begin{pmatrix} 1 & 6 & 15 & 20 & 15 & 6 & 1 \\ 6 & 36 & 90 & 120 & 90 & 36 & 6 \\ 15 & 90 & 225 & 300 & 225 & 90 & 15 \\ 20 & 120 & 300 & 400 & 300 & 120 & 20 \\ 15 & 90 & 225 & 300 & 225 & 90 & 15 \\ 6 & 36 & 90 & 120 & 90 & 36 & 6 \\ 1 & 6 & 15 & 20 & 15 & 6 & 1 \end{pmatrix}. \tag{15.4}$$

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 $1/4096$ 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 (15.3) and (15.4) to an image is shown in figure 15.9 (b) and (c) respectively. The smoothing effect is clearly visible.

Observation 15.12. *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.*

15.2.7 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 14.2 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

$$\frac{\partial P}{\partial x}(i,j) = \frac{p_{i+1,j} - p_{i-1,j}}{2}, \tag{15.5}$$

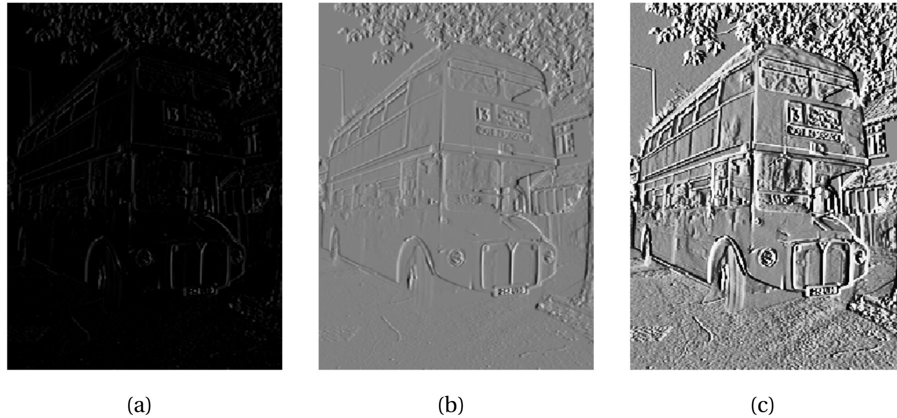
where we have used the convention $h = 1$ which means that the derivative is measured in terms of 'intensity per pixel'. We can run through all the pixels in the image and compute this partial derivative, but have to be careful for $i = 1$ and $i = m$ where the formula refers to non-existing pixels. We will adapt the simple convention of assuming that all pixels outside the image have intensity 0. The result is shown in figure 15.10a.

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 14.6 to each intensity value. The result is shown in figure 15.10c 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 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 15.10a corresponds to no change in intensity between the two pixels.

When the values are mapped to the interval $[0,1]$ in figure 15.10b, 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 15.10c these values have just been emphasised even more.

Figure 15.10c 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



(a) (b) (c)

Figure 15.10. The image in (a) shows the partial derivative in the x-direction for the image in 15.6. 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 15.1.

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 15.13. *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 15.14. *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 15.12b. The intensities have been normalised and the contrast enhanced by the function f_{50} in (15.1).

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

$$|\nabla P| = \sqrt{\left(\frac{\partial P}{\partial x}\right)^2 + \left(\frac{\partial P}{\partial y}\right)^2}.$$

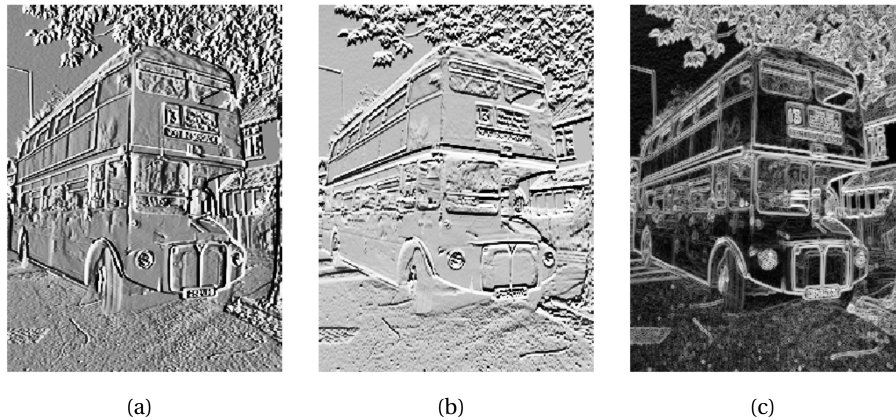
When the two first derivatives have been computed it is a simple matter to compute the gradient vector and its length; the resulting is shown as an image in figure 15.11c.

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 15.11a 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 15.11b.

(a) (b) (c)

Figure 15.11. 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.



(a) (b) (c)

Figure 15.12. 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.

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 15.8b to the intensities. If we use the function $g_{0.01}$ defined in equation(15.2) we obtain the image in figure 15.11c.

15.2.8 Comparing the first derivatives

Figure 15.12 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 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.

15.2.9 Second-order derivatives

To compute the three second order derivatives we apply the corresponding computational molecules which we described in section 14.2.

Observation 15.15 (Second order derivatives of an image). *The second order derivatives of an image P can be computed by applying the computational molecules*

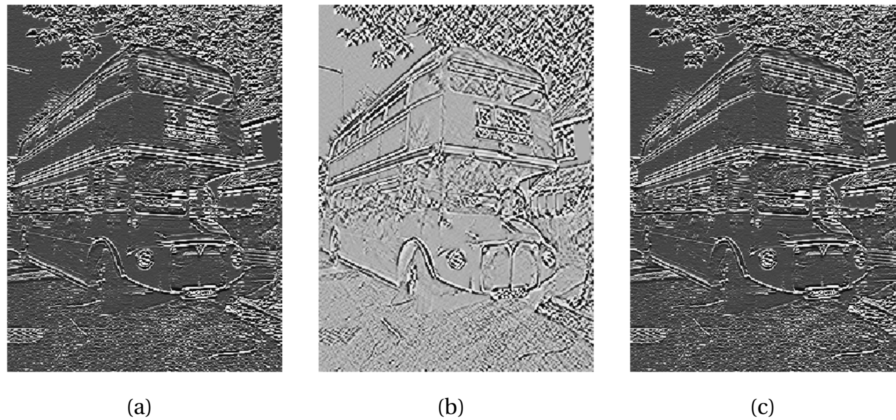
$$\frac{\partial^2 P}{\partial x^2}: \quad \begin{pmatrix} 0 & 0 & 0 \\ -1 & 2 & -1 \\ 0 & 0 & 0 \end{pmatrix},$$

$$\frac{\partial^2 P}{\partial y \partial x}: \quad \frac{1}{4}\begin{pmatrix} -1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & -1 \end{pmatrix},$$

$$\frac{\partial^2 P}{\partial y^2}: \quad \begin{pmatrix} 0 & 1 & 0 \\ 0 & 2 & 0 \\ 0 & -1 & 0 \end{pmatrix}.$$

With the information in observation 15.15 it is quite easy to compute the second-order derivatives, and the results are shown in figure 15.13. The computed derivatives were first normalised and then the contrast enhanced with the function f_{100} in each image, see equation 15.1.

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



(a) (b) (c)

Figure 15.13. 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.

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.

Exercises for Section 15.2

1. Mark each of the following statements as true or false.

(a). A computational molecule must always be symmetric around the center point.

(b). Sharp edges in an image correspond to large values of the second derivative along a line, i.e. large values of

$$p_{i+1,j} - 2p_{i,j} + p_{i+1,j},$$

which corresponds to the numerical expression for the second derivative found in Chapter 11.

15.3 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.

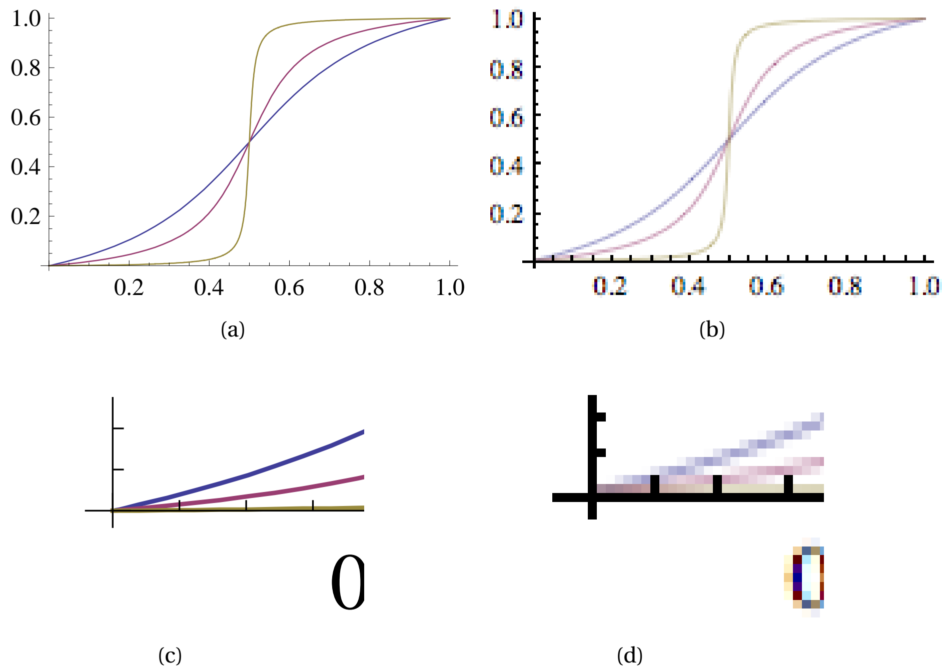


Figure 15.14. The difference between vector graphics ((a) and (c)) and raster graphics ((b) and (d)).

15.3.1 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 15.2.

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 15.14. 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

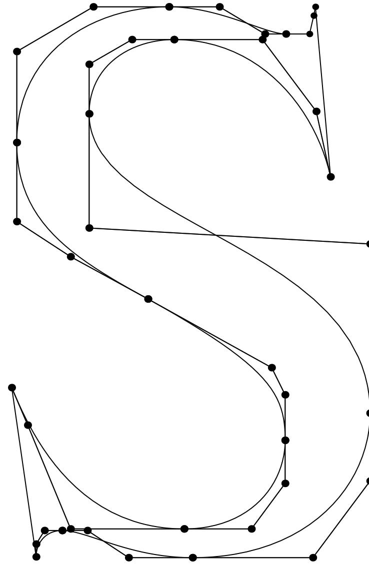


Figure 15.15. The character 'S' in the font Times Roman. The dots are parameters that control the shape of the curves.

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 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 15.15 for an example.

Fact 15.16. *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.

15.3.2 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`.

15.3.3 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`.

Exercises for Section 15.3

1. Mark each of the following statements as true or false.

(a). Vector graphics scales better than raster graphics when you zoom in closely on it.

APPENDIX

Answers

Section 1.5

Exercise 1(a).

$s1 := 0$; $s2 := 0$;
for $k := 1, 2, \dots, n$
  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.

Exercise 1(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 $k := 1, 2, \dots, n$
  if $a_k > 0$
    $s1 := s1 + a_k$;
    $pos := pos + 1$;
  else
    $s2 := s2 + a_k$;

$neg := neg + 1;$
$s2 := -s2;$

Exercise 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

```
if x1 + y1 < 10
    z1 := x1 + y1;
else
    x2 := x2 + 1;
    z1 := x1 + y1 − 10;
if x2 + y2 < 10
    z2 := x2 + y2;
else
    x3 := x3 + 1;
    z2 := x2 + y2 − 10;
if x3 + y3 < 10
    z3 := x3 + y3;
else
    z4 := 1;
    z3 := x3 + y3 − 10;
```

Exercise 3. We use the same representation as in the solution for exercise 3. Multiplication of two three-digit numbers x and y can then be performed by the formulas

$$product1 := x_1 * y1 + 10 * x_1 * y_2 + 100 * x_1 * y_3;$$
$$product2 := 10 * x_2 * y1 + 100 * x_2 * y_2 + 1000 * x_2 * y_3;$$
$$product3 := 100 * x_3 * y1 + 1000 * x_3 * y_2 + 10000 * x_3 * y_3;$$
$$product := product1 + product2 + product3;$$

Section 2.3

Exercise 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

Exercise 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

Section 3.1

Exercise 1 (a). False

Exercise 1 (b). True

Exercise 1 (c). False

Exercise 1 (d). True

Section 3.2

Exercise 1 (a). True

Exercise 1 (b). False

Exercise 1 (c). True.

Exercise 2(a). 220_4

Exercise 2(b). 32_5

Exercise 2(c). 10001_2

Exercise 2(d). 1022634_7

Exercise 2(e). 123456_7

Exercise 2(f). $7e_{16}$

Exercise 3(a). 131_8

Exercise 3(b). 67_8

Exercise 3(c). 252_8

Exercise 4(a). 100100_2

Exercise 4(b). 1000000_2

Exercise 4(c). 11010111_2

Exercise 5(a). $4\mathrm{d}_{16}$

Exercise 5(b). c

Exercise 5(c). 29e4

Exercise 5(d). 0.594

Exercise 5(e). 0.052

Exercise 5(f). 0.ff8

Exercise 6(a). 111100

Exercise 6(b). 100000000

Exercise 6(c). 111001010001

Exercise 6(d). 0.000010101010

Exercise 6(e). 0.000000000001

Exercise 6(f). 0.111100000001

Exercise 7(a). $7 = 10_7$, $37 = 10_{37}$ and $4 = 10_4$

Exercise 7(b). $\beta = 13, \beta = 100$

Exercise 8(a). $400 = 100_{20}$, $4 = 100_2$ and $278 = 100_{17}$

Exercise 8(b). $\beta = 5, \beta = 29$

Section 3.3

Exercise 1(a). True

Exercise 1(b). False

Exercise 1(c). False

Exercise 1(d). True

Exercise 2. $\beta = 6$

Exercise 3(a). 0.01_2

Exercise 3(b). $0.102120102120102120\ldots$

Exercise 3(c). 0.01_3

Exercise 3(d). $0.001111111\ldots$

Exercise 3(e). 0.7_8

Exercise 3(f). $0.6060606\ldots$

Exercise 3(g). $0.e_{16}$

Exercise 3(h). 0.24_8

Exercise 3(i). 0.343_6

Exercise 4. $\pi_9 \approx 3.1241_9$

Exercise 6. $c - 1$

Section 3.4

Exercise 1(a). The third alternative is correct

Exercise 1(b). The third alternative is correct

Exercise 1(c). 50.1_8

Exercise 2(a). 4_7

Exercise 2(b). 13_6

Exercise 2(c). 10001_2

Exercise 2(d). 1100_3

Exercise 2(e). 103_5

Exercise 2(f). $4_5 = 4_7$

Exercise 3(a). 3_8

Exercise 3(b). 11_2

Exercise 3(c). 174_8

Exercise 3(d). 112_3

Exercise 3(e). 24_5

Exercise 4(a). 1100_2

Exercise 4(b). 10010_2

Exercise 4(c). 1210_3

Exercise 4(d). 141_5

Exercise 4(e). 13620_8

Exercise 4(f). 10220_3

Exercise 4(g). 1111_2

Section 4.1

Exercise 1(a). The third alternative is correct.

Exercise 1(b). The third alternative is correct.

Exercise 1(c). The first alternative is correct.

Exercise 2(a). 0

Exercise 2(b). -8

Exercise 2(c). 7

Section 4.2

Exercise 1(a). False

Exercise 1(b). False

Exercise 1(c). True

Exercise 2. The third alternative is correct.

Exercise 3. Largest integer: $7fffffff_{16}$.
Smallest integer: 80000000_{16}.

Exercise 5(a). 0.4752735×10^7

Exercise 5(b). $0.602214179 \times 10^{24}$

Exercise 5(c). 0.8617343×10^{-4}.

Exercise 6. $0.1001\ 1100\ 1111\ 0101\ 1010\ldots \times 2^4$

Section 4.3

Exercise 1(a). True

Exercise 1(b). False

Exercise 1(c). False

Exercise 1(d). True

Exercise 2(a). The second alternative is correct

Exercise 2(b). The second alternative is correct

Exercise 2(c). The fourth alternative is correct

Exercise 3(a). $0101\ 1010_2 = 5a_{16}$

Exercise 3(b). $1100\ 0011\ 1011\ 0101_2 = c3b5_{16}$

Exercise 3(c). $1100\ 1111\ 1011\ 1000_2 = cfb8_{16}$

Exercise 3(d). $1110\ 1000\ 1011\ 1100\ 1011\ 0111_2 = e8bcb7_{16}$

Exercise 4. 0000 0000 0101 $1010_2 = 005a_{16}$

0000 00001111 $0101_2 = 00f5_{16}$

0000 0011 1111 $1000_2 = 03f8_{16}$

1000 1111 0011 $0111_2 = 8f37_{16}$

Exercise 5(a). æ, ø, and å.

Exercise 5(b). Nothing or error message; these codes are not valid UTF-8 codes.

Exercise 5(c). When stored as UTF-16 and read as ISO Latin1: NULæ, NULø, and NULå, where NUL is some representation of the nonprintable ASCII character with code 00. The other way does not give legal UTF-16 codes.

Exercise 5(d). Since the UTF-8 encodings here are valid two-byte Unicode characters, we just have to look up the Unicode character with code point equal to the UTF-8 encoding. This yields the following Hangul symbols:

æ (UTF8-encoding $c3a6_{16}$):쎦,

ø (UTF8-encoding $c3b8_{16}$): 쎸,

å (UTF8-encoding $c3a5_{16}$): 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.

Exercise 8. Enocded with UTF-8.

