Notes on Elementary Numerical Computation

Dieter Britz
Kemisk Institut, Aarhus Universitet

July 19, 2010

Preface

This is a translation of the Danish original "Elementær Numerisk Regning". That document developed from notes originally written, by various authors, for the course DatA, which later became DatK, mainly Poul Jørgensen (PJ), Niels Christian Nielsen (NCN) and the undersigned. PJ and NCN thus are the original authors of the chapter on linear systems, while NCN wrote on the Fourier transformation. I wrote the rest, and over the years added more material that I deemed to be of interest, such as function evaluation and digital signal processing, for which there never have been any exercises in the course. I have modified and added to the chapters written by others to some extent but not strongly except now in doing the translation. The drive to translate the notes (now in fact a book) was the recognition that we are host to an increasing number of foreign exchange students, who cannot read the Danish text, but find the course of interest.

There is no Index, but it is my hope that the Contents pages are sufficient.

Dieter Britz
July 2010

Contents

1 Introduction **1**
- 1.1 Programming language . . . 1
- 1.2 Precision and rounding . . . 1
- 1.3 Error finding . . . 2
- 1.4 Taylor series . . . 3
- 1.5 Error order . . . 4
- 1.6 Derivative approximations . . . 5

2 Roots of equations **7**
- 2.1 Binary search (bisection) . . . 7
- 2.2 Regula falsi . . . 8
- 2.3 Newton's method . . . 9
- 2.4 Derivatives of "difficult" functions . . . 10
- 2.5 Function inversion . . . 11
- 2.6 Min/max search . . . 12
- 2.7 The "G-method" - iterative solution . . . 12
- 2.8 Polynomials . . . 14

3 Integration **17**
- 3.1 Block integration . . . 17
- 3.2 Trapezium method . . . 18
- 3.3 Simpson's rule . . . 19
- 3.4 Errors . . . 20
- 3.5 Romberg integration . . . 21
- 3.6 Dynamic intervals . . . 22
- 3.7 Discrete function values . . . 24
- 3.8 Open forms . . . 24

4 Differential Equations **27**
- 4.1 Euler method . . . 28
- 4.2 Taylor expansion . . . 28
- 4.3 Runge Kutta (RK) . . . 29
- 4.4 Backward implicit (BI) . . . 31
- 4.5 Trapezium method . . . 32
- 4.6 Extrapolation . . . 33
- 4.7 Systems of differential equations . . . 33

4.8 Dynamic step length . . . 35
4.9 Differential equations and integration . . . 36

5 Interpolation **39**
5.1 Zero-order "interpolation" . . . 39
5.2 Linear (two-point) interpolation . . . 40
5.3 Parabolic interpolation . . . 41
5.4 General Lagrange formula . . . 42
5.5 Neville method . . . 42
5.6 Hermitian interpolation . . . 45

6 Least Squares Fitting **47**
6.1 Linear least squares . . . 49
6.2 Unknown errors . . . 54
6.3 Goodness of fit . . . 54
6.4 Nonlinear least squares . . . 56

7 Linear Systems and Matrices **59**
7.1 Elementary definitions . . . 59
7.2 Vector- and matrix norms . . . 60
7.3 Linear transformation . . . 61
7.4 Matrix multiplication. . . . 61
7.5 Determinants . . . 64
7.6 Matrix inversion . . . 65
7.7 Solution of linear systems of equations . . . 66
7.8 Eigenvalues and eigenvectors . . . 74

8 Function Evaluation **79**
8.1 General . . . 79
8.2 Numerical integration . . . 83
8.3 Newton's method . . . 84
8.4 Polynomials . . . 84
8.5 Asymptotic series . . . 88
8.6 Rational functions and continued fractions . . . 88

9 Fourier transformation **91**
9.1 Introduction . . . 91
9.2 Continuous Fourier transform . . . 93
9.3 The discrete Fourier transform DFT . . . 96
9.4 An example . . . 99
9.5 Sampling and aliasing . . . 103

10 Digital Signal Processing **105**
10.1 Filtering . . . 105
10.2 Frequency spectra of filter response . . . 110

Bibliography **113**

Chapter 1

Introduction

1.1 Programming language

In the course **DatK** (Numerical Computation in Chemistry) an elementary introduction is given to the main points in numerical computing. At the time of writing, three programming languages dominate in the field of scientific computation: `Fortran`, `Pascal` and `C++`.

`Fortran` (now in the modernised form of `Fortran 90/95`) has been the preferred language for scientific computing for many years, and there is a large body of freely available subroutines and functions ready for download, tried and tested by experts. `Fortran` is the language used in DatK. `Pascal` and `C++` have the advantage that very cheap compilers are available for a PC running under Windows, which has not always been the case with `Fortran 90/95`. However, if you work under `Linux`, you can download and install the free Intel `Fortran 90/95` compiler, which works fine. `Pascal` is often chosen for programming interfaces to lab instruments and the language is taken by some to be better structurable than `Fortran`, but it is often overlooked that there have been radical changes to `Fortran`, especially in the new version `Fortran 90/95`, which is now practically as elegant as other languages. In recent years, `C++` has become popular, but in the author's opinion, it is not as easy to learn as some other languages. We now also have `Java`, but it is not designed for numerical computation. Therefore we have chosen to use `Fortran 90/95` for DatK. As with spoken languages, learning the first computer programming language makes learning others easier, so if there is a need for it, anyone who has learned `Fortran` can easily learn another programming language.

1.2 Precision and rounding

In `Fortran 90/95` (and other computer languages) there are many different representations of numbers (or number variables); there are, for example, whole numbers (integers) and real numbers with fractional parts. When using integers, there are special problems with division. For example, the expression

1/2 computes to zero, the lower integer nearest to the mathematical result. Multiplication of integers is always exact, provided that the magnitude of the product does not exceed the maximum allowed integer size. With real numbers, there is generally some uncertainty in the last decimal digit, due to the way these numbers are stored in the computer. For example, in the decimal number system, the fraction 1/3 is represented as 0.333... with a finite number of decimals, which leaves the value a little short at the end. In the same manner, the computer's binary representation of most numbers has some uncertainty in the last fractional bit. Even a seemingly exact value like 0.1 becomes an infinite string of digits in binary form, which must be truncated somewhere, leading to a small truncation error. When such numbers are combined in an arithmetical operation, the result too is in error by some small amount, the truncation or rounding error. A given real number's precision is determined by the number of binary digits given to it in computer storage. Typical machines store real numbers by default with a precision of about 6-7 decimal digits, or a relative precision of 1 in $10^6 - 10^7$. The rounding error is then about 10^{-6} eller 10^{-7} times the number itself (the relative error). It is important to be aware of this. For example, two computed real numbers are unlikely to be equal to each other, even though we might expect them to be. If we add 0.01 to itself repeatedly, we might expect the sum at some point to be equal to 1.00.., but because of rounding errors, it will probably miss unity by some small amount. So, a loop in which we test whether the sum is equal to 1, will never detect this state. The condition must be expressed in a different way, e.g. $|\text{sum} - 1| < \epsilon$, where ϵ is some sufficiently small value, dependent on the context. There are techniques for minimising the accumulation of rounding errors and these are discussed in the `Fortran` part of the course.

An excellent introduction to the topic of errors is found in the Report by Løfstedt [1], which is made available in the course.

1.3 Error finding

It is rare for a computer program to be correct the first time. When a program is passed by the compiler - that is, when it is syntactically correct - there may still be errors, or "bugs". Some of these will be evident when the program runs. The `Fortran 90/95` compiler we use at the time of writing (2010) gives useful information especially on wrong syntax, but not so much on run-time errors, where error messages can be informative to some extent (but often not) but rarely tell us where in the program the error took place.

The simplest method of finding errors is to insert `print` statements such as `"At point A"` etc., in strategic places, to see how far the program gets, and then to narrow the search until (at worst) the very line where it happens is found. Often the error is obvious before this. One can also print out variable values to throw light on what is happening. When the error is found, these statements can be removed but if there are remaining doubts, they can be

commented out by inserting "!" at the beginning of each such line, so that these debugging statements can be activated again later by removing the "!".

1.4 Taylor series

This is a reminder of the very useful Taylor series or -expansions. If we have a function and know its value at some point x and its derivatives there, then the value at some distance from it, at $x+h$ (h typically small relative to x) is given by

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2!}f''(x) + \frac{h^3}{3!}f'''(x) + \dots \tag{1.1}$$

where $f'(x)$, $f''(x)$ are the function's first, second, ... derivatives with respect to x. Likewise, we have at $x-h$

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2!}f''(x) - \frac{h^3}{3!}f'''(x) + \dots \tag{1.2}$$

These two series lead to some very useful results. Note that in practice, these infinite series are truncated at some derivative level. since h is usually small, so that the error due to the truncation will normally also be small.

Just as with simple one-variable functions, Taylor series can be used to express sets of functions at shifted argument vectors. Let such a set be equations for a set of variables $x_1, x_2, \dots, x_N$ (or, in vector notation, $\mathbf{x}$), and so we have N functions $f_i, i = 1 \dots N$. Let $h_1, h_2, \dots, h_N$ (or the vector $\mathbf{h}$) be a set of distances or changes from the given set, resulting in $\mathbf{x}+\mathbf{h}$. The Taylor expansion at the new set is

$$\begin{aligned}
f_1(\mathbf{x}+\mathbf{h}) &= f_1(\mathbf{x}) + h_1\frac{\partial f_1}{\partial x_1} + h_2\frac{\partial f_1}{\partial x_2} + h_3\frac{\partial f_1}{\partial x_3} + \dots \\
f_2(\mathbf{x}+\mathbf{h}) &= f_2(\mathbf{x}) + h_1\frac{\partial f_2}{\partial x_1} + h_2\frac{\partial f_2}{\partial x_2} + h_3\frac{\partial f_2}{\partial x_3} + \dots \\
&\vdots \\
f_N(\mathbf{x}+\mathbf{h}) &= f_N(\mathbf{x}) + h_1\frac{\partial f_N}{\partial x_1} + h_2\frac{\partial f_N}{\partial x_2} + h_3\frac{\partial f_N}{\partial x_3} + \dots
\end{aligned} \tag{1.3}$$

where we only go as far as the first derivatives. The above can be written in the more compact matrix form,

$$f(\mathbf{x}+\mathbf{h}) = f(\mathbf{x}) + \mathbf{J}\cdot\mathbf{h} \tag{1.4}$$

in which $\mathbf{J}$, the Jacobian, is given by

$$\mathbf{J} = \begin{bmatrix} \dfrac{\partial f_1}{\partial x_1} & \dfrac{\partial f_1}{\partial x_2} & \cdots \dfrac{\partial f_1}{\partial x_N} \\ \vdots & & \\ \dfrac{\partial f_N}{\partial x_1} & \dfrac{\partial f_N}{\partial x_2} & \cdots \dfrac{\partial f_N}{\partial x_N} \end{bmatrix} . \tag{1.5}$$

1.5 Error order

When something is approximated (e.g. an integral, an interpolation, a derivative or a solution to a differential equation, etc.), there is often some interval (or a set of such) into which a stretch of function values is divided. The approximation is then in error e, which is a function of h. The function is usually a power series in h,

$$e = a\,h^p + b\,h^q + \dots\ , \tag{1.6}$$

with $a, b, \dots$ being unknown coefficients. Here p is the smallest power for which the coefficient is non-zero. Since h is usually chosen small, the higher terms in the polynomial have less significance than the lower terms, and the term in h^p will dominate. A good approximation of the error will thus be

$$e \approx a\,h^p \tag{1.7}$$

and we wish to know p. Normally we do not know the coefficient values, nor are they important. What is important is the socalled error **order** or the power of the interval, that dominates the error. One expresses the error in equation (1.7) as being of the order

$$e = O(h^p)\ . \tag{1.8}$$

It is this order that is of greatest interest in connection with a given approximation. If the order, for example, as here, is $O(h^p)$, and one tries to improve the approximation by halving h, we know from equation (1.8), that this reduces the error by the fraction 2^{-p}.

Estimation of the order

Sometimes it is not known, but desirable to know, the error order for a given calculation. There are several methods for its estimation.

Approximations such as derivatives or integrals are often computed from data points at intervals of h. The errors are then some power of h, $O(h^p)$, and we want to know p. Let us call the computed quantity $u(x, h)$, that is, the value of u at argument x computed with intervals h.

If we have an exact solution to compare with, we know the errors $e(h)$. In that case, we calculate $e(h)$ for this interval and again for αh, which produces the new error $e(\alpha h)$. Most often one chooses $\alpha = 2$. Now we can express the solution as

$$u = \hat{u} + a\,h^p + b\,h^q + \dots \tag{1.9}$$

or, if p dominates, simply

$$u = \hat{u} + a\,h^p \tag{1.10}$$

where $\hat{u}$ is the true solution.

If we know $\hat{u}$, it is easy. We perform two computations, one with interval h and one with $2h$. This produces the two results

$$\begin{aligned} u_h &= \hat{u} + a\,h^p \\ u_{2h} &= \hat{u} + a\,2^p\,h^p \end{aligned} \tag{1.11}$$

and we can write

$$\frac{u_{2h} - \hat{u}}{u_h - \hat{u}} = \frac{a\,2^p\,h^p}{a\,h^p} = 2^p \tag{1.12}$$

from which we obtain p,

$$p = \ln(2^p)/ln2\ . \tag{1.13}$$

If we do not know the true solution, there is another method, devised by Østerby [2], making use of three estimates. As before we compute u_h and u_{2h} but also now a third estimate with four times the interval, u_{4h}. This obeys the equation

$$u_{4h} = \hat{u} + a\,4^p\,h^p \tag{1.14}$$

and we can now compute the ratio

$$\frac{u_{4h} - u_{2h}}{u_{2h} - u_h} = \frac{(4^p - 2^p)\,a\,h^p}{(2^p - 1)\,a\,h^p} \tag{1.15}$$

which again yields 2^p.

Having found p, we can improve the method so as to eliminate it, leaving the next higher power q, decreasing the error, often greatly. See for example Romberg integration, p. 21 and extrapolation, p. 33.

1.6 Derivative approximations

Numerical differentiation is sometimes of interest, when we are dealing with a complicated evaluation of some function, which cannot directly be differentiated algebraically. This could, for example, be the sum of a number of terms in a series. We can get various approximations to first and higher derivatives from Taylor expansions (1.1) and (1.2). There are three different approximations to $\mathrm{d}f/\mathrm{d}x$. If we use (1.1), we get

$$\frac{\mathrm{d}f}{\mathrm{d}x} \approx \frac{f(x+h) - f(x)}{h} - \frac{h}{2}f''(x) - \ldots \tag{1.16}$$

which, for obvious reasons, is called a forward difference. If instead we use (1.2), we get the backward difference form,

$$\frac{\mathrm{d}f}{\mathrm{d}x} \approx \frac{f(x) - f(x-h)}{h} + \frac{h}{2}f''(x) - \ldots\ . \tag{1.17}$$

Both are $O(h)$, as can easily be seen. If we use both equations, and subtract (1.2) from (1.1), we get

$$\frac{\mathrm{d}f}{\mathrm{d}x} \approx \frac{f(x+h) - f(x-h)}{2h} + \frac{h^2}{3} f'''(x) - \dots \tag{1.18}$$

which is seen to be $O(h^2)$, as the term in h falls out. This form is the central difference and the most accurate of the three.

By adding the two Taylor expansions, we obtain a new result, which is an approximation to the second derivative:

$$\frac{\mathrm{d}^2 f}{\mathrm{d}x^2} \approx \frac{f(x-h) - 2f(x) + f(x+h)}{h^2} + \frac{h^2}{4} f'''(x) - \dots \ , \tag{1.19}$$

a central difference formula for the second derivative, and also $O(h^2)$.

It is possible to get better, that is, higher-order, approximations to these derivatives by using more points; one can even develop approximations based on a number of unequally spaced points, but this is outside the scope of these notes.

Chapter 2

Roots of equations

The task is to find, for a given function $f(x)$, that value or those values of x at which $f(x) = 0$. The method of solution depends upon what we know about the function. There exist many methods of root finding, and the following four will be described: binary search (bisection), regula falsi, Newton's method and an alternative, iterative, method.

2.1 Binary search (bisection)

As shown in Fig. 2.1, we must first find the x-values a and b which lie on either side of the root $x = \alpha$, such that $f(a) < 0$ and $f(b) > 0$. This is often easy to do. We then begin to narrow the interval $[a, b]$. The procedure is to calculate the function at the point midway between a and b:

$$x_{mid} = \tfrac{1}{2}(a + b). \tag{2.1}$$

If $f(x_{mid})$ lies below the zero line, i.e. $f(x_{mid}) < 0$, then we move a to x_{mid}; whereas if $f(x_{mid}) > 0$, we move b to x_{mid}. This process (step) is repeated until the interval $[a, b]$ (which keeps bounding $x = \alpha$) is sufficiently narrow. One defines a suitable quantity, ε, that is sufficiently small, and thus defines

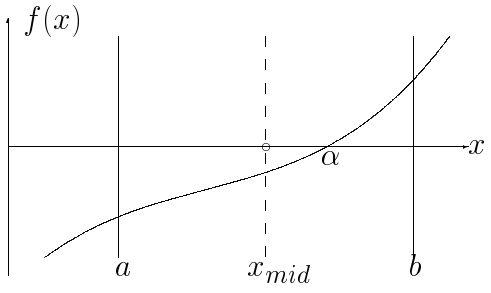

Figure 2.1: Binary search

the accuracy with which α is approximated. That is, one ends the process when $|a-b| \leq \varepsilon$ and the root is approximately $\alpha \pm \varepsilon$.

The above can be formalised into the following algorithm:

1. Find a og b, such that $f(a) < 0$ og $f(b) > 0$
2. Calculate $x_{mid} = \frac{1}{2}(a+b)$
3. If $f(x_{mid}) < 0$, $a \leftarrow x_{mid}$
   else $b \leftarrow x_{mid}$
4. If $|a-b| > \varepsilon$, repeat from 2.

Since one halves the interval at every step, it is possible to predict the number of *iterations* required for the interval to have narrowed to ε: it is $\log_2(|a-b|/\varepsilon)$. This method converges relatively slowly, but for well-behaved functions $f(x)$ it is very robust; that is, it will always find the solution, $\alpha \pm \varepsilon$.

2.2 Regula falsi

This is a variant of the binary search and converges slightly faster. We start with a pair of points, a og b, as with a binary search. As shown in Fig. 2.2, we now find the point x_0, where $f(x_0) = 0$ on the straight line drawn from $f(a)$ to $f(b)$. Simple geometry gives

$$x_0 = \frac{af(b) - bf(a)}{f(b) - f(a)} \,. \tag{2.2}$$

This is then treated as with a binary search. The algorithm is:

1. Find a og b, $f(a) < 0$ and $f(b) > 0$.
2. Calculate x_0

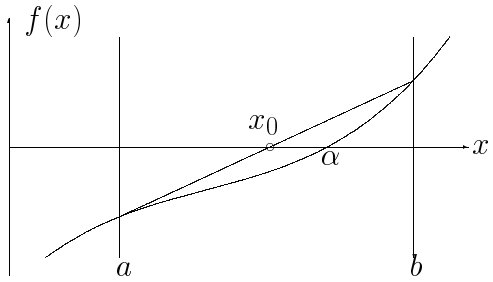

Figure 2.2: Regula falsi

3. If $f(x_0) < 0$, $a \leftarrow x_0$
   else $b \leftarrow x_0$

4. If $|a - b| > \varepsilon$, repeat from 2.

This method is as robust as the binary search (for smooth functions), but a little faster in converging.

2.3 Newton's method

This method is one of the fastest converging methods - if it converges, which is not always the case. The rule is that one should know a reasonable starting guess at the root, and that the function should be smooth in the vicinity of that root. If this is given, the method will converge in very few iterations. One must also be able to differentiate the function with respect to x, but this can always be done, if necessary numerically, see Sect. 2.4).

There are two ways to explain the method; by a geometric argument, or by algebra. In Fig. 2.3 we see a present guess at the root $x = x_n$ after n iterations and we improve it at the next iteration to the new value, x_{n+1}. We draw the tangent to the function at x_n and it cuts the x-axis at x_{n+1}. Clearly,

$$f'(x_n) = \frac{f(x_n)}{x_n - x_{n+1}} \tag{2.3}$$

which gives

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \ . \tag{2.4}$$

The corresponding algebraic argument makes use of the Taylor expansion. We have x_n and wish to find $x_n + \delta x = x_{n+1}$, such that $f(x_{n+1}) = 0$. The Taylor expansion about x_n is

$$f(x_n + \delta x) = f(x_n) + \delta x f'(x_n) + \frac{\delta x^2}{2!} f''(x_n) + \ldots \tag{2.5}$$

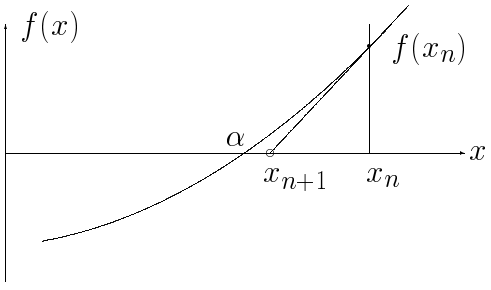

Figure 2.3: Newton's method

We set $f(x_n + \delta x) = 0$ and ignore terms with higher powers than δx:

$$\begin{aligned} 0 &= f(x_n) + \delta x f'(x_n) \\ \delta x &= -f(x_n)/f'(x_n) \end{aligned} \tag{2.6}$$

which yields the same formula

$$x_{n+1} = x_n + \delta x = x_n - \frac{f(x_n)}{f'(x_n)} \, . \tag{2.7}$$

This method of developing the formula has the advantage that it is easy to apply to functions of several variables, that is, to a set of functions.

Newton's method applied to a set of functions

In Sect. 1.4 the use of the Taylor expansion of a set of functions $f_i(\mathbf{x}),\ i = 1 \ldots N$ is described. This set has a corresponding set or roots $x_i,\ i = 1 \ldots N$. We arrived at the approximation

$$\mathbf{f}(\mathbf{x} + \mathbf{h}) = \mathbf{f}(\mathbf{x}) + \mathbf{J} \cdot \mathbf{h} \tag{2.8}$$

where $\mathbf{f}$ is the vector $[f_1\ f_2\ \ldots\ f_N]^T$, $\mathbf{h}$ is the vector $[\delta x_1\ \delta x_2\ \ldots\ \delta x_N]^T$ and $\mathbf{J}$ is the Jacobi matrix as defined in Sect. 1.4. Analogously with the procedure for a single variable, we set $\mathbf{f}(\mathbf{x} + \mathbf{h})$ equal to zero and obtain the correction vector $\mathbf{h}$ by

$$\mathbf{h} = -\mathbf{J}^{-1} \cdot \mathbf{f}(\mathbf{x}) \tag{2.9}$$

and therefore

$$\mathbf{x_{n+1}} = \mathbf{x_n} + \mathbf{h} = \mathbf{x_n} - \mathbf{J}^{-1} \cdot \mathbf{f}(\mathbf{x}) \tag{2.10}$$

Here, since $\mathbf{J}$ is a matrix, it must (in principle) be inverted, and the inverse multiplied by $\mathbf{f}(\mathbf{x})$ (but in practice, one solves the equation set $\mathbf{J} \cdot \mathbf{h} = -\mathbf{f}(\mathbf{x})$ for $\mathbf{h}$).

2.4 Derivatives of "difficult" functions

Newton's method requires function derivatives. Some functions cannot easily be differentiated algebraically. They may, for example, be the result of a complicated calculation and there may not be an explicit expression for the derivative. In such a case one approximates the derivative $f'(x)$ numerically. One chooses a small interval h and makes use of

$$f'(x) \approx \frac{f(x + \frac{h}{2}) - f(x - \frac{h}{2})}{h} \, . \tag{2.11}$$

which is illustrated in Fig. 2.4. The choice of h requires some "feel"; it should

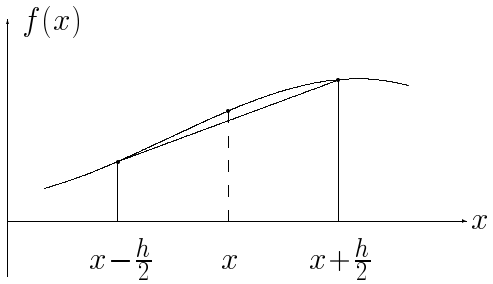

Figure 2.4: Numerical differentiation

be as small as possible, but not so small that the difference between the two function values is so small that numerical rounding problems arise, in which case the derivative becomes inaccurate. Here is an example. We approximate $f'(x)$ at $x = 1$ for the function $f(x) = \mathrm{e}^{-x}$. We know $f'(x)$, it is $-\mathrm{e}^{-1}$, or -0.3678794. Here is a table of approximations for a number of h values, using machine single precision (about 6-7 decimals).

h	f'(approx)	error
10^{0}	-0.383400	-0.015521
10^{-1}	-0.368033	-0.000153
10^{-2}	-0.367880	-0.000000
10^{-3}	-0.367880	-0.000000
10^{-4}	-0.367761	0.000119
10^{-5}	-0.366569	0.001311
10^{-6}	-0.357628	0.010252
10^{-7}	-0.000000	0.367879

Note that for $h > 10^{-3}$ the approximation becomes worse and worse, and finally yields zero. Normally, one must experiment with h, and possibly use a higher machine precision.

2.5 Function inversion

Many functions can be easily inverted. For example, if

$$y = f(x) = \mathrm{e}^{-x} \tag{2.12}$$

and we wish to find x for a given y, this is unproblematical, because the function can be inverted to

$$x = -\ln(y)\ . \tag{2.13}$$

This is not always the case, however. In electrochemistry one finds, for example, a function of the form

$$y = ae^{ux} + be^{vx} \tag{2.14}$$

and this cannot be inverted easily. If we however want an x value that satisfies a given y, (assuming that we know the parameters a, b, u and v), we must search for the answer. This is done by making it a root finding problem. We write

$$f(x) = ae^{ux} + be^{vx} - y \tag{2.15}$$

and find the root of the function, where $f(x) = 0$.

2.6 Min/max search

A related problem is to find a minimum or maximum of a function, see Fig. 2.5. The easiest method is to find the root of $f'(x)$, because the minimum or maximum will lie where $f'(x) = 0$. One must therefore find the root of $f'(x)$ using one the known root finding methods.

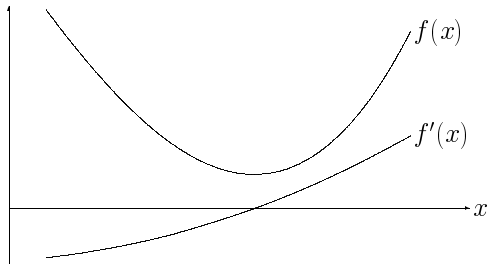

Figure 2.5: A function and its derivative

If the function is multivariate, $f(x_1, x_2, \ldots, x_N)$ then we obtain a set $f'_1, f'_2, \ldots, f'_N$ of derivatives

$$f'_i = \frac{\partial}{\partial x_i} f\,(x_1, x_2, \ldots, x_N) \qquad\qquad i = 1 \ldots N \tag{2.16}$$

for which we find the set of roots as described.