Exercise 10. Byte 6 is wrong. It should start with 10, since byte 5 starts with 110 and thus signals that the next byte should start with 10. The first four bytes represent three valid UTF-8 encoded characters.

Section 4.5

Section 5.2

Exercise 1(a). False

Exercise 1(b). False

Exercise 1(c). True

Exercise 2. The last expression.

Exercise 3(a). 1

Exercise 3(b). 90

Exercise 3(c). 100

Exercise 3(d). 10.5

Exercise 3(e). 10.4

Exercise 3(f). 4530

Exercise 4. A point at equal distance between two neighbours (such as 4.5 between 4 and 5) should be rounded up or down with equal probability.

Exercise 6(a). 0.1647×10^2

Exercise 6(b). 0.1228×10^2

Exercise 6(c). 0.4100×10^{-1}

Exercise 6(d). 0.6000×10^{-1}

Exercise 6(e). -0.5000×10^{-2}

Exercise 7(a). Normalised number in base β: A nonzero number a is written as

$$a = \alpha \times \beta^n$$

where $\beta^{-1} \le |\alpha| < 1$.

Exercise 8. One possible program:

$n := 1$;
while $1.0 + 2^{-n} > 1.0$
    $n := n + 1$;
print n;

Exercise 9. In a program the last value 2.0 is not written.

Exercise 10. The two commutative laws hold for floating point numbers. The other laws do not.

Section 5.3

Exercise 1(a). False

Exercise 1(b). True, unless $a = 0$, because then the relative error is not defined.

Exercise 1(c). False

Exercise 1(d). True

Exercise 2(a). $r = 0.0006$

Exercise 2(b). $r \approx 0.0183$

Exercise 2(c). $r \approx 2.7 \times 10^{-4}$

Exercise 2(d). $r \approx 0.94$

Exercise 3. (a). 0.0006
(b). 0.0187
(c). 0.000272
(d). 16.7
(a), (b), (c) agree with the last sentence in section 5.3.3, while (d) does not, since the relative errors are quite big.

Exercise 4. The relative errors are in examples 5.9–5.12 are $2.97 \times 10^{-4} \approx 10^{-4}$, $0.78 \times 10^{-4} \approx 10^{-4}$, 0, and $0.5 \times 10^{-1} \approx 10^{-1}$, respectively. These values are compatible with observation 5.20.

Exercise 6(b). 10×2^{-20}.

Exercise 6(c). ≈ 0.003433.

Section 5.4

Exercise 1(a). 3/2

Exercise 1(b). The last alternative is the correct one.

Exercise 2(a). Problematic for large x. Replace with $\frac{1}{\sqrt{x+1}+\sqrt{x}}$.

Exercise 2(b). Problematic for large x. Can be rewritten to $\ln\left(\frac{x}{x+1}\right)$.

Exercise 2(c). Problematic near near $x = \frac{\pi}{4}$. We can rewrite as $\cos(2x)$.

Section 6.1

Exercise 1. In simpler English the riddle says: Diophantus' youth lasted 1/6 of his life. He had the first beard in the next 1/12 of his life. At the end of the following 1/7 of his life Diophantus got married. Five years later his son was born. His son lived exactly 1/2 of Diophantus' life. Diophantus died 4 years after the death of his son. Solution: If d and s are the ages of Diophantus and his son when they died, then the epitaph corresponds to the two equations

$$d = (1/6 + 1/12 + 1/7)d + 5 + s + 4,$$
$$s = 1/2d.$$

If we solve these we obtain $s = 42$ years and $d = 84$ years.

Section 6.2

Exercise 1(a). The third alternative is correct

Exercise 1(b). The first alternative is correct.

Exercise 2(a). $x_2 = 1, x_3 = 2, x_4 = 5, x_5 = 13$

Exercise 2(b). $x_2 = 17, x_3 = 32, x_4 = 83, x_5 = 179$

Exercise 2(c). $x_2 = 4, x_3 = 16, x_4 = 128, x_5 = 4096$

Exercise 2(d). $x_2 \approx -2.4495$, $x_3 - 2.5396$, $x_4 \approx -2.5573$, $x_5 \approx -2.5607$

Exercise 2(e). $x_2 = \frac{3}{5}$, $x_3 = \frac{9}{25}$, $x_4 = \frac{62}{125}$, $x_5 = \frac{816}{625}$.

Exercise 2(f). $x_2, \ldots, x_5$ are not well-defined.

Exercise 3(a). Linear.

Exercise 3(b). Nonlinear.

Exercise 3(c). Nonlinear.

Exercise 3(d). Linear.

Section 6.3

Section 6.4

Exercise 2(a). $x_n = 3^n \cdot \frac{5}{3}$

Exercise 2(c). $x_n = (1 - 2n)(-1)^n$

Exercise 2(d). $x_n = \frac{3}{4} \cdot 3^n + \frac{5}{4}(8 - 1)^n$

Section 6.5

Exercise 1(a). False

Exercise 1(b). False

Exercise 2(a). The last alternative is correct.

Exercise 2(b). The last alternative is correct.

Exercise 2(c). The first alternative is correct.

Exercise 3(a). $x_n = 3 - 3^{-n}$.

Exercise 3(b). $x_n = 1/7$.

Exercise 3(c). $x_n = (2/3)^n$.

Exercise 4(b). We will eventually get overflow.

Exercise 6(a). Solution determined by the initial conditions: $x_n = 15^{-n}$.

Exercise 6(c). $n \approx 24$.

Exercise 7(a). Solution determined by the initial conditions: $x_n = 2^{-n}$.

Exercise 7(b). Eventually we will get overflow.

Section 7.1

Section 7.2

Exercise 1(a). False

Exercise 1(b). True

Exercise 1(c). False

Exercise 1(d). True

Exercise 2. The third alternative is correct.

Exercise 4(a). Use ternary trees instead of binary ones. (Each tree has either zero or three subtrees/children).

Exercise 4(b). Use n-nary trees. (Each tree has either zero or n subtrees/children)

Exercise 6. Frequencies used are all 1.

Section 7.3

Exercise 1(a). The statement is false.

Exercise 1(b). The statement is true

Exercise 1(c). The statement is false

Exercise 1(d). The statement is false.

Exercise 2. The third alternative is correct.

Exercise 3. $\log_2 x = \ln x / \ln 2$.

Exercise 4(a). ≈ 2.7534

Exercise 4(b). ≈ 2.2709

Exercise 4(c). ≈ 2.5306

Section 7.4

Exercise 1(a). True

Exercise 1(b). True, but modifications in how arithmetic coding is done can overcome this drawback.

Exercise 1(c). True

Exercise 2(a).

$$f(A) = 9, \qquad p(A) = 0.1,$$
$$f(B) = 1, \qquad p(B) = 0.9,$$

Exercise 2(b). 6 bits

Exercise 2(c). 011100

Exercise 3(a). $H = 2$

Exercise 3(b). 2 bits per symbol

Exercise 3(c). 2m + 1 bits $\frac{2m+1}{m} \approx 2$ bits per symbol

Exercise 3(d).

00 10 11 01 00 10

Exercise 3(e).

00 10 11 01 00 10 1

Exercise 4.

BCBBCBBBCB

Exercise 5.

01 01 11 10 00

Exercise 6.

$$f(x) = c + (y - a)\frac{d - c}{b - a}$$

Section 7.6

Section 8.1

Section 8.2

Section 9.1

Exercise 2(a). The second alternative is correct.

Exercise 2(b). The fourth alternative is correct.

Exercise 2(c). The fourth alternative is correct.

Exercise 3(a).

$$b_0 = f(a) - f'(a)a + \frac{f''(a)}{2}a^2, \qquad b_1 = -f''(a)a + f'(a), \qquad b_2 = f''(a)/2.$$

Exercise 3(b).

$$b_0 = f(a), \qquad b_1 = f'(a), \qquad b_2 = f''(a)/2$$

Exercise 4(a). $T_2(x;1) = 1 - 3x + 3x^2$.

Exercise 4(b). $T_2(x;0) = 12x^2 + 3x + 1$.

Exercise 4(c). $T_2(x;0) = 1 + x\ln 2 + (\ln 2)^2 x^2/2$.

Section 9.2

Exercise 1(a). True

Exercise 1(b). False

Exercise 1(c). False

Exercise 2. The first alternative is correct.

Exercise 3(a).

$$p_3(x) = -\frac{(x-1)(x-3)(x-4)}{12} - \frac{x(x-1)(x-4)}{3} + \frac{x(x-1)(x-3)}{12}.$$

Exercise 3(b).

$$p_3(x) = 1 - x + \frac{2}{3}x(x-1) - \frac{1}{3}x(x-1)(x-3).$$

Exercise 5(a). The Newton form is

$$p_2(x) = 2 - x.$$

Exercise 6(a). Linear interpolant p_1:

$$p_1(x) = y_1 + (y_2 - y_1)(x-1).$$

Error at x:

$$f[1,2,x](x-1)(x-2) = \frac{f''(\xi)}{2}(x-1)(x-2)$$

where ξ is a number in the smallest interval (a,b) that contains all of 1, 2, and x.

Error at $x = 3/2$:

$$\frac{f''(\xi_1)}{8}$$

where ξ is a number in the interval $(1,2)$.

Exercise 6(b). Cubic interpolant:

$$p_3(x) = y_0 + (y_1 - y_0)x + \frac{y_2 - 2y_1 + y_0}{2}x(x-1) + \frac{y_3 - 3y_2 + 3y_1 - y_0}{6}x(x-1)(x-2).$$

Error:

$$f[0,1,2,3,x]x(x-1)(x-2)(x-3) = \frac{f^{(iv)}(\xi)}{4!}x(x-1)(x-2)(x-3)$$

where ξ is now a number in the smallest open interval that contains all of 0, 1, 2, 3, and x. With $x = 3/2$ this becomes

$$\frac{3}{128}f^{(iv)}(\xi_3)$$

where ξ_3 is a number in the interval $(0,3)$.

Section 10.2

Exercise 1(a). True

Exercise 1(b). False

Exercise 1(c). True

Exercise 1(d). False

Exercise 1(e). True

Exercise 2(a). The last alternative is correct.

Exercise 2(b). The third alternative is correct.

Exercise 2. 25

Exercise 3(a). Approximation after 10 steps: 0.73876953125.

Exercise 3(b). To get 10 correct digits it is common to demand that the relative error is smaller than 5×10^{-11}, even though this does not always ensure that we have 10 correct digits. A challenge with the relative error is that it requires us to know the exact zero. In our case we have a very good approximation that we could use, but as we commented when we discussed properties of the relative error, it is sufficient to use a rough estimate, like 0.7 in this case. The required inequality is therefore

$$\frac{1}{2^N 0.7} \leq 5 \times 10^{-11}.$$

This inequality can be easily solved and leads to $N \geq 35$.

Exercise 3(c). Actual error: 1.3×10^{-11}

Exercise 5. The root 4π is selected

Exercise 6(a). The relative errors using the approximation (10.4) are
0.0909090909091, 0.0434782608696, 0.0222222222222, 0.010989010989, 0.00552486187845, 0.00275482093664, 0.00137931034483, 0.000690131124914, 0.0003451846738, 0.000172622130157.

Exercise 6(b). The relative errors are
0.0277281758685, 0.0164659979557, 0.00563108895642, 0.00541745449962, 0.000106817228398, 0.00265531863561, 0.00127425070361, 0.000583716737605, 0.000238449754604, 6.58162631029e-05.
These are seen to be quite close to the values from (a).

Section 10.3

Exercise 1(a). False

Exercise 1(b). True

Exercise 2. The last alternative is correct.

Exercise 3(a). $f(x) = x^2 - 3$. After one iteration we obtain the approximation 1.6666666666666667 which has two correct digits ($\sqrt{3} \approx 1.7320508075688772935$ with 20 correct digits). After 6 iterations we end up with the approximation 1.732050807568877.

Exercise 3(b). $f(x) = x^{12} - 2$.

Exercise 3(c). $f(x) = \ln x - 1$.

Section 10.4

Exercise 1(a). True

Exercise 1(b). False

Exercise 2(a). The last alternative is correct.

Exercise 2(b). The fourth alternative is correct.

Exercise 2(c). The second alternative is correct.

Exercise 2(d). The last alternative is correct.

Exercise 3. If you do the computations with 64-bit floating-point numbers, you have full machine accuracy after just 4 iterations. If you do 7 iterations you actually have about 164 correct digits.

Exercise 4(a). Midpoint after 10 iterations: 3.1416015625.

Exercise 4(b). Approximation after 4 iterations: 3.14159265358979.

Exercise 4(c). Approximation after 4 iterations: 3.14159265358979.

Exercise 5. Newton's method and the secant method behave differently, since the second derivative is not bounded away from zero close to the zero.

Exercise 6(b). $e_{n+1} = e_{n-1}e_n/(x_{n-1}+x_n)$, where $e_n = \sqrt{2} - x_n$.

Exercise 7(b). After 5 iterations we have the approximation 0.142857142857143 in which all the digits are correct (the fourth approximation has approximate error 6×10^{-10}). The code can look as follows:

```
N=30
epsilon=10**(-10)
i=0
xp=z=0.1
R=7.0
abserr=abs(z)
while i <= N and abserr >= epsilon*abs(z):
  z=xp*(2.0-R*xp)
  print(i,z)
  abserr=abs(z-xp)
  xp=z
  i=i+1
```

Exercise 8(c). An example where $x_n > c$ for $n > 0$ is $f(x) = x^2 - 2$ with $c = \sqrt{2}$ (choose for example $x_0 = 1$). If we use the same equation, but choose $x_0 = -1$, we converge to $-\sqrt{2}$ and have $x_n < c$ for large n (in fact $n > 0$).

An example where the iterations jump around is in computing an approximation to a zero of $f(x) = \sin x$, for example with $x_0 = 4$ (convergence to $c = \pi$).

Section 11.1

Section 11.2

Exercise 1(a). False. When h gets small enough, roundoff errors dominate the mathematical error.

Exercise 1(b). True

Exercise 1(c). True

Exercise 1(d). True

Exercise 1(e). False

Exercise 2(a). The last alternative is correct.

Exercise 2(b). The last alternative is correct.

Exercise 3(a). 10^{-8} is the power of 10 which gives the least error in the approximation.

Exercise 3(b). $h^* \approx 1.6733 \times 10^{-8}$.

Exercise 4(a). An error estimate is now

$$\frac{h}{2}|f''(a)| + \frac{h^2}{6} \max_{x\in[a,a+h]} |f'''(x)|.$$

Exercise 4(b). It is not possible to obtain an estimate of the truncation error using this kind of Taylor expansion.

Exercise 4(c). The linear Taylor polynomial is the best because it is is the shortest possible Taylor expansion which can give an estimate of $f'(a)$ (as we showed in (b)), and also the one which gives the simplest expression for the truncation error in that it does not depend on any derivatives higher than the second order (as we showed in (a)).

Section 11.3

Exercise 1. $f'(a) \approx p_2'(a) = -(f(a+2h) - 4f(a+h) + 3f(a))/(2h)$.

Section 11.4

Exercise 2(b). $h^* \approx 5.9 \times 10^{-6}$.

Exercise 3(b). With 6 digits:
$(f(a+h) - f(a))/h = 0.455902$, relative error: 0.088196.
$(f(a) - f(a-h))/h = 0.542432$, relative error: 0.084864.
$(f(a+h) - f(a-h))/(2h) = 0.499167$, relative error: 0.001666.

Exercise 4(c). With 6 digits:
$(f(a+h) - f(a))/h = 0.975$, relative error: 0.025.
$(f(a) - f(a-h))/h = 1.025$, relative error: 0.025.
$(f(a+h) - f(a-h))/(2h) = 1$, relative error: 8.88178×10^{-16}.

Exercise 5(a). Optimal h: 2.9×10^{-6}.

Exercise 5(b). Optimal h: 3.3×10^{-6}

Exercise 7(b). Optimal h: 2.24×10^{-4}.

Exercise 8(a). $c_1 = -1/(2h)$, $c_2 = 1/(2h)$.

Exercise 8(c). $c_1 = -1/h^2$, $c_2 = 2/h^2$, $c_3 = -1/h^2$.

Exercise 11(b). Opitmal h: 9.9×10^{-4}.

Section 12.1

Exercise 2(a). $\underline{I} \approx 1.63378$, $\overline{I} \approx 1.805628$.

Exercise 2(b). $\left|I - \underline{I}\right| \approx 0.085$, $\frac{|I-\underline{I}|}{|I|} = 0.0491781$.
$\left|I - \overline{I}\right| \approx 0.087$, $\frac{\left|I-\overline{I}\right|}{|I|} = 0.051$.

Section 12.2

Exercise 1(a). False

Exercise 1(b). True

Exercise 1(c). False

Exercise 1(d). True

Exercise 1(e). True

Exercise 2. 5/16.

Exercise 3. Approximation: 0.530624 (with 6 digits).

Exercise 4(a). Approximation with 10 subintervals: 1.71757 (with 6 digits)

Exercise 4(b). $h \le 2.97 \times 10^{-5}$.

Exercise 5. Approximation with 10 subintervals: 5.36648 (with 6 digits). $h \le 4.89 \times 10^{-5}$.

Section 12.3

Exercise 2. 3/8

Exercise 3. Approximation: 0.519725 (with 6 digits).

Exercise 4(a). Approximation with 10 subintervals: 1.71971 (with 6 digits).

Exercise 4(b). $h \le 2.1 \times 10^{-5}$.

Exercise 8. Approximation: 0.527217 (with 6 digits).

Exercise 9(a). 81 650 evaluations.

Exercise 9(b). 57 736 evaluations.

Exercise 9(c). 383 evaluations.

Exercise 10(a). Approximation with 10 subintervals: 1.718282782 (with 10 digits).

Exercise 10(b). $h \leq 1.8 \times 10^{-2}$.

Exercise 12. $w_1 = w_3 = (b-a)/6$, $w_2 = 2(b-a)/3$.

Section 13.1

Exercise 3(a). Linear.

Exercise 3(b). Nonlinear.

Exercise 3(c). Nonlinear.

Exercise 3(d). Nonlinear.

Exercise 3(e). Linear.

Section 13.2

Exercise 2. $x(t) = 1 - De^{\cos t}$, where D can be chosen freely.

Exercise 2(a). $x(t) = 1 - e^{\cos t}$.

Exercise 2(b). $x(t) = 1$.

Exercise 2(c). $x(t) = 1 + e^{\cos t}$.

Exercise 2(d). $x(t) = 1 + 2e^{\cos t}$

Exercise 3(a). $x(t) = 1$ will cause problems.

Exercise 3(b). The differential equation is not defined for $t = 1$.

Exercise 3(c). The equation is not defined when $x(t)$ is negative.

Exercise 3(d). The equation does not hold if $x'(t) = 0$ or $x(t) = 0$ for some t.

Exercise 3(e). The equation is not defined for $|x(t)| > 1$.

Exercise 3(f). The equation is not defined for $|x(t)| > 1$.

Section 13.3

Exercise 3(a). $x(0.3) \approx 1.362$.

Exercise 3(b). $x(0.3) \approx 0.297517$.

Exercise 3(c). $x(0.3) \approx 1.01495$.

Exercise 3(d). $x(1.3) \approx 1.27488$.

Exercise 3(e). $x(0.3) \approx 0.297489$.

Exercise 8.

$$|R_1(h)| \leq \frac{h^2}{4}.$$

Exercise 9(a). $x''(t) = 2t + (3x^2 - 1)x'(t)$.

Exercise 9(b). Quadratic Taylor with 1 step: $x(1) \approx 1$.
Quadratic Taylor with 2 steps: $x(1) \approx 1.3125$.
Quadratic Taylor with 5 steps: $x(1) \approx 1.62941067817$.

Exercise 9(c). Quadratic Taylor with 10 steps: $x(1) \approx 1.787456775$.
Quadratic Taylor with 100 steps: $x(1) \approx 1.90739098078$.
Quadratic Taylor with 1000 steps: $x(1) \approx 1.9095983769$.

Section 13.4

Exercise 2(a). $x(1) \approx 2$.

Exercise 2(b). $x(1) \approx 2.5$.

Exercise 2(c). $x(1) \approx 2.70833$.

Exercise 2(d). $x(1) \approx 2.71735$.

Exercise 2(e). If we use the fourth order Runge Kutta method we get the approximation 2.71827974414. If we change N to $100, 1000, 10000$ we get 2.71828182823, 2.71828182846, and 2.71828182846.

Exercise 2(f). It looks like the values converge to e.

Exercise 2(g). The equation is of first order with linear coefficients, and we see that the solution is $x(t) = e^t$, so that $x(1) = e$.

Exercise 3(a). Approximation at $t = 2\pi$:
Euler's method with 1 step: $x(2\pi) \approx 4.71828$.
Euler's method with 2 steps: $x(2\pi) \approx 4.71828$.
Euler's method with 5 steps: $x(2\pi) \approx 0.276243$.
Euler's method with 10 steps: $x(2\pi) \approx 2.14625$.

Exercise 3(b). Approximation at $t = 2\pi$:
Euler's midpoint method with 1 step: $x(2\pi) \approx 4.71828$.
Euler's midpoint method with 5 steps: $x(2\pi) \approx 3.89923$.

Section 13.5

Exercise 1. The third alternative is correct.

Section 13.6

Exercise 1(a). We set $x_1 = y$, $x_2 = y'$, $x_3 = x$, and $x_4 = x'$. This gives the system

$$\begin{aligned} x_1' &= x_2, \\ x_2' &= x_1^2 - x_3 + e^t, \\ x_3' &= x_4, \\ x_4' &= x_1 - x_3^2 - e^t. \end{aligned}$$

Exercise 1(b). We set $x_1 = x$, $x_2 = x'$, $x_3 = y$, and $x_4 = y'$. This gives the system

$$\begin{aligned} x_1' &= x_2, \\ x_2' &= 2x_3 - 4t^2 x_1, \\ x_3' &= x_4, \\ x_4' &= -2x_1 - 2tx_2. \end{aligned}$$

Exercise 1(c). We set $x_1 = x$, $x_2 = x'$, $x_3 = y$, and $x_4 = y'$. This gives the system

$$\begin{aligned} x_1' &= x_2 \\ x_2' &= -x_3 x_1 + (x_4)^2 x_1 \\ x_3' &= x_4 \\ x_4' &= -x_3. \end{aligned}$$

Exercise 1(d). We set $x_1 = x$, $x_2 = x'$, $x_3 = x''$, $x_4 = y$, and $x_5 = y'$. This gives the

system

$$\begin{aligned}
x_1' &= x_2\\
x_2' &= x_3\\
x_3' &= (t + x_2)(x_1)^2 - 3(x_5)^2 x_1\\
x_4' &= x_5\\
x_5' &= t + x_2.
\end{aligned}$$

Exercise 2. We define $x_1 = x$, $x_2 = x'$, $y_1 = y$, $y_2 = y'$, $y_3 = y''$, and get the following equations:

$$\begin{aligned}
x_1' &= x_2\\
x_2' &= t + x_1 + y_2\\
y_1' &= y_2\\
y_2' &= y_3\\
y_3' &= 1 + x_2 + 2y_3.
\end{aligned}$$

Exercise 3(a). With $x_1 = x$ and $x_2 = x'$ we obtain

$$\begin{aligned}
x_1' &= x_2,\\
x_2' &= (-3x_1 - t^2 x_2).
\end{aligned}$$

Exercise 3(b). With $x_1 = x$ and $x_2 = x'$ we obtain

$$\begin{aligned}
x_1' &= x_2,\\
x_2' &= (-k_s x_1 - k_d x_2)/m.
\end{aligned}$$

Exercise 3(c). With $x_1 = y$ and $x_2 = y'$ we obtain

$$\begin{aligned}
x_1' &= x_2\\
x_2' &= 2(e^{2t} - (x_1)^2)^{1/2}.
\end{aligned}$$

Exercise 3(d). With $x_1 = x$ and $x_2 = x'$ we obtain

$$\begin{aligned}
x_1' &= x_2\\
x_2' &= (5x_2 - x_1)/2.
\end{aligned}$$

Exercise 4. Euler with 2 steps:

$$x(2) \approx 7, \quad x'(2) \approx 6.53657, \quad y(2) \approx -1.33333, \quad y'(2) \approx -8.3619.$$

Euler's midpoint method with 2 steps:

$$x(2) \approx 7.06799, \quad x'(2) \approx -1.0262, \quad y(2) \approx -8.32262, \quad y'(2) \approx -15.2461.$$

Section 14.1

Section 14.2

Exercise 3.

$$\frac{\partial^3 f}{\partial x^2 \partial y} \approx \frac{f_{2,1} - f_{2,0} - 2f_{1,1} + 2f_{1,0} + f_{0,1} - f_{0,0}}{h_1^2 h_2}.$$

$$\frac{\partial^4 f}{\partial x^2 \partial y^2} \approx \frac{f_{2,2} - 2f_{2,1} + f_{2,0} - 2f_{1,2} + 4f_{1,1} - 2f_{1,0} + f_{0,2} - 2f_{0,1} + f_{0,0}}{h_1^2 h_2^2}.$$

Section 15.1

Section 15.2

Section 15.3

APPENDIX

Solutions

Section 1.5

Section 2.3

Section 3.1

Section 3.2

Exercise 2. The answers in this exercise can also be obtained by running the function `digits` in the module `baseconversion`, and printing out the results nicely with the function `string_from_digits` in the same module. The code for this can be found in the file `030202.py` in the folder for python code on the course webpage.

Exercise 2(a). We use algorithm 3.7 with $a = 40$ and $\beta = 4$. The first step in the algorithm gives

$$d_0 = a \,\%\, \beta = 40 \,\%\, 4 = 0 \text{ and } a = a \,//\, \beta = 40 \,//\, 4 = 10.$$

The second step in the algorithm gives

$$d_1 = a \,\%\, \beta = 10 \,\%\, 4 = 2 \text{ and } a = a \,//\, \beta = 10 \,//\, 4 = 2.$$

The third step in the algorithm gives

$$d_2 = a \,\%\, \beta = 2 \,\%\, 4 = 2 \text{ and } a = a \,//\, \beta = 2 \,//\, 4 = 0.$$

Since now $a = 0$, the algorithm terminates, and we have that $40 = (d_2 d_1 d_0)_4 = (220)_4$.

Exercise 5(a). We can convert four digits at a time. We write $1001101_2 = 01001101_2$ (we added an extra zero to obtain 2×4 digits). Since $0100_2 = 4_{16}$, and $1101_2 = \mathrm{d}_{16}$, we have that $1001101_2 = 4\mathrm{d}_{16}$.

Exercise 6(f). Since $\mathrm{f}_{16} = 1111_2$, $0_{16} = 0000_2$, and $1_{16} = 0001_2$, we have that $0.\mathrm{f}01_{16} = 0.111100000001_2$

Exercise 7(a). In general we have that $\beta = 10_\beta$ for any β. In particular we have that $7 = 10_7$, $37 = 10_{37}$, $4 = 10_4$.

Exercise 7(b). The equation $13 = 10_\beta$ gives that $\beta = 13$ from (a). The equation $100 = 10_\beta$ gives in the same way that $\beta = 100$. For all $a \in \mathbb{N}$ we can find a β which solves $a = 10_\beta$: It is enough to set $\beta = a$.

Section 3.3

Exercise 1(a). We have that $10_\beta = \beta$, so this is biggest for the largest base. In particular it is bigger for $\beta = 10$ when compared to $\beta = 9$, so that the statement is true.

Exercise 1(b). We have that $0.1_\beta = 1/\beta$. Now the smallest base gives the largest number, so that the statement is false.

Exercise 1(d). We can write

$$\frac{\ln\sqrt{e}^{\pi}}{\pi} = \frac{\ln(e^{\pi/2})}{\pi} = \frac{(\pi/2)\ln e}{\pi} = \frac{1}{2}.$$

Therefore, the number is rational.

Exercise 3. The answers in this exercise can also be obtained by running the function `decimal_digits_rational` in the module `baseconversion`, and printing out the results nicely with the function `string_from_digits` in the same module. The code for this can be found in the file `030303.py` in the folder for python code on the course webpage.

Exercise 4. We use algorithm 3.16 on the decimal part of π (i.e. $a = \pi - 3 = 0.14159265\ldots$) The first step in the algorithm gives

$$d_{-1} = \lfloor a*\beta \rfloor = \lfloor a*9 \rfloor = 1 \text{ and } a = a*\beta - d_{-1} = a*9-1$$

The next step gives

$$d_{-2} = \lfloor a*\beta \rfloor = \lfloor a*9 \rfloor = 2 \text{ and } a = a*\beta - d_{-2} = a*9-2.$$

The next step gives

$$d_{-3} = \lfloor a*\beta \rfloor = \lfloor a*9 \rfloor = 4 \text{ and } a = a*\beta - d_{-3} = a*9-4.$$

The next step gives

$$d_{-4} = \lfloor a*\beta \rfloor = \lfloor a*9 \rfloor = 1 \text{ and } a = a*\beta - d_{-4} = a*9-4.$$

we therefore get that the first four digits of π in base-9 are 3.1241_9.

This can also be be obtained by running the functions `decimal_digits_fractional` and `digits` in the module `baseconversion`, and printing out the results nicely with the function `string_from_digits` in the same module. The code for this can be found in the file `030304.py` in the folder for python code on the course webpage.

Exercise 5(a). $0.b_\beta$ is always a number in $[0,1)$, so that such a β exists only for $a<1$. Since $0.b_\beta = \frac{b}{\beta} = a = \frac{b}{c}$ it follows that $\beta = c$, so that we can find a unique β for all rational a on the form $a = 0.b_\beta$.

Exercise 5(b). Since $0.01_\beta = \frac{1}{\beta^2}$, if $a = \frac{b}{c} = 0.01_\beta$ we must have that $c = b\beta^2$. In other words, a rational number $\frac{b}{c}$ can be written on the form 0.01_β if and only if $c = b\beta^2$.

Exercise 5(c). Since $0.0b_\beta = \frac{b}{\beta^2}$, if $a = \frac{c}{d} = 0.0b_\beta$ we must have that $c = \frac{bd}{\beta^2}$. In other words, a rational number $\frac{c}{d}$ can be written on the form $0.0b_\beta$ if and only if $c = \frac{bd}{\beta^2}$.

Exercise 6. When we in algorithm 3.20 obtain a b which has been seen before, we will perform the same computations again, so that the sequence will repeat. There are c possibilities for this value since we compute the remainder with c. The longest possible repeating sequence is thus one where all values of b are observed. However, the value 0 will result in the rest of the digits being 0, so the maximum length repeating sequence is obtained when the values $1,...,c-1$ are observed for b in succession. This results in a repeating sequence of length $c-1$.

Exercise 7. This result is actually correct in any base, so we will here assume that the base is any number β. Assume that the digits of a are d_k, and that the digits $d_{-r},\ldots,d_{-r-t-1}$ eventually repeat (t is thus the length of the repeating sequence). We can write

$$a = b + b_0 + b_1 + b_2 + \ldots,$$

where b contains only the digits d_k with $k > -r$, and where b_s contains only the digits $d_{-r-st}, \dots, d_{-r-st-t-1}$. Since the digits repeat, b_s contains the same digits as b_0, shifted st digits to the right. This means that $b_s = \beta^{-st} b_0$, so that we can write

$$\begin{aligned} a &= b + b_0 + b_1 + b_2 + \dots = b + b_0 + \beta^{-t} b_0 + \beta^{-2t} b_0 + \beta^{-3t} b_0 + \dots \\ &= b + \frac{b_0}{1-\beta^{-t}} = b + \frac{b_0 \beta^t}{\beta^t - 1}, \end{aligned}$$

where we used the formula for the sum of a geometric series. Since b contains a finite number of digits it is rational (Exercise 8), and $\frac{b_0\beta^k}{\beta^t-1}$ is rational since the numerator and the denominator are integers. Since the sum of two rational numbers are rational, the results follows.

Exercise 8. Let the nonzero digits in base β be d_k for $k = r$ to s. If $r \geq 0$ the number is clearly rational, since it is an integer. When $r < 0$ the number can be written as

$$\sum_{k=r}^{s} d_k \beta^k = \frac{\sum_{k=r}^{s} d_k \beta^{r+k}}{\beta^r}.$$

Since here both the numerator and the denominator are integers, the number is rational.

Section 3.4

Exercise 1 (a). In the equation $7_\beta + 8_\beta = 13_\beta$, the left hand side is $7+8 = 15$. The right hand side is $13_\beta = \beta + 3$. Solving $\beta + 3 = 15$ we get that $\beta = 12$, so that the third alternative is correct.

Exercise 1 (b). We have that $4_9 + 5_9 = 10_9$, so that the first alternative is wrong.
We have that $3_9 + 3_9 = 6_9$, so that the second alternative is also wrong.
We have that $5_9 + 5_9 = 10 = 11_9$, so the third alternative is correct.
For the fourth alternative, we have that $11_9 - 3_9 = 10 - 3 = 7 = 7_9$, so that this alternative is also wrong. In summary, only the third alternative is correct.

Exercise 1 (c). We can write $40.125 = 5 \cdot 8 + 1/8 = 5 \cdot 8^1 + 0 \cdot 8^0 + 8^{-1}$. This can also be written as 50.1_8, so that the last alternative is the correct one.

Exercise 3 (a). $\begin{array}{r} 5_8 \\ -2_8 \\ \hline = 3_8 \end{array}$

Exercise 3(b).

$$\begin{array}{r} {\scriptstyle 2\,2\;\;} \\ \not{1}\not{0}0_2 \\ -1_2 \\ \hline =11_2 \end{array}$$

Exercise 3(c).

$$\begin{array}{r} {\scriptstyle 8\;\;\;\;\;\;} \\ \not{5}27_8 \\ -333_8 \\ \hline =174_8 \end{array}$$

Section 4.1

Exercise 1(c). The machine enters an infinite loop, since Python increases the precision used for numbers when the sum of two numbers is beyond the current limit. The machine will eventually run out of memory when too much recourses are required to represent the number, but this may take some billions of years. This means that the first alternative is correct.

Exercise 3. There is no way we can constuct a unique "threes's complement" in. One possibility is, in fact 4.3, to replace the representation of a negative number with the n first digits in $3^n - |x|$, where n still denotes the number of digits in x. With addition defined by neglecting digit $n+1$ as in two's complement, this representation of numbers will have the same properties as two's compliment. The only difference is that the numbers now represented only will cover two thirds of the numbers with $n+1$ digits: The lower third (which represents positive numbers), and the upper third (which represents negative numbers).

Section 4.2

Section 4.3

Exercise 3(a). Since $5a_{16} \leq 7f_{16}$, this is encoded with one byte, so that the UTF-8 encoding is the number itself, i.e. $5a_{16}$.

Exercise 3(b). We have that $80_{16} \leq f5_{16} \leq 7ff_{16}$, so that two bytes are used in the UTF8-encoding. Since $f5_{16} = 1111\ 0101_2$, we have that the UTF8-encoding is $1100\ 0011\ 1011\ 0101_2 = c3b5_{16}$.

Exercise 3(c). We have that $80_{16} \leq 3f8_{16} \leq 7ff_{16}$, so that two bytes are used in the UTF8-encoding also here. Since $3f8_{16} = 11\ 1111\ 1000_2$, we have that the UTF8-encoding is $1100\ 1111\ 1011\ 1000_2 = cfb8_{16}$.