2.7 The "G-method" - iterative solution

This method, also called the iterative method, is sometimes useful for finding the root of a nonlinear function,

$$f(x) = 0\ . \tag{2.17}$$

Assume that we can rearrange it to the form

$$x = g(x) \; . \tag{2.18}$$

This might be an implicit form of the first. If we are lucky, the following iterative formula can be used:

$$x_{n+1} = g(x_n) \tag{2.19}$$

for increasing n. We put the present estimate x_n into the function, and obtain the next estimate x_{n+1}, hopefully an improvement on x_n. One application of this occurs in chemistry, in the calculation of the concentration of H^+ when a a weak acid dissociates in water. The dissociation is described by

$$K = \frac{x^2}{c - x} \tag{2.20}$$

where $x = [H^+]$. Often we assume that x is insignificant in comparison with c (for small K-values), and in that case we can simplify the equation to

$$K = \frac{x^2}{c} \; , \tag{2.21}$$

i.e. directly

$$x = \sqrt{Kc} \; . \tag{2.22}$$

However, if that assumtion is not reasonable, we can avoid solving the quadratic equation by ussing the iterative formula

$$x_{n+1} = \sqrt{K\,(c - x_n)} \tag{2.23}$$

starting with $x_0 = 0$. Here is a concrete example: let $pK_a = 2$ ($K = 0.01$) and $c = 0.1M$. The sequence is

$$\begin{aligned}
x_1 &= \sqrt{0.01\,(0.1 - 0)} = \sqrt{0.001} && = 0.03162 \\
x_2 &= \sqrt{0.01\,(0.1 - 0.03162)} && = 0.02615 \\
x_3 &= \sqrt{0.01\,(0.1 - 0.02615)} && = 0.02718 \\
x_4 &= \sqrt{0.01\,(0.1 - 0.02718)} && = 0.02699 \\
&\vdots \\
x_7 & && = 0.0270156 \; .
\end{aligned} \tag{2.24}$$

The method is also "pocket calculator friendly" - try it yourself. One does not have to type in intermediate results, only some constants. Note also that a quadratic equation has two roots, and that we here ignored the other root, which is meaningless in this context.

This method will be mentioned again in Chap. 7 in connection with the solution of linear systems, where it is called Gauss-Seidel.

2.8 Polynomials

Polynomials are a subject of special interest, and there are special methods for finding their zeroes.

For polynomials up to the fourth degree there exist analytical methods of solution. The method for quadratic equations is well known, and Abramowitz & Stegun [3] presents methods for cubic and quartic equations. For higher degrees, things become increasingly difficult and one normally uses numerical methods. There are some exceptions in those cases where we can easily factorise the polynomial into lower order polynomial factors. The zeroes of these factors are also the zeroes of the original polynomial. Division of the polynomial by such factors is called "deflation". In favourable cases, deflation might result in a sufficiently low order polynomial whose zeroes can be found analytically.

In the book by Press & al. [4] there are several methods for finding polynomial zeroes, of which we will describe two here, which are specially useful.

Laguerre method

Let the polynomial be $P_n(x)$, with zeroes $x_i, i = 1, \ldots n$:

$$P_n(x) = (x - x_1)(x - x_2)\ldots(x - x_n)\ . \tag{2.25}$$

We know none of the zeroes, but we have a start guess at one of them. The logarithm of the function is

$$\ln|P_n(x)| = \ln|x - x_1| + \ln|x - x_2| + \cdots + \ln|x - x_n| \tag{2.26}$$

which after differentiation becomes

$$\frac{\mathrm{d}\ln|P_n(x)|}{\mathrm{d}x} = \frac{1}{x - x_1} + \frac{1}{x - x_2} + \cdots + \frac{1}{x - x_n} = \frac{P_n'}{P_n} \equiv G \tag{2.27}$$

where P_n' is the first derivative of P_n. We then differentiate again,

$$-\frac{\mathrm{d}^2\ln|P_n(x)|}{\mathrm{d}x^2} = \frac{1}{(x - x_1)^2} + \frac{1}{(x - x_2)^2} + \cdots + \frac{1}{(x - x_n)^2} = \left(\frac{P_n'}{P_n}\right)^2 - \frac{P_n''}{P_n} \equiv H \tag{2.28}$$

where P_n'' is the second derivative. Both G og H can be evaluated using the present estimate of x.

We assume (hope) that the zero x_1 is not too far from our present guess x. The deviation from the true zero is $a = x - x_1$, and we therefore wish to find a. Strangely enough, the method also involves the assumption that all other zeroes deviate by another quantity b from our guess, i.e., $x - x_i =$

$b, i = 2, 3, \ldots, n$. With these assumptions, we reformulate equations (2.27) and (2.28) to the new

$$G = \frac{1}{a} + \frac{n-1}{b} \tag{2.29}$$

and

$$H = \frac{1}{a^2} + \frac{n-1}{b^2} \tag{2.30}$$

which we can solve for a:

$$a = \frac{n}{G \pm \sqrt{(n-1)(nH - G^2)}} \,, \tag{2.31}$$

where we choose the sign such that the denominator assumes the larger value. This yields the deviation a of our guess, and we can correct it. Note that the terms within the square root can be imaginary, so we can find a complex root with this method.

Clearly, we do not find the true zero at the first try, and we must repeat the process (iterate). After a number of iterations, a will converge to zero, or as close as we are willing to accept. We have then found one of the zeroes, and can deflate the polynomial P_n to P_{n-1} by dividing by the factor $(x - x_1)$. After that we can start the same procedure to find another zero, and so on for all the others.

Having found a number of roots, we can use the "root polishing" process described by Press & al. [4] to render them more precise. See the last section of this chapter.

An example

We wish to find the roots of the polynomial equation

$$P_4(x) = x^4 - 8x^3 + 17x^2 + 2x - 24 = 0 \tag{2.32}$$

which has the roots -1, 2, 3, 4, and we guess at $x = 1$ as a start. We use the above equations and the first three iterations produce respectively 1.7882, 1.9964 and 2.000, which does not change thereafter. Deflation of $P_4(x)$ by $(x - 2)$ results in the cubic equation

$$P_3(x) = x^3 - 6x + 5x + 12 \,, \tag{2.33}$$

for which we can find the roots by the method described by Abramowitz & Stegun [3]. However, while we are at it, we can also continue with the Laguerre method. We make another guess at another root of $P(x) = 0$, $x = 1$, and obtain a new sequence of solutions 2.3703, 2.9506 and 3.0000. Deflation with $(x - 3)$ results in the quadratric equation

$$P_2(x) = x^2 - 3x - 4 \,, \tag{2.34}$$

which we can easily solve for the remaining two zeroes.

Eigenvalue method

Press & al. [4] describe the following method. It turns the problem of finding eigenvalues of a matrix on its head. As mentioned in Chap. 7, page 74, the definition of the eigenvalues via the determinant of the matrix leads naturally to a polynomial whose zeroes are the eigenvalues. This is however the least effective method of finding eigenvalues, as there are much more effective iterative methods. These methods can be adapted to solve for the zeroes of polynomials. A given polynomial can be expressed as

$$P_n(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1} + x^n \tag{2.35}$$

where the polynomial has been divided by a_n if $a_n \neq 1$. We set up a so-called *companion matrix*

$$\mathbf{A} = \begin{bmatrix} -a_{n-1} & -a_{n-2} & \cdots & -a_1 & -a_0 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & & & & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix} . \tag{2.36}$$

The eigenvalues of this matrix are also the zeroes of the original polynomial.

An example

If we again take the polynomial (2.32), the companion matrix is

$$\mathbf{A} = \begin{bmatrix} 8 & -17 & -2 & 24 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.37}$$

and a suitable procedure for finding eigenvalues then finds all four polynomial roots.

Root polishing

Both methods mentioned above are iterative methods, and therefore it may be that we are not satisfied with the accuracy of the results. With "root polishing" they can be improved. This is based on the Newton method for root finding. We start, for each root, with the (presumably slightly inaccurate) value, using the polynomial itself as the function, and its derivative. Normally, only one or two Newton steps are required to converge to up to machine accuracy. The formula is

$$x_{n+1} = x_n - \frac{P_n(x_n)}{P_n'(x_n)} . \tag{2.38}$$

The formula works for complex roots too, provided that the program is written accordingly, with variables of the type `complex`.

Chapter 3

Integration

We are sometimes interested in calculating an integral of a function

$$I = \int_a^b f(x)\mathrm{d}x \tag{3.1}$$

which we cannot integrate analytically. We divide the range $[a, b]$ into equal intervals h (see Fig. 3.1).

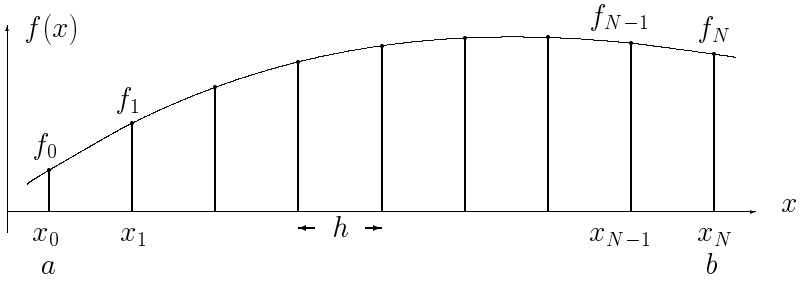

Figure 3.1: Partitioning of a function

We look first at a function which we are able to evaluate at every point x in the range wanted. Later functions which are represented as a discrete set of values will be considered. The function f is given by its values $f_0, f_1, f_2, \ldots, f_N$ at the points $x_0, x_1, x_2, \ldots, x_N$. The integral is the area under $f(x)$ in the range $[a, b]$.

There are several methods of integration, to be described below.

3.1 Block integration

This term describes an approximation that makes use of a constant across every interval. Fig. 3.2 shows one interval out of the N intervals in Fig. 3.1. We can choose three different constant values, shown in the figure by the three

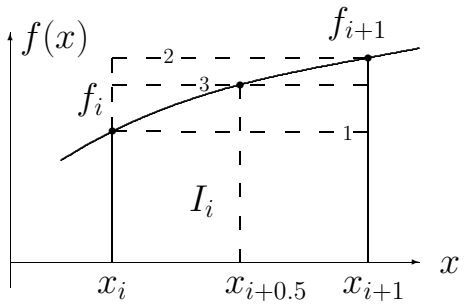

Figure 3.2: Block integration, a single element

stippled lines 1, 2 and 3. Lines 1 and 2 are clearly less satisfactory than line 3, which intuitively seems the best. If we use line 1, then the integral I_i is

$$I_i = f_i \ (x_{i+1} - x_i) = h f_i \ , \tag{3.2}$$

but if we use line 2, the approximation is

$$I_i = h f_{i+1} \ . \tag{3.3}$$

The condition for a minimum error here is, that the function be rather constant over the interval. It is easy to show that these choices yield an approximation with error $O(h)$; that is, if we attempt to improve the approximation by multiplying the number of intervals by some number m, the error becomes smaller by the same factor. This can be improved upon. An obvious choice is line 3, which means the approximation

$$I_i = h f_{i+0.5} \ . \tag{3.4}$$

This approximation, the midpoint method, produces an error of $O(h^2)$, which is much better. If we again improve the calculation by using m times as many intervals, the error now decreases by the factor m^2.

The total integral I is the sum of all I_i, and if we use the midpoint method, the expanded formula becomes

$$I = \sum_{i=0}^{N-1} I_i = h \sum_{i=0}^{N-1} f(x_{i+0.5}) \tag{3.5}$$

where the integral symbol has been replaced by the sum symbol $\sum$.

3.2 Trapezium method

Fig. 3.3 clearly points to the use of the trapezium formed by drawing a straight line between (x_i, f_i) and (x_{i+1}, f_{i+1}). Now the condition for a minimum error is

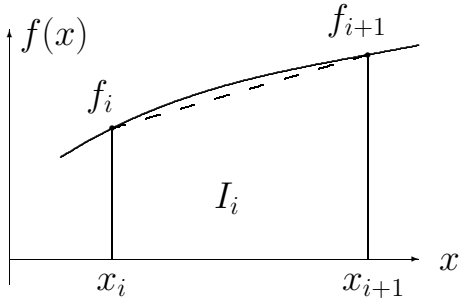

Figure 3.3: Trapezium integration, single element

that the function does not curve very much over the interval. Ideally it should itself be a straight line. The formula is:

$$I_i = \frac{h}{2}\,(f_i + f_{i+1}) \tag{3.6}$$

and when we put all elements together:

$$\begin{aligned} I &= I_0 + I_1 + \cdots + I_{N-1} \\ &= \frac{h}{2}\,(f_0 + f_1) \\ &+ \frac{h}{2}\,(f_1 + f_2) \\ &+ \ldots \\ &+ \frac{h}{2}\,(f_{N-2} + f_{N-1}) \\ &+ \frac{h}{2}\,(f_{N-1} + f_N) \end{aligned} \tag{3.7}$$

we get

$$I = h\left(\tfrac{1}{2}f_0 + f_1 + f_2 + \cdots + f_{N-1} + \tfrac{1}{2}f_N\right). \tag{3.8}$$

As with the midpoint integration method, this formula has an error of $O(h^2)$.

3.3 Simpson's rule

With block integration we used a rough block as the integral approximation over an interval and preferred the function $f(x)$ to be rather constant over that interval. With the trapezium method, $f(x)$ need not be constant but preferably linear over the interval. If we now use more than two x values, we

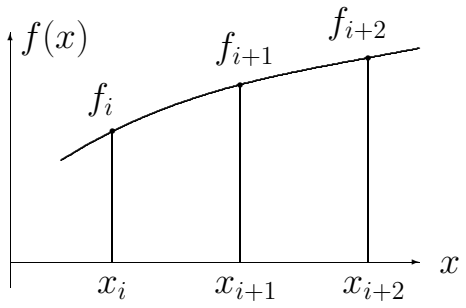

Figure 3.4: Simpson's rule integration, two elements

can make use of the function's curvature. In Fig. 3.4 we see two successive intervals, using three points. It is easy to fit a parabola to three function points, and the integral of the fitted parabola then is given by

$$I = \frac{h}{3}\big(f_i + 4f_{i+1} + f_{i+2}\big) \ . \tag{3.9}$$

Putting all the double intervals together yields

$$I = h\Big(\tfrac{1}{3}f_0 + \tfrac{4}{3}f_1 + \tfrac{2}{3}f_2 + \cdots + \tfrac{2}{3}f_{N-2} + \tfrac{4}{3}f_{N-1} + \tfrac{1}{3}f_N\Big) \ , \tag{3.10}$$

which is the well known Simpson's rule. Note that in order to make the coefficients $\frac{1}{3}, \frac{4}{3}, \frac{2}{3}, \frac{4}{3}, \ldots, \frac{2}{3}, \frac{4}{3}, \frac{1}{3}$ fit, N must be an even number, that is, there must an even number of intervals or an odd number of function values.

The above Simpson formula can be expressed as

$$I = \frac{h}{3}\left(f_0 + f_N + 4S_o + 2S_e\right) \tag{3.11}$$

where $S_o = f_1 + f_3 + \cdots + f_{N-1}$ (the sum of the odd values), and $S_e = f_2 + f_4 + \cdots + f_{N-2}$ (the sum of the evens). This way of expressing the formula becomes useful in a following paragraph.

3.4 Errors

For completeness' sake we present a summary of the methods and the error orders. We have

simple block integration: $O(N^{-1})$ or $O(h)$

midtpoint block integration: $O(N^{-2})$ or $O(h^2)$

trapezium rule: $O(N^{-2})$ or $O(h^2)$

Simpson's rule: $O(N^{-4})$ or $O(h^4)$.

This says, for example that, using Simpson's rule, if we increase the number of intervals N to $2N$, we divide the error by $2^4 = 16$. The order information can be used to optimise or improve a given approximation, see the next paragraph and Sect. 3.6.

3.5 Romberg integration

This method is described clearly by Østerby [5], whose article can be obtained from DAIMI. Consider again block integration at the midpoints. The approximating integral I_h, using intervals of size h, can be expressed as

$$I_h = \hat{I} + a\,h^2 + b\,h^4 + c\,h^6 + \ldots \tag{3.12}$$

where $\hat{I}$ is the true value that we wish to approximate, and the other terms are error terms. The term with the lowest power, here $a\,h^2$, will dominate, which is why we say that the error is of $O(h^2)$. If we could eliminate this dominating term, we would achieve a higher order, here $O(h^4)$; this can be done.

Generaly, without specifying the method, the general form of (3.12) is

$$I_h = \hat{I} + a\,h^p + b\,h^q + c\,h^r + \ldots \tag{3.13}$$

where $a, b, c, \ldots$ are unknown constants and $p, q, r, \ldots$ are unspecified powers. We can either calculate these or, if we cannot, we can determine them by numerical experiments, see Sect. 1.5. The method makes use of two estimates of the integral; first using N steps with interval length h over the integration range, with the result I_h; followed by a second integration with $N/2$ steps of length $2h$ with the result I_{2h}. We can then write the equations

$$\begin{aligned} I_h &= \hat{I} + a\,h^p + b\,h^q + c\,h^r + \ldots \\ I_{2h} &= \hat{I} + a\,2^p\,h^p + b\,2^q\,h^q + c\,2^r\,h^r + \ldots \end{aligned} \tag{3.14}$$

and if we multiply the first by 2^p and subtract the second second, we obtain, after some rearrangement,

$$I_h + \frac{I_h - I_{2h}}{2^p - 1} = \hat{I} + d\,h^q + \ldots \tag{3.15}$$

where d is a new constant. It is seen that the approximation is now of the next higher order q. We can keep doing this, computing yet another integral with $N/4$ steps of length $4h$, and calculate yet another integral, also of $O(h^q)$, and repeat the Romberg process and eliminate the term in h^q, increasing the error order to $O(h^r)$, etc. This is a very effective method of improving integration.

A numerical example

We calculate the integral $\int_0^1 exp(-x)\mathrm{d}x$. We know the result, $1 - exp(-1)$. We use midpoint integration and keep doubling the number of intervals N, until the error is sufficiently small, here defined as $< 10^{-6}$. The result is

N	Error without Romberg	Error with Romberg
4	-0.025590	-0.000186
8	-0.006537	-0.000012
16	-0.001643	-0.000001
32	-0.000411	0.000000
64	-0.000103	0.000000
128	-0.000026	0.000000
256	-0.000006	0.000000
512	-0.000002	0.000000
1024	0.000000	0.000000

Note that without Romberg we need 1024 intervals to achieve the desired error, whereas only 16 to achieve the same result with Romberg. Although Romberg involves more calculations per interval, it wins comfortably.

3.6 Dynamic intervals

We wish to integrate a function $f(x)$ over a given range but we do not know which value of N is needed to obtain the required accuracy, expressed as the maximum error we are willing to tolerate. Let us say we are using Simpson. We do not know the error resulting from a given N, but we do know its dependence on N (or h). Let this error be e_N. If we now double N, we know that the new error will be reduced by a factor of 16, that is, $e_{2N} = \frac{1}{16}e_N$. Thus if we compute the integral using N intervals to obtain the integral I_N , and then compute I_{2N} using $2N$ intervals, we can conclude that the difference between the two values is an expression of the actual errors, since the true integral value $\hat{I}$ is the same in both cases. We can make the following statements:

$$\begin{aligned} I_N &= \hat{I} + e_N \\ I_{2N} &= \hat{I} + e_{2N} \end{aligned}$$

and

$$e_{2N} = \tfrac{1}{16}e_N \ .$$

We then get

$$\begin{aligned} I_N - I_{2N} &= e_N - e_{2N} \\ &= e_N \left(1 - \tfrac{1}{16}\right) \\ &= \tfrac{15}{16}e_N \approx e_N \ . \end{aligned} \tag{3.16}$$

Normally we know what error level we are satisfied with. Thus the procedure is to repeatedly double the number of intervals N, look at the difference $|I_N - I_{2N}|$, and when this is sufficiently small, stop the process. We then know that

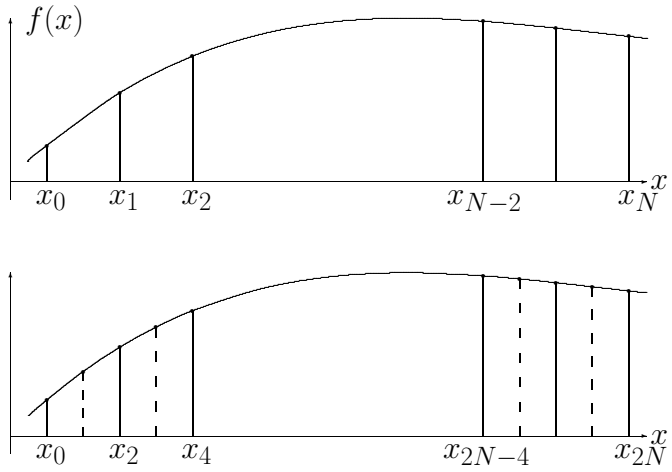

Figure 3.5: Doubling of the number of intervals N

the last integral estimate I_{2N} has an error that is smaller than the maximum acceptable.

This procedure is rather simple to apply to Simpson's rule; Fig. 3.5 shows a doubling of N. In the upper part we have N intervals, and

$$I_N = \frac{h}{3}\{f_0 + f_N + 4U + 2L\} \tag{3.17}$$

with the definitions of U and L as above. We have to compute three sums:

ends, $f_0 + f_N$

the odd positions, $f_1 + f_3 + \cdots + f_{N-1}$

the even positions, $f_2 + f_4 + \cdots + f_{N-2}$

and put them together to obtain I_N as in the formula. When we double the number of intervals, we can use all three again. We only have to compute the new function values in between (the dashed lines, all at odd positions). The ends are of course constant for all N. It is clear from Fig. 3.5 that, for $2N$, all the new evens are the previous evens and odds, so we can say

$$(\text{evens sum})\ (2N) = (\text{odds sum})\ (N) + (\text{evens sum})\ (N)\ . \tag{3.18}$$

This is very easy to program. Let the maximum tolerable error be ϵ. The algorithm is:

1. Start with $N = 1$ ($\rightarrow f_0, f_1$)
2. Ends sum $E = f_0 + f_1$ (constant hereafter)
3. Evens sum $S_e = 0$ (no points yet)

4. Odds sum $S_o = 0$ (no points yet)
5. Evaluate the integral, I_N
6. Double $N \to 2N$, halve $h \to h/2$; $S_e(2N) = S_o(N) + S_e(N)$
7. Calculate the new odds $S_o = f_1 + f_3 + \ldots$
8. Calculate the new integral I_{2N}
9. Evaluate $e = |I_N - I_{2N}|$
10. Set $I_N := I_{2N}$
11. If $|e| > \epsilon$, return to 6, otherwise I_N is the desired integral.

3.7 Discrete function values

In the above, it was assumed that we can evaluate the function to be integrated at any point of its argument. However, some integrations are of functions represented as a series of discrete points, perhaps produced by an instrument, or a simulation. Here we must make do with the values we have available. Fig. 3.1 shows such a point sequence. We cannot use the midpoint method directly but can approximate it by taking the mean of neighbouring points:

$$f_{i+0.5} \approx \frac{1}{2}(f_i + f_{i+1}) \; . \tag{3.19}$$

When we put all these together as in (3.5), we end with the trapezium method. We can also use Romberg here, by doing the integration using every other point, and the Romberg formula. Another possibility, if the number of points is odd, is to use Simpson's rule. So there are no special problems with a discrete sequence of values to be integrated.

3.8 Open forms

There are functions, which one cannot evaluate at one end of the range $f(a)$ and/or $f(b)$, for example

$$\int_0^1 x^{-\frac{1}{2}} \, \mathrm{d}x \; . \tag{3.20}$$

If the function can be evaluated at all points other than at a singularity in the integration range as in (3.20), the midpoint method can be used, because it never needs values at the interval ends. For such integrations, we often

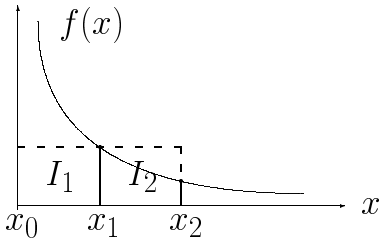

Figure 3.6: Open form, block integration

find non-integer error orders. If we integrate (3.20), for example, using the midpoint method, we find that the integral given by

$$I_h = \hat{I} + a\,h^{\frac{1}{2}} + b\,h^2 + \ldots \tag{3.21}$$

and other open forms also lead to non-integer polynomial powers for the error. If one knows the sequence of orders (which can be measured), they can go into the Romberg method to improve the integration.

If, however, the function is in the form of discrete values at given points, we have a problem at the ends. We can then approximate the end value by using extrapolation. Fig. 3.6 shows such a function. With block integration we can let both I_1 and I_2 be equal to hf_1, and thus we will not need f_0. Clearly, we can do better, for example as shown in Fig. 3.7, where we use the trapezium method and obtain the approximation to f_0 by extrapolating backwards on a straight line between f_1 and f_2. Since we have equal intervals, this is

$$\begin{aligned} f_0 &= f_1 + (f_1 - f_2) \\ &= 2f_1 - f_2\ . \end{aligned} \tag{3.22}$$

Then the trapezium formula here becomes

$$I_1 = \tfrac{1}{2}\,(f_1 - f_2)\ . \tag{3.23}$$

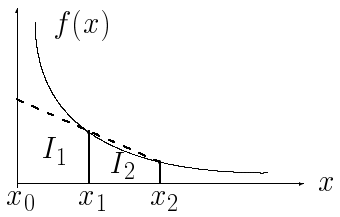

Figure 3.7: Open form, trapezium integration

Chapter 4

Differential Equations

Many phenomena in the sciences can be described mathematically in the form of differential equations of various orders, either as ordinary (**ode**) or partial differential equations (**pde**s). There are also systems of these equations. Examples are the dynamics of radioactive decay, chemical reactions, or mass transport (diffusion etc.). Although it is possible in some cases to find analytical solutions, this is often difficult or even impossible and we must apply numerical methods to obtain solutions. We restrict the discussion here to ordinary differential equations, **ode**s, and begin with simple types of these. Generally they can be written as

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f(y,t)\ . \tag{4.1}$$

In many cases the function f has only one variable (y or t). We write y' instead of $\mathrm{d}y/\mathrm{d}t$ for brevity. To solve such an equation means finding the underlying function $y(t)$. The function $f(y,t)$ expresses a gradient, and so we require more information to solve the equation: a value for y at some t-value or a socalled **boundary condition**. Fig. 4.1 shows what we have to start the solution with,

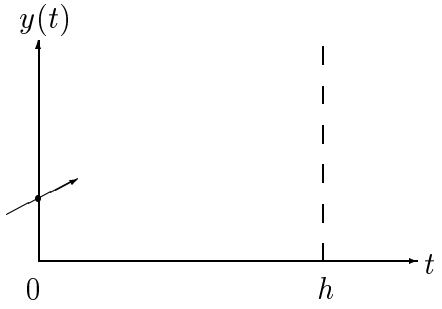

Figure 4.1: Beginning of the numerical solution of an ode

a value at $t = 0$ and the gradient y' there. We now compute new values at the discrete t-values h, $2h$, , ... etc.

The following notation will be used here: we assume that our function shall be solved at a number of points in $t = 0, h, 2h, \ldots$, where h is the chosen time interval. We thus compute a number of y-values $y_0, y_1, \ldots, y_N$, and most often we compute a new value y_{n+1} at t_{n+1} from a known previous value y_n at t_n.

We use as an example a differential equation whose solution we know:

$$y' = -y\,; \qquad y(0) = 1 \tag{4.2}$$

which has the solution $y(t) = \exp(-t)$.

4.1 Euler method

This simple method can be described graphically as moving along the gradient from a known point at t_n to the next at $t + h$ or t_{n+1}. Mathematically, we replace y' by the discretisation

$$y' \approx \frac{y(t+h) - y(t)}{h} \tag{4.3}$$

which we know to be a first-order approximation with respect to h if it is to apply at t (forward difference) or also at $t + h$ (backward difference). With the Euler method, it is a forward difference. By setting y' equal to $f(y, t)$ we obtain

$$y(t+h) = y(t) + h\, f(y, t) \tag{4.4}$$

or

$$y_{n+1} = y_n + h\, f(y_n, t)\ . \tag{4.5}$$

For our example (4.2) this becomes

$$y_{n+1} = y_n(1 - h)\ . \tag{4.6}$$

The process is repeated (iterated) for a number of values until we reach the last desired t-value $t + Nh$ or t_N. Fig. 4.2 shows the exact $y(t_{n+1})$ together with the one we get at t_{n+1} from the Euler method and it is clear that the method results in a certain error. This can be reduced by taking more steps with smaller h. We find that the error after N steps is proportional with h, that is, $O(h)$. There are methods that do better than this.