Exercise 3(d). We have that $800_{16} \leq 8f37_{16} \leq ffff_{16}$, so that three bytes are used in the UTF8-encoding also here. Since $8f37_{16} = 1000\ 1111\ 0011\ 0111_2$, we have that the UTF8-encoding is $1110\ 1000\ 1011\ 1100\ 1011\ 0111_2 = e8bcb7_{16}$.

Exercise 5(a). These characters have code points $e6_{16} = 1110\ 0110_2$, $f8_{16} = 1111\ 1000_2$, and $e5_{16} = 1110\ 0101_2$. All of them are stored with 2 bytes in UTF-8.

- The UTF8-encoding of 'æ' is $1100\ 0011\ 1010\ 0110_2 = c3a6_{16}$, which corresponds to the two characters Ã¦ in the ISO Latin1 character set.
- The UTF8-encoding of 'ø' is $1100\ 0011\ 1011\ 1000_2 = c3b8_{16}$, which corresponds to the two characters Ã¸ in the ISO Latin1 character set.
- The UTF8-encoding of 'å' is $1100\ 0011\ 1010\ 0101_2 = c3a5_{16}$, which corresponds to the two characters Ã¥ in the ISO Latin1 character set.

Exercise 5(b). None of the three codepoints for 'æ', 'ø', 'å, are seen to be valid UTF8-codes.

Exercise 5(c). All three characters are also stored with 2 bytes in UTF-16, and as $00e6_{16}$, $00f8_{16}$, and $00e5_{16}$, respectively. These are displayed as NULæ, NULø, and NULå, respectively, where NUL is some representation of the nonprintable ASCII character with code 00. The other way, the ISO Latin1 encoding of the three characters is three bytes, which does not give a legal UTF-16 code. UTF-16 encoding.

Exercise 5(d). Assume that the characters are stored with UTF-8. As shown in (a), the UTF-8 encodings are $c3a6_{16}$, $c3b8_{16}$, and = $c3a5_{16}$, which are the valid two-byte Unicode characters 쎦, 쎸, and 쎥. These are also called Hangul symbols.

Conversely, assume that the characters are stored with UTF-16. The first byte in the code $1110\ 0110_2$ for 'æ', indicates that it should be stored with 3 bytes, but then the second byte should start with 10, which it does not. The same applies for 'ø'. Finally, for the code $1111\ 1000_2$ for 'å',, there are no bytes in UTF-8 which start with 11111, so that all characters are invalid UTF-8 characters.

Exercise 8. We have that

$$41\ 42\ 43\ 44\ 45_{16} = 0100\ 0001\ 0100\ 0010\ 0100\ 0011\ 0100\ 0100\ 0100\ 0101_2$$

This is clearly a valid UTF-8 encoding of 5 ASCII characters. Since 5 bytes are used, it can not be a UTF-16 code, since that would require an even number of bytes being used.

Exercise 10. We have that

$$
\begin{aligned}
41\,C3\,98\,41\,C3\,41\,41\,C3\,98\,98\,41_{16} = \\
0100\,0001\,1100\,0011\,1001\,1000\,0100\,0001\,1100\,0011\,0100 \\
0001\,0100\,0001\,1100\,0011\,1001\,1000\,1001\,1000\,0100\,0001_2.
\end{aligned}
$$

The first byte, $0100\,0001_2$, is a valid UTF-8 character. The next two bytes,

$$1100\,0011\,1001\,1000_2,$$

is a valid two-byte UTF-8 character. The next byte, 0100 0001, is again a valid UTF-8 character. The next two bytes, $1100\,0011\,0100\,0001_2$ are not valid, since the second byte should start with 10 when the first starts with 110.

Section 4.5

Section 5.2

Exercise 2. The last expression may give large relative error when calculated on a machine using floating point arithmetic, since it is possible for $\sin(-x^2)$ to be close to $-1/2$ (cancellation). The other expressions can not give cancellation.

Exercise 6(b). The biggest number of 9.834 and 2.45 is 9.834, and this can be written on normalised form as 0.9834×10^1. The other number can be written as 0.2450×10^1 when we use the same exponent (we added a 0 to get 4 digits). We add the significands and get $0.9834 + 0.2450 = 1.2284$. At the end we convert 1.2284×10^1 to normalized form and get 0.1228×10^2, where we at the end had to do a rounding since the significand should be represented by 4 digits only.

Exercise 7(a). A normalised number in base β is represented as $\alpha \times \beta^n$ where n is a one digit number, and α is a number between β^{-1} and 1 represented with a four digit number in base β ($0.1 = 10^{-1}$ was exchanged with β^{-1}).

Exercise 7(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.

In the hexadecimal numeral system the rules are similar: If the last digit is one of 8, 9, a,b,c,d,e, or f, round up, otherwise round down.

Exercise 9. The Python code is

```
x=0.0
while x<= 2.0:
  print(x)
  x=x+0.1
```

In my program the last value 2.0 is not written. The explanation is that 0.1 can not be represented exactly by the computer. What actually is the case here is that the machine represents 0.1 with a number slightly bigger than 0.1. When this number is added with itself 20 times, we have a number which is bigger than 2.0, so the number is not printed. 2.0 can, however, be represented exactly by the computer.

Exercise 10. The Python code is

```
from random import random

def test_law(ls,rs):
    count = 0
    x = 0; y = 0; z = 0
    for k in range(10000):
        x=random(); y = random(); z = random()

        law_holds = (ls(x,y,z) == rs(x,y,z))
        if not law_holds:
            count = count+1
    print("Law failed %i percent of the times" % (100*count/10000))
    if count > 0:
        print("Last numbers where the law failed:", x, y, z)
    print("\n")

# (a)
print("Testing the distributive law")
ls_dist = lambda x,y,z: (x+y)*z
rs_dist = lambda x,y,z: x*z + y*z
test_law(ls_dist,rs_dist)

# (b)
```

```
print("Testing the associative law")
ls_ass = lambda x,y,z: (x+y)+z
rs_ass = lambda x,y,z: x+(y+z)
test_law(ls_ass,rs_ass)

# (c)
print("Testing the commutative law")
ls_comm = lambda x,y,z: x+y
rs_comm = lambda x,y,z: y+x
test_law(ls_comm,rs_comm)

# (d)
print("Testing the associative law for multiplication")
ls_ass_mult = lambda x,y,z: (x*y)*z
rs_ass_mult = lambda x,y,z: x*(y*z)
test_law(ls_ass_mult,rs_ass_mult)

print("Testing the commutative law for multiplication")
ls_comm_mult = lambda x,y,z: x*y
rs_comm_mult = lambda x,y,z: y*x
test_law(ls_comm_mult,rs_comm_mult)
```

The code uses a generic tester for laws, where the laws to be tested are sent as parameters.

Section 5.3

Exercise 2(a). The absolute error is $|a - \tilde{a}| = |1 - 0.9994| = 0.0006$. The relative error is $\frac{|a-\tilde{a}|}{|a|} = \frac{0.0006}{1} = 0.0006$. The relative error can also be written as $0.6 \times 10^{-3} \approx 10^{-3}$, and observation 5.20 says that about the 3 most significant digits in a and $\tilde{a}$ should agree. This is not so far from the truth in this case, since 0.999 (where we have included the three most significant digits) will be rounded to 1.000.

Exercise 2(b). The absolute error is $|a-\tilde{a}| = |24-23.56| = 0.44$. The relative error is $\frac{|a-\tilde{a}|}{|a|} = \frac{0.44}{24} = 0.01833$. The relative error can also be written as $1.8 \times 10^{-2} \approx 10^{-2}$, and observation 5.20 says that about the 2 most significant digits in a and $\tilde{a}$ should agree. This is in fact the case, since 23.56 with the two most significant digits gives 24.

Exercise 2(c). The absolute error is $|a - \tilde{a}| = |-1267 + 1267.345| = 0.345$. The relative error is $\frac{|a-\tilde{a}|}{|a|} = \frac{0.345}{1267} = 0.000272$. The relative error can also be written

as $2.7 \times 10^{-4} \approx 10^{-4}$, and observation 5.20 says that about the 4 most significant digits in a and $\tilde{a}$ should agree. This is in fact the case, since -1267.345 with the two most significant digits gives -1267.

Exercise 2(d). The absolute error is $|a-\tilde{a}| = |124-7| = 117$. The relative error is $\frac{|a-\tilde{a}|}{|a|} = \frac{117}{124} = 0.9435$. The relative error can also be written as $0.94 \times 10^0 \approx 10^0$, and observation 5.20 says that no significant digits in a and $\tilde{a}$ should agree. This is the case here.

Exercise 3. (a). The absolute error the other way is $\frac{|\tilde{a}-a|}{|\tilde{a}|} = \frac{0.0006}{0.9994} = 0.0006$, so that we have approximately the same relative error. This agrees with the sentence in section 5.3.3, which says that the two relative errors should be quite close, in cases where the two relative errors are quite small.
(b). The absolute error the other way is $\frac{|\tilde{a}-a|}{|\tilde{a}|} = \frac{0.44}{23.56} = 0.0187$, so that we have approximately the same relative error. This agrees with the sentence in section 5.3.3, as for a).
(c). The absolute error the other way is $\frac{|\tilde{a}-a|}{|\tilde{a}|} = \frac{0.345}{1267.345} = 0.000272$, so that we have approximately the same relative error. This agrees with the sentence in section 5.3.3.
(d). The absolute error the other way is $\frac{|\tilde{a}-a|}{|\tilde{a}|} = \frac{117}{7} \approx 16.7$, so that we here have relative errors quite far apart. This has to do with that the relative errors are quite big, so that the observation is not valid.

Exercise 4. In example 5.9 we found the approximation $\tilde{a} = 0.1247 \times 10^2 = 13.47$ to the sum $5.645 + 7.821 = 13.466$. The relative error is

$$\frac{|a-\tilde{a}|}{|a|} = \frac{0.004}{13.466} \approx 0.000297 = 2.97 \times 10^{-4} \approx 10^{-4}.$$

From the observation the numbers should agree in the four most significant digits. This is the case since the four most significant digits in 13.466 give 13.47.

In example 5.10 we found the approximation $0.4234 \times 10^2 = 42.34$ to the sum $42.34 + 0.0033 = 42.3433$. The relative error is

$$\frac{|a-\tilde{a}|}{|a|} = \frac{0.0033}{42.3433} \approx 0.000078 = 0.78 \times 10^{-4} \approx 10^{-4}.$$

From the observation the numbers should agree in the four most significant digits. This is the case since the four most significant digits in 42.3433 give 42.34.

In example 5.11 we found the approximation $0.7 \times 10^{-1} = 0.07$ to the difference $10.34 - 10.27 = 0.07$. The relative error here is 0, and all significant digits should agree, which they do since the numbers are equal.

In example 5.12 we found the approximation $0.9000 \times 10^{-3} = 0.0009$ to the difference $10/7 - 1.42 \approx 0.8571 \times 10^{-3} = 0.0008571$. The relative error is

$$\frac{|a-\tilde{a}|}{|a|} = \frac{0.0000429}{0.0008571} \approx 0.05 = 0.5 \times 10^{-1} \approx 10^{-1}.$$

From the observation the numbers should agree in just the most significant digit. This is the case since 0.0008571 with only the most significant digit is 0.0009.

Exercise 6(a). We see that the bits after the 23 first bits in $1/10$ are 11001100.... These are the same bits as the ones occuring in $1/10$, but delayed with 20 bits. This means that the truncation error is 0.1×2^{-20}, and that the truncation of $1/10$ to the first 23 decimal bits equals $0.1 - 0.1 \times 2^{-20} = 0.1 \times (1 - 2^{-20})$.

Exercise 6(c). After one hour the sum of the increments are $3600 \times 10 = 36000$, i.e. the number of tenths of second in an hour. The loop then computes

$$\sum_t i_t * c \approx c \sum_t i_t = c \times 36000 = 36000 \times 0.1 \times (1 - 2^{-20}) = 3600(1 - 2^{-20}).$$

Here we made an approximation, since we change the summation order, we may loose some digits in precision. If there were no roundoff errors, we would have computed $36000 \times 0.1 = 3600$. The absolute error is thus approximately $|3600(1 - 2^{-20}) - 3600 = | = 3600 \times 2^{-20} \approx 0.003433$.

Exercise 6(d). The `while`-loop could sum the time increments i_t, without multiplying with c. When needed, the accumulated time could be converted from tenths of a second to seconds by performing division by 10, using Algorithm 3.20. This avoids the accumulation of errors

Section 5.4

Exercise 1(a). We can write

$$\frac{5-\sqrt{5}}{5+\sqrt{5}} + \frac{\sqrt{5}}{2} = \frac{(5-\sqrt{5})(5-\sqrt{5})}{5^2-5} + \frac{\sqrt{5}}{2} = \frac{25-10\sqrt{5}+5}{20} + \frac{\sqrt{5}}{2} = \frac{3}{2}.$$

Therefore, the last alternative is the correct one.

Exercise 1(b). The last alternative is the correct one. The reason is that several bits are allocated for the exponent, see Fact 4.8. The first alternative is wrong, positive numbers can also give roundoff errors. The second alternative is wrong because we typically have limitations on the computer in the significand and the

exponent, and even if the types can expand the bits used for these as in Python, the computer has at the end a limited amount of memory. The third alternative is wrong, since 64-bit integers represents numbers in the interval $[-2^{63}, 2^{63}-1]$ (Fact 4.2).

Exercise 2(a). If x is very large we will in the expression $\sqrt{x+1}-\sqrt{x}$ subtract two large numbers very close to each other. According to observation 5.13 we then can loose precision with many digits (i.e. cancellation). If we multiply with $\sqrt{x+1}+\sqrt{x}$ up and down we get $\frac{1}{\sqrt{x+1}+\sqrt{x}}$, where we have avoided the problem of subtracting two numbers close to each other.

Exercise 2(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.

Exercise 2(c). We can write $\cos^2 x - \sin^2 x = \cos(2x)$,so that we avoid subtraction of two almost equal numbers near $x = \frac{\pi}{4}$.

Section 6.1

Section 6.2

Exercise 2(a). We have that $x_{n+2} = f(n, x_n, x_{n+1}) = 3x_{n+1} - x_n$. We compute

$$\begin{aligned}
x_2 &= 3x_1 - x_0 = 3 - 2 = 1 \\
x_3 &= 3x_2 - x_1 = 3 - 1 = 2 \\
x_4 &= 3x_3 - x_2 = 3\times 2 - 1 = 5 \\
x_5 &= 3x_4 - x_3 = 3\times 5 - 2 = 13.
\end{aligned}$$

Exercise 2(d). We have that $x_{n+1} = f(n, x_n) = -\sqrt{4 - x_n}$. We compute

$$\begin{aligned}
x_1 &= -\sqrt{4 - x_0} = -\sqrt{4 - 0} = -2 \\
x_2 &= -\sqrt{4 - x_1} = -\sqrt{4 + 2} = -\sqrt{6} \approx -2.4495 \\
x_3 &= -\sqrt{4 - x_2} = -\sqrt{4 + 2.4495} \approx -2.5396 \\
x_4 &= -\sqrt{4 - x_3} = -\sqrt{4 + 2.5396} \approx -2.5573 \\
x_5 &= -\sqrt{4 - x_4} = -\sqrt{4 + 2.5573} \approx -2.5607.
\end{aligned}$$

Exercise 2(e). We have that $x_{n+2} = f(n, x_n, x_{n+1}) = \frac{1}{5}(3x_{n+1} - x_n + n)$. We compute

$$
\begin{aligned}
x_2 &= \frac{1}{5}(3-0+0) = \frac{3}{5} \\
x_3 &= \frac{1}{5}(\frac{9}{5} - 1 + 1) = \frac{9}{25} \\
x_4 &= \frac{1}{5}(\frac{27}{25} - \frac{3}{5} + 2) = \frac{62}{125} \\
x_5 &= \frac{1}{5}(\frac{186}{125} - \frac{9}{25} + 3) = \frac{816}{625}.
\end{aligned}
$$

Exercise 2(f). If we insert $x_0 = 3$ we get that $x_1^2 = -15 + 1 = -14$, which clearly has no solution. Furthermore, x_{n+1} is not uniquely determined for other initial conditions, since $x_{n+1} = \pm\sqrt{1 - 5x_n}$.

Section 6.3

Exercise 1. Examples of such functions, with more general versions as well, can be found in the following listing:

```
def difference_eq_print1(f, x0, N):
    """
    Iterate a first order difference equation, and print the values.

    f: a function which computes the next value in the difference equation from the previous value
    x0: The initial value for the difference equation
    N: The number of iterations
    """
    xp = x0
    for n in range(N):
        x=f(xp, n)
        print(x)
        xp=x

def difference_eq_print2(f, x0, x1, N):
    """
    Iterate a second order difference equation, and print the values.

    f: a function which computes the next value in the difference equation from the two previous va
    x0, x1: The initial values for the difference equation
    N: The number of iterations
```

```
    """
    xpp, xp = x0, x1
    for n in range(N):
        x=f(xpp, xp, n)
        print(x)
        xpp=xp
        xp=x

def difference_eq_print3(f, x0, x1, x2, N):
    """
    Iterate a third order difference equation, and print the values.

    f: a function which computes the next value in the difference equation from the three
    x0, x1, x2: The initial values for the difference equation
    N: The number of iterations
    """
    xppp, xpp, xp = x0, x1, x2
    for n in range(N):
        x=f(xppp, xpp, xp, n)
        print(x)
        xppp = xpp
        xpp = xp
        xp=x

def difference_eq2(f, x0, x1, N):
    """
    Return the next n values of a second order difference equation

    f: a function which computes the next value in the difference equation from the two p
    x0, x1: The initial values for the difference equation
    N: The number of iterations
    """
    x = [0.]*N
    x[0] = f(x0, x1,0)
    x[1] = f(x1, x[0],1)
    for n in range(2,N):
        x[n] = f(x[n-2], x[n-1],n)
    return x
```

Exercise 2. The code can look as follows:

```
from difference_eqs import difference_eq_print2
import sys

# Read N from the command line
# Usage: python 060302.py 10
def fibonacci2(xp,xpp,n):
    return xp+xpp

N=int(sys.argv[1])
difference_eq_print2(fibonacci2, 0, 1, N)
```

Exercise 3. The code can look as follows:

```
from difference_eqs import difference_eq_print3
import sys

def fibonacci3(xp,xpp,xppp,n):
    return xp+xpp+xppp

N=int(sys.argv[1])
difference_eq_print3(fibonacci3, 0, 1, 1, N)
```

Section 6.4

Section 6.5

Exercise 2. The code can look as follows:

```
from difference_eqs import difference_eq_print1, difference_eq_print2

print('a')

def fa(xpp,xp,n):
    return (-4.0*xp+4.0*xpp)/3.0

difference_eq_print2(fa,1.,2/3.,100)
input('Press any key to continue')


print('b')
```

```
def fb(xp,n):
    return (1+xp/3.)/3.

difference_eq_print1(fb,1.,100)
input('Press any key to continue')

print('c')

def fc(xp,n):
    return 2.0+xp/3.0

difference_eq_print1(fc,2.,30)
```

Exercise 2(a). The characteristic equation is $3r^2+4r-4=0$, which has roots $r=\frac{-4\pm\sqrt{16+48}}{6}=\frac{-2\pm 4}{3}$. The roots are thus -2 and $2/3$, so that the solution to the difference equation is $x_n = C(-2)^n + D(2/3)^n$. The initial values give

$$\begin{aligned} C+D &= 1 \\ -2C+\frac{2}{3}D &= 2/3, \end{aligned}$$

which gives that $\frac{8}{3}D=\frac{8}{3}$, so that $D=1$ and $C=0$. The solution is thus $x_n = (2/3)^n$. This will go to zero, but due to roundoff in the second initial condition there will be a term on the form $\hat{\epsilon}(-2)^n$ also contributing in the simulation, so that we will eventually get overflow, i.e. $x_n=\pm\infty$ for one choice of the sign. Since $x_{n+2}=(-4x_{n+1}+4x_n)/3$, we will then get $x_{n+1}=\mp\infty$, i.e. the sign is reversed due to the additional minus sign. For the next iterations, if x_{n+1} and x_n are ∞ with opposite signs, then x_{n+2} is seen to be ∞ with opposite sign from x_{n+1}. Therefore, the computed solution will give overflows with alternating signs for large n. The last alternative is thus correct.

Exercise 2(b). It is straightforward to check that the exact solution is $x_n=\frac{3}{8}+\frac{45}{8}\left(\frac{1}{9}\right)^n$. It is clear that simulations will converge to $3/8$.

Exercise 4(b). Due to the roundoff error in the second initial value, the computer will compute values on the form $\hat{\epsilon}_1 3^n - \frac{5}{14}(1-\hat{\epsilon}_2)5^{-n}$, where $\hat{\epsilon}_i$ are small. Due to the presence of the term $\hat{\epsilon}_1 3^n$, we will eventually get overflow.

Exercise 4(c). The code can look as follows:

```
from difference_eqs import difference_eq_print1
import sys

def f(xp,n):
    return 3*xp + 5**(-n)

N = int(sys.argv[1]) # 40
difference_eq_print1(f, -5.0/14, N)
```

Exercise 5(a). The characteristic equation is $r^2 - r - 1 = 0$, which has roots $r = \frac{1\pm\sqrt{1+4}}{2} = \frac{1\pm\sqrt{5}}{2}$. This means that we have two different real roots, so that the general solution is

$$x_n = C\left(\frac{1+\sqrt{5}}{2}\right)^n + D\left(\frac{1-\sqrt{5}}{2}\right)^n.$$

The two initial values give

$$\begin{aligned} C + D &= 1 \\ C\frac{1+\sqrt{5}}{2} + D\frac{1-\sqrt{5}}{2} &= \frac{1-\sqrt{5}}{2}. \end{aligned}$$

Substituting the first in the second gives

$$C\sqrt{5} + \frac{1-\sqrt{5}}{2} = (1-\sqrt{5})/2,$$

so that $C = 0$ and $D = 1$. This gives the solution $x_n = \left(\frac{1-\sqrt{5}}{2}\right)^n$.

Exercise 5(b). Due to rounding in the second initial condition, the computer will simulate values of the form

$$x_n = \hat{\epsilon}\left(\frac{1+\sqrt{5}}{2}\right)^n + (1-\hat{\epsilon})\left(\frac{1-\sqrt{5}}{2}\right)^n.$$

This means that the values of x_n eventually will overflow. In the beginning of the calculations, the values seem to converge to zero, since the term $(1-\hat{\epsilon})\left(\frac{1-\sqrt{5}}{2}\right)^n$ dominates in the beginning, and this term converges to zero.

Exercise 5(c). The code can look as follows:

```
from difference_eqs import difference_eq_print2
from math import sqrt
import sys

def fibonacci2(xp,xpp,n):
    return xp+xpp

N=int(sys.argv[1]) # 1560
difference_eq_print2(fibonacci2, 1.0, (1-sqrt(5.0))/2., N)
```

Exercise 6(a). The characteristic equation is $r^2 - \frac{2}{5}r + \frac{1}{45} = 0$, which has roots

$$r = \frac{\frac{2}{5} \pm \sqrt{\frac{4}{25} - \frac{4}{45}}}{2} = \frac{\frac{2}{5} \pm \sqrt{\frac{36-20}{225}}}{2} = \frac{\frac{2}{5} \pm \frac{4}{15}}{2} = \frac{1}{5} \pm \frac{2}{15},$$

so that $r = \frac{1}{3}$ eller $r = \frac{1}{15}$. Therefore the general solution to the difference equation is $x_n = A\left(\frac{1}{15}\right)^n + B\left(\frac{1}{3}\right)^n$. The initial values $x_0 = 1$, $x_1 = \frac{1}{15}$ give

$$\begin{aligned} A + B &= 1 \\ \frac{1}{15}A + \frac{1}{3}B &= \frac{1}{15}. \end{aligned}$$

These equations can also be written as

$$\begin{aligned} A + B &= 1 \\ A + 5B &= 1. \end{aligned}$$

We quickly see that the dolution to this is $A = 1$, $B = 0$, so that the solution to the difference equation is $x_n = \left(\frac{1}{15}\right)^n = 15^{-n}$.

Exercise 6(b). The other initial condition cannot be represented exactly on the computer, so that the computer instead will find a solution on the form

$$\hat{x_n} = (1 - \hat{\epsilon})\left(\frac{1}{15}\right)^n + \hat{\epsilon}\left(\frac{1}{3}\right)^n,$$

where $\hat{\epsilon}$ is a small number representing the roundoff error committet by the computer. When n becomes large, the "error" $\hat{\epsilon}\left(\frac{1}{3}\right)^n$ dominates in this expression, which explains why we must expect numerical inaccuracies for large n. Note that the absolute error is not large since $\hat{\epsilon}\left(\frac{1}{3}\right)^n$ is a small number, but that the relative error is very large since, since $\hat{\epsilon}\left(\frac{1}{3}\right)^n$ is relatively much larger than $\left(\frac{1}{15}\right)^n$ for large n.

Exercise 6(c). $\hat{\epsilon}$ represents approximately the smallest number the machine can represent. If we use 64 bits this corresponds to $\approx 2^{-63} \approx 10^{-17}$. We have lost all significant digits when the "error" $\hat{\epsilon}\left(\frac{1}{3}\right)^n$ becomes larger that the actual solution $\left(\frac{1}{15}\right)^n$, i.e. $10^{-17}\left(\frac{1}{3}\right)^n > \left(\frac{1}{15}\right)^n$. This corresponds to $5^n > 10^{17}$, which gives $n > \frac{17\ln 10}{\ln 5} \approx 24$. The arguments given here are not exact. For example, it can be that the estimate for $\hat{\epsilon}$ is not very exact.

Exercise 6(d). The code can look as follows:

```
from difference_eqs import difference_eq_print2
import sys

N=int(sys.argv[1]) # 100

def f(xp,xpp,n):
    return 2*xp/5. + xpp/45.

difference_eq_print2(f, 1., 1.0/15, N)
```

Exercise 7(a). The characteristic equation is $r^2 - \frac{5}{2}r + 1 = 0$, which has roots 2 and 1/2 The general solution is thus $C2^n + D2^{-n}$. The initial values give that $C + D = 1$, $2C + D/2 = 1/2$, which give that $C = 0$, $D = 1$, so that $x_n = 2^{-n}$.

Exercise 7(b). There are no roundoff errors in the initial conditions and the coefficients here, so that the iterations will compute 2^{-n} exactly. When n gets large, however, the computer can't represent 2^{-n} exactly anymore (there will be an underflow in the exponent), so that this will be rounded to zero. Due to this roundoff error, the computer will eventually compute $x_n = \hat{\epsilon}_1 2^n + (1-\hat{\epsilon}_2)2^{-n}$ for $\hat{\epsilon}_i$ very small, and this will eventually give $\pm\infty$. From the difference equation it is clear that the next iteration will also give $\pm\infty$ with the same sign. The next iteration will then give NaN, since we subtract $\pm\infty$ from itself.

Exercise 7(c). The code can look as follows:

```
from difference_eqs import difference_eq2
from numpy import *
import matplotlib.pyplot as plt

def f(xpp,xp,n):
    return 5*xp/2. - xpp
```

```
N = 3180
x = difference_eq2(f, 1.0, 0.5, N)

n = array(arange(2,12))
plt.plot(n, x[0:10], 'r', n, 2**(-array(arange(2,12, dtype=float))),'g')
plt.show()

print(x[1068:1078] - 2**(-array(arange(1070,1080))))
print(x[3168:])
```

Section 7.1

Section 7.2

Exercise 3(a).

$$
\begin{array}{rrr}
f(t)=2, & f(a)=2, & f(o)=2, \\
f(h)=2, & f(m)=1, & f(l)=2, \\
f(e)=6, & f(n)=2, & f(i)=1, \\
f(r)=3, & f(y)=1, & f(w)=1, \\
f(\sqcup)=6, & f(p)=2, & f(d)=1.
\end{array}
$$

Exercise 3(b). An example of a Huffman tree for this text can be seen in figure 16:

Exercise 3(c). The Huffman coding for the text "there are many people in the world" is then:

$$
\begin{aligned}
&0100\ 0101\ 11\ 0010\ 11\ 000\ 0110\ 0010\ 11\ 000 \\
&00111\ 0110\ 0111\ 10110\ 000\ 1000\ 11\ 1001\ 1000 \\
&1010\ 11\ 000\ 10111\ 0111\ 000\ 0100\ 0101\ 11\ 000 \\
&001100\ 1001\ 0010\ 1010\ 001101
\end{aligned}
$$

The entropy is:

$$H = 3.6325 \tag{.6}$$

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.

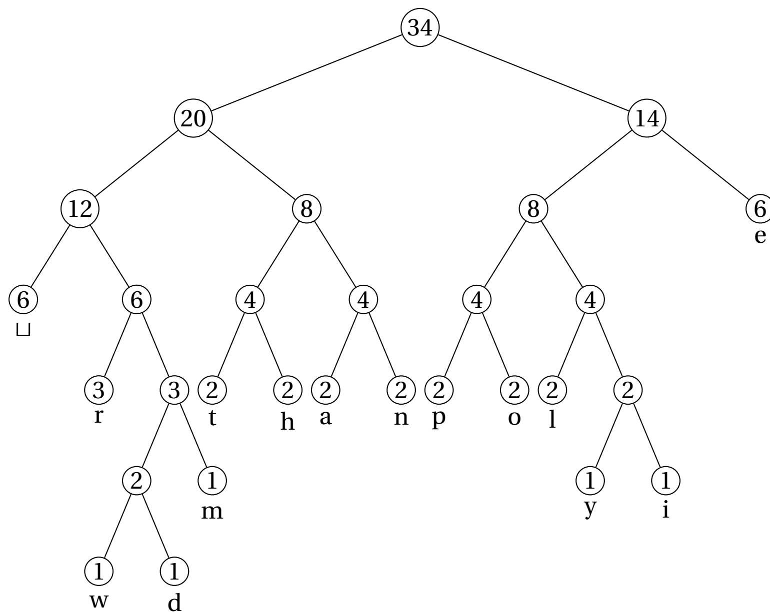


Figure 16. The Huffman tree for the text 'there are many people in the world'.

Exercise 5(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

$$H = 1.5 \tag{.7}$$

This gives an optimal coding with 12 bits for 8 symbols, which is just what the Huffman coding gave.

Exercise 5(b). Dividing all the frequencies by 2 and interchanging A with C in the tree in (a) gives the tree for this problem. We can now use the same Huffman codes (with the code of A interchanged by the code of C), and Huffman coding is clearly still optimal (the entropy is unchanged).

Exercise 6. If we assume all letters have equal frequency, we can in the 64 first steps of constructing the Huffman tree simply combine adjacent letters into new nodes. We then have 64 nodes, each with frequency 2. We can repeat this procedure and obtain 32 new nodes with frequency 4 each, then 16 new nodes with frequency 8 each, and so on. We end up with a binary tree with 8 levels, where all nodes have two branches, and where the letters all are leaf nodes at the lowest level. The letters are then represented by all possible combination of 7 bits, since any path from the root node can be followed down to the leaf nodes.

Section 7.3

Exercise 1(a). The first text has probabilities $p(A) = 0.5$, $p(B) = 0.5$, so that the entropy is $-0.5\log_2(0.5) - 0.5\log_2(0.5) = 1$. The second text has probabilities $p(A) = 5/9$, $p(B) = 4/9$, so that the entropy is $-5/9\log_2(5/9) - 4/9\log_2(4/9) = 0.99108$. The statement is therefore false.

Exercise 1(b). We have that the probability of the symbol is $p(A) = 1$. Since $\log_2(1) = 0$, the informtaion entropy is 0, so that the statement is true.

Exercise 1(c). The statement is false. As an example, if we repeat a short text many times, the information entropy is unchanged.

Exercise 1(d). The answer to this question consists of a value of true or false. If p is the probability of true, the information entropy is $-p\log_2 p - (1-p)\log_2(1-p)$. If $p \geq 1-p$, we have that

$$-p\log_2 p-(1-p)\log_2(1-p) < -p\log_2(1-p)-(1-p)\log_2(1-p) = -\log_2(1-p) < 1.$$

Similarly when $p < 1-p$. The statement is therefore false.

Exercise 2. The information entropy for the first text is

$$-0.5\log_2 0.5 - 0.5\log_2 0.5 = 1.$$

Clearly Huffman coding here uses 1 bit per symbol also, so this achieves a minimum number of bits per symbol.

The information entropy for the second text is

$$2(-0.25\log_2 0.25) - 0.5\log_2 0.5 = 1.5.$$

Huffman coding here uses codes of length $l(A) = l(B) = 2$, and $l(C) = 1$, so that it uses

$$p(A)l(A) + p(B)l(B) + p(C)l(C) = 0.25 \times 2 + 0.25 \times 2 + 0.5 \times 1 = 1.5$$

bits per symbol, so this achieves a minimum number of bits per symbol as well.

The information entropy for the third text is $-0.25\log_2 0.25 - 0.75\log_2 0.75 \approx 0.81128$. Clearly Huffman coding here uses 1 bit per symbol, so this does not achieve a minimum number of bits per symbol.

The information entropy for the fourth text is $4(-0.25\log_2 0.25) = 2$. Clearly Huffman coding here uses 2 bits per symbol also, so this achieves a minimum number of bits per symbol.

Exercise 4(a). We have that $f(t) = 2$, $f(o) = 3$, $f(b) = 1$, $f(e) = 1$, $f(i) = 1$, $f(s) = 1$, $f(d) = 1$, and $f(\sqcup) = 4$, so that the probabilities are $p(t) = 2/14$, $p(o) = 3/14$, $p(b) = 1/14$, $p(e) = 1/14$, $p(i) = 1/14$, $p(s) = 1/14$, $p(d) = 1/14$, and $p(\sqcup) = 4/14$. The information entropy is

$$-5 \times \frac{1}{14}\log_2(1/14) - \frac{2}{14}\log_2(2/14) - \frac{3}{14}\log_2(3/14) - \frac{4}{14}\log_2(4/14) \approx 2.7534.$$

Exercise 4(b). We have that $f(\sqcup) = 4$, $f(d) = 3$, $f(o) = 3$, $f(b) = 2$, and $f(e) = 2$, so that the probabilities are $p(\sqcup) = 2/7$, $p(d) = 3/14$, $p(o) = 3/14$, $p(b) = 1/7$, and $p(e) = 1/7$. The information entropy is

$$-\frac{2}{7}\log_2(2/7) - 2\frac{3}{14}\log_2(3/14) - 2\frac{1}{7}\log_2(1/7) \approx 2.2709.$$

Exercise 4(c). We have that $f(o) = 6$, $f(\sqcup) = f(d) = f(b) = f(y) = 2$, and $f(s) = f(c) = 1$, , so that the probabilities are $p(o) = 3/8$, $p(\sqcup) = p(d) = p(b) = p(y) = 1/8$, and $p(s) = p(c) = 1/16$. The information entropy is

$$-\frac{3}{8}\log_2(3/8) - 4\frac{1}{8}\log_2(1/8) - 2\frac{1}{16}\log_2(1/16) = -\frac{3}{8}\log_2(3/8) + 3/2 + 1/2 \approx 2.5306.$$

Section 7.4

Exercise 2(a). Clearly we have that $f(A) = 9$, $f(B) = 1$, so that $p(A) = 9/10$, $p(B) = 1/10$.