4.2 Taylor expansion

Using the Euler method, we in fact make use, in effect, of a short piece of a Taylor expansion. If we have $y(t)$ and want to calculate $y(t + h)$, then Taylor

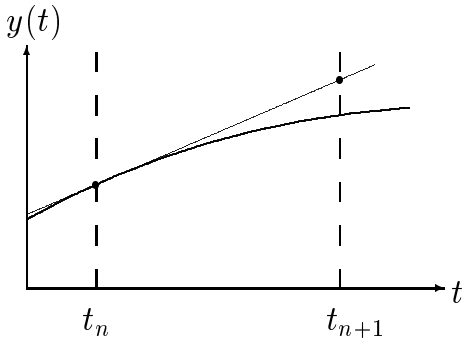

Figure 4.2: A single step using the Euler method

expansion gives

$$y(t+h) = y(t) + hy'(t) + \frac{h^2}{2!}y''(t) + \ldots \tag{4.7}$$

and with Euler we use only the first term on the right-hand side, giving (4.5). If we can differentiate y', we could improve the approximation by one order by making use of the next term with $\frac{h^2}{2}y''(t)$. This is however not convenient and is not easy to automate; a computer program for this will be specific to a given ode. The method is therefore not widely used.

4.3 Runge Kutta (RK)

Fig. 4.3 shows one way to look at the RK method. First we take an Euler step up line 1 and land at the open circle, which is the Euler solution. Now we determine y' there, that is, the gradient at the approximate solution at t_{n+1} (line 2). A gradient midway between the two positions (line 3) should be better than either of the two others, and we now use that gradient, starting again from the solution at t_n (line 4). This produces an improved solution at t_{n+1} (at least for the curve as drawn). All this is expressed mathematically using a special notation characteristic for the RK method. One computes a number of k_i, all expressing changes in y. Let us formalise the method in the following manner.

$$\begin{aligned} k_1 &= h\, f(y_n, t_n) \\ y_{n+1} &= y_n + k_1\ . \end{aligned} \tag{4.8}$$

This is the Euler method expressed in the RK form, which can be considered as a first-order RK method.

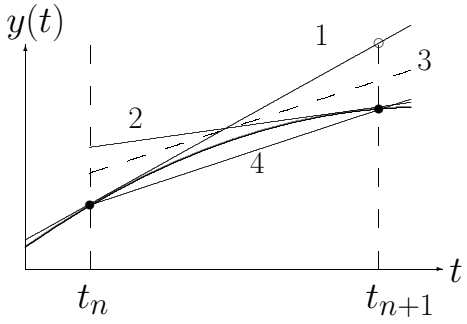

Figure 4.3: One RK step

Second order:

The above arguments about an improved gradient can be expressed in two ways, both with k-values, k_1 and k_2:

$$\begin{aligned} k_1 &= h\, f(y_n, t_n) \\ k_2 &= h\, f(y_n + k_1, t_n + h) \qquad (4.9) \\ y_{n+1} &= y_n + \tfrac{1}{2}\,(k_1 + k_2) \qquad (4.10) \end{aligned}$$

(this corresponds to the above, the mean of the two gradients), or

$$\begin{aligned} k_1 &= h\, f(y_n, t_n) \qquad \text{(som før)} \\ k_2 &= h\,(y_n + \tfrac{1}{2}k_1, t_n + \tfrac{1}{2}h) \qquad (4.11) \\ y_{n+1} &= y_n + k_2\,, \end{aligned}$$

which corresponds to a gradient computed at an approximated mid-point $t + \frac{1}{2}h$. The two variants are equally good, both $O(h^2)$.

For our example (4.2), equation (4.11) becomes

$$\begin{aligned} k_1 &= -h\, y_n \\ k_2 &= -h\,(y_n + k_1) \qquad (4.12) \\ y_{n+1} &= y_n + \tfrac{1}{2}(k_1 + k_2)\,. \end{aligned}$$

Even greater improvements are possible, going to yet higher orders. We name two of these here, third- and fourth-order variants:

Third order:

$$\begin{aligned} k_1 &= h\, f(y_n, t_n) \\ k_2 &= h\, f(y_n + \tfrac{1}{2}k_1, t_n + \tfrac{1}{2}h) \\ k_3 &= h\, f(y_n - k_1 + 2k_2, t_n + h) \qquad (4.13) \\ y_{n+1} &= y_n + \tfrac{1}{6}\,(k_1 + 4k_2 + k_3) + \mathrm{O}(h^3)\,. \end{aligned}$$

Fourth order:

$$\begin{aligned} k_1 &= h\, f(y_n, t_n) \\ k_2 &= h\, f(y_n + \tfrac{1}{2}k_1, t_n + \tfrac{1}{2}h) \\ k_3 &= h\, f(y_n + \tfrac{1}{2}k_2, t_n + \tfrac{1}{2}h) \\ k_4 &= h\, f(y_n + k_3, t_n + h) \\ y_{n+1} &= y_n + \tfrac{1}{6}\,(k_1 + 2k_2 + 2k_3 + k_4) + \mathrm{O}(h^4)\ . \end{aligned} \tag{4.14}$$

There is more to compute here, but this is compensated for by the fact that we can take larger steps with higher orders.

4.4 Backward implicit (BI)

The above-mentioned methods are all **explicit**; that is, they compute a new value using only old known values, with a discrete equation of the form

$$y_{n+1} = f(y_n, \dots)\ , \tag{4.15}$$

where there are known parameters in the function on the right-hand side. There exist also **implicit** methods, which have certain advantages, and in which the unknown value to be calculated is present among the arguments of the function. The simplest of these is the use of a backward difference and is therefore called **backward implicit** or **BI**. The idea is to use the gradient at the, as yet unknown, point y_{n+1} and to move along that gradient to it from y_n. Fig. 4.4 shows this; line 1 is the gradient, line 2 is parallel to it, drawn from the starting point. We note that this yields a result that seems no better than the one we get from the Euler method, shown in Fig. 4.2. In fact BI, like the Euler method, is a first-order method. It can, however, provide the basis for higher-order methods, as will be shown below.

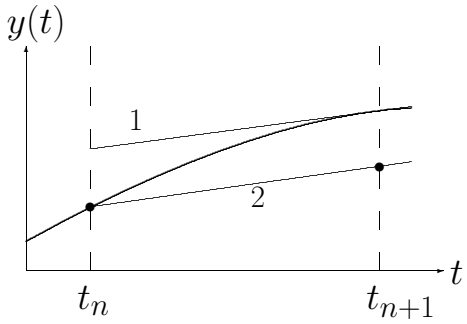

Figure 4.4: A single step with BI

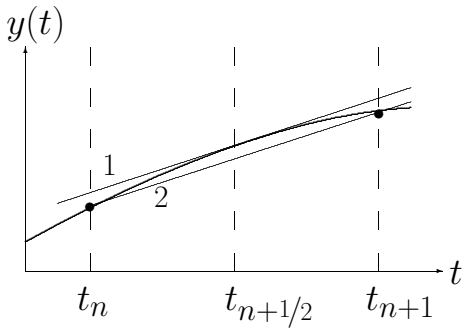

Figure 4.5: Trapezium method

The method can be expressed mathematically by

$$y_{n+1} = y_n + h\, f(y_{n+1}, t_{n+1}) \; . \tag{4.16}$$

Depending on what the function f is, this can be awkward to express, but normally this is not the case. Interestingly, the discrete expression for the derivative on the left-hand side is the same as with Euler, but now it refers to t_{n+1}, hence the name. In the case of our example (4.2), there is no problem: we write

$$\frac{y_{n+1} - y_n}{h} = -y_{n+1} \tag{4.17}$$

which produces

$$y_{n+1} = y_n \, \frac{1}{1+h} \; . \tag{4.18}$$

4.5 Trapezium method

We saw that for the differential equation $y' = f(y,t)$, that both the Euler- and BI methods have the drawback that y' is approximated by an asymmetric one-sided expression, respectively the forward and backward derivative discretisation $(y_{n+1} - y_n)/h$. It is interesting to note that one and the same expression *is* "forward" or "backward" depending on where it is intended to apply. Fig. 4.5 makes clear that it applies much better to the midpoint $t + \frac{1}{2}h$, where it becomes a central difference. The gradient at the midpoint is more likely to be similar to that of a line drawn between the two points, and this idea leads to another implicit method. We write generally

$$\frac{y_{n+1} - y_n}{h} = f(y_{n+1/2}, t_{n+1/2}) \; . \tag{4.19}$$

We know the middle t value but not that of y, but we can approximate this by the mean of the known y_n and the (as yet unknown) y_{n+1}. The discrete equation is thus

$$\frac{y_{n+1} - y_n}{h} = f\Big(\frac{y_n + y_{n+1}}{2}, t_{n+1/2}\Big) \tag{4.20}$$

which is implicit, containing the unknown value on both sides. In our example this becomes

$$\frac{y_{n+1} - y_n}{h} = -\frac{y_n + y_{n+1}}{2} \tag{4.21}$$

which gives

$$y_{n+1} = y_n \frac{1 - 2h}{1 + 2h} \; . \tag{4.22}$$

This method, being $O(h^2)$, works rather well and it it provides the basis for a number of related methods in other contexts, such as the numerical solution of partial differential equations and in systems of ordinary or partial differential equations.

4.6 Extrapolation

The BI method is itself not of great interest, as it is not very accurate but it suggests itself as the basis for improved methods of higher orders. It is, as we have seen, of $O(h)$, and the use of extrapolation can increase the order. The simplest second-order variant works as follows. We take a first BI step with length h to get a first estimate of y_{n+1}; let us call this $y(1)$. Now we compute another estimate $y(2)$, taking two steps each of length $h/2$. Since BI is $O(h)$, we know that the error in $y(2)$ is half of that in $y(1)$ and we can therefore eliminate it and achieve a better result from the combination

$$y_{n+1} = 2y(2) - y(1) \; . \tag{4.23}$$

Now the error is $O(h^2)$. There are variants of higher order still, making use of combinations of yet more partial steps, and there exist other methods based on BI. Together with some other properties of BI, it is these methods that make BI interesting.

4.7 Systems of differential equations

Many chemical and physical processes are described by a number of differential equations, perhaps coupled, that must be solved together. Atmospheric chemistry produces some of the worst examples, where hundreds of reacting

species are involved in complex reaction systems, leading to large systems of differential equations. In general, such systems can be written in the form

$$\begin{array}{rcl} y_1' &=& f_1(y_1, y_2, \cdots, y_N, t) \\ y_2' &=& f_2(y_1, y_2, \cdots, y_N, t) \\ \cdots & & \\ y_N' &=& f_N(y_1, y_2, \cdots, y_N, t) \end{array} \tag{4.24}$$

or in vector notation

$$\mathbf{y}' = \mathbf{f}(\mathbf{y}, t) \ , \tag{4.25}$$

where the vector $\mathbf{y} = [y_1, y_2, \cdots, y_N]^T$ denotes the N functions of time t, that must be found. In order to solve such systems, extensions of the same methods described can be used, for example Euler, RK, BI, trapezium etc. We use as an example a small system of two differential equations, both not functions of t:

$$\left.\begin{array}{rcl} x' &=& -y \\ y' &=& x \end{array}\right\} \tag{4.26}$$

where $y' \equiv \mathrm{d}y/\mathrm{d}t$, and t is a third, independent, variable along which x and y move. As always we must have starting (boundary) values:

$$t = 0: \quad x = x_0, \quad y = y_0 \ . \tag{4.27}$$

We can develop x- and y values for $t = h, 2h, \cdots$, etc. The Euler method is here simply

$$\left.\begin{array}{rcl} x_{n+1} &=& x_n - h\, y_n \\ y_{n+1} &=& y_n + h\, x_n \end{array}\right\} \ . \tag{4.28}$$

Runke-Kutta is almost as simple, but we now require a series $k_1, k_2, \ldots$ for each of the functions x and y, and we must make sure of the correct sequence, in which they are computed; it is $k_{1x}, k_{1y}; k_{2x}, k_{2y}$ etc. For second-order RK this is

$$\left.\begin{array}{rcl} k_{1x} &=& -h\, y_n \\ k_{1y} &=& h\, x_n \end{array}\right\} \qquad \left.\begin{array}{rcl} k_{2x} &=& -h\,(y_n + k_{1y}) \\ k_{2y} &=& h\,(x_n + k_{1x}) \end{array}\right\} \ . \tag{4.29}$$

Finally the new values are given by

$$\left.\begin{array}{rcl} x_{n+1} &=& x_n + \frac{1}{2}\,(k_{1x} + k_{2x}) \\ y_{n+1} &=& y_n + \frac{1}{2}\,(k_{1y} + k_{2y}) \end{array}\right\} \ . \tag{4.30}$$

This can be extended to higher-order RK variants.

The trapezium method was seen to be a good one for a single function, so we also try it here:

$$\left.\begin{array}{rcl} x_{n+1} &=& x_n - \frac{h}{2}\,(y_{n+1} + y_n) \\ y_{n+1} &=& y_n + \frac{h}{2}\,(x_{n+1} + x_n) \end{array}\right\} \ . \tag{4.31}$$

Here we meet the small problem that the equation system is implicit, being a system with two unknowns. A little reorganising produces

$$\begin{bmatrix} 1 & \frac{h}{2} \\ -\frac{h}{2} & 1 \end{bmatrix} \begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = \begin{bmatrix} x_n - \frac{1}{2}hy_n \\ \frac{1}{2}hx_n + y_n \end{bmatrix} .$$

Although this might seem a little awkward, it is worth the effort because of the accuracy of the method.

Similarly, BI can be used to advantage here also, with extrapolation. BI is very robust (stable), errors being reduced smoothly with the number of steps taken, and can be used for so-called stiff systems, in which some variables change much faster than others, which gives problems with simpler methods like Euler or RK methods, such as error oscillation or even divergence.

4.8 Dynamic step length

Sometimes the function we are solving has very different slopes with respect to the variable t. It is then appropriate to vary the step length, so that small steps are taken where the function is steep, and larger steps where it is less steep. Although there is some computational overhead in the calculation of the optimal step length, this is nevertheless rewarding because much computer time can be saved by the method.

The basis for this is to agree on an acceptable error size ε. Then it remains to determine that step size h which leads to an error not exceeding ε. This can be done by using what is known of the error order of whatever solution method to be used. Usually rather higher-order methods are used in this connection, such as, for example, fourth-order Runge-Kutta, for which we know that the error upon a number of steps to a given t is $O(h^4)$. We start with an estimate $y(t+h)$, knowing $y(t)$, and let the error be e_1, whose magnitude we do not know. Let us call this solution $y(1)$. We now recompute for y taking two steps each of size $\frac{1}{2}h$, resulting in another estimate, $y(2)$. From the order we know that the error is now only $(\frac{1}{2})^4$ of that of the first solution. We can write

$$y(1) = \hat{y} + e_1 \tag{4.32}$$

where $\hat{y}$ is the (unknown) true y-value at $t+h$, and also

$$y(2) = \hat{y} + e_2 \, . \tag{4.33}$$

Subtracting one from the other we get

$$y(1) - y(2) = e_1 - e_2 \tag{4.34}$$

and since we know that $e_2 = \frac{1}{16}e_1$,

$$y(1) - y(2) = \tfrac{15}{16}e_1 \approx e_1 \, . \tag{4.35}$$

The difference between the two computed estimates is therefore a good approximation of the actual error e_1. Now we can compute the appropriate step size h^* that gives us the acceptable error ε:

$$h^* = h \left(\frac{\varepsilon}{e_1} \right)^{1/4} . \tag{4.36}$$

If $e_1 < \varepsilon$, we can use the just calculated y_{n+1} and take the optimal step size h^* for the next step. If $e_1 > \varepsilon$, we re-evaluate y_{n+1} using h^*.

All this might seem bothersome, but under certain conditions (for certain functions) it can save much computing time. An added advantage is that one does not have to guess at a suitable step length before starting the computation, as the program finds the optimal size automatically.

One disadvantage of this method is that the result, that is the function y, consists of a number of discrete values at varying intervals. If one wishes equal intervals for some reason (perhaps for a display), this can be achieved with interpolation.

4.9 Differential equations and integration

Solving a differential equation is equivalent to integrating it. Mathematically it works as follows. The differential equation

$$y' = f(y, t) \tag{4.37}$$

is integrated over the interval between t_n and t_{n+1}:

$$y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} f(y, t)\, \mathrm{d}t . \tag{4.38}$$

The function $f(y_n, t_n)$ is known, so that we can use all the integration methods we already know from Chap. 3.

We start with block integration, Fig. 4.6: the new y value at t_{n+1} is the old value at t, $y(t_n)$ plus the integral over the interval h . Block integration means that the integral piece is simply $h\, f(t)$, that is

$$y_{n+1} = y_n + h\, f(t_n) \tag{4.39}$$

which we note is exactly the Euler method described in Sect. 4.1.

We know from Chap. 3 that the trapezium formula shown in Fig. 4.7 is better than block integration. There is a possible problem that f might be a function of both y and t, and that we do not yet know y_{n+1}. Nevertheless we can write the integral

$$y_{n+1} = y_n + \tfrac{1}{2} h \left(f(y_n, t_n) + f(y_{n+1}, t_{n+1}) \right) \tag{4.40}$$

which corresponds to the trapezium method described in Sect. 4.5.

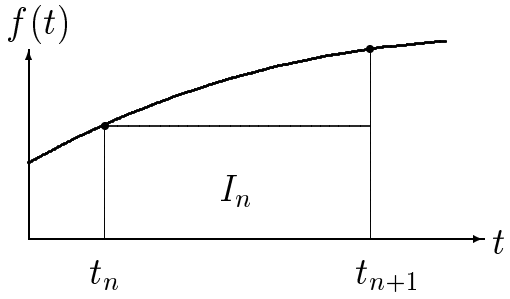

Figure 4.6: Block integration of a differential equation

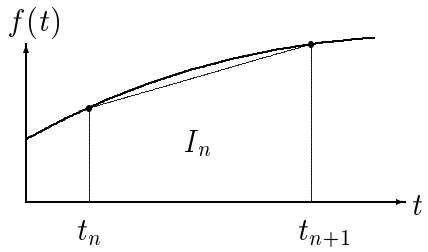

Figure 4.7: Trapezium integration of a differential equation

Chapter 5

Interpolation

We are given a function defined by a sequence of discrete points at x-values $x_i\,(i = 0 \ldots n)$ (Fig. 5.1), and we wish to find a value at a position x (open circle in the Figure) that lies in between the given positions. This is called interpolation. The intervals between the given points need not be equal. Interpolation

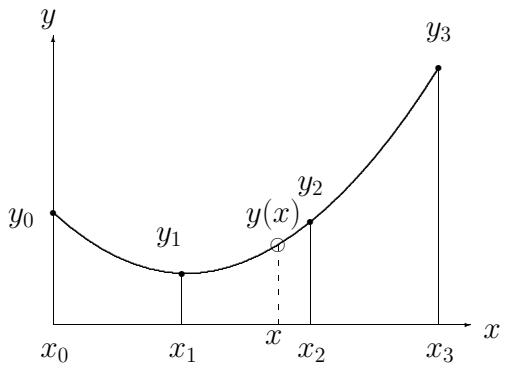

Figure 5.1: Function defined at some points

is often done by fitting a polynomial to a number of known points, and where in the following an "order" is mentioned, it is the order of the polynomial that is meant. The order of the error in the interpolated value with respect to the interval sizes is not a simple matter; often it turns out to be one order higher than the polynomial order used.

5.1 Zero-order "interpolation"

The simplest - and roughest - interpolation method is that which takes the same value as that of the closest point preceding the point at which the interpolation is to be. This involves only one of the defined points. In Fig. 5.2 we

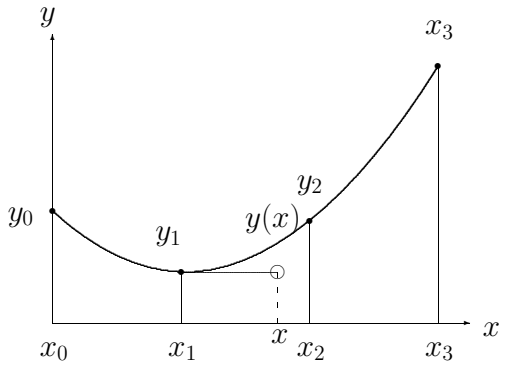

Figure 5.2: Zero-order interpolation

would thus say for the interpolated value $y(x)$ that

$$y(x) = y_1 \ . \tag{5.1}$$

This corresponds to continuation of y_1 "horizontally" to x. The interpolation is then

$$y = y_1 \ . \tag{5.2}$$

5.2 Linear (two-point) interpolation

Fig. 5.3 shows a two-point interplation, which corresponds to drawing a straight line between the two surrounding defined points and taking the interpolated

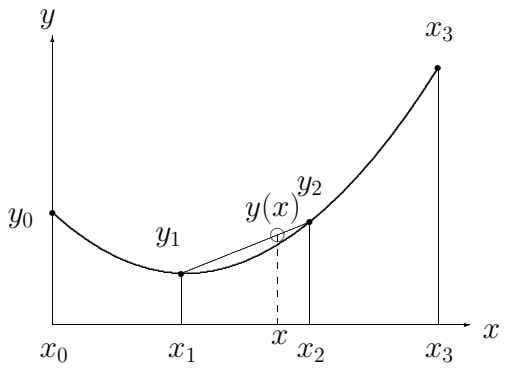

Figure 5.3: First-order (linear) interpolation

value from the line. Simple geometry leads to

$$\begin{aligned} y &= y_1 + \frac{x - x_1}{x_2 - x_1} \cdot (y_2 - y_1) \\ &= \frac{(x_2 - x)\, y_1 + (x - x_1)\, y_2}{x_2 - x_1} \end{aligned} \tag{5.3}$$

but we can do it by algebra. Let the line be given by

$$y = a_0 + a_1 x \tag{5.4}$$

and since we have two defined values, we have two equations:

$$\begin{aligned} y_1 &= a_0 + a_1 x_1 \\ y_2 &= a_0 + a_1 x_2 \end{aligned} \tag{5.5}$$

which can be solved for a_0 and a_1:

$$\begin{aligned} a_1 &= \frac{y_2 - y_1}{x_2 - x_1} \\ a_0 &= \frac{x_1 y_2 - x_2 y_1}{x_1 - x_2} \end{aligned} \tag{5.6}$$

and by substituting for x we get

$$\begin{aligned} y &= a_0 + a_1 x \\ &= \frac{x_1 y_2 - x_2 y_1}{x_1 - x_2} + \frac{y_2 - y_1}{x_2 - x_1} x \end{aligned} \tag{5.7}$$

which, after some rearrangement, produces the same result as in (5.3).

5.3 Parabolic interpolation

Fig 5.4 shows the use of three defined points (at x_1, x_2 and x_3). Clearly we can expect a better result for $y(x)$ than a linear interpolation between x_1 and x_2, if we fit a parabola to the three points. In this way we involve the underlying function's curvature to some degree. It is not possible now to see the fitted function as distinct from the underlying function, as the two are so similar. The parabola is given by

$$y = a_0 + a_1 x + a_2 x^2 \tag{5.8}$$

and by using all three known values we write

$$\left.\begin{aligned} y_1 &= a_0 + a_1 x_1 + a_2 x_1^2 \\ y_2 &= a_0 + a_1 x_2 + a_2 x_2^2 \\ y_3 &= a_0 + a_1 x_3 + a_2 x_3^2 \end{aligned}\right\} . \tag{5.9}$$

As before, this yields the coefficients a_0, a_1 and a_2, and we express $y(x)$ using them. The reader is spared the details, but after some work the end result is

$$y(x) = \frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)} y_1 + \frac{(x - x_1)(x - x_3)}{(x_2 - x_1)(x_2 - x_3)} y_2 + \frac{(x - x_1)(x - x_2)}{(x_3 - x_1)(x_3 - x_2)} y_3 \, . \tag{5.10}$$

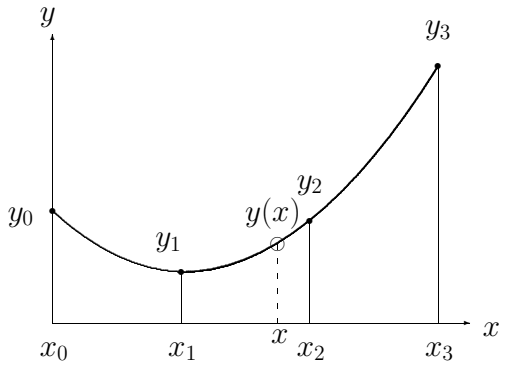

Figure 5.4: Second-order (parabolic) interpolation

5.4 General Lagrange formula

If first order is better than zero order, and second order better still, it is logical to assume that we will get even better results from increasing the order further, by including more and more defined points in order to calculate an interpolated value. An n-point interpolation is the polynomial

$$y = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1} \tag{5.11}$$

and it becomes more and more awkward to find the coefficients as n increases. Fortunately, this is done for us by the Lagrange formula which, for n points is

$$\begin{aligned} y(x) &= \frac{(x-x_2)(x-x_3)\ldots(x-x_n)}{(x_1-x_2)(x_1-x_3)\ldots(x_1-x_n)} y_1 \\ &+ \frac{(x-x_1)(x-x_3)\ldots(x-x_n)}{(x_2-x_1)(x_2-x_3)\ldots(x_2-x_n)} y_2 \\ &+ \vdots \\ &+ \frac{(x-x_1)(x-x_2)\ldots(x-x_{n-1})}{(x_n-x_1)(x_n-x_2)\ldots(x_n-x_{n-1})} y_n \ , \end{aligned} \tag{5.12}$$

where the i^{th} coefficient (for y_i) in the sum is computed from all products $(x - x_j)$ for all $j \neq i$, in the numerator, and all products $(x_i - x_j)$ for all $j \neq i$ in the denominator. Note that (5.10) is a special case of (5.12) with $n = 3$.

5.5 Neville method

The Lagrange method, shown above, is a little unwieldy. With some argument it is possible to achieve an easier method, which is fully equivalent to the Lagrange method. It furthermore has the advantage of providing an estimate

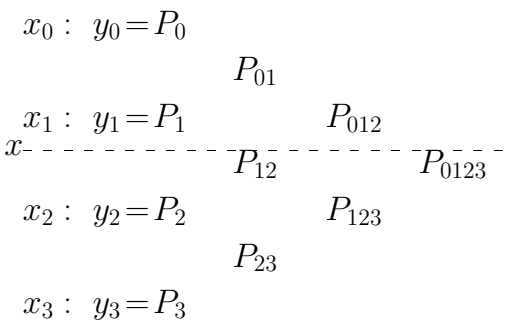

Figure 5.5: Neville diagram

of the error in the calculated $y(x)$ value, which the Lagrange method does not yield.

The principle is to go upward in the order from zero to $(n-1)$, and to improve the new estimates of $y(x)$ as a function of those previously calculated. Regarding Fig. 5.2, we have two possible zero-order interpolations for $y(x)$: y_1 and y_2. These two thus constitute a set of zero-order approximations, in Neville method terminology called P_0, P_1, etc., where the number of indices show the number of points used for the particular interpolation, here 1. The two, P_1 og P_2, can now lead to a 2-point first-order interpolation ($y(x)$ shown in Fig. 5.2); in Sect. 5.1 we obtained an expression for this, and we now use the two zero-order functions $P_1 = y_1$ and $P_2 = y_2$, and get

$$P_{12} = \frac{(x_2 - x)\ P_1 + (x - x_1)\ P_2}{(x_2 - x_1)} \ , \tag{5.13}$$

that is, we have improved our two roughest formulae, P_1 and P_2, to a better one, P_{12}. There is a general recursive formula for all P. The word "recursive" means that a new P-polynomial is computed from lower polynomials. The formula is

$$P_{i(i+1)\ldots(i+m)} = \frac{(x - x_{i+m})\, P_{i(i+1)\ldots(i+m-1)} + (x_i - x)\, P_{(i+1)(i+2)\ldots(i+m)}}{x_i - x_{i+m}} \ . \tag{5.14}$$

P_{12} can be derived from the formula by substituting $i = 1$ and $m = 1$, which leads to (5.3). The "Neville diagram", seen in Fig. 5.5 for four levels, makes the process clearer. In practice we choose (that is, the subroutine `POLINT` chooses) the polynomial based on the desired number of points to be used, and which lies closest to the x values at which the interpolation is to be computed. For example, in Fig. 5.5, we could use P_1 $(n = 1)$, P_{12} $(n = 2)$, P_{012} $(n = 3)$ or P_{0123} $(n = 4)$.