Exercise 2(b). We have that $\lceil -\log_2(0.9^9 0.1)\rceil + 1 = 6$, so 6 bits are needed.

Exercise 2(c). The first seven A's restrict us to the interval $[0, 0.9^7]$. The next B restricts us to the interval $[0.9^8, 0.9^7]$, and the two last A's restrict us further to the interval $[0.9^8, 0.9^8 + 0.9^2(0.9^7 - 0.9^8)]$. The midpoint in this interval is $0.9^8 + 0.9^2(0.9^7 - 0.9^8)/2 \approx 0.44983823445$.

- The first bit in this is clearly 0.

- To compute the second bit we compute $2 \times 0.44983823445 \approx 0.8996764689$, so that the second bit is 1.
- We then compute $2 \times 0.8996764689 - 1 \approx 0.7999353$, so that the third bit is 1.
- We then compute $2 \times 0.7999353 - 1 \approx 0.5987058756$, so that the fourth bit is 1.
- We then compute $2 \times 0.5987058756 - 1 \approx 0.1974117512$, so that the fifth bit is 0.
- Finally we compute $2 \times 0.1974117512 \approx 0.3948235024$, so that the sixth and final bit is 0.

The arithmetic code is thus 011100 (we could here have omitted the two trailing zeros in the arithmetic code as well, since 0.0111 and 0.011100 correspond to the same number).

Exercise 3(a). The information entropy is $-4 \times 0.25 \log_2(0.25) = 2$.

Exercise 3(c). Arithmetic coding requires $\lceil -\log_2(0.25^m) \rceil + 1 = 2m + 1$ bits, so that we require $\frac{2m+1}{m}$ bits per symbol. When m is large this is very close to 2.

Exercise 3(e). We map A to the interval $[0, 0.25]$, B to the interval $[0.25, 0.5]$, C to the interval $[0.5, 0.75]$, D to the interval $[0.75, 1]$. This means that the four intervals correspond to numbers where the first bits are 00, 01, 10, and 11, respectively. Note that these are exactly the Huffman codewords for the four letters. Clearly this also means that, when we refine the arithmetic code with a new letter, this means that we simply add the corresponding bits to the code This means that the arithmetic code is the same as the Huffman code, with the exception that we add a bit at the end. This bit represents that we restrict to the midpoint on the interval, which is achieved if we set the bit to 1 (i.e. $0.b1$ is the midpoint in $[0.b, 0.(b+1)]$).

Exercise 4. The letters A,B, and C correspond to the intervals $[0, 0.1]$, $[0.1, 0.7]$, and $[0.7, 1]$, respectively. The number 1001101 corresponds to $2^{-1} + 2^{-4} + 2^{-5} + 2^{-7} = 0.6015625$. This lies in the second interval, so that the first letter is B.

1. We now apply the mapping

$$h_2(0.6015625) = (0.6015625 - 0.1)/(0.7 - 0.1) = 0.8359375,$$

   and this value lies in the third interval, so that the second letter is C.

2. We now apply the mapping

$$h_3(0.8359375) = (0.6015625 - 0.7)/(1 - 0.7) = 0.453125,$$

and this value lies in the second interval, so that the third letter is B.

3. We now apply the mapping

$$h_2(0.453125) = (0.453125 - 0.1)/(0.7 - 0.1) = 0.58854166666666,$$

and this value lies in the second interval, so that the forth letter is B.

4. We now apply the mapping

$$h_2(0.58854166666666) = (0.58854166666666 - 0.1)/(0.7 - 0.1) \\ = 0.81423611111111,$$

and this value lies in the third interval, so that the fifth letter is C.

5. We now apply the mapping

$$h_3(0.81423611111111) = (0.81423611111111 - 0.7)/(1 - 0.7) \\ = 0.380787037037043,$$

and this value lies in the second interval, so that the sixth letter is B.

6. We now apply the mapping

$$h_2(0.380787037037043) = (0.380787037037043 - 0.1)/(0.7 - 0.1) \\ = 0.467978395061738,$$

and this value lies in the second interval, so that the seventh letter is B.

7. We now apply the mapping

$$h_2(0.467978395061738) = (0.467978395061738 - 0.1)/(0.7 - 0.1) \\ = 0.613297325102897,$$

and this value lies in the second interval, so that the eighth letter is B.

8. We now apply the mapping

$$h_2(0.613297325102897) = (0.613297325102897 - 0.1)/(0.7 - 0.1) \\ = 0.855495541838162,$$

and this value lies in the third interval, so that the nineth letter is C.

9. We now apply the mapping

$$h_3(0.855495541838162) = (0.855495541838162 - 0.7)/(1 - 0.7) = 0.518318472793872,$$

and this value lies in the second interval, so that the tenth letter is B.

In summary, the text is BCBBCBBBCB.

Exercise 5. The first 99 A's restrict us to the interval $[0, 0.99^{99}]$. The last B restricts us to the interval $[0.99^{100}, 0.99^{99}]$. The arithmetic code is thus $0.99^{99}(1 + 0.99)/2 = 0.995 \times 0.99^{99} \approx 0.367880989461478$. We need $\lceil -\log_2(0.99^{99} 0.01) \rceil + 1 = 10$ bits for the arithmetic code.

- Clearly the first bit is 0.
- We compute $2 \times 0.367880989461478 \approx 0.735761978922956$, so that the second bit is 1.
- We then compute $2 \times 0.735761978922956 - 1 \approx 0.471523957845911$ so that the third bit is 0.
- We then compute $2 \times 0.471523957845911 = 0.943047915691822$ so that the fourth bit is 1.
- We then compute $2 \times 0.943047915691822 - 1 \approx 0.886095831383645$, so that the fifth bit is 1.
- We then compute $2 \times 0.886095831383645 - 1 \approx 0.772191662767289$, so that the sixth bit is 1.
- We then compute $2 \times 0.772191662767289 - 1 \approx 0.544383325534579$, so that the seventh bit is 1.
- We then compute $2 \times 0.544383325534579 - 1 \approx 0.088766651069157$, so that the eighth bit is 0.
- We then compute $2 \times 0.088766651069157 \approx 0.177533302138315$, so that the nineth bit is 0.
- Finally we compute $2 \times 0.177533302138315 \approx 0.355066604276630$, so that the final bit is 0.

The arithmetic code is thus 0101111000.

Section 7.6

Section 8.1

Section 8.2

Section 9.1

Exercise 3(a). $p''(a) = f''(a)$ gives that $2b_2 = f''(a)$, so that

$$b_2 = f''(a)/2.$$

$p'(a) = f'(a)$ then gives that $b_1 + 2b_2 a = f'(a)$, so that

$$b_1 = -2b_2 a + f'(a) = -f''(a)a + f'(a).$$

$p(a) = f(a)$ then gives that $b_0 + b_1 a + b_2 a^2 = f(a)$, so that

$$\begin{aligned} b_0 &= -b_1 a - b_2 a^2 + f(a) = -(-f''(a)a + f'(a))a - f''(a)a^2/2 + f(a) \\ &= f(a) - f'(a)a + \frac{f''(a)}{2}a^2. \end{aligned}$$

Exercise 3(b). The condition $p''(a) = f''(a)$ gives as before that $2b_2 = f''(a)$, so that $b_2 = f''(a)/2$. $p'(a) = f'(a)$ now gives that $b_1 = f'(a)$, while $p(a) = f(a)$ gives that $b_0 = f(a)$.

Section 9.2

Exercise 3(a). Setting $x = 0$ we get that $1 = c_0(-1)(-3)(-4)$, so that $c_0 = -1/12$.
Setting $x = 1$ we get that $0 = c_1(1)(1-3)(1-4)$, so that $c_1 = 0$.
Setting $x = 3$ we get that $2 = c_2(3)(3-1)(3-4)$, so that $c_2 = -1/3$.
Setting $x = 4$ we get that $1 = c_3(4)(4-1)(4-3)$, so that $c_3 = 1/12$.
We thus get that

$$p_3(x) = -\frac{(x-1)(x-3)(x-4)}{12} - \frac{x(x-1)(x-4)}{3} + \frac{x(x-1)(x-3)}{12}.$$

Exercise 3(b). Setting $p_3(x) = c_0 + c_1 x + c_2 x(x-1) + c_3 x(x-1)(x-3)$, we get the equations

$$\begin{aligned} 1 &= c_0 \\ 0 &= c_0 + c_1 \\ 2 &= c_0 + 3c_1 + 6c_2 \\ 1 &= c_0 + 4c_1 + 12c_2 + 12c_3. \end{aligned}$$

The first two equations give that $c_0 = 1$ and $c_1 = -1$. Putting this in to the third equation gives that $c_2 = (2-1+3)/6 = 2/3$. Putting this in to the fourth equation gives that $c_3 = (1-1+4-8)/12 = -1/3$. This gives

$$p_3(x) = 1 - x + \frac{2}{3}x(x-1) - \frac{1}{3}x(x-1)(x-3).$$

Exercise 4(a). Since $p_1(x_0) = p_2(x_0) = f(x_0)$, $p_1(x_1) = p_2(x_1) = f(x_1)$, $p_1(x_2) = p_2(x_2) = f(x_2)$, we get that

$$\begin{aligned}
p(x_0) &= p_2(x_0) - p_1(x_0) = f(x_0) - f(x_0) = 0\\
p(x_1) &= p_2(x_1) - p_1(x_1) = f(x_1) - f(x_1) = 0\\
p(x_2) &= p_2(x_2) - p_1(x_2) = f(x_2) - f(x_2) = 0.
\end{aligned}$$

Therefore, the values at the interpolation points are 0.

Exercise 4(b). The second degree polynomial p has three distinct zeros, but a non-zero second degree polynomial has at most two zeros. We therefore must have that $p = 0$, so that $p_1 = p_2$, so that the interpolating polynomial is unique.

Exercise 4(c). Defining $p = p_1 - p_2$ as above, we see as before that $p(x_0) = p(x_1) = \ldots = p(x_n) = 0$, so that p is a polynomial of degree n with $n+1$ distinct zeros. From the fundamental theorem of algebra (i.e. that a nonzero polynomial of degree n has at most n roots) it follows that $p = 0$, so that $p_1 = p_2$, so that the interpolating polynomial is unique also in this case.

Exercise 5(a). The Newton form of the quadratic, interpolating polynomial is

$$p_2(x) = f(x_0) + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1).$$

We have that

$$\begin{aligned}
f[x_0, x_1] &= \frac{f(x_1) - f(x_0)}{x_1 - x_0} = \frac{1-2}{1-0} = -1\\
f[x_1, x_2] &= \frac{f(x_2) - f(x_1)}{x_2 - x_1} = \frac{0-1}{2-1} = -1\\
f[x_0, x_1, x_2] &= \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0} = \frac{-1-(-1)}{2-0} = 0.
\end{aligned}$$

Inserting this in $p_2(x)$ we get that $p_2(x) = 2 - (x - x_0) = 2 - x$.

Exercise 5(b). Both f and p_n are interpolating polynomials of degree n, and we know then from 4 that they must be equal. In (a) the function was already a second degree polynomial, so that it must be equal to its interpolant too. This is why we obtained that p_s equaled the function itself.

Section 10.2

Exercise 2(a). The zeros of f can be found to be $1,2,3$, and f changes sign at each of these. Clearly $f(0)<0$ and $f(3.5)>0$. The first midpoint chosen by the bisection method is $(0+3.5)/2=1.75$. Since $f(1.75)>0$, the first iteration of the bisection method restricts to the interval $[0,1.75]$. Within this interval 1 is the only zero, so that the bisection method is guaranteed to find this zero. The last alternative is therefore correct.

Exercise 2(b). $\cos x$ has the zeros $\pi/2 \approx 1.57$, $3\pi/2 \approx 4.71$, and $5\pi/2 \approx 7.85$ on $[0,10]$. We have that $\cos(0)>0$, $\cos(10)<0$, and $\cos 5>0$, so that the first iteration of the bisection method restricts to the interval $[5,10]$. $5\pi/2$ is the only zero of f within this interval, so that the bisection method will find this zero. The third alternative is therefore correct.

Exercise 2. If we did no iterations, the error would be bounded by $1/2$. Each iteration cuts the error by one third, so that the error after N iterations is bounded by $0.5\cdot 3^{-N}$. If we want this to be less than 10^{-12}, we must have that $0.5\cdot 3^{-N}\leq 10^{-12}$, so that $3^N \geq 0.5\cdot 10^{12}$, so that $N\geq \log(0.5\cdot 10^{12})/\log 3 \approx 24.52$. We this need to take at least 25 iterations.

Exercise 3. The following code can be used:

```
from math import cos
from zerofinding import bisection

def f(x):
  return x-cos(x)

bisection(f,0,1,10**(-20),10) #a
bisection(f,0,1,10**(-10),60) #b
bisection(f,0,1,10**(-20),60) #d
```

Exercise 5. The following code can be used:

```
from math import sin,pi
from zerofinding import bisection

def f(x):
    return sin(x)

bisection(f,-1,20,10**(-10),60,4*pi)
```

Exercise 6. The following code can be used:

```
from math import sqrt
from zerofinding import bisection

def f(x):
    return x**2-2

bisection(f,1,1.5,10**(-10),10) #a
bisection(f,1,1.5,10**(-10),10, sqrt(2)) #b
```

Section 10.3

Exercise 3. The following code can be used

```
from math import log,exp,sqrt
from zerofinding import secant

def fa(x):
    return x**2-3

def fb(x):
    return x**12-2

def fc(x):
    return log(x)-1

secant(fa,0.1,1,10**(-15),60,sqrt(3)) #a
secant(fb,0.1,1,10**(-15),60,2**(1.0/12)) #b
secant(fc,0.1,1,10**(-15),60,exp(1)) #c
```

Exercise 5. For the function $f(x) = (x-1)^3$ the secant method gives

$$\begin{aligned} x_n &= x_{n-1} - \frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})} f(x_{n-1}) \\ &= x_{n-1} - \frac{x_{n-1} - x_{n-2}}{(x_{n-1}-1)^3 - (x_{n-2}-1)^3}(x_{n-1}-1)^3 \\ &= x_{n-1} - \frac{x_{n-1} - x_{n-2}}{(x_{n-1}-1)^3 - (x_{n-2}-1)^3}(x_{n-1}-1)^3 \end{aligned}$$

The following code can be used in this exercise:

```
from zerofinding import secant

def f(x):
    return (x-1)**3

secant(f,0.5,1.2,10**(-15),7,1)
```

Exercise 5(a). The first 7 iterations give

$$1.15789473684$$
$$1.11710677382$$
$$1.08899801740$$
$$1.06700831862$$
$$1.05063360476$$
$$1.03820748989$$
$$1.02884624587$$

Exercise 5(b). We see that we still have a deviation bigger than 10^{-2} after 7 iterations, so that we don't seem to get 62% correct digits per iteration, as promised by the error estimates for the secant method (we should at least obtain one new correct digit for every second iteration). This example therefore does not agree with Observation 10.15. But notice that $f'(1) = 0$, so that we cannot find a γ as demanded by Theorem 10.14. This is the reason why we do not observe the convergence speed noted in Observation 10.15.

Section 10.4

Exercise 4. The following code can be used

```
from math import sin,cos,pi
from zerofinding import bisection,secant,newton

def f(x):
    return sin(x)

def df(x):
    return cos(x)

print('Bisection method')
```

```
bisection(f,3,4,10**(-20),10,pi) #a
print('Secant method')
secant(f,4,3,10**(-20),10,pi) #b
print('Newton method')
newton(f,df,3,10**(-20),10,pi) #c
```

Exercise 5. The following code can be used

```
from zerofinding import bisection,secant,newton

def f(x):
    return (x-10.0/3.0)**5

def df(x):
    return 5.0*(x-10.0/3.0)**4

print('Bisection method')
bisection(f,3,4,10**(-20),10,10.0/3) #a
print('Secant method')
secant(f,4,3,10**(-20),10,10.0/3) #b
print('Newton method')
newton(f,df,3,10**(-20),10,10.0/3) #c
```

Exercise 6(a). We have that $f'(x) = 2x$, so that the Newton iteration takes the form

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{(x_n)^2 - 2}{2x_n} = \frac{x_n^2 + 2}{2x_n}.$$

if we subtract $\sqrt{2}$ and substitute $e_n = x_n - \sqrt{2}$ on both sides we obtain

$$e_{n+1} = \frac{x_n^2 + 2}{2x_n} - \sqrt{2} = \frac{x_n^2 + 2 - 2\sqrt{2}x_n}{2x_n} = \frac{(x_n - \sqrt{2})^2}{2x_n} = \frac{e_n^2}{2x_n}$$

Exercise 6(b). The secant method is

$$\begin{aligned} x_n &= x_{n-1} - \frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})} f(x_{n-1}) \\ &= x_{n-1} - \frac{x_{n-1} - x_{n-2}}{(x_{n-1})^2 - (x_{n-2})^2}((x_{n-1})^2 - 2) = x_{n-1} - \frac{(x_{n-1})^2 - 2}{x_{n-1} + x_{n-2}}. \end{aligned}$$

If we subtract $\sqrt{2}$ on both sides we can write this as

$$\begin{aligned}
e_n &= e_{n-1} - \frac{(x_{n-1}+\sqrt{2})(x_{n-1}-\sqrt{2})}{x_{n-1}+x_{n-2}}\\
&= e_{n-1} - e_{n-1}\frac{x_{n-1}+\sqrt{2}}{x_{n-1}+x_{n-2}} = e_{n-1}\left(1-\frac{x_{n-1}+\sqrt{2}}{x_{n-1}+x_{n-2})}\right)\\
&= e_{n-1}\frac{x_{n-1}+x_{n-2}-x_{n-1}-\sqrt{2}}{x_{n-1}+x_{n-2}} = e_{n-1}\frac{x_{n-2}-\sqrt{2}}{x_{n-1}+x_{n-2}}\\
&= \frac{e_{n-1}e_{n-2}}{x_{n-1}+x_{n-2}}.
\end{aligned}$$

Exercise 7(a). We have that $f'(x) = -1/x^2$, so that the Newton iteration is

$$\begin{aligned}
x_{n+1} &= x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{1/x_n - R}{-1/(x_n)^2}\\
&= x_n + x_n - R(x_n)^2 = x_n(2-Rx_n).
\end{aligned}$$

In this formula we do not need to perform division, only multiplication. By iterating this formula we can therefore approximate $1/R$ by only applying multiplications.

Section 11.1

Section 11.2

Exercise 2(a). The first order Taylor polynomial with remainder can be written

$$f(a+h) = f(a) + f'(a)h + f''(c)h^2/2,$$

where c is a number between a and $a+h$. This can also be written as

$$\frac{f(a+h)-f(a)}{h} - f'(a) = f''(c)h/2.$$

Since $f''(c) = -\cos c$, we have that the right side is less than $h/2$ in absolute value, so that $|(f(a+h)-f(a))/h - f'(a)| \leq h/2$, so that the last alternative is correct.

Exercise 2(b). The total error is

$$f'(a) - \frac{\overline{f(a+h)} - \overline{f(a)}}{h}.$$

Equation (11.11) said that $\overline{f(a)} = f(a)(1+\epsilon_1)$ and $\overline{f(a+h)} = f(a+h)(1+\epsilon_2)$, so that

$$\begin{aligned} f'(a) - \frac{\overline{f(a+h)} - \overline{f(a)}}{h} &= f'(a) - \frac{f(a+h)(1+\epsilon_2) - f(a)(1+\epsilon_1)}{h} \\ &= f'(a) - \frac{f(a+h)-f(a)}{h} - \frac{f(a+h)\epsilon_2 - f(a)\epsilon_1}{h} \end{aligned}$$

Taking absolute values and using the triangle equality, we see that

$$\left| f'(a) - \frac{\overline{f(a+h)} - \overline{f(a)}}{h} \right| \le \frac{h}{2} \max_{x\in[a,a+h]} |f''(x)| + \frac{2\epsilon^*}{h} \max_{x\in[a,a+h]} |f(x)|,$$

so that the last alternative is correct.

Exercise 3(a). On my computer 10^{-8} is the power of 10 which gives the least error in the approximation. This can be obtained by running the following program:

```
from math import *

for p in range(4,15):
  h=10.0**(-p)
  approx=(exp(1+h)-exp(1))/h
  print(p, approx, abs((approx-exp(1))/exp(1)))
```

Exercise 3(b). If we use the values $\epsilon^* = 7 \times 10^{-17}$ from Example 11.14, then Lemma 11.13 gives the optimal h $h^* = 2\sqrt{\epsilon^*} \approx 1.6733 \times 10^{-8}$ (terms cancel since $f(a) = f''(a)$).

Exercise 4(a). We now write

$$f(a+h) = f(a) + hf'(a) + \frac{h^2}{2} f''(a) + \frac{h^3}{6} f'''(\xi_h)$$

for some ξ_h in the interval between a and $a+h$. We can now write

$$f'(a) - \frac{f(a+h)-f(a)}{h} = -\frac{h}{2} f''(a) - \frac{h^2}{6} f'''(\xi_h).$$

After taking absolute values, the error estimate becomes

$$\frac{h}{2}|f''(a)| + \frac{h^2}{6} \max_{x\in[a,a+h]} |f'''(x)|.$$

This error estimate is similar to the one we obtained when we used a first degree Taylor expansion, but we also need a general bound on the third derivative. we still need to bound the second derivative, but only at the point a.

Exercise 4(b). We now write $f(a+h) = f(a) + f'(\xi_h)h$ for some ξ_h in the interval between a and $a+h$. In this expression $f'(a)$ is not present, so it is not possible to obtain an estimate of the truncation error using this Taylor expansion.

Exercise 4(c). The linear Taylor polynomial is the best because it is is the shortest possible Taylor expansion which can give an estimate of $f'(a)$ (as we showed in (b)), and also the one which gives the simplest expression for the truncation error in that it does not depend on any derivatives higher than the second order (as we showed in (a)).

Section 11.3

Exercise 1. The Newton form of the interpolating polynomial is

$$p_2(x) = f(a) + f[a,a+h](x-a) + f[a,a+h,a+2h](x-a)(x-(a+h)).$$

We compute that $p_2'(a) = f[a,a+h] - f[a,a+h,a+2h]h$. The divided differences can be computed as

$$\begin{aligned} f[a,a+h] &= \frac{f(a+h)-f(a)}{h} \\ f[a+h,a+2h] &= \frac{f(a+2h)-f(a+h)}{h} \\ f[a,a+h,a+2h] &= \frac{f(a+2h)-2f(a+h)+f(a)}{2h^2}. \end{aligned}$$

The approximation is thus

$$\begin{aligned} f'(a) \approx p_2'(a) &= f[a,a+h] - f[a,a+h,a+2h]h \\ &= \frac{f(a+h)-f(a)}{h} - \frac{f(a+2h)-2f(a+h)+f(a)}{2h} \\ &= -\frac{f(a+2h)-4f(a+h)+3f(a)}{2h} \end{aligned}$$

Section 11.4

Exercise 2. The code can look as follows

```
from math import exp

for p in range(3,15):
  h=10.0**(-p)
  approx=(exp(1+h)-exp(1-h))/(2*h)
  print(p, approx, abs((approx-exp(1))/exp(1)))
```

Exercise 4(b). We can plot the curve together with the secants as follows with Python:

```
from numpy import *
import matplotlib.pyplot as plt

x=arange(0,6,0.05,float)
plt.plot(x,(-x**2+10*x-5)/4)
plt.plot([1,3],[(-1**2+10*1-5)/4,(-3**2+10*3-5)/4])
plt.plot([1,5],[(-1**2+10*1-5)/4,(-5**2+10*5-5)/4])
plt.plot([3,5],[(-3**2+10*3-5)/4,(-5**2+10*5-5)/4])
plt.show()
```

Exercise 7(a). On my computer 10^{-4} is the power of 10 which gives the least error in the approximation. This can be tested by running the following program:

```
from math import *

for p in range(15):
  h=10.0**(-p)
  print(p, abs((exp(1-h)-2*exp(1)+exp(1+h))/h**2-exp(1)))
```

Exercise 7(b). If we use the value $\epsilon^* = 7\times 10^{-17}$ from Example 11.14 then Observation **??** gives the optimal choice of h $h^* = \sqrt[4]{36\epsilon^*} \approx 2.2405\times 10^{-4}$ (terms cancel since $f(a) = f^{(4)}(a)$).

Exercise 8(a). If the approximation method $f'(a) \approx c_1 f(a-h) + c_2 f(a+h)$ is to be exact for $f(x) = 1$, we must have that $0 = c_1 + c_2$, since $f(a-h) = f(a+h) = 1$, and since $f'(x) = 0$. Therefore we must have that $c_2 = -c_1$.

If the method is to be exact for $f(x) = x$ we must in the same way have that

$$1 = c_1(a-h) + c_2(a+h) = c_1(a-h) - c_1(a+h) = -2c_1 h,$$

so that $c_1 = -\frac{1}{2h}$,so that also $c_2 = \frac{1}{2h}$. The method therefore becomes $-\frac{1}{2h} f(a-h) + \frac{1}{2h} f(a+h) = \frac{f(a+h)-f(a-h)}{2h}$

Exercise 8(b). If $f(x) = cx + d$ we have that $f'(x) = c$, and the method takes the form

$$\frac{f(a+h)-f(a-h)}{2h} = \frac{c(a+h)+d-(c(a-h)+d)}{2h} = \frac{2ch}{2h} = c,$$

so that the method is exact for all polynomials of degree ≤ 1.We see that the method coincides with the symmetric Newton-method for differentiation, and it therefore has an error of order $\frac{1}{h^2}$, which is better than the Newton's quotient (which has an error of order $\frac{1}{h}$). It is worse than the four point method for numerical differentiation,which has order $\frac{1}{h^4}$.

Here it also should have been mentioned that the method also is exact for polynomials of degree ≤ 2 (also see 4). There are several ways to see this. First, the error estimate from Section 11.4 uses $f^{(3)}(x)$, and since all second degree polynomials have a third derivative equal to 0, the error must be zero. One could also as above substitute $f(x) = x^2$ into the formula:

$$\frac{f(a+h) - f(a-h)}{2h} = \frac{(a+h)^2 - (a-h)^2}{2h} = \frac{4ah}{2h} = 2a,$$

which also is $f'(a)$. Finally, the symmetric Newton quotient was defined as the derivativee at a of the unique parabola interpolating f at $a-h, a$, and $a+h$. If f itself is a parabola it is equal to this interpolant since it is unique, so that the symmetric Newton quotient must return the derivative.

Exercise 8(c). If the approximation method $f''(a) \approx c_1 f(a-h) + c_2 f(a) + c_3 f(a+h)$ is exact for $f(x) = 1$, we must have that $0 = c_1 + c_2 + c_3$. If it is exact for $f(x) = x$ we must have that

$$0 = c_1(a-h) + c_2 a + c_3(a+h) = a(c_1 + c_2 + c_3) + h(-c_1 + c_3) = h(-c_1 + c_3),$$

which gives that $c_1 = c_3$. If it is exact for $f(x) = x^2$ we must have that

$$\begin{aligned} 2 &= c_1(a-h)^2 + c_2 a^2 + c_3(a+h)^2 \\ &= a^2(c_1 + c_2 + c_3) - 2ahc_1 + 2ahc_3 + h^2(c_1 + c_3) = 2c_1 h^2, \end{aligned}$$

which gives that $c_1 = \frac{1}{h^2}$. We therefore also get that $c_3 = \frac{1}{h^2}$, and that $c_2 = -c_1 - c_2 = -\frac{2}{h^2}$, so that the method becomes

$$\frac{1}{2h} f(a-h) - \frac{1}{h} f(a) + \frac{1}{2h} f(a+h) = \frac{f(a-h) - 2f(a) + f(a+h)}{h^2}.$$

We see that this coincides with the already seen three point method to compute the second derivative in this section.

Exercise 8(d). All third degree polynomials have a fourth derivative equal to 0, and therefore the truncation error becomes 0 ($M_1 = 0$ in Theorem 11.19). Alter-

natively we can substitute $f(x) = x^3$ into the formula:

$$\begin{aligned}
&\frac{f(a-h) - 2f(a) + f(a+h)}{2h} \\
&= \frac{(a-h)^3 - 2a^3 + (a+h)^3}{h^2} \\
&= \frac{a^3 - 3a^2h + 3ah^2 - h^3 - 2a^3 + a^3 + 3a^2h + 3ah^2 + h^3}{h^2} \\
&= \frac{6ah^2}{h^2} = 6a,
\end{aligned}$$

which coincides with the second derivative of f in a.

Exercise 9. We see that the coefficients in the new approximation are taken from row 2 of Pascal's triangle with alternating sign. This means that also this approximation reduces bass. Since the values are taken from a higher row in Pascal's triangle, one is lead to believe that it reduces more bass than the Newton difference quotient.

Exercise 11(a). On my computer 10^{-3} is the power of 10 which gives the least error in the approximation. This can be tested by running the following program:

```
from math import *

for p in range(15):
  h=10.0**(-p)
  print(p, abs((exp(1-2*h)-8*exp(1-h)+8*exp(1+h) - exp(1+2*h))/(12*h)-exp(1)))
```

Exercise 11(b). If we use the value $\epsilon^* = 7 \times 10^{-17}$ from Example 11.14 then the relation (11.28) gives the optimal h $h^* = \sqrt[5]{\frac{27\epsilon^*}{2}} \approx 9.8875 \times 10^{-4}$ (terms cancel since $f(a) = f^{(5)}(a)$)

Section 12.1

Section 12.2

Exercise 2. We get that

$$I_{mid} = \frac{1}{2}f(1/4) + \frac{1}{2}f(3/4) = \frac{1}{2}\frac{1}{16} + \frac{1}{2}\frac{9}{16} = \frac{1}{2}\frac{10}{16} = \frac{5}{16},$$

so that the first alternative is correct.

Exercise 3. The interval $[0,\pi/2]$ is here split into the subintervals

$$[0,\pi/12],\ [\pi/12,2\pi/12],\ [2\pi/12,3\pi/12],\ [3\pi/12,4\pi/12],\ [4\pi/12,5\pi/12],\ [5\pi/12,6\pi/12].$$

The midpoints on these intervals are $\pi/24$, $3\pi/24$, $5\pi/24$, $7\pi/24$, $9\pi/24$, $11\pi/24$. With $h=\pi/12$ the mid-point method yields

$$\begin{aligned}
I_{mid} &= \frac{\pi}{12}\left(f(\pi/24)+f(3\pi/24)+f(5\pi/24)+f(7\pi/24)+f(9\pi/24)+f(11\pi/24)\right)\\
&= \frac{\pi}{12}\left(\frac{\sin(\pi/24)}{1+(\pi/24)^2}+\frac{\sin(3\pi/24)}{1+(3\pi/24)^2}+\frac{\sin(5\pi/24)}{1+(5\pi/24)^2}+\frac{\sin(7\pi/24)}{1+(7\pi/24)^2}\right.\\
&\quad\left.+\frac{\sin(9\pi/24)}{1+(9\pi/24)^2}+\frac{\sin(11\pi/24)}{1+(11\pi/24)^2}\right)\\
&\approx 0.530624.
\end{aligned}$$

Exercise 4. The code can look like this

```
from integration import midpointn,midpoint
from math import exp

def f(x):
  return exp(x)
print(midpointn(f,0,1,10)) #a
print(midpoint(f,0,1,10**(-10),60,exp(1)-1)) #b
```

Exercise 4(b). From Theorem 12.8 we know that the error is bounded by $(b-a)\frac{h^2}{24}\max_{x\in[a,b]}\left|f''(x)\right|$. Since $f''(x)=e^x$, $\max_{x\in[0,1]}\left|f''(x)\right|=e$, so that this is $\frac{eh^2}{24}$. We must thus ensure that $\frac{eh^2}{24}<10^{-10}$, so that $h<\sqrt{\frac{24}{e}}10^{-5}\approx 2.97\times 10^{-5}$.

Section 12.3

Exercise 2. We have that

$$I_{trap}=\frac{1}{2}\left(\frac{f(0)+f(1)}{2}+f\left(\frac{1}{2}\right)\right)=\frac{1}{2}\left(\frac{0+1}{2}+\frac{1}{4}\right)=\frac{3}{8},$$

so that the second alternative is correct.

Exercise 4. The code can look like this

```
from integration import trapezoidaln,trapezoidal
from math import exp

def f(x):
    return exp(x)
print(trapezoidaln(f,0,1,10)) #a
print(trapezoidal(f,0,1,10**(-10),60,exp(1)-1)) #b
```

Exercise 4(b). The error is bounded by

$$(b-a)\frac{h^2}{12}\max_{x\in[0,1]}|f''(x)| = \frac{h^2}{12}e.$$

We have that $\frac{h^2}{12}e \le 10^{-10}$ when $h \le \sqrt{12/e}\times 10^{-5} \approx 2.1\times 10^{-5}$.

Exercise 9(a). Since the error in the trapezoidal rule is bounded by

$$(b-a)\frac{h^2}{12}\max_{x\in[a,b]}|f''(x)|,$$

and since $f''(x) = \frac{8}{(1+2x)^3}$ (which attains maximum absolute value for $x = 0$), we need to choose h so that

$$(b-a)\frac{h^2}{12}\max_{x\in[a,b]}|f''(x)| = 8\frac{h^2}{12} \le 10^{-10},$$

which gives that $h \le 10^{-5}\sqrt{3/2}$. The number of function evaluations is then $\approx 1/h = 10^5/\sqrt{2/3} \approx 81649.66$, which means that at least 81650 evaluations are needed.

Exercise 9(b). Since the error in the midpoint rule is bounded by

$$(b-a)\frac{h^2}{24}\max_{x\in[a,b]}|f''(x)|,$$

we obtain as in (a) that

$$(b-a)\frac{h^2}{24}\max_{x\in[a,b]}|f''(x)| = 2\frac{h^2}{6} \le 10^{-10},$$

which gives that $h \le 10^{-5}\sqrt{3}$. The number of function evaluations is therefore roughly $1/h = 10^5/\sqrt{3} \approx 57735.03$. In other words, at least 57736 evaluations are needed.

Exercise 9(c). Since the error in Simpson's rule is bounded by

$$(b-a)\frac{h^4}{180}\max_{x\in[a,b]}|f^{(iv)}(x)|,$$

and since $f^{(iv)}(x) = \frac{384}{(1+2x)^5}$ (which attains maximum absolute value for $x = 0$), we need to choose h so that

$$(b-a)\frac{h^4}{180}\max_{x\in[a,b]}|f^{(iv)}(x)| = 384\frac{h^4}{180} \le 10^{-10},$$

which mens that h is bounded by $h \le (180\cdot 10^{-10}/384)^{1/4}$. The number of function evaluations is then $\approx 1/h = (180\cdot 10^{-10}/384)^{-1/4} \approx 382.17$, which means that at least 383 evaluations are needed.