In fact, `POLINT` makes use of a further improvement of the method. It is not necessary to compute all P values from scratch. Instead the subroutine

computes changes from one P level to the next higher, using again recursive formulae to do this.

Let us say that we have the approximations P_1 and P_2, and use them to compute P_{12}. We can express a change C_{11}, so that

$$P_{12} = P_1 + C_{11} \tag{5.15}$$

and therefore the new approximation becomes

$$y_{app} = P_1 + C_{11} \ . \tag{5.16}$$

C_{11} functions as a correction to P_1. In the same manner we can compute another to correct P_2. The procedure is (see Fig. 5.5) that when we go downwards in the Neville diagram, we produce corrections C and when we go upwards we produce corrections D. They are marked with subscripts C_{mi} and D_{mi}, given by expressions

$$C_{mi} = P_{i\ldots(i+m)} - P_{i\ldots(i+m-1)} \tag{5.17}$$

and

$$D_{mi} = P_{i\ldots(i+m)} - P_{i+1\ldots(i+m)} \ . \tag{5.18}$$

Fig. 5.6 shows the picture. The first index numbers along the row, the other the level downward.

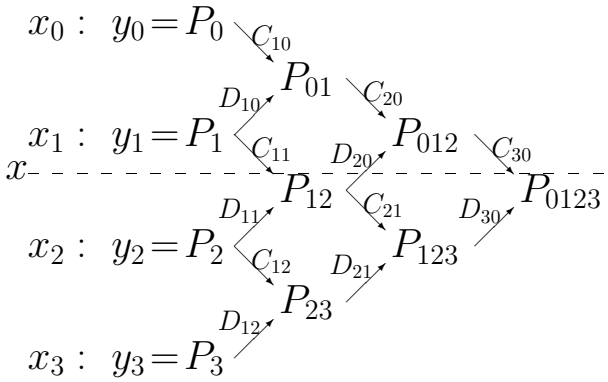

Figure 5.6: Neville diagram indicating C and D

There is a recursive expression for this as well:

$$C_{m+1,i} = \frac{(x_i - x)(C_{m,i+1} - D_{m,i})}{x_i - x_{i+m+1}} \tag{5.19}$$

and

$$D_{m+1,i} = \frac{(x_{i+m+1} - x)(C_{m,i+1} - D_{m,i})}{x_i - x_{i+m+1}} \ . \tag{5.20}$$

The interpolation is finally a sum of these changes plus that y value that lies closest to x. The procedure is not only easier, but it also finally provides an error estimate, equal to to the last computed change. In our example the best four-point approximation will be the sum

$$y_{app} = P_2 + C_{11} + D_{20} + C_{30} = P_{0123} \tag{5.21}$$

and the error estimate is equal to C_{30}, the term computed last.

Which is more effective? We can count the operations carried out. One often assumes that an addition (or subtraction) and a multiplication (or division) take the same amount of computer time, so we need only count these together. Looking at the formulae, we find that the recursive P computations require the least time, and the other two are very similar in this respect. We nevertheless prefer the method of changes C and D, even though it takes more time, because of the error estimate we gain.

5.6 Hermitian interpolation

There is a way of raising the error order of a polynomial interpolation without using more points, invented by the French mathematician Hermite. The essence of the method is that as well as function values at the given points, function derivatives at the same points are also made use of. The method is described very clearly by Kopal [6, p.31]. In the book, he presents the following table:

$$\begin{array}{llll} f(x_0) & f'(x_0) & f''(x_0) & \cdots \\ f(x_1) & f'(x_1) & f''(x_1) & \cdots \\ f(x_2) & f'(x_2) & f''(x_2) & \cdots \\ \vdots & \vdots & \vdots & \\ f(x_n) & f'(x_n) & f''(x_n) & \cdots \end{array}$$

What we have seen until now makes use of the first column, the function values at the $n+1$ points $x_0 \cdots x_n$. It is also possible to use any given row, which amounts to using the Taylor expansion around a single point, with the well known weights. The Hermitian method uses the first two columns. From $n+1$ points, we can then write $2n+2$ equations using the known function values and their derivatives. For compactness, we write $y_i = f(x_i)$ here. Let us progress from the simplest one-point interpolation near the point at x_1, but now fitting a straight line

$$y = a_0 + a_1 x \ ; \tag{5.22}$$

we can write two equations:

$$\begin{aligned} y_1 &= a_0 + a_1 x_1 \\ y_1' &= a_1 \ . \end{aligned} \tag{5.23}$$

This easily yields the two coefficients, and we have raised the zero-order interpolation based on a single point from zero to first order.

We now use two points, x_1 and x_2, and fit a second-order polynomial

$$y = a_0 + a_1 x + a_2 x^2 \tag{5.24}$$

to them. We have a choice of using the derivative at x_1 or x_2 and we choose the first of these. So we can write the three equations

$$\begin{aligned} y_1 &= a_0 + a_1 x_1 + a_2 x_1^2 \\ y_2 &= a_0 + a_1 x_2 + a_2 x_2^2 \\ y_1' &= \quad a_1 + 2a_2 x_1 \ . \end{aligned} \tag{5.25}$$

This can readily be solved for the three coefficients, and we obtain a second order formula. However, why not use both known derivatives, and fit a cubic equation

$$y = a_0 + a_1 x + a_2 x^2 + a_3 x^3 \tag{5.26}$$

to the points? We write the four equations

$$\begin{aligned} y_1 &= a_0 + a_1 x_1 + a_2 x_1^2 + a_3 x_1^3 \\ y_2 &= a_0 + a_1 x_2 + a_2 x_2^2 + a_3 x_2^3 \\ y_1' &= \quad a_1 + 2a_2 x_1 + 3a_3 x_1^2 \\ y_2' &= \quad a_1 + 2a_2 x_2 + 3a_3 x_2^2 \end{aligned} \tag{5.27}$$

which again can be solved for the coefficients (using a linear solver), resulting in a third order interpolation.

Kopal presents results of the third order formula applied to an interpolation between two sine function values, at $x = 0.4$ and $x = 0.6$, at $x = 0.5$. We follow Kopal here and the above formulae produce the results in Table 5.1. The advantages of using this methods are clear.

Table 5.1: Errors for interpolation at $x = 0.5$ between two sine function values. True value: 0.479426

	Interpolated value	Error	% error
Simple one-point formula	0.389418	-0.090007	-18.7740
Hermitian one-point formula	0.481524	0.002009	0.4378
Two-point parabolic formula	0.479277	-0.000148	-0.0309
Two-point cubic formula	0.479424	-0.000002	-0.0004

Chapter 6

Least Squares Fitting

We often deal with a number of (x, y) points, either measured or calculated, which all carry errors in the y direction; in most cases there are no errors to speak of in the x direction. In many cases we have an idea of a function type that fits the point sequence, and the task is to find those values for the adjustable function parameters that allow the function to fit the points optimally.

Another situation is that, rather than the exact function that underlies the point sequence - that is a mathematical description of the physical properties of the measured or calculated values - we simply want to find some function that might closely match such an underlying function. There are certain functions that serve well for this, such as polynomials, sums of exponentials and sometimes trigonometrical functions (think of Fourier series). Here too we seek to find the trial function's parameters that provide a best fit.

Since there are errors in the y-values, the function $f(x)$ cannot go through all (or perhaps any) of the points, and we must define what we mean by a "best fit". Fig. 6.1 shows a number N of points all carrying errors, and a function that has been fitted such that it fits as well as possible, in a sense that will be defined below. Fig. 6.2 shows the details from the fit around one point, at

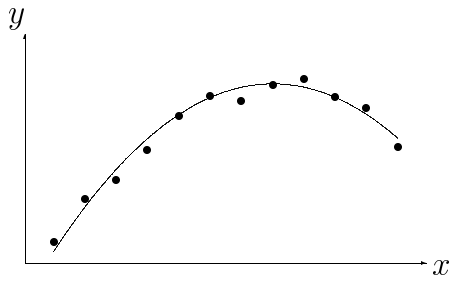

Figure 6.1: A point sequence and the fitted function

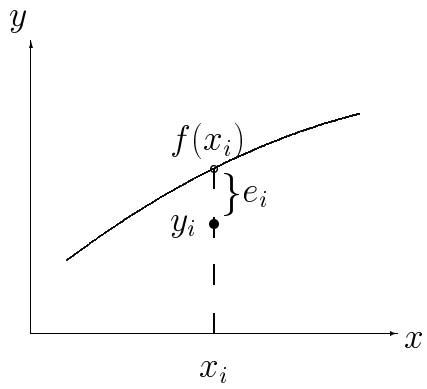

Figure 6.2: Detail out of Fig. 6.1

x_i, and the fitted function $f(x_i)$, where the point's value y_i deviates from the function value with an error equal to e_i. We shall operate with these errors in order to find a best-fit function.

The least squares method consists of defining the "best fit" such that one finds those function parameters in the fitted function $f(x)$, that result in a minimum sum S of squared errors:

$$S = \sum_{i=1}^{N} e_i^2 = \sum_{i=1}^{N} \left(y_i - f(x_i)\right)^2 \ . \tag{6.1}$$

Often we know the standard deviation of the values, and they may in principle be different for all points - for example, if different measurement ranges were used for the measurements. We can generalise the above formula using the variances σ_i^2 and this produces the quantity χ^2:

$$\chi^2 = \sum_{i=1}^{N} \left(\frac{y_i - f(x_i)}{\sigma_i} \right)^2 \tag{6.2}$$

that we seek to minimise. By dividing each error by its standard deviation, we make it relative, so that what we now minimise is the sum of the relative squared errors. As we will see, χ^2 also carries information about the goodness of the fit. If the σ_i are not known, we set them all equal to unity and obtain S. This does not provide any information on the goodness of fit, but does provide a best fit.

Let us say that we fit a function $f(x, p_1, p_2, \ldots p_M)$, where the set $p_1, p_2, \ldots, p_M$ are M parameters that describe the function. For example, if the function is a straight line, we have two parameters, $f(x) = p_1 + p_2 x$. A fit means finding the best set of parameters. For each of them, χ^2 should show a minimum,

which sets the condition

$$\frac{\partial \chi^2}{\partial p_i} = 0 \ . \tag{6.3}$$

We must therefore differentiate (6.1) or (6.2) with respect to all parameters, and set all derivatives to zero. This produces a set of M equations to solve for all M parameters, with more or less difficulty, depending on the function, which might result in an easy set of linear equations or a more complicated nonlinear equation set to solve. Both situations will be described here.

6.1 Linear least squares

In many cases we have to do with polynomials. The word "linear" here refers not to the fitted function but the linear equation system that yields the function parameter values. A polynomial of degree $M-1$ is the function

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{M-1} x^{M-1} \ , \tag{6.4}$$

with M parameters.

Constant

We begin with a simple example to illustrate the method, using a "polynomial" that is just a constant ($M = 1$):

$$f(x) = a_0 \ . \tag{6.5}$$

This is to be fitted to the set of points shown in Fig. 6.3 (the straight line shows the fitted function).

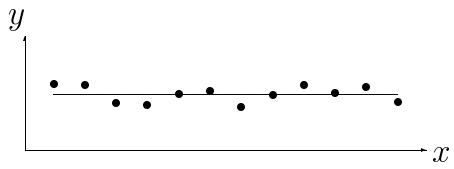

Figure 6.3: Set of points and the fitted function, here a constant

We make use of (6.2), that is, we know the standard deviations for all points in the set. There is only one parameter, a_0, so we differentiate with respect to it, and get the single equation

$$\frac{\partial \chi^2}{\partial a_0} = 2\sum_{i=1}^{N} \frac{(y_i - a_0)}{\sigma_i}(-1) = 0 \tag{6.6}$$

which gives us

$$a_0 = \frac{\sum_{i=1}^{N} y_i/\sigma_i^2}{\sum_{i=1}^{N} 1/\sigma_i^2} \; . \tag{6.7}$$

If all σ_i are the same, this becomes

$$a_0 = \frac{1}{N}\sum_{i=1}^{N} y_i \tag{6.8}$$

which we recognise as the mean value of all y_i (as we might have expected).

Straight (regression) line

Now we set $M = 2$, that is, we fit a straight line to the N points, so

$$f(x) = a_0 + a_1 x \; . \tag{6.9}$$

Here we must differentiate (6.2) with respect to both a_0 and a_1, and we get the two equations

$$\begin{aligned} \frac{\partial \chi^2}{\partial a_0} &= 2\sum_{i=1}^{N} \left(\frac{y_i - a_0 - a_1 x_i}{\sigma_i^2} \right)(-1) = 0 \\ \frac{\partial \chi^2}{\partial a_1} &= 2\sum_{i=1}^{N} \left(\frac{y_i - a_0 - a_1 x_i}{\sigma_i^2} \right)(-x_i) = 0 \end{aligned} \tag{6.10}$$

which produce the linear system

$$\begin{aligned} a_0 \sum_{i=1}^{N} \frac{1}{\sigma_i^2} + a_1 \sum_{i=1}^{N} \frac{x_i}{\sigma_i^2} &= \sum_{i=1}^{N} \frac{y_i}{\sigma_i^2} \\ a_0 \sum_{i=1}^{N} \frac{x_i}{\sigma_i^2} + a_1 \sum_{i=1}^{N} \frac{x_i^2}{\sigma_i^2} &= \sum_{i=1}^{N} \frac{x_i y_i}{\sigma_i^2} \; . \end{aligned} \tag{6.11}$$

Here we introduce a convenient notation: let

$$S_{nm} \equiv \sum_{i=1}^{N} \frac{x_i^n y_i^m}{\sigma_i^2} \; . \tag{6.12}$$

Then we can rewrite (6.11) as

$$\begin{aligned} S_{00}\, a_0 + S_{10}\, a_1 &= S_{01} \\ S_{10}\, a_0 + S_{20}\, a_1 &= S_{11} \end{aligned} \tag{6.13}$$

which has solutions

$$\begin{aligned} a_0 &= \frac{S_{20}S_{01} - S_{10}S_{11}}{\Delta} \\ a_1 &= \frac{S_{00}S_{11} - S_{10}S_{01}}{\Delta} \end{aligned} \tag{6.14}$$

in which the determinant Δ is

$$\Delta = S_{00}S_{20} - S_{10}^2 \, . \tag{6.15}$$

One way to solve this is to express (6.13) in matrix form:

$$\mathbf{A}\mathbf{a} = \mathbf{b} \tag{6.16}$$

where

$$\mathbf{A} \equiv \begin{bmatrix} S_{00} & S_{10} \\ S_{10} & S_{20} \end{bmatrix} \, , \tag{6.17}$$

$$\mathbf{a} \equiv \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \tag{6.18}$$

and

$$\mathbf{b} \equiv \begin{bmatrix} S_{01} \\ S_{11} \end{bmatrix} \, . \tag{6.19}$$

The solution is then

$$\mathbf{a} = \mathbf{A}^{-1}\mathbf{b} \tag{6.20}$$

and since this is only a 2×2 system, it is not difficult to solve, giving

$$\mathbf{A}^{-1} = \frac{1}{\Delta} \begin{bmatrix} S_{20} & -S_{10} \\ -S_{10} & S_{00} \end{bmatrix} \tag{6.21}$$

which leads to the solution of (6.14). We shall use the above in the next Section.

Parameter errors

We make use of the simple case from the above, $M = 2$, to describe how we can obtain the standard deviation (or its square, the variance) of the fitted function parameters; because there are errors on the point set, there must also be errors on the parameters calculated. The procedure is based on error propagation theory. A function f of several variables p_i, each of which has a standard deviation σ_i or a variance σ_i^2, has itself a variance σ_f^2, given by

$$\sigma_f^2 = \sum_i \sigma_i^2 \left(\frac{\partial f}{\partial p_i} \right)^2 . \tag{6.22}$$

Both a_0 and a_1 for the straight line function are functions of all y_i, which here are the variables carrying errors, and therefore we write (6.22)

$$\sigma_{a_0}^2 = \sum_{i=1}^{N} \sigma_i^2 \left(\frac{\partial a_0}{\partial y_i} \right)^2 \tag{6.23}$$

and

$$\sigma_{a_1}^2 = \sum_{i=1}^{N} \sigma_i^2 \left(\frac{\partial a_1}{\partial y_i} \right)^2 . \tag{6.24}$$

We always operate with the derivatives $\partial a_0/\partial y_i$ and $\partial a_1/\partial y_i$, for all $i = 1 \ldots N$. When we look at the solution for a_0 in (6.14) we note that there are some quantities that are independent of y, and some that depend on y. To emphasise the latter, we write the solution for a_0 in the form

$$a_0 = \frac{1}{\Delta} \left(S_{20} \sum_{i=1}^{N} \frac{y_i}{\sigma_i^2} - S_{10} \sum_{i=1}^{N} \frac{x_i y_i}{\sigma_i^2} \right) \tag{6.25}$$

(since Δ, S_{20} and S_{10} are independent of y). Now we can differentiate for a given i:

$$\frac{\partial a_0}{\partial y_i} = \frac{1}{\Delta} \left(S_{20} \cdot \frac{1}{\sigma_i^2} - S_{10} \cdot \frac{x_i}{\sigma_i^2} \right) \tag{6.26}$$

and immediately obtain, by using (6.22),

$$\sigma_{a_0}^2 = \sum_{i=1}^{N} \frac{1}{\Delta^2} \left\{ S_{20}^2 \cdot \frac{1}{\sigma_i^2} - 2 S_{20} S_{10} \cdot \frac{x_i}{\sigma_i^2} + s_{10}^2 \cdot \frac{x_i^2}{\sigma_i^2} \right\} . \tag{6.27}$$

This leads directly to

$$\sigma_{a_0}^2 = \frac{1}{\Delta^2} \left\{ S_{20}^2 S_{00} - 2 S_{20} S_{10} S_{10} + S_{10}^2 S_{20} \right\} \tag{6.28}$$

and after a little cleaning up (and considering (6.15)), this gives

$$\sigma_{a_0} = \frac{S_{20}}{\Delta} \,. \tag{6.29}$$

A similar procedure for a_1 yields

$$\sigma_{a_1} = \frac{S_{00}}{\Delta} \,. \tag{6.30}$$

There is however another method, rooted in the solution (6.20) of the matrix equation (6.16). Regarding the solution closely, that is, the matrix inverse (6.21), the above results for σ_{a_0} and σ_{a_1} are seen in the diagonal of the inverse. One can thus obtain the parameter uncertainties simply by inverting the matrix $\mathbf{A}$ in (6.16). For our straight line example we have the definition of $\mathbf{A}$ in equation (6.17), and the matrix is easily inverted to (6.21), from which we directly can read off the solutions (6.27) and (6.28).

This method is fruitful not only in fitting higher-degree polynomials, but also for nonlinear cases (see Sect. 6.4).

General polynomial

If we want to fit higher-order polynomials, we use essentially the same procedure as for the above simpler examples. The matrices become larger. For a polynomial as in (6.4), where we must estimate M parameters, we must solve (6.16) for $a_0 \ldots a_M$. We rewrite the polynomial expression in the (slightly) simpler form

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_p x^p \tag{6.31}$$

in which $p = M - 1$ is the polynomial degree. Then the matrix $\mathbf{A}$ is given by

$$\mathbf{A} = \begin{bmatrix} S_{00} & S_{10} & S_{20} & \ldots & S_{p0} \\ S_{10} & S_{20} & S_{30} & \cdots & \\ S_{20} & S_{30} & S_{40} & \cdots & \\ \vdots & & & & \vdots \\ S_{p0} & \ldots & & & S_{2p0} \end{bmatrix} \tag{6.32}$$

and

$$\mathbf{b} = [S_{01} \; S_{11} \; \ldots \; S_{p1})^T \tag{6.33}$$

with the S terms as defined in (6.12). For a parabola we then obtain

$$\begin{bmatrix} S_{00} & S_{10} & S_{20} \\ S_{10} & S_{20} & S_{30} \\ S_{20} & S_{30} & S_{40} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} S_{01} \\ S_{11} \\ S_{21} \end{bmatrix} \tag{6.34}$$

or, explicitly

$$\begin{bmatrix} \sum_{i=1}^{N} \frac{1}{\sigma_i^2} & \sum_{i=1}^{N} \frac{x_i}{\sigma_i^2} & \sum_{i=1}^{N} \frac{x_i^2}{\sigma_i^2} \\ \sum_{i=1}^{N} \frac{x_i}{\sigma_i^2} & \sum_{i=1}^{N} \frac{x_i^2}{\sigma_i^2} & \sum_{i=1}^{N} \frac{x_i^3}{\sigma_i^2} \\ \sum_{i=1}^{N} \frac{x_i^2}{\sigma_i^2} & \sum_{i=1}^{N} \frac{x_i^3}{\sigma_i^2} & \sum_{i=1}^{N} \frac{x_i^4}{\sigma_i^2} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{N} \frac{y_i}{\sigma_i^2} \\ \sum_{i=1}^{N} \frac{x_i\, y_i}{\sigma_i^2} \\ \sum_{i=1}^{N} \frac{x_i^2\, y_i}{\sigma_i^2} \end{bmatrix} . \tag{6.35}$$

The linear system $\mathbf{Aa} = \mathbf{b}$ must be solved for all parameters. We will normally not do this by inverting $\mathbf{A}$, but if we do, then we obtain at the same time the parameter variances lying on the inverse diagonal, as described in the previous section. As well, as will be shown in Chap. 7, there are quite effective algorithms for matrix inversion.

For a straight line the system reduces to (6.17), and the solution to (6.21).

6.2 Unknown errors

In the above, it was assumed that we know all standard deviations σ_i; it was mentioned that we arbitrarily set them all to unity if we do not know them. We can then calculate the set $\mathbf{a}$ for a minimum S, which is a scaled expression for χ^2, with an unknown scaling factor. In this case we cannot judge the goodness of fit, other than by a visual judgement (which in fact is often not bad). The various fitted polynomials of varying degrees shown in Fig. 6.4 clearly show the difference between the poor fits for $M = 2$ and $M = 3$, as well as the too-good fits for, for example, $M \geq 9$.

6.3 Goodness of fit

If we have chosen a suitable function to fit to a set of points, then the minimal $\chi^2/(N - M)$ should reflect the variances σ_i^2. In other words, the fitted curve should lie within the standard deviation limits for all points. We might choose a poor fitting function and this will be reflected in the value of $\chi^2/(N-M)$. An unsuitable fit can be too poor, but also too good. The determining quantity is $\chi^2/(N - M)$, where $N - M$ is the number of degrees of freedom. Often $N >> M$, and we can simply use χ^2/N. There are three cases:

$$\begin{aligned} \chi^2/(N-M) &>> 1 \quad \text{(Poor fit)} \\ \chi^2/(N-M) &\approx 1 \quad \text{(Good fit)} \\ \chi^2/(N-M) &<< 1 \quad \text{(Too-good fit)} \end{aligned} . \tag{6.36}$$

Fig. 6.4 shows some examples. The first case holds if the fitted function does not fit, and lies outside the standard deviation limits of the points, for example,

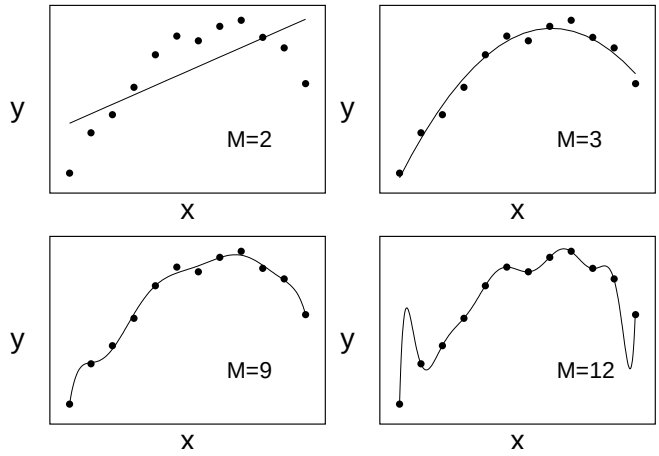

Figure 6.4: Some polynomial fits to a set of 12 points

for $M = 2$ in Fig. 6.4, where a straight line was "fitted" to to a set of points that clearly do not lie on a straight line. The last case can occur if one chooses a polynomial of too high a degree, so that the fitted function passes precisely through all points. We see a hint of this for $M = 9$, and most clearly for $M = N = 12$, where the function passes through all points exactly. Polynomials of intermediate degree work best, here $M = 3$. Table 6.1 shows the $\chi^2/(N - M)$ values for all M values.

We note no great difference in the region $3 \leq M \leq 8$, but for $M = 3$ we have the value closest to ideal, unity. For $M > 8$ the values decrease towards zero, indicating a too-good fit.

Table 6.1: $\chi^2/(N - M)$ for fitted functions in Fig. 6.4 against M

M	$\chi^2/(N - M)$
2	10.8
3	0.56
4	0.38
5	0.42
6	0.50
7	0.51
8	0.45
9	0.37
10	0.10
11	0.02
12	0

6.4 Nonlinear least squares

If we want to fit a function such that the derivatives of the expression for χ^2 are nonlinear in the function parameters, the solution is not so easy. Such a function could for example be

$$y = a_1 \exp(a_2 x) \tag{6.37}$$

where we have a problem with a_2: writing the usual equation for χ^2:

$$\chi^2 = \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \left\{ y_i - a_1 \exp(a_2 x_i) \right\}^2 \tag{6.38}$$

and minimising χ^2 by searching for $\partial\chi^2/\partial a_1 = \partial\chi^2/\partial a_2 = 0$, it becomes clear that we cannot simply solve the resulting equation system:

$$\begin{aligned} \frac{\partial \chi^2}{\partial a_1} &= 0 = -2 \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \left\{ y_i - a_1 \exp(a_2 x_i) \right\} \exp(a_2 x_i) \\ \frac{\partial \chi^2}{\partial a_2} &= 0 = -2 \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \left\{ y_i - a_1 \exp(a_2 x_i) \right\} a_1 x_i \exp(a_2 x_i) \end{aligned} \tag{6.39}$$

so that we must use other methods, such as are described elsewhere, for example in Press & al. [4]. There is however a relatively easy method of getting around the problem, that is, the linearisation method. If we already have a set of parameter values $(a_1, a_2, \ldots, a_M)$, that are reasonably close to the solution set to be found, we can use the Taylor expansion to correct this set. Let the set we have be $\mathbf{a}$ and let the vector $\mathbf{F}$ have elements $\partial\chi^2/\partial a_i, i = 1 \ldots M$, all of which are subequations to be set to zero. A correction set $\delta\mathbf{a}$ should then produce

$$\mathbf{F}\left(\mathbf{a}+\delta\mathbf{a}\right) = 0 \tag{6.40}$$

and from a Taylor expansion we have

$$\mathbf{F}(\mathbf{a}+\delta\mathbf{a}) = \mathbf{F}(\mathbf{a}) + \mathbf{F}'(\mathbf{a}) \cdot \delta\mathbf{a} + \ldots \tag{6.41}$$

(ignoring terms of higher order than $\mathbf{F}'$). Then it follows that

$$\mathbf{F}(\mathbf{a}) + \mathbf{F}'(\mathbf{a}) \cdot \delta\mathbf{a} = 0 \tag{6.42}$$

which yields a set of equations linear in the correction set $\delta\mathbf{a}$. It is necessary to evaluate both $\mathbf{F}$ and $\mathbf{F}'$, the matrix of derivatives, but this can always be done. Once we have solved for $\delta\mathbf{a}$, we may find that $\mathbf{F}\left(\mathbf{a} + \delta\mathbf{a}\right)$ is still not equal to zero, because we have truncated the Taylor series; but normally, we will have advanced closer to the solution. We add $\delta\mathbf{a}$ to $\mathbf{a}$ and use the sum as the new set $\mathbf{a}$. Normally, this process must be repeated a number of times, before an

acceptable set **a** is obtained. We must define what is acceptable. This can for example be that some norm of the vector **F** is less than some number we have set as our aim.