Exercise 10. The code can look like this

```
from integration import simpsonn,simpson
from math import exp

def f(x):
    return exp(x)
print(simpsonn(f,0,1,10)) #a
print(simpson(f,0,1,10**(-10),60,exp(1)-1)) #b
```

Exercise 11(a). It is enough to verify Simpson's rule on each interval. The integrals become

$$\int_{a-h}^{a+h} x^3\,dx = \frac{1}{4}((a+h)^4-(a-h)^4) = \frac{1}{4}(8a^3h+8ah^3) = 2a^3h+2ah^3$$
$$\int_{a-h}^{a+h} x^2\,dx = \frac{1}{3}((a+h)^3-(a-h)^3) = \frac{1}{3}(6a^2h+2h^3) = \frac{h}{3}(6a^2+2h^2)$$
$$\int_{a-h}^{a+h} x\,dx = \frac{1}{2}((a+h)^2-(a-h)^2) = \frac{1}{2}4ah = 2ah$$
$$\int_{a-h}^{a+h} dx = 2h.$$

We also get that the rule itself becomes

$$\frac{h}{3}((a-h)^3+4a^3+(a+h)^3)=\frac{h}{3}(6a^3+6ah^2)=2a^3h+2ah^3$$
$$\frac{h}{3}((a-h)^2+4a^2+(a+h)^2)=\frac{h}{3}(6a^2+2h^2)$$
$$\frac{h}{3}((a-h)+4a+(a+h))=2ah$$
$$\frac{h}{3}(1+4+1)=2h.$$

This shows that Simpsons' rule is exact for the given polynomials.

Exercise 11(b). For $f(x)=bx^3+cx^2+dx+e$ the integral is

$$\int_{a-h}^{a+h} f(x)dx = b\int_{a-h}^{a+h} x^3dx + c\int_{a-h}^{a+h} x^2dx + d\int_{a-h}^{a+h} xdx + e\int_{a-h}^{a+h} 1dx.$$

The rule itself now gives

$$\begin{aligned}&\frac{h}{3}(f(a-h)+4f(a)+f(a+h))\\&=b\frac{h}{3}((a-h)^3+4a^3+(a+h)^3)+c\frac{h}{3}((a-h)^2+4a^2+(a+h)^2)\\&+d\frac{h}{3}((a-h)+4a+(a+h))+e\frac{h}{3}(1+4+1)\end{aligned}$$

We see that equality follows from that we have equality for $x^3, x^2, x, 1$.

Exercise 11(c). Since $f^{(4)}(x)=0$ for all x for every third degree polynomial, it follows directly from the error estimate that the method is exact for such functions.

Section 13.1

Section 13.2

Exercise 2. The differential equation is separable since it can be written as $\frac{x'}{1-x}=\sin t$. This gives that $-\ln|1-x|=-\cos t+C$, so that $1-x=De^{\cos t}$, so that $x(t)=1-De^{\cos t}$.

Exercise 2(a). The solution in this case is $x(t)=1-e^{\cos t}$.

Exercise 2(b). Inserting $x(4)=1$ we obtain that $1=1-De^{\cos 4}$, so that $D=0$, so that $x(t)=1$.

Exercise 2(c). Setting $t=\pi/2$ we obtain $2=1-D$, so that $D=-1$, so that $x(t)=1+e^{\cos t}$.

Exercise 2(d). The solution in this case is $x(t)=1+2e^{\cos t}$.

Section 13.3

Exercise 3(a). We get that

$$
\begin{aligned}
x_1 &= x_0 + hf(t_0, x_0) = 1 + 0.1f(0,1) = 1 + 0.1(0+1) = 1.1 \\
x_2 &= x_1 + hf(t_1, x_1) = 1.1 + 0.1f(0.1, 1.1) = 1.1 + 0.1(0.1 + 1.1) = 1.22 \\
x_3 &= x_2 + hf(t_2, x_2) = 1.22 + 0.1f(0.2, 1.22) = 1.22 + 0.1(0.2 + 1.22) = 1.362.
\end{aligned}
$$

Exercise 4. The module `differentialeqs.py` contains a general implementation of Euler's method in Python.

Exercise 5. The code can be changed as follows:

```
h = (b-a)/float(n)
t[0] = b
for k in range(n):
    x[k+1] = x[k] - h*f(t[k],x[k])
    t[k+1] = b - (k + 1)*h
```

Here we simply have used Euler's method with negative step size.

Exercise 6. If we insert $x'(t) = f(t, x(t))$ in the approximation we find that

$$
f(t,x) \approx \frac{x(t+h) - x(t)}{h}.
$$

This can be written $x(t+h) - x(t) \approx hf(t, x(t))$, which means that $x(t+h) \approx x(t) + hf(t, x(t))$. The right hand side here is equivalent to one step with Euler's method.

Exercise 8. If the step length is h, we obtain the approximation

$$
x(h) \approx x(0) + hf(t,x) = 1 + h \sin h.
$$

The error is given by

$$
R_1(h) = \frac{h^2}{2} x''(\xi)
$$

where $\xi \in (0, h)$. Since $x'(t) = \sin x(t)$, we have

$$
x''(t) = x'(t) \cos x(t) = \sin x(t) \cos x(t) = \frac{\sin\bigl(2x(t)\bigr)}{2}
$$

We therefore have $|x''(t)| \le 1/2$, so

$$
|R_1(h)| \le \frac{h^2}{4}.
$$

Exercise 9. The code can look as follows:

```
from differentialeqs import quadratictaylor

def f(t,x):
    return t**2+x**3-x

def df(t,x):
    return 2.0*t+(3.0*x**2-1)*f(t,x)

print('b')
for n in [1,2,5]:
    print(quadratictaylor(f,df,0,1,1,n))

print('c')
for n in [10,100,1000]:
    print(quadratictaylor(f,df,0,1,1,n))
```

Exercise 9(a). We get that

$$x''(t) = 2t + 3x^2 x'(t) - x'(t) = 2t + (3x^2 - 1)x'(t),$$

where we substitute $t^2 + x^3 - x$ for x'.

Exercise 9(b). One step with the quadratic Taylor method here becomes

$$x_{k+1} = x_k + hx'_k + \frac{h^2}{2}(2t_k + (3x_k^2 - 1)x'_k)$$

where $x'_k = t_k^2 + x_k^3 - x_k$. Since $x'_0 = 0 + 1 - 1 = 0$, one step with the quadratic Taylor method therefore gives $x_1 = 1$. The first step with the quadratic Taylor method clearly gives $x_1 = 1$, no matter what h is. If we use more steps the next step becomes

$$x'_1 = t_1^2 + x_1^3 - x_1 = t_1^2$$
$$x_2 = x_1 + ht_1^2 + \frac{h^2}{2}(2t_1 + (3x_1^2 - 1)t_1^2) = 1 + ht_1^2 + \frac{h^2}{2}(2t_1 + 2t_1^2)$$

If $h = 0.5$ we get that $x_2 = 1 + \frac{1}{8} + \frac{1}{8}(1 + \frac{1}{2}) = 1 + \frac{1}{8} + \frac{1}{8} + \frac{1}{16} = \frac{21}{16} \approx 1.3125$. If $h = 0.2$ we get in the same way that the next steps give $x_2 = 1.0176$, $x_3 = 1.08109021746$, $x_4 = 1.24076835064$, $x_5 = 1.62941067817$.

Exercise 9(c). If we run the code we get for $N = 10, 100, 1000$ the approximations $x(1) \approx 1.787456775$, $x(1) \approx 1.90739098078$, and $x(1) \approx 1.9095983769$, respectively.

Section 13.4

Exercise 2. The code looks as follows if we use the fourth order Runge Kutta method in the last part of the exercise:

```
from differentialeqs import *

def f(t,x):
    return x

print('a')
print(euler(f,0,1,1,1))
print('c')
print(eulermidpoint(f,0,1,1,1))
print('d')
print(rungekutta4(f,0,1,1,1))
print('e')
print(rungekutta4(f,0,1,1,2))

print('f')
for N in [10,100,1000,10000]:
    x = rungekutta4(f,0,1,1,N)
    print(x[-1:-6:-1]) # Print the last 5 estimates
```

Exercise 2(a). One step with Euler's method gives

$$x_1 = x_0 + x_0 = 2.$$

Exercise 2(b). One step with Euler's midpoint method gives

$$\begin{aligned} x_{1/2} &= x_0 + \frac{1}{2}x_0 = \frac{3}{2} \\ x_1 &= x_0 + x_{1/2} = \frac{5}{2} = 2.5. \end{aligned}$$

Exercise 2(c). With Runge-Kuttas method we get

$$k_0 = 1 \qquad k_1 = 1 + \frac{k_0}{2} = 1.5 \qquad k_2 = 1 + \frac{k_1}{2} = 1.75 \qquad k_3 = 1 + 1.75 = 2.75$$

$$x_1 = x_0 + \frac{1}{6}(k_0 + 2k_1 + 2k_2 + k_3) = 1 + \frac{1}{6}(1 + 3 + 3.5 + 2.75) \approx 2.7083.$$

Exercise 2(d). With two steps with Runge-Kuttas method we get

$$k_0 = 1 \quad k_1 = 1 + \frac{k_0}{4} = 1.25 \quad k_2 = 1 + \frac{k_1}{4} = 1.3125 \quad k_3 = 1 + 0.60625 = 1.6025$$

$$x_1 = x_0 + \frac{1}{12}(k_0 + 2k_1 + 2k_2 + k_3) \approx 1.6440$$

$$k_0 = x_1 = 1.6440 \qquad k_1 = x_1 + \frac{k_0}{4} \qquad k_2 = x_1 + \frac{k_1}{4} \qquad k_3 = x_1 + \frac{k_2}{2}$$

$$x_2 = x_1 + \frac{1}{12}(k_0 + 2k_1 + 2k_2 + k_3) \approx 2.7100$$

Exercise 2(e). If we use the fourth order Runge Kutta method we get he approximation 2.71827974414. If we change N to 100, 1000, 10000 we get 2.71828182823, 2.71828182846, and 2.71828182846.

Exercise 2(f). It looks like the values converge to e.

Exercise 2(g). The equation is of first order with linear coefficients, and we see that the solution is $x(t) = e^t$, so that $x(1) = e$.

Exercise 3. The code can look as follows:

```
from math import *
from differentialeqs import euler,eulermidpoint

def f(t,x):
    return -x*sin(t)+sin(t)

print('a')
print(euler(f,0,2*pi,2+exp(1),1))
print(euler(f,0,2*pi,2+exp(1),2))
print(euler(f,0,2*pi,2+exp(1),5))
print(euler(f,0,2*pi,2+exp(1),10))
print(euler(f,0,2*pi,2+exp(1),1000))

print('b')
print(eulermidpoint(f,0,2*pi,2+exp(1),1))
print(eulermidpoint(f,0,2*pi,2+exp(1),5))
print(eulermidpoint(f,0,2*pi,2+exp(1),1000))
```

Exercise 3(a). Euler's method here takes the form $x_{k+1} = x_k + h(-x_k \sin t_k + \sin t_k)$.

Exercise 3(b). Euler's midpoint method here takes the form

$$x_{k+1/2} = x_k + \frac{h}{2}(-x_k \sin t_k + \sin t_k)$$
$$x_1 = x_k + h(-x_{k+1/2} \sin(t_k + h/2) + \sin(t_k + h/2))$$

Exercise 3(c). This differential equation is separable, and one can show that the solution is on the form $1 + De^{\cos(t)}$. In particular, the solution is periodic with period 2π, so that the solution should satisfy $x(2\pi) = 2 + e$. If we run the code for different N you will see that it is first for N larger than 1000 that we begin to get close, so that we can not say anything for N as small as those given in the exercise.

Exercise 4(a). One step with Euler's method gives that

$$x_{n+1} = x_n - h\lambda x_n = (1 - h\lambda)x_n.$$

Clearly then $x_n = (1 - h\lambda)^n$. This remains bounded when $-1 \le 1 - h\lambda \le 1$, i.e. when $0 \le h\lambda \le 2$, i.e. when $0 \le h \le 2/\lambda$ (when we assume that $\lambda > 0$).

Exercise 4(b). One step with Euler's midpoint method gives first

$$x_{n+1/2} = x_n - h\lambda x_n/2 = (1 - h\lambda/2)x_n,$$

and then

$$x_{n+1} = x_n - h\lambda(1 - h\lambda/2)x_n = (1 - h\lambda + h^2\lambda^2/2)x_n.$$

Clearly then $x_n = (1 - h\lambda + h^2\lambda^2/2)^n$. This remains bounded when

$$-1 \le 1 - h\lambda + h^2\lambda^2/2 \le 1,$$

so that

$$-2 \le -h\lambda + (h\lambda)^2/2 \le 0.$$

The function $f(x) = x^2/2 - x$ has its minimum at $(1, -1/2)$, and $x^2/2 - x = 0$ when $x = 0$ or $x = 2$. Therefore, we must have that $0 \le h\lambda \le 2$, i.e. when $0 \le h \le 2/\lambda$, which is the same as for Euler's method.

Section 13.5

Exercise 1. We define $x_1 = x$, $x_2 = x_1'$. The equation can then be written as $x_2' + \sin(tx_2) - x_1^2 = e^t$, so that $x_2' = e^t - \sin(tx_2) + x_1^2$. The third alternative is therefore correct.

Section 13.6

Exercise 2. If we differentiate the first equation we get that $x''' = 1 + x' + y''$. This inserted in the second equation gives that $y''' = 1 + x' + y'' + y'' = 1 + x' + 2y''$. We define $x_1 = x$, $x_2 = x'$, $y_1 = y$, $y_2 = y'$, $y_3 = y''$, we get the following equations:

$$\begin{aligned}
x_1' &= x_2 \\
x_2' &= t + x_1 + y_2 \\
y_1' &= y_2 \\
y_2' &= y_3 \\
y_3' &= 1 + x_2 + 2y_3.
\end{aligned}$$

where the second equation corresponds to the first original equation, and the fifth equation corresponds to the rewritten equation $y''' = 1 + x' + y'' + y'' = 1 + x' + 2y''$.

Exercise 4. In order to solve this exercise, it is most convenient to implement a function which returns the values for $\boldsymbol{f}(t, \boldsymbol{x})$. The two equations can be written as a first order system as

$$\begin{aligned}
x_1' &= x_2 \\
x_2' &= 2y_1 - \sin\left(4t^2 x_1\right) \\
y_1' &= y_2 \\
y_2' &= -2x_1 - \frac{1}{2t^2(x_2)^2 + 3},
\end{aligned}$$

with the initial condition $(x_1(0), x_2(0), y_1(0), y_2(0)) = \boldsymbol{x}_0 = (1, 2, 1, 0)$. We get that $\boldsymbol{f}(0, \boldsymbol{x}_0) = (2, 2 - \sin 0, 0, -2 - 1/(0+3)) = (2, 2, 0, -7/3)$, so that the first step with Euler's method with $h = 1$ gives

$$\boldsymbol{x}_1 = \boldsymbol{x}_0 + \boldsymbol{f}(0, \boldsymbol{x}_0) = (1, 2, 1, 0) + (2, 2, 0, -7/3) = (3, 4, 1, -7/3).$$

We now get that

$$f(1, \boldsymbol{x}_1) = (4, 2 - \sin(12), -7/3, -6 - 1/(32+3)) \approx (4, 2.5366, -7/3, -6.0286)$$

so that the second step with Euler's method gives

$$\begin{aligned}
\boldsymbol{x}_2 = \boldsymbol{x}_1 + \boldsymbol{f}(1, \boldsymbol{x}_1) &\approx (3, 4, 1, -7/3) + (4, 2.5366, -7/3, -6.0286) \\
&\approx (7, 6.5366, -1.3333, -8.3619).
\end{aligned}$$

This means that our approximations become

$$x(2) \approx 7, \quad x'(2) \approx 6.5366, \quad y(2) \approx -1.3333, \quad y'(2) \approx -8.3619.$$

For Euler's midpoint method the first step becomes

$$\begin{aligned}
\boldsymbol{x}_{1/2} &= \boldsymbol{x}_0 + f(0, \boldsymbol{x}_0)/2 = (1,2,1,0) + (2,2,0,-7/3)/2 = (2,3,1,-7/6) \\
f(1/2, \boldsymbol{x}_{1/2}) &= (3, 2 - \sin 2, -7/6, -4 - 1/(9/2+3)) \approx (3, 1.0907, -1.1667, -4.1333) \\
\boldsymbol{x}_1 &= \boldsymbol{x}_0 + f(1/2, \boldsymbol{x}_{1/2}) \approx (1,2,1,0) + (3, 1.0907, -1.1667, -4.1333) \\
&= (4, 3.0907, -0.1667, -4.1333).
\end{aligned}$$

The second step becomes

$$\begin{aligned}
f(1, \boldsymbol{x}_1) &\approx (3.0907, -0.0454, -4.1333, -8.0452) \\
\boldsymbol{x}_{3/2} &= \boldsymbol{x}_1 + f(1, \boldsymbol{x}_1)/2 \\
&\approx (4, 3.0907, -0.1667, -4.1333) + (3.0907, -0.0454, -4.1333, -8.0452)/2 \\
&= (5.5454, 3.0680, -2.2333, -8.1560) \\
f(3/2, \boldsymbol{x}_{3/2}) &\approx (3.0680, -4.1169, -8.1560, -11.1128) \\
\boldsymbol{x}_2 &= \boldsymbol{x}_1 + f(3/2, \boldsymbol{x}_{3/2}) \approx (7.0680, -1.0262, -8.3226, -15.2461).
\end{aligned}$$

This means that our approximations become

$$x(2) \approx 7.0680, \quad x'(2) \approx -1.0262, \quad y(2) \approx -8.3226, \quad y'(2) \approx -15.2461.$$

Section 14.1

Section 14.2

Section 15.1

Section 15.2

Section 15.3