Let us take the above function $y = a_1 \exp(a_2 x)$ as an example. Here,

$$\frac{\partial \chi^2}{\partial a_1} = F_1 \qquad \text{og} \qquad \frac{\partial \chi^2}{\partial a_2} = F_2 \tag{6.43}$$

that is,

$$\begin{aligned} F_1(a_1, a_2) &= -2 \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \{y_i - a_1 \exp(a_2 x_i)) \exp(a_2 x_i) \\ F_2(a_1, a_2) &= -2 \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \{y_i - a_1 \exp(a_2 x_i)\} a_1 x_i \exp(a_2 x_i) \end{aligned} \tag{6.44}$$

and we have for the correction δa_1 and δa_2,

$$\begin{aligned} F_1(a_1{+}\delta a_1, a_2{+}\delta a_2) &= F_1(a_1, a_2) + \delta a_1 \frac{\partial F_1(a_1, a_2)}{\partial a_1} + \delta a_2 \frac{\partial F_1(a_1, a_2)}{\partial a_2} = 0 \\ F_2(a_1{+}\delta a_1, a_2{+}\delta a_2) &= F_2(a_1, a_2) + \delta a_1 \frac{\partial F_2(a_1, a_2)}{\partial a_1} + \delta a_2 \frac{\partial F_2(a_1, a_2)}{\partial a_2} = 0 \end{aligned} \tag{6.45}$$

or

$$\begin{bmatrix} \dfrac{\partial F_1}{\partial a_1} & \dfrac{\partial F_1}{\partial a_2} \\ \dfrac{\partial F_2}{\partial a_1} & \dfrac{\partial F_2}{\partial a_2} \end{bmatrix} \begin{bmatrix} \delta a_1 \\ \\ \delta a_2 \end{bmatrix} = \begin{bmatrix} -F_1 \\ \\ -F_2 \end{bmatrix} \tag{6.46}$$

(where F_1 and F_2 denote $F_1(a_1, a_2)$ etc., with the present, uncorrected parameters a_1 and a_2). The derivatives in the matrix are in fact double derivatives of χ^2 (see the definitions of F_1 and F_2), so we can also write the equation in the form

$$\begin{bmatrix} \dfrac{\partial^2 \chi^2}{\partial a_1^2} & \dfrac{\partial^2 \chi^2}{\partial a_1 \partial a_2} \\ \dfrac{\partial^2 \chi^2}{\partial a_2 \partial a_1} & \dfrac{\partial^2 \chi^2}{\partial a_2^2} \end{bmatrix} \begin{bmatrix} \delta a_1 \\ \\ \delta a_2 \end{bmatrix} = \begin{bmatrix} -\dfrac{\partial \chi^2}{\partial a_1} \\ -\dfrac{\partial \chi^2}{\partial a_2} \end{bmatrix} . \tag{6.47}$$

We then solve $\mathbf{J} \cdot \delta \mathbf{a} = -\mathbf{F}$ (**J** being the Jacobian).

The double derivatives (there are three) are

$$\begin{aligned}
\frac{\partial^2\chi^2}{\partial a_1^2} &= 2\sum_{i=1}^{N}\frac{\exp(2a_2x_i)}{\sigma_i^2} \\
\frac{\partial^2\chi^2}{\partial a_1\partial a_2} &= -2\sum_{i=1}^{N}\frac{x_iy_i\exp(a_2x_i)-2a_1x_i\exp(2a_2x_i)}{\sigma_i^2} \qquad (6.48)\\
\frac{\partial^2\chi^2}{\partial a_2^2} &= -2\sum_{i=1}^{N}\frac{a_1x_i^2y_i\exp(a_2x_i)-2a_1^2x_i^2\exp(2a_2x_i)}{\sigma_i^2}\,. \qquad (6.49)
\end{aligned}$$

Finally, as always, we are interested in the uncertainties of the computed parameters, or their variances. They are the same as the variances of the corrections $\delta\mathbf{a}$. As before, for general least squares, they are found on the diagonal of the inverse matrix $\mathbf{J}^{-1}$.

The process described above can be generalised for fitting a general function $f(x,\mathbf{a})$, where the vector $\mathbf{a}$ consists of a number M of parameters. We always solve the equation $\mathbf{J}\cdot\delta\mathbf{a}=-\mathbf{F}$, the matrix $\mathbf{J}$ being

$$\mathbf{J}=\begin{bmatrix}
\frac{\partial^2\chi^2}{\partial a_1^2} & \frac{\partial^2\chi^2}{\partial a_1\partial a_2} & \cdots & \frac{\partial^2\chi^2}{\partial a_1\partial a_M}\\
\frac{\partial^2\chi^2}{\partial a_2\partial a_1} & \frac{\partial^2\chi^2}{\partial a_2^2} & \cdots & \frac{\partial^2\chi^2}{\partial a_2\partial a_M}\\
\vdots & & & \vdots\\
\frac{\partial^2\chi^2}{\partial a_M\partial a_1} & \frac{\partial^2\chi^2}{\partial a_M\partial a_2} & \cdots & \frac{\partial^2\chi^2}{\partial a_M^2}
\end{bmatrix}\qquad\text{and}\qquad
\mathbf{F}=\begin{bmatrix}
\frac{\partial\chi^2}{\partial a_1}\\
\frac{\partial\chi^2}{\partial a_2}\\
\vdots\\
\frac{\partial\chi^2}{\partial a_M}
\end{bmatrix} \qquad (6.50)$$

and, as above, the inverse of $\mathbf{J}$ yields the variances of the parameters on the diagonal.

Chapter 7

Linear Systems and Matrices

7.1 Elementary definitions

A (real or complex) vector space $\mathbf{V}$ over a given vector field $\mathbf{F}$ consists of a set of vectors

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad , \qquad \mathbf{x^T} = \begin{bmatrix} x_1, x_2, \ldots, x_n \end{bmatrix} \ .$$

The following operations are of interest. The sum

$$\mathbf{x} + \mathbf{y} = \begin{bmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \cdots \\ x_n + y_n \end{bmatrix} \ ,$$

and the outer (direct) product

$$\mathbf{x} \otimes \mathbf{y^T} = \begin{bmatrix} x_1\mathbf{y^T} \\ x_2\mathbf{y^T} \\ \cdots \\ x_n\mathbf{y^T} \end{bmatrix} \quad \text{i.e. matrix} \quad \begin{bmatrix} x_1y_1 & x_1y_2 & x_1y_3 & \ldots & x_1y_n \\ x_2y_1 & x_2y_2 & x_2y_3 & \ldots & x_2y_n \\ \cdots & & & & \\ x_ny_1 & x_ny_2 & x_ny_3 & \ldots & x_ny_n \end{bmatrix} \ .$$

A matrix $\mathbf{A}$ and its transpose $\mathbf{A^T}$ are defined as

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{bmatrix} \ , \mathbf{A^T} = \begin{bmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & \cdots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \ldots & a_{mn} \end{bmatrix} \ .$$

Matrix $\mathbf{A}$ is orthonormal if

$$\mathbf{A^T A} = \mathbf{I}$$

where **I** is the unit matrix. Before going further, we present some commonly occurring forms of "sparse"matrices, that is, matrices with many zeroes systematically distributed through the matrix:

$$\begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_{nn} \end{bmatrix} \quad \text{(diagonal)},$$

$$\begin{bmatrix} a_{11} & a_{12} & 0 & \cdots & 0 \\ a_{21} & a_{22} & a_{23} & \cdots & 0 \\ \ddots & \ddots & \ddots & & \vdots \\ & & \ddots & \ddots & \ddots \\ 0 & 0 & a_{m-1,n-2} & a_{m-1,n-1} & a_{m-1,n} \\ 0 & 0 & \cdots & a_{m,n-1} & a_{mn} \end{bmatrix} \quad \text{(tridiagonal)},$$

$$\begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \quad \text{(lower triangular)},$$

$$\begin{bmatrix} a_{11} & a_{12} & & \cdots & a_{1n} \\ 0 & a_{22} & a_{23} & \cdots & a_{2n} \\ \vdots & & \ddots & \ddots & \vdots \\ & \cdots & & a_{m-1,n-1} & a_{m-1,n} \\ 0 & \cdots & & 0 & a_{mn} \end{bmatrix} \quad \text{(upper triangular)}.$$

7.2 Vector- and matrix norms

The norm of a vector or matrix is written as $\|\mathbf{A}\|$. The following statements hold for the norm of a vector or matrix:

1. The norm must be larger than zero if not all of the elements of the vector or matrix are zero, i.e.
   $\|\mathbf{A}\| > 0$; $\|\mathbf{A}\| = 0$ **iff** $\mathbf{A}$=0 (where **iff** means "if and only if")

2. The norm is multiplied by the scalar k's magnitude if all elements of the vector or matrix are multiplied by k, i.e.
   $\|k\mathbf{A}\| = |k| \, \|\mathbf{A}\|$

3. The norm of the sum of two vectors or matrices does not exceed the sum of their respective norms, i.e.
   $\|\mathbf{A} + \mathbf{B}\| \leq \|\mathbf{A}\| + \|\mathbf{B}\|$

4. The norm of the product of the norms of two vectors or matrices does not exceed the product of the two respective norms, i.e. $\|\mathbf{AB}\| \leq \|\mathbf{A}\|\|\mathbf{B}\|$.

There are several definitions of norms for vectors and matrices. For vectors the general form is

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p} . \tag{7.1}$$

The Euclidian norm $\|\mathbf{x}\|_2$ (or 2-norm) is given by $p = 2$, and the infinity norm $\|\mathbf{x}\|_\infty$ is defined for $p \to \infty$, as $\max_{1 \leq i \leq n} |x_i|$, where n is the dimension of the vector.

A matrix norm can be defined in various ways, in a similar manner to that of a vector. For example, for $p = 1$,

$$\|\mathbf{A}\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^{n} |a_{ij}| \tag{7.2}$$

which is the largest column sum, while the infinity norm here is

$$\|\mathbf{A}\|_\infty = \max_{1 \leq i \leq n} \sum_{j=1}^{n} |a_{ij}| \tag{7.3}$$

which is the largest row sum.

7.3 Linear transformation

A linear transformation can be considered as follows. Assume that we have a vector $\mathbf{x}$, which, by means of an operator $\mathbf{A}$, is transformed into a new vector $\mathbf{y}$ (within the same vector space). If $\mathbf{A}$ represents the transformation matrix, this process is described by a matrix-vector multiplication of the type

$$\mathbf{y} = \mathbf{Ax} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} , \quad y_i = \sum_{j=1}^{m} a_{ij} x_j . \tag{7.4}$$

7.4 Matrix multiplication.

The product $\mathbf{C}$ of the $m \times n$ matrix $\mathbf{A}$ and the $n \times p$ matrix $\mathbf{B}$ is given by

$$\mathbf{C} = \mathbf{AB} = \begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1p} \\ C_{21} & C_{22} & \cdots & C_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ C_{m1} & C_{m2} & \ldots & C_{mp} \end{bmatrix} , \qquad C_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} , \tag{7.5}$$

where the dimension of the product matrix $\mathbf{C}$ is $m \times p$. Note that element C_{ij} is the sum of the products of the elements in the i^{th} row in $\mathbf{A}$ and the corresponding elements in the j^{th} column in $\mathbf{B}$. Matrix multiplication is distributive, that is,

$$\mathbf{A}\left(\mathbf{B}+\mathbf{C}\right)=\mathbf{AB}+\mathbf{AC} \qquad \text{og} \qquad \left(\mathbf{B}+\mathbf{C}\right)\mathbf{D}=\mathbf{BD}+\mathbf{CD}$$

but not necessarily commutative; that is,

$$\mathbf{AB} \neq \mathbf{BA}$$

will often be the case.

Since matrix multiplication is central to numerical computation, it is useful to understand how the computer implements the procedure. For convenience, let matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ all be $n \times n$ in dimension. Multiplication as defined above is carried out in Fortran90 using the operation (intrinsic function) `MATMUL` after suitable declaration of the matrices. One might program the operation like this:

```
c = 0
do i = 1,n
  do j = 1,n
    do k = 1,n
      c(i,j) = c(i,j) + a(i,k)*b(k,j)
    enddo
  enddo
enddo
```

Note that `c` is set to zero initially.

If we define a *flop* (floating point operation) as an addition (or subtraction) or a multiplication (or division), both of which we have in the middle line, a matrix multiplication will require $2n^3$ flops. Note that a number of flops per second is often used as a measure of a computer's speed (about 4000 Mflop for `uran2`, corresponding to 2.5×10^{-10} s/flop). For normal 32-bit real numbers (4 bytes of 8 bits each), a 1000×1000 matrix will take up $4n^2$ or 4 Mb. The computer memory has a finite capacity, and typically resides partly in RAM (Random Access Memory) with "pages" (e.g. 512 kb) and partly in virtual memory (some Gb on a hard disk) - it is of some importance that the above operation make use of the computer's storage of matrices in order to minimise paging and (especially) disk reading. Disk reads are up to two orders of magnitude slower that memory access. To get an impression of the processing speed for matrix multiplication it can be mentioned that the machine `uran2` can perform a 100×100 matrix multiplication in 1.6 ms, but takes 1.6 s for a 1000×1000 multiplication (written in 2010).

Since processing times for large matrices, even on fast computers, are significant, it is worth the trouble to optimise matrix multiplication - for example,

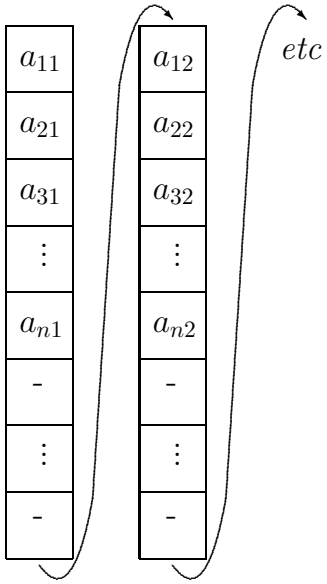

Figure 7.1: Fortran's order of matrix storage in memory

by minimising unneccesary data transport. Assume that the three matrices **A**, **B** and **C** have *physical* dimensions $m \times m$ (dimensions defined in the declarations of three variables), while the *logical* dimensions are $n \times n$ (the dimensions of the arrays actually to be used), where $n < m$. The computer stores the three matrices columnwise, as illustrated in Fig. 7.1.

If we now perform a matrix multiplication according to the above scheme, the computation of, for example, the element C_{11} will involve multiplication of element numbers 1 (in **A**) and n^2+1 (in **B**), $n+1$ and n^2+2, etc. and this will entail many large jumps in memory. These can be reduced considerably by using the following procedure:

```
c = 0
do j = 1,n
  do k = 1,n
    do i = 1,n
      c(i,j) = c(i,j) + a(i,k)*b(k,j)
    enddo
  enddo
enddo
```

where the corresponding operations involve elements 1 and n^2+1, 2 and n^2+2 etc. We have changed the innermost loop operations (which go fastest) from row- to column operations, which make better use of the storage order in memory. As a general rule we should always strive for a minimum of large jumps in memory, which for matrix operations is achieved by ensuring that

the first index of an array runs fastest. A good F90/95 compiler might do this automatically for `MATMUL`. It may be added that programs with minimal paging generally also vectorise and parallelise effectively - which is of importance in the use of vector processors or clusters of computers.

7.5 Determinants

The (recursive) way of defining a determinant is also a recipe for evaluating it; but it is the least efficient way of doing it. There are some properties of matrices that indicate more efficient methods but let us see first how to implement the definition-based method. Let us say we have an $n \times n$ matrix $\mathbf{A}$. Its determinant is the sum of all the elements in a row (or column) multiplied with the determinant of the cofactor, with alternating signs along the elements, (sign= $(-1)^{(i+j)}$). The cofactor for the (i, j)th element is the smaller matrix resulting from the matrix $\mathbf{A}$ with the i^{th} row and the j^{th} column left out. For example, if

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 5 & 7 & 8 \end{bmatrix}$$

then the cofactor for the first row element a_{11} is the submatrix $\begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$, etc. The determinant of $\mathbf{A}$ is thus

$$\begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 5 & 7 & 8 \end{vmatrix} = 1 \begin{vmatrix} 5 & 6 \\ 7 & 8 \end{vmatrix} - 2 \begin{vmatrix} 4 & 6 \\ 5 & 8 \end{vmatrix} + 3 \begin{vmatrix} 4 & 5 \\ 5 & 7 \end{vmatrix}$$

which expands to

$$\begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 5 & 7 & 8 \end{vmatrix} = 1(5 \times 8 - 7 \times 6) - 2(4 \times 8 - 5 \times 6) + 3(4 \times 7 - 5 \times 5) = 3 \; . \quad (7.6)$$

The definition is recursive because it uses the word "determinant" within the definition. If the matrix had been larger than just 3×3, there would have been many more operations after expansion. One need not expand further than to the 2×2 matrix stage, where each determinant can be evaluated easily.

A number of matrix properties can be used to make the evaluation of a determinant more efficient:

1. If two rows of a matrix are exchanged, this changes the sign, but not the magnitude, of the determinant.

2. If one row is added to another, this does not change the sign nor the magnitude of the determinant.

3. If a matrix has been converted to an upper or lower triangular form or diagonalised, for example by Gauss elimination or LUD (see later), the determinant is given by the product of the diagonal elements.

As we shall see, items (1) og (2) arise from the solution of linear systems and are quite efficient. When making use of (3), one must keep track of how many times row exchange has been performed, because (1) indicates a change of sign for every pivoting operation. All this is described in Sect. 7.7. Efficient evaluation of a determinant is thus given by conversion of the matrix to upper or lower triangular or diagonal form.

7.6 Matrix inversion

Inversion of a matrix can be a time-consuming affair, unless certain tricks are used. The way inversion is often described is the least efficient method. We wish to invert the matrix $\mathbf{A}$. First we must calculate its determinant $|\mathbf{A}|$ (which in itself can be time-consuming). Then we transpose the matrix and every element of the inverse $\mathbf{V}$, V_{ij}, is then given by the cofactor of the corresponding element in $\mathbf{A}^T$ (that is, the determinant of the minor matrix resulting from leaving out the ith row and the jth column), and with a sign change at all even indices, just as with the calculation of a determinant. Here is a simple example.

$$\mathbf{A} \equiv \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 5 & 7 & 8 \end{bmatrix} .$$

The determinant is equal to 3, as was seen in (7.6). The inverse is now computed from the transposed matrix:

$$\mathbf{A}^T = \begin{bmatrix} 1 & 4 & 5 \\ 2 & 5 & 7 \\ 3 & 6 & 8 \end{bmatrix}$$

and the elements in the inverse $\mathbf{V}$ are the cofactors along a row or column:

$$V_{11} = \frac{1}{3}\begin{vmatrix} 5 & 7 \\ 6 & 8 \end{vmatrix} ,$$

$$V_{12} = -\frac{1}{3}\begin{vmatrix} 2 & 7 \\ 3 & 8 \end{vmatrix} ,$$

$$V_{13} = \frac{1}{3}\begin{vmatrix} 2 & 5 \\ 3 & 6 \end{vmatrix} ,$$

etc., and the final result is

$$\mathbf{V} = -\frac{1}{3}\begin{bmatrix} -2 & 5 & -3 \\ -2 & -7 & 6 \\ 3 & 3 & -3 \end{bmatrix} .$$

This method, as mentioned above, is very time-consuming, even using a computer, but can be used for matrices up to 3×3. For larger matrices it is better to use more efficient methods, see below in Sect. 7.7.

7.7 Solution of linear systems of equations

In chemistry and physics we often meet linear equation systems such as

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2n}x_n &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \cdots + a_{3n}x_n &= b_3 \\ \cdots & \qquad (7.7) \\ a_{m1}x_1 + a_{m2}x_2 + a_{m3}x_3 + \cdots + a_{mn}x_n &= b_n \end{aligned}$$

(7.8)

with n unknowns $x_1, x_2, \cdots, x_n$ related through m equations. In the case $n = m$ there are as many equations as unknowns and the system can be solved, as long as all equations are linearly independent (the non-singular case). The system is singular, if two or more equations are linearly dependent (often denoted as row-degenerate).

The above system of equations is written in matrix form

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \,,$$

where $\mathbf{A}$ is the so-called coefficient matrix, while $\mathbf{b}$ and $\mathbf{x}$ are column vectors, the latter being the unknowns

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{bmatrix} \,, \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_n \end{bmatrix} \,, \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \cdots \\ b_n \end{bmatrix} \,.$$

When performing these operations, the word "row" refers not only to the row of the matrix, but also to the corresponding element in the vector $\mathbf{b}$ on the right-hand side of the system (7.7). It will be seen later that we are using an *augmented* matrix, containing both the matrix coefficients and an extra column consisting of the right-hand side, which is treated along with the matrix itself.

There exists a number of more or less efficient methods for solving linear systems. In this and following sections we describe some of the simplest methods. First we describe the so-called *Gauss elimination* method which, although not the most robust or fastest, is easy to understand and is useful in practice, for example for matrix inversion.

Triangular matrices

Many methods for solving linear systems of equations are based on the conversion of a given matrix to an upper or lower triangular matrix.

An upper triangular matrix has non-zero elements on and above the diagonal. All elements below the diagonal are equal to zero. If the equation system to be solved consists of such a matrix, it is easily solved:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n &= b_1 \\ a_{22}x_2 + a_{23}x_3 + \cdots + a_{2n}x_n &= b_2 \\ a_{33}x_3 + \cdots + a_{3n}x_n &= b_3 \\ \cdots &= \cdots \\ a_{nn}x_n &= b_n \ . \end{aligned} \tag{7.9}$$

As long as all $a_{ii} \neq 0$, x_n can be determined from the last equation above, then x_{n-1} by using that solution, and so on up through the system. This method, called *back substitution*, can be expressed recursively as

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=i+1}^{n} a_{ij}x_j \right) \ . \tag{7.10}$$

A lower triangular matrix has non-zero elements on and under the diagonal, all other elements being equal to zero. Such a system is also easy to solve, using *forward substitution*:

$$\begin{aligned} a_{11}x_1 &= b_1 \\ a_{21}x_1 + a_{22}x_2 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3 \\ \cdots & \\ a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \cdots + a_{nn}x_n &= b_n \ . \end{aligned} \tag{7.11}$$

Here we start with the first equation, directly giving us x_1, and so on, moving down the system. The general recursive description is here

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j \right) \ . \tag{7.12}$$

Gauss- and Gauss-Jordan elimination

The above equation system (7.7) can be solved for the case $m = n$ and the system is upper or lower triangular, or can be converted to such. This can be done by the so-called *Gauss elimination*. The method systematically eliminates elements, such that finally an upper triangular matrix is left.

There are some useful rules to know for what may be done to a matrix, that makes elimination possible. It is permitted to

- exchange rows
- multiply a row by a constant
- add rows, that is, replace one row by a (weighted) sum of that row and another.

The last operation can eliminate elements which we want to set to zero. For example, in the system (7.7), we would like to have all elements directly under the top left element set to zero. Generally the procedure is that, if we want to eliminate $a_{i,1}$, that we replace line i by the sum of line 1 multiplied by a_{i1} and line i multiplied by $-a_{11}$, etc. An example will illustrate this. Let us say that we have this system:

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & -2 & -1 \\ 3 & -1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 11 \\ 2 \\ 12 \end{bmatrix} . \tag{7.13}$$

It is of advantage if the system is diagonally dominant as far as possible, that is, as far as possible the element of largest magnitude in a given row is the diagonal element; this is not the case here. To achieve this we exchange rows 1 and 3. This is called **pivoting**. Before we do this, we must describe another process. When we exchange rows or add rows to each other, we must do the same for the right-hand side of the equation system, vector **b**. To make this easier, we extend the coefficient matrix, which thus becomes the *augmented matrix*, with the right-hand side added as an extra column. The matrix is then represented in its augmented (and pivoted) form as

$$\left[\begin{array}{ccc|c} 3 & -1 & 2 & 12 \\ 2 & -2 & -1 & 2 \\ 1 & 2 & 3 & 11 \end{array}\right] . \tag{7.14}$$

We can now begin the elimination. Let R_i denote row i. We begin by replacing R_2 with the sum $-2 \times R_1 + 3 \times R_2$, and R_3 with $-R_1 + 3 \times R_3$, which leads to

$$\left[\begin{array}{ccc|c} 3 & -1 & 2 & 12 \\ 0 & -4 & -7 & -18 \\ 0 & 7 & 7 & 21 \end{array}\right] . \tag{7.15}$$

We only need to eliminate one more element, which we do by replacing R_3 by $-7 \times R_2 + (-4) \times R_3$, and we end with

$$\left[\begin{array}{ccc|c} 3 & -1 & 2 & 12 \\ 0 & -4 & -7 & -18 \\ 0 & 0 & 21 & 42 \end{array}\right] . \tag{7.16}$$

We now have an upper triangular matrix, and can start the back substitution, from the bottom up. We directly obtain $x_3 = 2$, and go up the system and finally have the solution of system (7.13), $(x_1, x_2, x_3) = (3, 1, 2)$.

We can also, after computing (7.16), continue by eliminating elements above the diagonal, instead of doing the back substitution. This takes place in the same manner as the above, and we end with

$$\left[\begin{array}{rrr|r} 1764 & 0 & 0 & 5292 \\ 0 & -84 & 0 & -84 \\ 0 & 0 & 21 & 42 \end{array}\right] . \tag{7.17}$$

This is the Gauss-Jordan method.

Note that the coefficients increase in magnitude, but we can now simply read off the solution. There is an alternative method which prevents the repeated growth of the coefficients' magnitude. Rather than multiplying rows by coefficients when adding rows, we can divide by them. The first elimination step then replaces R_2 with $-\frac{1}{3}R_1 + \frac{1}{2}R_2$, which also eliminates an element, and so forth. This produces

$$\left[\begin{array}{rrr|r} 3 & -1 & 2 & 12 \\ 0 & \frac{2}{3} & \frac{7}{6} & 3 \\ 0 & 0 & \frac{7}{12} & \frac{7}{6} \end{array}\right] \tag{7.18}$$

which not unexpectedly gives the same result. If we go on and use Gauss-Jordan, the diagonalised system becomes

$$\left[\begin{array}{rrr|r} \frac{7}{5} & 0 & 0 & \frac{21}{5} \\ 0 & -\frac{4}{7} & 0 & -\frac{4}{7} \\ 0 & 0 & \frac{7}{12} & \frac{7}{6} \end{array}\right] . \tag{7.19}$$

This can be scaled by dividing by the diagonal elements, which is in fact a way to represent the solution. It leads to

$$\left[\begin{array}{rrr|r} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 2 \end{array}\right] \tag{7.20}$$

in which the solution is directly in the right-hand side. This is useful for matrix inversion, as will be seen.

LU decomposition (LUD). Crout's method

An efficient and flexible method of solution of linear systems is **lower-upper decomposition** or **LUD**. It is assumed that there is a pair of lower and upper triangular matrices **L** and **U**, respectively, whose product is the matrix **A** in the system

$$\mathbf{Ax} = \mathbf{b} \tag{7.21}$$

so that

$$\mathbf{LU} = \mathbf{A} \; . \tag{7.22}$$

Then, after multiplication of (7.21) with $\mathbf{L}^{-1}$,

$$\mathbf{Ux} = \mathbf{L}^{-1}\mathbf{b} = \mathbf{y} \; , \tag{7.23}$$

$$\mathbf{Ly} = \mathbf{b} \; . \tag{7.24}$$

Using the last equation we can solve for $\mathbf{y}$ by forward substitution, an easy operation, since matrix $\mathbf{L}$ is lower triangular. The solution $\mathbf{y}$ is then used in (7.23) to calculate $\mathbf{x}$ by back substitution, again an easy operation because $\mathbf{U}$ is a lower triangular matrix.

The main problem is the decomposition, that is, finding $\mathbf{L}$ and $\mathbf{U}$. They are found by making use of (7.22). For this to work, we must have unity values on the diagonal either of $\mathbf{L}$ or $\mathbf{U}$. Assuming this, we can successively and recursively, compute all elements of the two matrices. We use again the above example system (7.13), after exchanging rows. When LU-decomposing, the right-hand side $\mathbf{b}$ is not involved, until $\mathbf{L}$ and $\mathbf{U}$ are finally used. The matrix to be LU decomposed is thus the product

$$\begin{bmatrix} 1 & 0 & 0 \\ \ell_{21} & 1 & 0 \\ \ell_{31} & \ell_{32} & 1 \end{bmatrix} \times \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} = \begin{bmatrix} 3 & -1 & 2 \\ 2 & -2 & -1 \\ 1 & 2 & 3 \end{bmatrix} \; . \tag{7.25}$$

The process is readily understood. We can immediately insert the upper row into $\mathbf{U}$, which is $(3, -1, 2)$ given that $1 \times u_{11} = 3$, etc. Now that we know these three u values, we can determine the first column of $\mathbf{L}$: by multiplying row i of $\mathbf{L}$ by the first column of $\mathbf{U}$, for $i = 2, 3$. Thereafter we use these known values. For example, we obtain u_{22} by using $\ell_{21}u_{12} + u_{22} = -2$, and so on. The final result is

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ \frac{2}{3} & 1 & 0 \\ \frac{1}{3} & -\frac{7}{4} & 1 \end{bmatrix} \tag{7.26}$$

and

$$\mathbf{U} = \begin{bmatrix} 3 & -1 & 2 \\ 0 & -\frac{4}{3} & -\frac{7}{3} \\ 0 & 0 & -\frac{7}{4} \end{bmatrix} \; . \tag{7.27}$$

In practice, since we know that the diagonal of $\mathbf{L}$ is unity, we can combine the two triangular matrices into a single one:

$$\begin{bmatrix} 3 & -1 & 2 \\ \frac{2}{3} & -\frac{4}{3} & -\frac{7}{3} \\ \frac{1}{3} & -\frac{7}{4} & -\frac{7}{4} \end{bmatrix} \tag{7.28}$$

with the diagonal understood and therefore left out. This is the way computer LUD routines store both matrices in one.

The above example illustrates the procedure. The formal description is as follows. For all $j = 1, \ldots, n$, the two operations

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} \ell_{ik} u_{kj} \qquad i = 1, 2, \ldots, j \tag{7.29}$$

and

$$l_{ij} = \frac{1}{u_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} \ell_{ik} u_{kj} \right) \qquad i = j+1, j+2, \ldots, n \tag{7.30}$$

are carried out, alternating from one to the other, where a_{ij} is the corresponding element is matrix $\mathbf{A}$.

Now for the reasons why LUD is so interesting. Assume, in our example, that the right-hand side $\mathbf{b}$ is (appropriate to the pivoted system (7.14)), the vector $[12, 2, 11]^T$. Note equation (7.24). Since we know $\mathbf{L}$, we can compute the vector $\mathbf{y}$. It becomes $[12, -6, -3.5]^T$. Once we have this, we can go on to the desired solution $\mathbf{x}$, arising from equation (7.23). Both operations are very easy, as they involve only triangular matrices. If we now want to solve for another $\mathbf{b}$, we can quickly compute a new $\mathbf{x}$, without carrying out a new LUD. In chemical-physical computations, one often has to do with a constant coefficient matrix and a number of different right-hand sides, and LUD makes this very efficient.

Matrix inversion

In Sect. 7.6, a systematic method for matrix inversion was described, and it was noted that the method is very inefficient for matrices larger than 3×3. For larger matrices, we need something more efficient. LUD is one such method of computing the inverse $\mathbf{A}^{-1}$ of matrix $\mathbf{A}$. Inversion can be performed by solving the n linear equations $\mathbf{A}x_i = e_i$ in

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{I} \qquad \rightarrow \qquad \mathbf{A} \begin{bmatrix} x_{11} & x_{12} & \ldots & x_{1n} \\ x_{21} & x_{22} & \ldots & x_{2n} \\ \cdot & \cdot & \ldots & \cdot \\ x_{n1} & x_{n2} & \ldots & x_{nn} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 \\ \cdot & \cdot & \ldots & \cdot \\ 0 & 0 & \cdot & 1 \end{bmatrix} \tag{7.31}$$

where $\mathbf{A}$ is solved for a number of right-hand sides, so that the augmented matrix becomes

$$\left[\begin{array}{cccc|cccc} a_{11} & a_{12} & \ldots & a_{1n} & 1 & 0 & \ldots & 0 \\ a_{21} & \ldots & & & 0 & 1 & 0 & \ldots \\ \vdots & & & & \vdots & & & \\ a_{n1} & \ldots & & a_{nn} & 0 & \ldots & & 1 \end{array}\right] . \tag{7.32}$$

After $\mathbf{A}$ is reduced to a diagonal (that is, it is solved for all right-hand sides), where there are only unity values on the diagonal, then $\mathbf{A}^{-1}$ is sitting in the augmented part. This can be done very efficiently with LUD. We first LU-decompose matrix $\mathbf{A}$, and then solve for n right-hand sides, each of whose elements consist of a vector of all zeros but one, in the i^{th} position, which results in the i^{th} column of the inverse, i running from 1 to n.

Tridiagonal systems

Not infrequently, we come across systems of equations whose coefficients lie on the three central diagonals, that is, in every row, only the three elements around the diagonal are non-zero (except in the first and last rows, where there are only two non-zero elements). It would be a waste of computing time to solve such systems with the above methods, because they would address many zeroes. There are more efficient methods for these systems, that make use of the special properties of tridiagonal systems. Consider the system

$$\begin{bmatrix} a_{11} & a_{12} & 0 & 0 & \dots & \\ a_{21} & a_{22} & a_{23} & 0 & \dots & \\ 0 & a_{32} & a_{33} & a_{34} & 0 & \dots \\ \vdots & \ddots & \ddots & \ddots & & \\ 0 & \dots & & a_{n-1,n-2} & a_{n-1,n-1} & a_{n-1,n} \\ 0 & & & \dots & a_{n,n-1} & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{n-1} \\ b_n \end{bmatrix} . \quad (7.33)$$

We often know either a_{11} or a_{nn}. If the latter is the case, we can move backwards from the last line, reducing the system to just two diagonals. From the bottom upwards, we substitute for x_{i+1} in every line i, producing a new set, in which every line now has only two elements, until we reach the top, where we are now left with only one, for x_1, which is thereby solved. After this, we can do a forward substitution downward and solve for all other unknowns. Read more about this in Press & al. [4].

Iterative solution (Jacobi, Gauss-Seidel)

The iterative method, already described in Sect. 2.7, page 12, for single equations, can also be used for systems of equations, either linear or nonlinear. The method that results from the "g-method" in Sect. 2.7 by applying it to systems, is called the Jacobi method (not to be confused with the Jacobi method for calculating eigenvalues). The method is best illustrated by an example. Assume that we have a set of three equations to solve:

$$\begin{aligned} 8x_1 + x_2 - x_3 &= 8 \\ 2x_1 + x_2 + 9x_3 &= 12 \\ x_1 - 7x_2 + 2x_3 &= -4 \end{aligned} \quad (7.34)$$

(the solution is that all three unknowns are equal to unity, but we pretend not to know this). We start with a guess at the solution vector. The procedure is now to exchange equations (rows) such that the diagonal is dominant as far as possible. We can do this here by exchanging equation 3 with equation 2. The system is then expressed generally as

$$\mathbf{Ax} = \mathbf{b} \tag{7.35}$$

and the method is the successive application of the equation

$$x_i^{(n+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1, j\neq i}^{n} a_{ij} x_j^{(n)} \right) \tag{7.36}$$

for successive i. In the above example, after the exchange, we have

$$\begin{aligned} x_1 &= \frac{1}{8}\big(8 - x_2 + x_3\big) \\ x_2 &= -\frac{1}{7}\big(-4 - x_1 - 2x_3\big) \\ x_3 &= \frac{1}{9}\big(12 - 2x_1 - x_2\big) \end{aligned} \tag{7.37}$$

and we begin with a guessed set, for example, $\left[x_1^{(0)}, x_2^{(0)}, x_3^{(0)}\right]^T = [0,0,0]^T$. This results in the first calculation

$$x_1^{(1)} = \tfrac{8}{8} - \big(\tfrac{1}{8}\cdot 0 - \tfrac{1}{8}\cdot 0\big) = 1.000$$

$$x_2^{(1)} = \tfrac{-4}{-7} - \big(\tfrac{1}{-7}\cdot 0 + \tfrac{2}{-7}\cdot 0\big) = 0.571 \tag{7.38}$$

$$x_3^{(1)} = \tfrac{12}{9} - \big(\tfrac{2}{9}\cdot 0 + \tfrac{1}{9}\cdot 0\big) = 1.333\ . \tag{7.39}$$

The process is repeated. Here is a table of the values computed:

trin:	0	1	2	3	...	7
x_1	0	1.000	1.095	0.995		1.001
x_2	0	0.571	1.095	1.026	...	1.001
x_3	0	1.333	1.048	0.969		1.001

(some steps have been left out). This is the Jacobi method. It is seen that the values converge to the solution, but somewhat slowly. It is possible to speed up convergence a little by making use, at every step, of the new values already computed in earlier lines. In the above example, the start set was used in all three lines; that is, in line 2 the value $x_1 = 0$ was used, even though it has just been recalculated to be unity. If we use the new values immediately, the

method is called Gauss-Seidel and for this method, the first three calculations are

$$\begin{aligned} x_1^{(1)} &= \tfrac{8}{8} - \left(\tfrac{1}{8}\cdot 0 - \tfrac{1}{8}\cdot 0\right) = 1.000 \\ x_2^{(1)} &= \tfrac{-4}{-7} - \left(\tfrac{1}{-7}\cdot 1 + \tfrac{2}{-7}\cdot 0\right) = 0.714 \qquad (7.40) \\ x_3^{(1)} &= \tfrac{12}{9} - \left(\tfrac{2}{9}\cdot 1 + \tfrac{1}{9}\cdot 0.714\right) = 1.032\ . \qquad (7.41) \end{aligned}$$

The table now becomes

trin:	0	1	2	3	4
x_1	0	1.000	1.041	0.997	1.001
x_2	0	0.741	1.014	0.996	1.000
x_3	0	1.032	0.990	1.002	1.000

and clearly convergence is faster.

This method does not, however, always work. For example, it cannot be used for the system (7.13); the calculated solution vector diverges more and more with increasing numbers of steps.

7.8 Eigenvalues and eigenvectors

The eigenvalues and -vectors of a matrix are defined by

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x}\ , \qquad (7.42)$$

where λ is an eigenvalue and $\mathbf{x}$ the corresponding eigenvector. λ is an eigenvalue if and only if

$$|\mathbf{A} - \lambda\mathbf{I}| = 0\ , \qquad (7.43)$$

where $|\ldots|$ denotes the determinant. Thus, the calculation of the eigenvalues of $\mathbf{A}$ means finding the roots of the characteristic polynomial equation

$$\lambda^n + a_{n-1}\lambda^{n-1} + \cdots + a_1\lambda + a_0 = 0\ , \qquad (7.44)$$

arising from the expansion of (7.44). One might "simply" find the roots of (7.44). The problem is, however, that this is a very computer-intensive procedure. In fact, the opposite is done: since we have much more efficient methods for finding eigenvalues, these are used to find polynomial roots, see Chap. 2, Sect. 2.8.

There are no direct methods for finding eigenvalues or -vectors; all methods are iterative to some extent. We name only a few here, without a lot of detail. Press & al. [4] thus describe the Jacobi transformation, which can be used for real symmetric matrices (which have real eigenvalues), and **QR**-decomposition for all other matrices. Press & al. [4] also recommend the use of professional subroutines, and we follow their advice here, except for the iterative methods to be described in the next Section.

"Power" method for the largest eigenvalue

One begins with a guess at an eigenvector; this could for example be the vector $[1, 1, \ldots, 1]^T$. We then multiply the matrix by this vector, and an estimate of the largest eigenvalue is that element of the product vector having the largest magnitude. The vector is divided by this value, and the result becomes the next approximation to the final eigenvector. The process is repeated until convergence is obtained. Here is an example. We want the largest eigenvalue of the matrix $\begin{bmatrix} 4 & 1 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & 1 \end{bmatrix}$. Our first guess is

$$\begin{bmatrix} 4 & 1 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 5 \\ 3 \\ 1 \end{bmatrix} = 5 \begin{bmatrix} 1 \\ 0.6 \\ 0.2 \end{bmatrix} \tag{7.45}$$

and the first estimate of the eigenvalue becomes 5. We repeat:

$$\begin{bmatrix} 4 & 1 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0.6 \\ 0.2 \end{bmatrix} = \begin{bmatrix} 4.6 \\ 1.0 \\ 0.2 \end{bmatrix} = 4.6 \times \begin{bmatrix} 1 \\ 0.217 \\ 0.0435 \end{bmatrix} . \tag{7.46}$$

After a number of iterations we finally obtain an eigenvalue equal to 4 and an eigenvector equal to $[1, 0, 0]^T$.

If we instead want the smallest eigenvalue, we start by inverting the matrix, finding the largest eigenvalue of the inverse and then invert that value.

Iterative method of finding the smallest eigenvalue

Matrix diagonalisation normally entails a number of floating point operations (flops) of the order of n^3, so that the direct determination of all eigenvalues and eigenvectors will often be very time- and memory-consuming for matrices of size $n > 1000$. In some special cases, where the matrix $\mathbf{A}$ is "sparse" (that is, contains many zeroes), and if we are only interested in one or a few smallest eigenvalues, iterative methods can be of advantage.

Assume that we have a sparse symmetric matrix $\mathbf{A}$ of dimension n for which we can easily carry out the linear transformation as seen in (7.4), page 61,

$$\mathbf{y} = \overbrace{\mathbf{Ax}} . \tag{7.47}$$

The overbrace is used here and in what follows indicates a linear transformation, for clarity. Determination of $\mathbf{A}$'s i'th eigenvalue via the equation

$$\mathbf{Ax}_i = \lambda_i \mathbf{x}_i \tag{7.48}$$

on the basis of a start set $\mathbf{x}^{(0)}$ for the eigenvector can be done as follows: (7.48), with $\mathbf{x}^{(0)}$ substituted for $\mathbf{x}_i$, is multiplied by $\left(\mathbf{x}^{(0)}\right)^T$

$$\left(\mathbf{x}^{(0)}\right)^T \overbrace{\mathbf{Ax}^{(0)}} = \lambda^{(0)} \left(\mathbf{x}^{(0)}\right)^T \mathbf{x}^{(0)} \tag{7.49}$$

and, as $\left(\mathbf{x}^{(0)}\right)^T \mathbf{x}^{(0)}$ is a scalar,

$$\lambda^{(0)} = \frac{\left(\mathbf{x}^{(0)}\right)^T \overbrace{\mathbf{A}\mathbf{x}^{(0)}}}{\left(\mathbf{x}^{(0)}\right)^T \mathbf{x}^{(0)}} \,. \tag{7.50}$$

Next we assume that $\mathbf{A}$ can be rewritten as

$$\mathbf{A} = \mathbf{A}^{(0)} + \mathbf{A}^{(1)} \,, \tag{7.51}$$

where $\mathbf{A}^{(0)}$ is the dominant and readily inverted, (normally diagonal) component and $\mathbf{A}^{(1)}$ a correction set. Correspondingly we apply

$$\lambda = \lambda^{(0)} + \lambda^{(1)} \,; \qquad \mathbf{x} = \mathbf{x}^{(0)} + \mathbf{x}^{(1)} \,, \tag{7.52}$$

where the left-hand side represents exact values. By inserting these rewritten equations in (7.48) we obtain

$$\overbrace{\left[\mathbf{A}^{(0)} + \mathbf{A}^{(1)}\right]\left[\mathbf{x}^{(0)} + \mathbf{x}^{(1)}\right]} = \left[\lambda^{(0)} + \lambda^{(1)}\right]\left[\mathbf{x}^{(0)} + \mathbf{x}^{(1)}\right] \,, \tag{7.53}$$

which, neglecting second-order terms in the correction set, is easily written as

$$\mathbf{x}^{(1)} = -\left[\mathbf{A}^{(0)} - \lambda^{(0)}\mathbf{I}\right]^{-1} \overbrace{\left[\mathbf{A} - \lambda^{(0)}\mathbf{I} - \lambda^{(1)}\mathbf{I}\right]\mathbf{x}^{(0)}} \,. \tag{7.54}$$

It is noted that the contents of $[\ldots]$ in the above equation corresponds to the matrix $\mathbf{A}$ with the λ's subtracted from the diagonal, after multiplication by the unit matrix $\mathbf{I}$. If we demand orthogonality

$$\left(\mathbf{x}^{(0)}\right)^T \mathbf{x}^{(1)} = 0 \tag{7.55}$$

then $\lambda^{(1)}$ can be obtained from (7.53) by multiplication by $\left(\mathbf{x}^{(0)}\right)^T$ from the left on both sides

$$\begin{aligned} 0 = &-\left(\mathbf{x}^{(0)}\right)^T \left[\mathbf{A}^{(0)} - \lambda^{(0)}\mathbf{I}\right]^{-1} \overbrace{\left[\mathbf{A} - \lambda^{(0)}\mathbf{I}\right]\mathbf{x}^{(0)}} \\ &+ \left(\mathbf{x}^{(0)}\right)^T \left[\mathbf{A}^{(0)} - \lambda^{(0)}\mathbf{I}\right]^{-1} \lambda^{(1)}\mathbf{x}^{(0)} \end{aligned}$$

which gives

$$\begin{aligned} \lambda^{(1)} = &\left\{\left(\mathbf{x}^{(0)}\right)^T \left[\mathbf{A}^{(0)} - \lambda^{(0)}\mathbf{I}\right]^{-1} \mathbf{x}^{(0)}\right\}^{-1} \\ &\cdot \left(\mathbf{x}^{(0)}\right)^T \left[\mathbf{A}^{(0)} - \lambda^{(0)}\mathbf{I}\right]^{-1} \overbrace{\left[\mathbf{A} - \lambda^{(0)}\mathbf{I}\right]\mathbf{x}^{(0)}} \,. \end{aligned} \tag{7.56}$$

As is seen from the example below, this expression can be simplified, if matrix $\mathbf{A}$ is not only diagonally dominant but where all diagonal elements are equal.

From the above it is seen that the correction terms for both eigenvalue $\lambda^{(1)}$ and eigenvector $\mathbf{x}^{(1)}$ are easily determined from equations (7.56) and (7.54) by

application of the linear transformation (indicated by the overbraces) and the inner scalar vector product (7.47)). The method is iterative, and each new correction term $\mathbf{x}^{(1)}$ to the original guess at $\mathbf{x}^{(0)}$ is an improvement,

$$\mathbf{x}_{improved} = \mathbf{x}^{(0)} + \mathbf{x}^{(1)} \ , \tag{7.57}$$

and $\mathbf{x}^{(0)}$ is used for the determination of a new $\lambda^{(0)}$ by (7.50) and the whole process is repeated. When the norm of the residual,

$$R = \| \left[\mathbf{A} - \lambda^{(0)} \mathbf{I} \right] \mathbf{x}^{(0)} \| < \delta \tag{7.58}$$

falls below a given (small) value δ, $\lambda^{(0)}$ and $\mathbf{x}^{(0)}$ represent the final eigenvalue and -vector pair, that is, we have convergence. The value of δ must be chosen appropriately.

Remark on the linear transformation

We are often dealing with very large matrices, and it might seem that we must carry out lengthy matrix multiplications for the linear transformation. This is however not the case. The matrices are sparse with a known pattern of the non-zero elements, and we carry out the multiplication using our knowledge of this pattern. An example is the Hückel matrix,

$$\mathbf{A} = \begin{bmatrix} \alpha & \beta & & \dots & & & \beta \\ \beta & \alpha & \beta & & & & \\ \ddots & \ddots & \ddots & & & & \\ & & & & & & \\ & & & & \ddots & \ddots & \ddots \\ & & & & \beta & \alpha & \beta \\ \beta & & & \dots & & \beta & \alpha \end{bmatrix} \ . \tag{7.59}$$

When multiplying this matrix with vector $\mathbf{x}$ in order to get the product vector $\mathbf{y}$, the procedure is the following:

$$y_1 = \alpha x_1 + \beta x_2 + \beta x_n \ , \tag{7.60}$$

whereafter, for all $i = 2 \dots n-1$,

$$y_i = \beta x_{i-1} + \alpha x_i + \beta x_{i+1} \ ; \tag{7.61}$$

and finally

$$y_n = \beta x_1 + \beta x_{n-1} + \alpha x_n \ . \tag{7.62}$$

For an $n \times n$ matrix $\mathbf{A}$, this means $3n$ multiplications and n additions, which for large n, is much less than the about n^2 operations needed for the multiplication of a general, non-sparse matrix.

Example: the Hückel matrix

If we wish to find the smallest eigenvalue of a Hückel matrix as defined (7.59), we can simplify (7.56) to

$$\lambda^{(1)} = \frac{\left(\mathbf{x}^{(0)}\right)^T \overbrace{\left[\mathbf{A} - \lambda^{(0)}\mathbf{I}\right]\mathbf{x}^{(0)}}}{\left(\mathbf{x}^{(0)}\right)^T \mathbf{x}^{(0)}} \tag{7.63}$$

because all diagonal elements are the same. The procedure is then:

(a) set starting values for $\mathbf{x}^{(0)}$; these could for example be the vector $[0, 0, \frac{1}{\sqrt{n}}, \cdots, \frac{1}{\sqrt{n}}]^T$;

(b) carry out the linear transformation (7.47) and compute $\lambda^{(0)}$ from (7.50);

(c) check for convergence by using (7.58); if $R < \delta$, exit from the iteration;

(d) determine $\lambda^{(1)}$, eqn. (7.63);

(e) determine $\mathbf{x}^{(1)}$, eqn. (7.54);

(f) find $x_{improved}$ and use it for the new $\mathbf{x}^{(0)}$;

(g) go to (a).

A suitable value for δ must of course be set before starting.

Chapter 8

Function Evaluation

Computer languages such as Fortran, Pascal, C++, etc, have inbuilt functions like `SQRT`, `SIN`, `EXP`, `LOG`, etc, which must be evaluated on the spot when called or invoked. If one wants to write a new function that is not implemented already, one begins to think about how to program the function. Often one simply downloads ready professional program packages but it is in any case of interest to know how it is done best. An example is the error function `erf` and its complement, `erfc` used in, for example, the course `DatK`, where it is evaluated by integration.

The following methods, among others, can be used:

- Numerical integration
- Newton's method
- approximating polynomials
- rational functions
- asymptotic series.

In many textbooks we find certain programming tricks designed to improve the efficiency of function evaluation. Many of these have their origin in the past, when people did the calculations by hand, so that some of them are little more than curiosities today, when computers are fast, and the number of iterations or of the terms in a series are less critical. However, some of these are mentioned here, for interest, since they can provide an idea of programming strategy in numerical computing.

For a great number of approximations to many functions, see the Mathematical Handbook by Abramowitz & Stegun [3], a classical text.

8.1 General

Precision

Some aspects of precision are:

- Calculated values should have the same precision as the basic computer operations, whatever precision has been set. This is not too demanding for single precision real variables, but may be for double precision (that is, normally about 16 decimals or more).
- Some functions have known ranges and the calculations must result in values within these. For example, sin x ought not be evaluated outside its range [-1,+1].

There are two ways to define errors: absolute or relative. If we denote the computed value as v and the true value as $\hat{v}$, then the absolute error is defined as

$$e_a = v - \hat{v} \tag{8.1}$$

and the relative error by

$$e_r = \frac{v - \hat{v}}{\hat{v}} \, . \tag{8.2}$$

The former can be more relevant for a function with a modest range, such as the sine function, while a function such as the exponential or a square root might be better specified to some relative precision.

Where do the formulae come from?

Some formulae simply come from known expansions (e.g. the so-called McLaurin expansions). An example is the exponential function, which can be expressed as

$$\mathrm{e}^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots \tag{8.3}$$

which for certain arguments x (less than 1 in magnitude) can be used to calculate the function value. Many functions, such as the logarithm, the trigonometric functions and others can be represented in the form of such series. Others are approximated by polynomials; one example is the error function erf and the complementary error function erfc, both of which can be expressed as McLaurin series, but these are not very suitable for evaluation. Polynomials are often determined empirically as the best possible approximation, over a restricted argument range.

Relations between functions

Some functions can be expressed as functions of other functions, for which we might already have a method of evaluation. In principle, for example, we need not approximate the cosine function if we have a method of calculating the sine function, because they have a known relation to each other. It may however

happen that we might prefer a separate approximation to such a function in any case. An example is the pair erf and erfc, related by

$$\mathrm{erfc}(x) = 1 - \mathrm{erf}(x) \tag{8.4}$$

so one also implies the other. But for large x there is a much more accurate formula for erfc than that for erf subtracted from 1 and so it is better to use (8.4). Press & al. [4] recommend using the incomplete gamma function to compute the error function, while there is a number of polynomial approximations for it in Abramowitz & Stegun [3].

Segmentation

In some cases it is best to use different formulae for different ranges of the arguments of a function. Thus for erfc, calculation becomes less and less accurate for large arguments if (8.4) is used, because $\mathrm{erf}(x)$ approaches unity for large x, and we are subtracting one number from another close to it, a classic situation for inaccuracy. As mentioned above, there are separate approximate formulae for erfc for this situation. This is called "segmentation". It is also seen for the hyperbolic functions. For example, we can use approximations for the exponential function for $\sinh(x)$, but if $|x| \ll 1$, then certain polynomial expressions are better, as otherwise we would be subtracting two values close to unity, with resulting inaccuracy.

Range reduction

If we consider the $\log_{10}$ function as an example, it is clear that it is not necessary to find an approximation for any argument, no matter how large or small. If we write a given argument x in the form

$$x = y \times 10^n \tag{8.5}$$

where n is a whole number and $1 < y \leq 10$, then the resulting logarithm is

$$\log_{10} x = n + \log_{10} y \; . \tag{8.6}$$

This is range reduction, reducing the problem to a limited range of the argument. It can be done for a number of functions whose arguments potentially have an infinite range. For the natural log ln, we can use

$$x = y \times 2^n \; , \tag{8.7}$$

which reduces the problem to the range $0.75 \leq y < 1.5$ and the final formula is then

$$\ln x = n \ln 2 + \ln y \; . \tag{8.8}$$

The value of $\ln 2$ can be computed once and for all and stored.

Range reduction is used in many cases, including the square root. Obviously here we can use the reduction

$$x = y \times n^2 \tag{8.9}$$

which reduces y to $0.25 \leq y < 1$. The formula is

$$\sqrt{x} = n\sqrt{y}\ . \tag{8.10}$$

For the trigonometric functions, with period 2π, one could, for large arguments, subtract a sufficient multiple of 2π, until the argument lies in the range $0 < x < 2\pi$, but this too is unnecessarily large. Here we can make use of the even smaller range $0 < x < \pi/2$, together with the known simple relations between the four quadrants.

Chebyshev polynomials

There is a family of polynomials that is very useful in several connections, among them function approximation. One group of these is denoted by T_n, with $n = 0, 1, \ldots$. The first few terms are:

$$\begin{aligned}
T_0 &= 1 \\
T_1 &= x \\
T_2 &= 2x^2 - 1 \\
T_3 &= 4x^3 - 3x \\
T_4 &= 8x^4 - 8x^2 + 1 \\
T_5 &= 16x^5 - 20x^3 + 5x \\
T_6 &= 32x^6 - 48x^4 + 18x^2 - 1 \\
T_7 &= 64x^7 - 112x^5 + 56x^3 - 7x \\
T_8 &= 128x^8 - 256x^6 + 160x^4 - 32x^2 + 1 \\
T_9 &= 256x^9 - 576x^7 + 432x^5 - 120x^3 + 9x \\
T_{10} &= 512x^{10} - 1280x^8 + 1120x^6 - 400x^4 + 50x^2 - 1\ .
\end{aligned} \tag{8.11}$$

There is a general expression,

$$T_n(x) = \cos\left(n \arccos x\right) \tag{8.12}$$

which generates them all. However, the above expressions are more convenient to work with.

An important property of all these polynomials is that, within the range $-1 \leq x \leq 1$, the function values remain in the same range $[-1, +1]$. Fig. 8.1 shows some of the functions. One can also invert the functions and this can

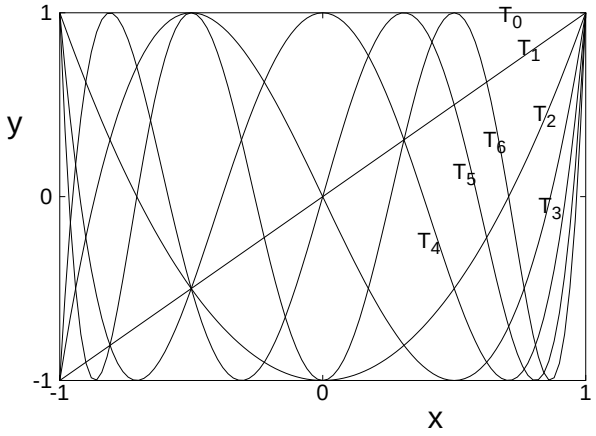

Figure 8.1: The first 7 Chebyshev functions in $[-1, +1]$

be very useful, as we will see below:

$$\begin{aligned}
1 &= T_0 \\
x &= T_1 \\
x^2 &= \tfrac{1}{2}\left(T_0 + T_2\right) \\
x^3 &= \tfrac{1}{4}\left(3T_1 + T_3\right) \\
x^4 &= \tfrac{1}{8}\left(3T_0 + 4T_2 + T_4\right) \\
x^5 &= \tfrac{1}{16}\left(10T_1 + 5T_3 + T_5\right) \\
x^6 &= \tfrac{1}{32}\left(10T_0 + 15T_2 + 6T_4 + T_6\right) \\
x^7 &= \tfrac{1}{64}\left(35T_1 + 21T_5 + 7T_5 + T_7\right) \\
x^8 &= \tfrac{1}{128}\left(35T_0 + 56T_2 + 28T_4 + 8T_6 + T_8\right) \\
x^9 &= \tfrac{1}{256}\left(126T_1 + 84T_3 + 35T_5 + 9T_7 + T_9\right) \\
x^{10} &= \tfrac{1}{512}\left(126T_0 + 210T_2 + 120T_4 + 45T_6 + 10T_8 + T_{10}\right) \ .
\end{aligned} \tag{8.13}$$

Both these tables can be used to produce efficient polynomial series, to be described in Sect. 8.4.

8.2 Numerical integration

Although numerical integration is not prominent in text books on the subject, it can be used to advantage in cases where we require high efficiency and easy programming. For example, if we are experienced in integration and need the error function, it can be evaluated by this means more easily than the various approximation formulae to be seen in Abramowitz & Stegun, [3], based on polynomials. But in general, integration is not the preferred method of evaluating functions. See Chap. 3 for the details of numerical integration.

8.3 Newton's method

Newton's method can be used in cases where the function to be evaluated can easily be inverted. One such function is the square root. If we wish to know $\sqrt{a}$, we can evaluate it as the root of the equation

$$f(x) = x^2 - a \tag{8.14}$$

which is easily found using Newton's method. One would probably use range reduction before beginning the process. Here we must find a starting guess at the solution (root). This can be the argument a itself, especially if range reduction has been used. The process is then the following. Find n in equation (8.9), replace (8.14) with

$$f(y) = y^2 - a/n^2 \tag{8.15}$$

and solve for $f(y) = 0$. Newton's general formula (2.4) on page 9 becomes here

$$y_{n+1} = \frac{1}{2}\left(y_n + \frac{a}{n^2 y_n}\right) . \tag{8.16}$$

The method can also be used to find a cubic root, instead of using logarithms (which is what computers generally would do for such roots). If we want the cubic root, we write

$$f(x) = x^3 - a \tag{8.17}$$

and solve that. Fike [7] recommends dividing by x, which gives

$$f(x) = x^2 - a/x . \tag{8.18}$$

Newton's formula is then

$$x_{n+1} = x_n - \frac{x_n^2 - a/x_n}{2x_n + a/x_n^2} . \tag{8.19}$$

This form is believed to lead to faster convergence. Naturally here too range reduction is of advantage.

8.4 Polynomials

Polynomials are much used for function approximation. We all know the polynomial expansion for $\exp(x)$, the McLaurin series

$$\mathrm{e}^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} \cdots \tag{8.20}$$

and a large number of such series is found in Abramowitz & Stegun [3]. Often there are variants, each each being most suitable for certain argument ranges. For example, there are several polynomial series for $\ln(x)$, $\ln(1+x)$ and $\ln(\frac{x+1}{x-1})$, as well as others, see Abramowitz & Stegun [3].

Efficient evaluation of a polynomial

A polynomial is expressed in the form

$$p(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_n x^n \tag{8.21}$$

and this leads one initially to a code such as

```
p = a(0)
xpow = x
do i = 1, n
  p = p + a(i)*xpow
  xpow = xpow * x
enddo
```

(assuming that `a(0:n)` is declared as an array and that its values are known). This algorithm uses $2n$ multiplications and n additions, that is all in all $3n$ flops. We avoided using the more costly powers `x**i`, but still we can improve the algorithm, based on rewriting of (8.21) in the form

$$p(x) = (\cdots(a_n x + a_{n-1})x + a_{n-2})x + \cdots + a_1)x + a_0 \ , \tag{8.22}$$

for example for a fourth-degree polynomial

$$p(x) = (((a_4 x + a_3)x + a_2)x + a_1)x + a_0 \ . \tag{8.23}$$

This can be programmed backwards like this:

```
p = 0
do i = n, 1, -1
  p = (p+a(i)) * x
enddo
```

and this uses only $2n$ flops. If n is large, we may save much computing time by using this algorithm. This is probably one of the above-named curiosities, since the savings are modest by today's standards, and of little importance. Against the time saving advantage we have a code that is harder to follow than the first and this might outweigh the savings in about 30% computing time.

Application of Chebyshev polynomials

There are two applications of Chebyshev polynomials (see Sect. 8.1 for their definitions) in the context of polynomial approximations. One of these enables us to shorten a given polynomial series while preserving its accuracy; and the other produces a transformed series which, for the same number of terms in a series, increases the accuracy of evaluation.

The first of these is called "economised power series". The method is described here by means of the example (8.20) for the exponential function. We can write is as the seven-term approximation

$$\mathrm{e}^x \approx 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + \frac{x^6}{720} \; . \tag{8.24}$$

Note the table of Chebyshev polynomials in (8.11). Now let us subtract from the sum in (8.24) a quantity equal to $\frac{1}{720}\frac{T_6}{32}$. We know that, if x lies within $[-1, +1]$, then T_6, like all T_n, also lies within the same range, and therefore subtraction of this term means a change of no more than $\frac{1}{720\times 32}$ or 0.000043. So, if we are satisfied by a four-figure accuracy, the subtraction makes no difference. But now note what the series becomes after the subtraction:

$$\begin{aligned} \mathrm{e}^x \approx 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + \frac{x^6}{720} \\ - \frac{1}{720 \times 32}\left(32x^6 - 48x^4 + 18x^2 - 1\right) \; , \end{aligned} \tag{8.25}$$

which eliminates the term in x^6 and finally becomes

$$\mathrm{e}^x \approx 1.000043 + x + 0.499219x^2 + \frac{1}{6}x^3 + 0.043750x^4 + \frac{1}{120}x^5 \; . \tag{8.26}$$

We have shortened the series by one term, but preserved the accuracy. We can continue the shortening process. This is a rather rough example, and one would normally attack polynomials with more terms.

This, too, can today be seen as a curiosity, as there is not much cpu time saved using it.

The other application of Chebyshev polynomials involves replacing one polynomial with another with the same number of terms but a somewhat higher accuracy. We again use the exponential function as an example, this time using (8.20) only up to the term in x^4. All powers are replaced by the inverse Chebyshev expressions seen in Table (8.13). The approximation (8.20) then becomes

$$\begin{aligned} \mathrm{e}^x \approx T_0 + T_1 + \frac{1}{4}(T_0 + T_2) + \frac{1}{24}(3T_1 + T_3) + \frac{1}{192}(3T_0 + 4T_2 + \cdots) \\ + \frac{1}{1920}(10T_1 + 5T_3 + \cdots) + \frac{1}{23040}(10T_0 + 15T_2 + \cdots) + \cdots \end{aligned} \tag{8.27}$$

where we have left out all terms in T_n for $n > 3$. After a little effort this produces the new series

$$\mathrm{e}^x \approx 1.2661T_0 + 1.1303T_1 + 0.2715T_2 + 0.0443T_3 \tag{8.28}$$

and, if we wish to have it with powers of x instead, we can expand all T_n as in Table (8.11), which finally leads to the new formula

$$\mathrm{e}^x \approx 0.9946 + 0.9973x + 0.5430x^2 + 0.1773x^3 \; . \tag{8.29}$$

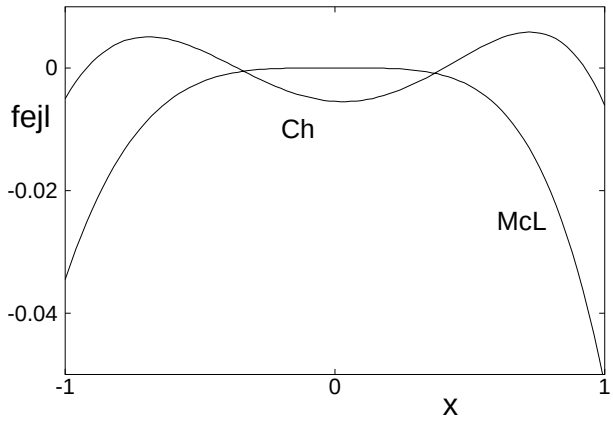

Figure 8.2: Errors of McLaurin (McL) and Chebyshev polynomial (Ch) for $\exp(x)$

Fig. 8.2 shows the difference between the two formulae, the McLaurin series (8.20) going to x^4 and (8.29). The latter has a larger error at $x = 0$, but the errors are generally smaller in the range $[-1, +1]$ than those from the McLaurin formula, and this was achieved with fewer terms.

Minimax polynomials

If we wish to fit a polynomial $P_n(x)$ to a function $f(x)$ without using a McLaurin series, then there exists an optimum polynomial for the fit. The error is $|P_n(x) - f(x)|$ (absolute) or $|(P_n(x) - f(x))/f(x)|$ (relative). All polynomials will have an error that varies over the range, and within that range there will be a error of maximum magnitude. This should be minimised, and a suitable minimax polynomial is that which has the smallest maximum error. Fike [7] has a detailed description of how such a polynomial approximation can be derived, by an iterative search for both the polynomial's coefficients and those points in the range at which there are error maxima. The reader is referred to Fike [7] for details. We restrict ourselves here to a very simple example. If the function is the square root function $\sqrt{x}$ within the range $[\frac{1}{16}, 1]$, and we wish to fit a first-order polynomial $a_0 + a_1 x$ to it, it is easy to find the minimax function $\frac{2}{9} + \frac{8}{9}x$, which has three maxima in the relative error, at $x = \frac{1}{16}, 0.25$ and 1. The errors are all of the same magnitude, respectively $\frac{1}{9}, -\frac{1}{9}$ and $\frac{1}{9}$, or about 11%. This does not seem very satisfactory as an approximation but it does provide an optimal starting value for a Newton solution, as described in Sect. 8.3, and might lead to fewer Newton iterations than another start would need for convergence.

As mentioned in Press & al. [4], a Chebyshev polynomial is often very close to a minimax approximation, and these are rather easy to compute. See the example in the preceding Section.

8.5 Asymptotic series

There exist functions for which it is not possible to use range reduction and for which one is forced to design an algorithm for large arguments. In these cases a polynomial often cannot be used. One such function is the error function erf and also the related complementary error function erfc. For such functions, asymptotic series or expansions are often the answer. They can also be useful for functions which can be range-reduced, or for which there exist other approximations.

The general form of an asymptotic expansion is

$$f(x) \approx b_0 + b_1 x^{-1} + b_2 x^{-2} + \cdots \ . \tag{8.30}$$

For erfc we have the series

$$\mathrm{erfc}(x) = \frac{1}{\sqrt{\pi} x \exp(x^2)} \left(1 - \frac{1}{2} x^{-2} + \frac{1 \cdot 3}{2^2} x^{-4} + \frac{1 \cdot 3 \cdot 5}{2^3} x^{-6} \cdots \right) \ . \tag{8.31}$$

For the gamma function, defined as

$$\Gamma(x) = \int_0^\infty t^{x-1} \exp(-t) \mathrm{d}t \ , \tag{8.32}$$

we find in Abramowitz & Stegun [3] a few polynomial series with various accuracies and an asymptotic expansion for large arguments.

8.6 Rational functions and continued fractions

These encompass more methods of approximating functions, and are dealt with together here, as they are related to each other.

A rational function is a fraction with polynomials both in the numerator and denominator; that is, generally,

$$f(x) = \frac{a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n}{1 + b_1 x + b_2 x^2 + \cdots + b_m x^m} \tag{8.33}$$

(here b_0 has been factored out). Such a fraction can be converted to a continued fraction with a little effort, either ending as in

$$f(x) = b_0 + \cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \cfrac{a_3}{b_3 + \cfrac{a_4}{b_4 + \ddots + \cfrac{a_n}{b_n}}}}} \tag{8.34}$$

or one that continues indefinitely, and we decide where it should be terminated.

Many text books use a different method of presenting these formulae, which gave book printers a practical problem. The above equation (8.34) is thus often shown in the form

$$f(x) = b_0 + \frac{a_1}{b_1+}\ \frac{a_2}{b_2+}\ \frac{a_3}{b_3+}\ \frac{a_4}{b_4+} \cdots \frac{a_n}{b_n} \tag{8.35}$$

and there are similar variants.

An expression such as (8.34) can be coded in Fortran in the following way, starting from the bottom and moving up:

```
sum = a(n) / b(n)
do i = n-1, 1, -1
  sum = a(i) / (b(i) + sum)
enddo
func = b(0) + sum
```

Here the terminating term was the starting value of the running variable `sum`. Rational fractions can easily be evaluated in the same manner as polynomials, as described above in (8.22).

As mentioned, one can often convert a rational function to a continued fraction by algebraic division. Examples can be found in Fike [7]. This can lead to simpler formulae, but it must be admitted that the device became less interesting with the advent of fast computers.

Chapter 9

Fourier transformation

9.1 Introduction

The Fourier transform is an extremely useful device in modern data processing and is indispensable as an analytical numerical tool in a large number of mathematical, chemical or physical disciplines.

We first describe something that is not a Fourier transform but is related to it and casts light on what is happening when the Fourier transform is used.

Let us look at the cosine and sine transform of some trigonometric functions $h(t)$ of time t. We transform them by integrating the function multiplied by the cosine or sine funtion. This gives the two formulae

$$H_c(f) = \frac{1}{T}\int_0^T h(t)\cos(2\pi f t)\ \mathrm{d}t \tag{9.1}$$

and

$$H_s(f) = \frac{1}{T}\int_0^T h(t)\sin(2\pi f t)\ \mathrm{d}t\ . \tag{9.2}$$

Here f is the frequency, normally in units of Hz or s^{-1} (but the function need not be a function of time). Often we shorten the expressions by using angular frequency $\omega = 2\pi f$, which allows the more compact formulae

$$H_c(\omega) = \frac{1}{T}\int_0^T h(t)\cos\omega t\ \mathrm{d}t \tag{9.3}$$

and

$$H_s(\omega) = \frac{1}{T}\int_0^T h(t)\sin\omega t\ \mathrm{d}t\ . \tag{9.4}$$

These transformations differ from the real Fourier transformation in that they are specific for the cosine or sine function, that we integrate over a limited time range, and that we divide by the time range T. This makes it possible

to transform pure trigonometric functions over a limited range, which, strictly speaking, is not possible to do with the Fourier transform.

Let $h(t) = \cos\omega t$ and $T = 2\pi/\omega$, that is, we integrate over one whole period of the function $h(t)$. Then, by looking up tables of integrals, it is easy to find that $H_c(\omega) = \frac{1}{2}$. If on the other hand we make use of (9.4), we obtain the result $H_s(\omega) = 0$.

If we repeat the process for $h(t) = \sin\omega t$, we obtain the opposite: now $H_c(\omega) = 0$ and $H_s(\omega) = \frac{1}{2}$ for all values of ω (except 0).

Now we let the function be a trigonometric function with another frequency, while we again multiply it by either $\cos\omega t$ or $\sin\omega t$: let $h(t) = \cos a\omega t$, with $a \neq 1$. We integrate over a variable interval $[0, T]$. The process can be carried out with the aid of the identity

$$\cos x \cos y = \tfrac{1}{2}\left(\cos(x - y) + \cos(x + y)\right) \tag{9.5}$$

and a table lookup, and we finally obtain

$$H_c(\omega) = \frac{(a+1)\sin([a-1]\omega T) + (a-1)\sin([a+1]\omega T)}{2(a^2-1)\omega T}\,. \tag{9.6}$$

Inspection of the result reveals that this is a decreasing function of T, going to zero for large T. The same procedure for functions with our sine function (9.4) produces a similar result, once again approaching zero for large T.

Here we have learned something. If we let $h(t) = \cos a\omega t$ and let a increase from zero upwards and let T be large and a multiple of 2π, then we obtain the result that $H_c(\omega)$ is close to zero, except when $a = 1$. The transformation (9.3) thus reacts (for large T) only to a cosine wave of the same frequency ω, and not to other frequencies. Neither does it react to a sine wave of any frequency. Analogously the sine function (9.4) reacts only to another sine wave of the same frequency ω, and not to any cosine wave.

Measured signals often consist of, or can be represented as, a number of cosine and sine waves of different frequencies. We wish to find these components. It is easy now to see that the two transformations can select all the components from such a mixture. The operations (9.3) and (9.4) are linear, that is, the transformation of $bh(t)$ results in b times that for $h(t)$ (b being a constant). If the signal consists of a mixture of cosine waves of different frequencies and we transform with variable ω and (9.3), every time we hit an ω value that matches one of the component's frequency, the result is that component's (half) amplitude. The collection of these is called the spectrum of the signal.

We need just one more thing. If we let our function be a cosine wave with a phase shift: $h(t) = \cos(\omega t + \theta)$, what will be the result? By using (9.3) we obtain $H_c(\omega) = \frac{1}{2}\cos\theta$, and $H_s(\omega) = \frac{1}{2}\sin\theta$ for all ω. Now both transformations react, and the results reflect the phase shift. That is, if we apply both transformations, we obtain two numbers that give us information about the wave's amplitude and phase. We can also express this as having

separated the wave into its cosine and sine components. It is therefore a good idea to apply both transformations at the same time.

We are now ready for the real Fourier transformation.

9.2 Continuous Fourier transform

We restrict ourselves here to the notation used by Press & al. [4]. There are several variants for the presentation of the transform. One of the definitions is

$$H(f) = \int_{-\infty}^{\infty} h(t)\ \mathrm{e}^{i2\pi ft}\mathrm{d}t \tag{9.7}$$

and for the inverse

$$h(t) = \int_{-\infty}^{\infty} H(f)\ \mathrm{e}^{-i2\pi ft}\mathrm{d}f \tag{9.8}$$

where $i = \sqrt{-1}$. Press & al. [4] use the above order of the sign in the exponentials, but most texts define the transforms with the opposite signs. There exist also many variants of the factors, with which the integrals are multiplied, but all variants are useful in practice. The first of the two brings us from the time- to the frequency domain, and the second back again. We write functions of time with lower case letters as $h(t)$, and their transform as $H(f)$ using capitals. We have here integrals with limits from $-\infty$ to $+\infty$, and this leads to certain limitations or conditions.

Recall the identity

$$\cos x + i \sin x = \mathrm{e}^{ix} \tag{9.9}$$

and it is seen that the integration (9.7) in fact represents both a cosine and a sine transform, so that we thereby produce both transforms from the one operation, stored in the real and imaginary parts of the result. We are dealing with complex numbers, which also goes - in principle - for the function $h(t)$, which however most often is real. We will see below, that the time function must be provided as a complex function, usually with a zero imaginary component.

Once again we use the notation of Press & al [4] notation, which represents a transform pair in the form

$$h(t) \Leftrightarrow H(f) \tag{9.10}$$

and which simplifies the following expressions.

Conditions

There are some conditions to be satisfied before performing these operations. The most important, that is sufficient but not quite necessary, is that that the

function $h(t)$ satisfies

$$\int_{-\infty}^{\infty} |h(t)| \mathrm{d}t < \infty \tag{9.11}$$

which apparently excludes the trigonometric functions we used in the above, as well as some others. But the condition is not stringent; it guarantees that the transformation can be carried out, but there are functions that do not satisfy the condition, which nevertheless can be transformed. One of these is the sinc function $\sin(\omega t)/\omega t$. The reader is referred to Brigham [8] for the theoretical details. In practice, we deal with signals (functions of time or distance) that lie within a limited range, and one integrates within the limits that are given. This also provides the connection with the discrete transform, dealt with below. But even with the infinite limits we can obtain a transformation. Thus we can say that if $h(t) = \cos(2\pi f_0 t)$, then $H(f)$ is two lines at $f = \mp f_0$, and of infinite magnitude. This corresponds to "delta functions", $\frac{a}{2}\delta(f \mp f_0)$. In this way we are allowed to speak about transformations of these functions, even though they do not satisfy condition (9.11).

Some relationships

Press & al. [4] show a table which concisely presents some relatonships, and we follow their style:

If ...	then...
$h(t)$ is real	$H(-f) = H^*(f)$
$h(t)$ is imaginary	$H(-f) = -H^*(f)$
$h(t)$ is even	$H(-f) = H(f)$, i.e. $H(f)$ is even)
$h(t)$ is odd	$H(-f) = -H(f)$, i.e. $H(f)$ is odd)
$h(t)$ is real and even	$H(f)$ is real and even
$h(t)$ is real and odd	$H(f)$ is imaginary and odd
$h(t)$ is imaginary and even	$H(f)$ is imaginary and even
$h(t)$ is imaginary and odd	$H(f)$ is real and odd.

The notation $H^*(f)$ indicates the complex complement.

The following relatonships are also of interest:

$ah(t)$	$\Leftrightarrow$	$aH(f)$	linearity
$h(t)+g(t)$	$\Leftrightarrow$	$H(f)+G(f)$	additivity
$h(at)$	$\Leftrightarrow$	$\frac{1}{\lvert a\rvert}H(\frac{f}{a})$	time scaling
$\frac{1}{\lvert b\rvert}h(\frac{t}{b})$	$\Leftrightarrow$	$H(bf)$	frequency scaling
$h(t-t_0)$	$\Leftrightarrow$	$H(f)\mathrm{e}^{i2\pi f t_0}$	time shift
$h(t)\mathrm{e}^{-i2\pi f_0 t}$	$\Leftrightarrow$	$H(f-f_0)$	frequency shift
$\frac{\mathrm{d}^n}{\mathrm{d}t^n}(h(t))$	$\Leftrightarrow$	$(-i2\pi f)^n H(f)$	differentiation

Convolution

Two functions, $h(t)$ and $g(t)$, can be transformed by convolution according to the definition

$$h(t) * g(t) \equiv \int_{-\infty}^{\infty} g(\tau)h(t-\tau)\mathrm{d}\tau \; . \tag{9.12}$$

This operation, for a range of t, is time-consuming but can be speeded up on the basis of the transform pair

$$h(t) * g(t) \Leftrightarrow G(f)\, H(f) = H(f)\, G(f) \; , \tag{9.13}$$

that is, can be computed in the frequency domain by a simple multiplication of the transforms of the two functions. This is much less computer intensive, thanks to the fast Fourier transform FFT, to be described below.

Correlation

Correlation between two functions is related to convolution and here too there is a useful connection to Fourier transformation. Correlation is defined as

$$\mathrm{corr}(g,h) \equiv \int_{-\infty}^{\infty} g(t+\tau)h(\tau)\mathrm{d}\tau \tag{9.14}$$

and we have the transform pair

$$\mathrm{corr}(g,h) \Leftrightarrow G(f)\, H^*(f) \; . \tag{9.15}$$

If the functions are different, we have cross correlation over a range t and if the two functions are identical, we have autocorrelation. This can be thought of as a function of t that indicates how much a function is self-similar at a distance t. If, for example, we have a cosine wave and $t = 2\pi t_0$, that is, a distance of one whole wave length, then the function is exactly self-similar (the operation produces the same as the function squared $\cos^2 \omega t_0$), which

results in an autocorrelaton of 1. On the other hand, distance $t = \pi t_0$, results in an autocorrelation of -1, if $t = \frac{1}{2}\pi t_0$ it is zero.

Autocorrelation is defined as the continuous function

$$r(\tau) = \lim_{T\to\infty} \frac{1}{T} \int_0^T h(t)h(t+\tau)\mathrm{d}t \tag{9.16}$$

where we now use τ as the symbol for distance (or offset, also called lag). This formula changes for a discrete sequence of length n (see the next paragraph) to

$$r(\tau) = \frac{1}{n-\tau} \sum_{i=1}^{n-\tau} h_i h_{i+\tau} \ . \tag{9.17}$$

The reason for letting i run only to $n-\tau$ is of course that there is no value at $i > n$. This formula, which for large n is computer-intensive, can be computed much more efficiently by the discrete Fourier transformation, on the basis of the relation between the autocorrelation and the power spectrum of a signal.

9.3 The discrete Fourier transform DFT

We are often dealing with so-called time series or, generally, sequences of values, perhaps measured, with equal intervals between them, often of time (but not necessarily). So here we have a number of "samples" h_k from an underlying continuous function $h(t)$. It is said that the sequence is sampled in the time domain at a certain sampling interval δt or at a sampling frequency $f_s = \delta t^{-1}$. Now we wish to calculate a Fourier transform of the point sequence, such that it will resemble the Fourier transform of the function. Therefore we must redefine the continuous function form of the transform to that for a discrete sequence, by suitably changing the integrals (9.7) and (9.8). Clearly we do not integrate from $-\infty$ to $+\infty$, and our sequence is not infinite either. A hidden assumption will be that the sequence $h_k, k = 0 \ldots N-1$ repeats itself infinitely, and we are looking at one period of that infinite periodic sequence.

We must establish some important relationships between quantities in the time domain and the frequency domain. In the time domain we have N points $h_k, k = 0, \ldots N-1$, and the sequence corresponds to a range of length T. The point numbering is $0 \ldots N-1$, and this eases the expression for the transform. That is, the intervals δt are equal to T/N (we stay with the time domain, although some signals lie along other dimensions, such as distance; in such cases, the symbols will be a little different, but the treatment is the same). When we transform such a sequence, we obtain a new sequence of points called a spectrum, where the points lie along the frequency axis f (or generally the inverse of the dimension of the original sequence). The two independent variables do not appear in the sequences; we must ourselves be aware of them. That is, for example, we must know T and δt. A complicating factor is that the

spectrum is a function of frequency going from negative values through zero to positive values, and this too we must keep in mind. The smallest possible frequency contained in the sequence is that for which the length of the sequence is one whole period or wave length. It is thus T^{-1}. All other frequencies are multiples of this, and it has the symbol δf. The spectrum consists of N points $H_k, k = -\frac{N}{2}, \ldots \frac{N}{2} - 1$, for the corrresponding frequencies f_k, which run from $-\frac{N}{2}\delta f$ through zero and up to $(\frac{N}{2} - 1)\delta f$.

Here is a summary of the above:

- The time series is $h_k, k = 0, \ldots N-1$
- This sequence corresponds to $t_k = k\delta t, k = 0, \ldots N-1$.
- The total time is $T = N\delta t$.
- Sampling intervals are $\delta t = \frac{T}{N}$.
- Spectrum frequencies are $H_k, k = -\frac{N}{2}, \ldots \frac{N}{2} - 1$.
- The spectrum's frequencies are $f_k, -\frac{N}{2}\delta f, \ldots (\frac{N}{2} - 1)\delta f$
- The frequency intervals are $\delta f = \frac{1}{T}$.
- The sampling frequency is $f_s = \frac{1}{\delta t}$.

The continuous integrals (9.7) and (9.8) can now be expressed in discrete form as sums. Recall that the sequence to be transformed in itself carries no information of the sampling intervals δt; these are arbitrarily set to unity. The total time T is then equal to N. This means that $\delta f = N^{-1}$. One must suitably scale the spectrum axis. The formulae are

$$H_j = \sum_{k=0}^{N-1} h_k \mathrm{e}^{i2\pi f_k t_j} \tag{9.18}$$

and, since $f_k = k\delta f = k/N$ and $t_j = j\delta t = j$, this becomes

$$H_j = \sum_{k=0}^{N-1} h_k \mathrm{e}^{i2\pi jk/N} \ . \tag{9.19}$$

Note that we should multiply by δt but this was set equal to unity. The inverse transform is

$$h_j = \sum_{k=0}^{N-1} H_k \mathrm{e}^{-i2\pi t_k f_j} \tag{9.20}$$

and it becomes the corresponding

$$h_j = \sum_{k=0}^{N-1} H_k \mathrm{e}^{-i2\pi jk/N} \ . \tag{9.21}$$

Each of these two formulae is applied for all possible (that is, N) f_k or t_k, and for each N, exponentials must be computed. This means N^2 calculations, and the exponentials require the most computing time. The two formulae are called the discrete Fourier transform or DFT.

There is a minor device for improvement: what we are doing here is to replace an integral with a sum, seen to be based on block integration, which is the least accurate numerical integration method. From Chap. 3 we know, however, that the trapezium method is much better, and that it differs from block integration only in that the first and last point in the sequence are halved. We can do this here, and obtain a better integration result. We must apply this to the given sequence, because the professional DFT routines or FFT (see below) do not apply the halving.

There is another minor complication. When the transformation (9.19) is performed for all frequencies, we have a spectrum, but in a strange order, see Fig. 9.1. The first $N/2$ points in the $H(f)$-sequence represent, perhaps as expected, the frequencies $0, \delta f, 2\delta f, \ldots, (\frac{N}{2}-1)\delta f$. But there are also negative frequencies and these are in the second half, in the order $-\frac{N}{2}\delta f, (-\frac{N}{2}+1)\delta f, \ldots, -\delta f$. If we want to present the spectrum of the sequence, we must exchange the two halves. We then obtain points corresponding to the frequency sequence $-\frac{N}{2}\delta f, (-\frac{N}{2}+1)\delta f, \ldots, -\delta f, 0, \delta f, 2\delta f, \ldots (\frac{N}{2}-1)\delta f$.

We are dealing with complex numbers in both domains. Finally, there is the aspect of scaling. We should be able to transform from the time domain to the frequency domain, and back again. When we do this, we find that the original values are multiplied by N, and we must be aware of this if the scaling matters. Normally this is not important.

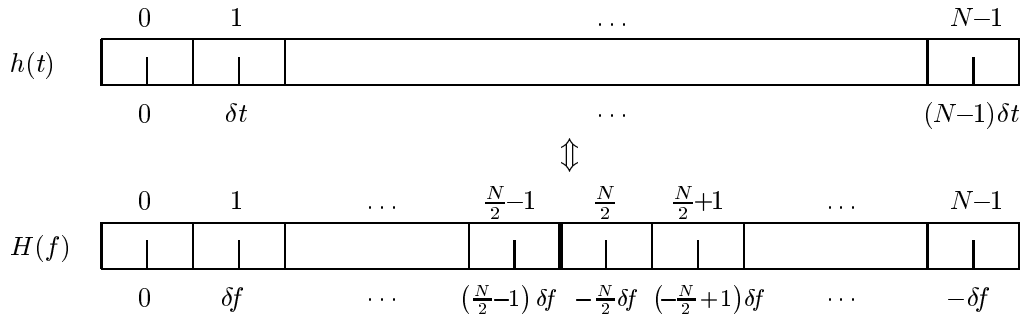

Figure 9.1: Numbering and order with Fourier transformation. Every box has a real and imaginary part. In the frequency domain the positive frequencies come first, followed by the negative half.

Fast Fouriertransform FFT

A Fourier transform computed according to equations (9.19) or (9.21) demands N^2 multiplications for a sequence of N points and, worse, as many computations of exponentials. This can be very heavy computing as N becomes large. Fortunately there are ways of reducing the computation drastically. One of these was discovered by Runge in 1903 [9]. He noted that there are not N^2 different exponential values to be computed but only N, because many of them can be range-reduced to the first N. If the argument in e^{ix} exceeds $i2\pi$, then $i2\pi$ can be subtracted from it (see range reduction in Chap. 8), which results in an argument for which the calculation has already been done. This reduces the computation time significantly.

Another way of improving the process is described in Press & al. [4] and is based on a discovery made by Danielson and Lanczos in 1942 [10]. Briefly, the method consists of splitting a Fourier transformation of N points into two, each of $N/2$ points, one being those with even numbering $H^{(e)}$ and the other with odd numering $H^{(o)}$. The formula is

$$H(j) = H^{(e)}(j) + \mathrm{e}^{i2\pi j/N} H^{(o)}(j) \; . \tag{9.22}$$

If this is done just once, we have already saved much time, because we now have only $N^2/4$ operations to perform. But it does not stop here. The statement also goes for the half-lengths, which also can be split up in the same manner, etc. This requires a deal of administration, to keep track of the exponentials and what must be added to what, but in the end the method requires only $N \log_2 N$ operations for an N-point sequence. If for example $N = 1000$, we need 10^6 operations for the DFT, but only about 10^4 for the improved algorithm, which is called the **Fast Fourier Transform** or **FFT**. The interested reader is referred to Press & al. [4] or Brigham [8] for a more detailed description of the algorithm. The end result is of course precisely the same sequence of numbers as from using the DFT.

In order for the FFT to be optimally used, the number of points in the sequence to be transformed should be a power of 2, that is, $N = 2^m$.

9.4 An example

In order to elucidate the above, a concrete example is given here. There exists a file consisting of (x, h) coordinates, where x is the distance in nm along a surface scanned by a needle passed over it and h is the height of the needle above the surface so that it just touches. The surface is that of a Ni crystal with Au deposited on it. The needle must go up and down, so as to skim along the tops of the atoms. The data file has 256 measurement points that stretch over a distance $X = 10.33$ nm. This is a signal from a so-called *scanning tunnelling microscope* or STM. There is some uncertainty in the measurements, and the Fourier transform can be used to find the mean

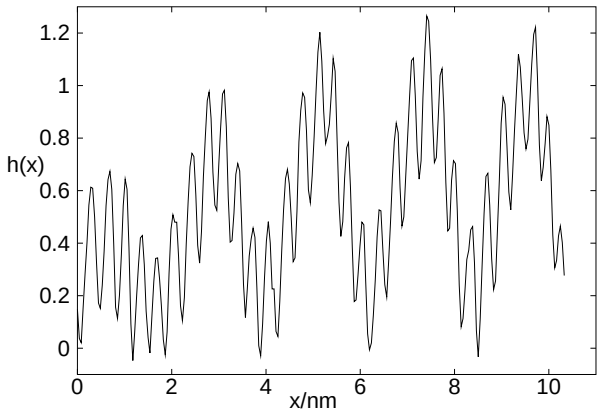

Figure 9.2: STM signal, the height h of a needle from a surface as a function of distance x

distance between neighbouring Ni atoms and the Au atoms in the top layer. There is some jitter in the distances between wave tops. We can immediately obtain an estimate of the mean distances by counting the number of waves. There are 31 of the short waves and about 5 of the longer waves. This produces mean distances (or wave lengths) of 0.32 and 2 nm respectively. We can now refine these values.

We have the following information on the signal that we can use:

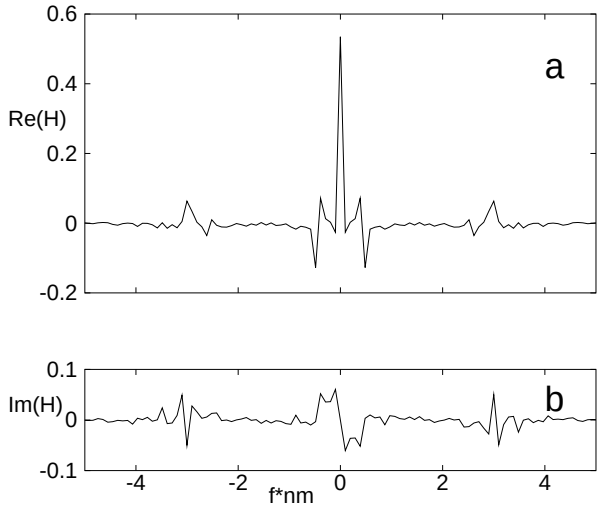

Figure 9.3: Fourier spectrum H of the signal in Fig. 9.2. (a): the real component and (b), the imaginary component.

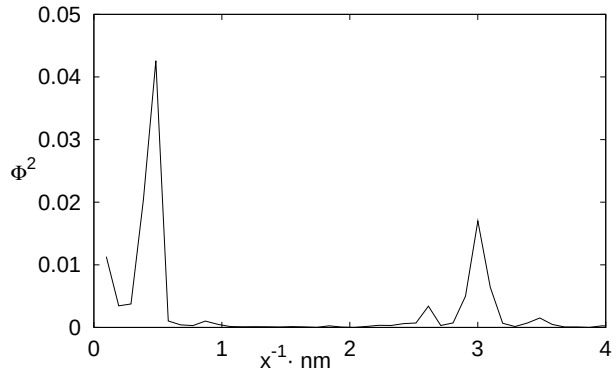

Figure 9.4: Power spectrum computed from the spectrum in Fig. 9.3

- $X = 10.33$ nm (total distance)
- This implies a minimum frequency $\delta f = 1/X = 1/10.33$ nm^{-1}.
- There are 256 points so that $\delta x = X/256 = 10.33/256$.
- The sampling frequency is $f_s = 1/\delta x = 256/10.33$ nm^{-1}.
- The maximum frequency is $F_{max} = 1/(2\delta x) = 128/10.33$. This corresponds to a wave length $2\delta x$.

The mean distance between atoms should produce marked peaks in the spectrum at the frequencies corresponding to the two inverse distances (here in units of nm^{-1}). We therefore compute a Fourier transform of the sequence, which must first be stored as an array of type `complex`. Fig. 9.3 shows both the real and imaginary components of the resulting spectrum. The smallest frequency is 10.33^{-1} and the largest, given from the 256 points, is thus 127.5/10.33 or about 12.3. There is however not much to see outside the range $[-5, 5]$, so only this range is shown.

A remark on the frequency axis is in order. It has been computed from our knowledge that $\delta f = 1/10.33$ and that N = 256. That which is transformed is only the sequence of h values, and there is no information in the spectrum on the actual frequencies. The spectrum sequence H is simply indexed $0 \ldots 255$, and we must translate these indices to frequencies, see also Fig. 9.1).

We note, in the real component of the spectrum, a marked peak at $f = 0$. This reflects the mean value of the signal h, as is clearly seen from (9.19). This will be removed later. We also note other peaks but they are not very prominent; these lie at about 0.5 and 3, corresponding to distances 2 and 0.33 nm. We now convert the complex spectrum into a power spectrum by

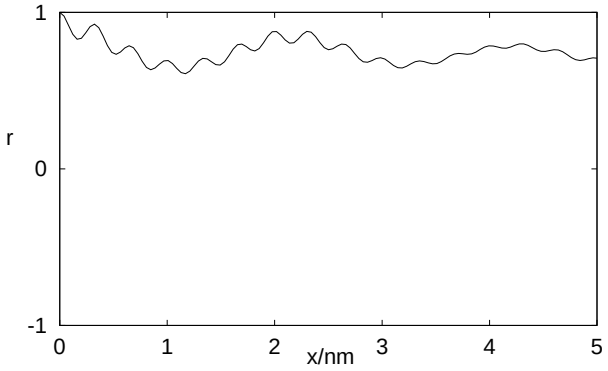

Figure 9.5: Autocorrelation function computed from the power spectrum in Fig. 9.4, without zeroing the value at $f = 0$.

computing for each element

$$\Phi^2(j) = |H(j)|^2 \tag{9.23}$$

which produces a sequence of power values shown in Fig. 9.4, where we now show only that half of the spectrum at positive frequencies. The element at zero index still contains the effect of mean value, and is not shown. We see the clear peaks, whose position can be determined. They lie close to 0.5 and 3 nm^{-1}, respectively, corresponding to the approximate estimates of 2.0 and 0.33 nm mentioned above. As a last step we can compute the autocorrelation function. The original signal consists of periodic waves, and this should be reflected in $r(\tau)$ which should also be periodic with the same wave lengths.

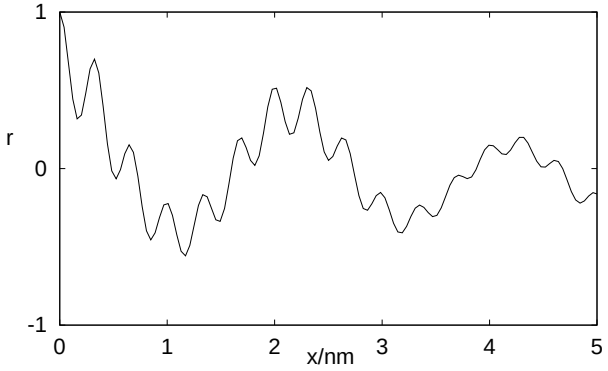

Figure 9.6: Autocorrelation function computed from the power spectrum in Fig. 9.4, after zeroing at $f = 0$.

We know that $r(0) = 1$, and we can therefore scale the whole r sequence by dividing by $r(0)$. The x axis is again computed, using $\delta x = 10.33/255$ and the indexing of the sequence. Fig. 9.5 does show periodicities, but the whole curve lies close to unity, which is due to the effect of the mean signal value, which has not yet been removed. There ought to be some negative r values. So we remove the zero-frequency element of the power spectrum by setting $\Phi(0) = 0$. Fig. 9.6 shows the result. Now r ranges between -1 and +1 as it should and we can use the plot again to calculate distances. We count 15 short waves over a distance of 5 nm, giving 0.33 nm; and 2 long waves up to about $x = 4.2$, so they have length 2.1 nm, although there is some uncertainty in this estimate.

9.5 Sampling and aliasing

When one samples a continuous signal, it is important to ensure the correct sampling frequency f_s or the sampling interval $\delta t = f_s^{-1}$. This is most easily illustrated as follows. If the signal contains components up to a highest frequency f_{max}, then the sequence of samples should be able to represent the signal up to this highest frequency. If we do not sample sufficiently densely, there appears a strange effect called **aliasing**. Fig. 9.7 shows this situation. The solid line is a sine wave, and the points are samples of the curve, but the sampling intervals are a little larger than the wavelength. Clearly, the points follow a curve that does not represent the function to be sampled. Fig. 9.8 however is an example of sufficiently dense sampling, with several samples per wave. If we draw a line through these points, the resulting curve represents the original reasonably well.

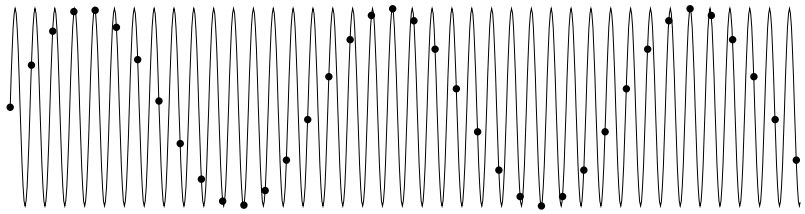

Figure 9.7: Aliasing of a sine wave with insufficient sampling

The condition for sufficiently dense sampling is that there must be at least two samples per wave of the highest-frequency component of the signal. That is,

$$f_s \geq 2f_{max} \; . \tag{9.24}$$

This is called the **Nyquist criterion**. There are two practical ways to satisfy the criterion. The first is to find out what f_{max} is, and to set f_s appropriately. The other, which is often dictated when there is maximum possible sampling

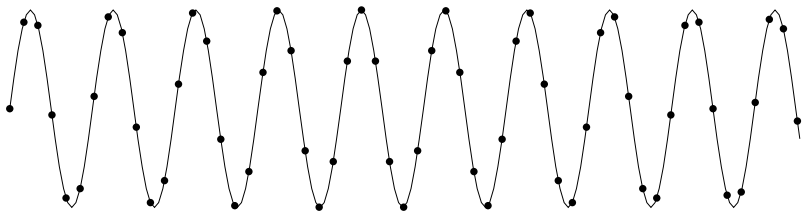

Figure 9.8: Aliasing of a sine wave with sufficient sampling

frequency or the number of samples must be limited, is to change f_{max} by means of filtering of the measured signal, so that the criterion is satisfied. Filtering is a complex subject. There is some information in Chap. 10 on digital filtering.

Chapter 10

Digital Signal Processing

By digital signals we mean a sequence of points representing an underlying continuous function or signal. The points might come from "sampling" the analogue signal, or from a computation. Since the values are in the form of discrete points, we speak of a discrete sequence or, if they are points in time, a time series. Just as there are physical methods for processing electrical signals, such as filtering, smoothing, differentiating etc., there exist digital methods for discrete series for these operations. The subject falls under the term digital signal processing. There exist many textbooks with that term in the title, and here are described some of the basics.

10.1 Filtering

The term filtering often means smoothing of a signal, but other uses exist such as removing a base line (high-pass filtering) or differentiation. Here some of the common filters are described. They are mean value-, least squares- and recursive filters. All these fall into one or other of the two categories "finite impulse response" (FIR) or "infinite impulse response" (IIR) filters.

Let us say we have a sequence of N values $s_i, i = 1 \ldots N$, and want to filter them, producing a new sequence $u_i, i = 1 \ldots N$, which we call the filter response. Generally all digital filters can be mathematically expressed as

$$u_i = \sum_{k=i-m}^{i+m} a_k u_k + \sum_{k=i-m}^{i+m} b_k s_k \tag{10.1}$$

where m is a half window, that is, we filter over a "window" of width $w = 2m + 1$. If some of the coefficients a_i are nonzero, we are dealing with a recursive or IIR filter, because for every i, u_k uses values that are part of the response, already computed. This brings with it the phenomenon of infinite impulse response IIR, meaning that a given s_i value will cause a response in all further u_i values. If we for example imagine the sequence of points $1, 0, 0, \ldots 0$ and a pure IIR filter, like

$$u_i = 0.5u_{i-1} \quad (u_1 = s_1) \tag{10.2}$$

then we will see the response sequence $1, 0.5, 0.25, ...\left(\frac{1}{2}\right)^{N-1}$ and it is clear that the single 1 in the signal is felt in the whole response sequence. If on the other hand all a_i are equal to zero, we have an FIR filter. The same sequence filtered by

$$u_i = s_{i-1} + s_i + s_{i+1} \quad (u_1 = s_1) \tag{10.3}$$

has the response sequence $1, 1, 0, \ldots, 0$, so that the single 1 has led to a response that lasts only one interval.

In the following the two kinds of filters are described, under three headings. A sequence of 256 points serves as example for all and is seen in Fig.10.1. It is a Gaussian curve carrying some noise. The noise itself is also distributed Gaussian, that is, it has a normal distribution.

Mean value filter

A mean value filter moves over the sequence and takes the mean value over a window of width $w = 2m + 1$ at every point. The formula is

$$u_i = \frac{1}{2m+1} \sum_{k=i-m}^{i+m} s_k \ . \tag{10.4}$$

This is of the FIR type. In practice there is a minor problem at both ends, where there are not enough points under the window to make a sum. Here we begin with $m = 0$ for u_1, $m = 1$ for u_2 (3 points), and so on, until we can get the required w points. When approaching the end, we let the window shrink again in the opposite sense. Fig. 10.2 shows the result of a filter on the curve in Fig. 10.1, now smoothed with respectively $w = 5$ and $w = 19$ (w must be odd). This filter is simple to program but is not especially good. It has the disadvantage of lowering any peaks in the signal, as is seen in Fig. 10.1 for $w = 19$.

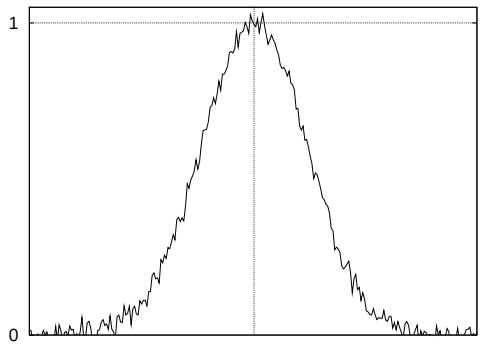

Figure 10.1: A noisy signal

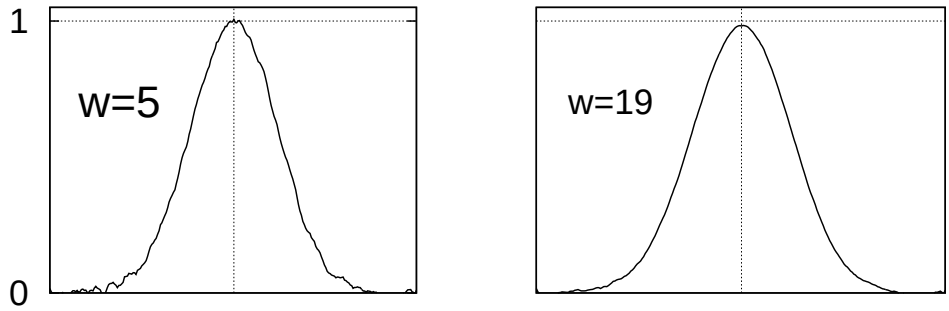

Figure 10.2: Signal in Fig. 10.1 smoothed by a mean value filter

Least squares filters

The reason that the mean value filter reduces the height of a peak is clearly that it cannot follow the curvature of the signal. This is possible for a filter based on a least squares polynomial over a given window, where the centre point of the fitted polynomial replaces the old point. Often a parabola or cubic function is used for this. As higher and higher order polynomials are used, they begin to show undesirable oscillations, which a parabola or cubic does not. This type of filter is also of the FIR kind. It is not necessary to compute coefficients for every window, which would make the filter very inefficient. This is avoided by precomputing a number of weighting factors or coefficients b_k for a given polynomial. A relatively simple example shows how to obtain these.

Assume that we have five points in each window and, for convenience, we centre the window around the centre point so that the points lie at $x_i, i = 1, \ldots 5$, and they are $-2, -1, 0, 1, 2$. At these positions we have unspecified general values $s_i, i = 1, \ldots, 5$. We wish to fit a parabola to the five points, that is, the function

$$u = a_0 + a_1 x + a_2 x^2 \ . \tag{10.5}$$

The method of least squares is described in Chap. 6, and here we have for the coefficients

$$\begin{aligned} -2\sum_{i=1}^{5}\left(y_i - a_0 - a_1 x_i - a_2 x_i^2\right) &= 0 \\ -2\sum_{i=1}^{5}\left(y_i - a_0 - a_1 x_i - a_2 x_i^2\right)x_i &= 0 \\ -2\sum_{i=1}^{5}\left(y_i - a_0 - a_1 x_i - a_2 x_i^2\right)x_i^2 &= 0 \end{aligned} \tag{10.6}$$

which we can rewrite by separating and rearranging terms, to obtain

$$\begin{aligned} a_0 \sum_i 1 \; + a_1 \sum_i x_i + a_2 \sum_i x_i^2 &= \sum_i y_i \\ a_0 \sum_i x_i + a_1 \sum_i x_i^2 + a_2 \sum_i x_i^3 &= \sum_i x_i y_i \\ a_0 \sum_i x_i^2 + a_1 \sum_i x_i^3 + a_2 \sum_i x_i^4 &= \sum_i x_i^2 y_i \end{aligned} \tag{10.7}$$

(sums are written without limits for easier reading). We know all x_i, and can substitute for them on the right-hand side. We then get

$$\begin{aligned} 5a_0 \qquad\quad +10a_2 &= \sum_i y_i \\ 10a_1 \qquad\quad &= \sum_i x_i y_i \\ 10a_0 \qquad\quad +34a_2 &= \sum_i x_i^2 y_i \; . \end{aligned} \tag{10.8}$$

We are only interested in a_0, because we only want the single value at the centre of the parabola over the window at $x = 0$. It is easy to solve the system to get this coefficient

$$a_0 = \frac{1}{35}\Big(-3y_1 + 12y_2 + 17y_3 + 12y_4 - 3y_5 \Big) \; . \tag{10.9}$$

To use this as a filter, we must apply the following formula to every window

$$u_i = \frac{1}{35}\Big(-3s_{i-2} + 12s_{i-1} + 17s_i + 12s_{i+1} - 3s_{i+2} \Big) \; . \tag{10.10}$$

The hard work was done in computing the coefficients or weights, but this has only to be done once. There are extensive tables for many polynomial orders and different filters in the classic paper by Savitsky and Golay [11]. Such filters

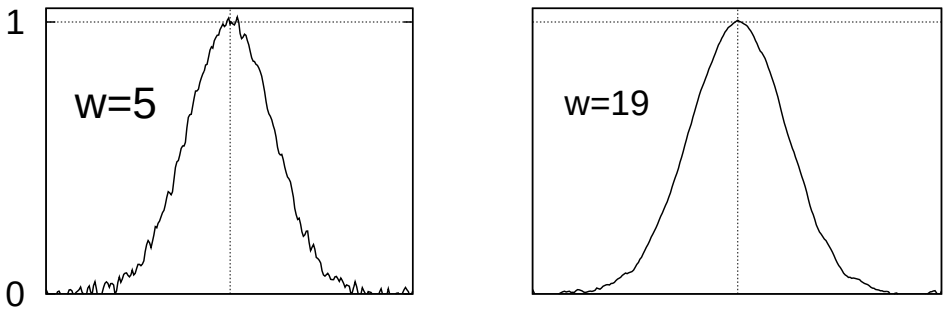

Figure 10.3: Signal in Fig. 10.1 smoothed with a least squares filter

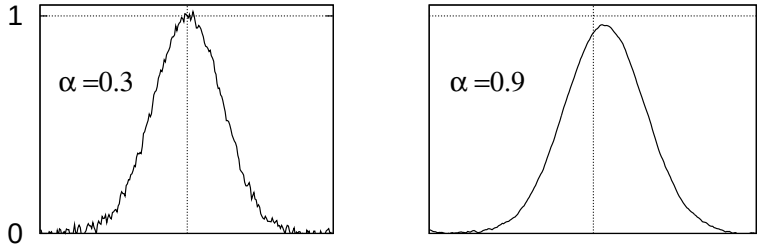

Figure 10.4: Recursive IIR filter, with α as indicated

are named after these two workers, that is, Savitsky-Golay filters. The tables include also weights for differentiating filters, see below.

Fig. 10.3 shows the result for a fitted parabola with two different window widths. The filter does not smooth much better than the moving average but it does preserve the signal peak.

Recursive (IIR) filters

As mentioned above, a recursive or IIR filter is one for which some a-coefficients are non-zero in the general filter equation (10.1). One example is

$$u_i = \alpha u_{i-1} + (1 - \alpha)s_i \; . \tag{10.11}$$

The larger α is, the more the formula makes use of already filtered u values.

Fig. 10.4 shows the result. For $\alpha = 0.3$ there is not much smoothing, but there is for $\alpha = 0.9$. On the other hand the peak has clearly been shifted to the right and reduced. This filter is a discrete approximation of a so-called first order low-pass filter that can be realised using electronic components, as shown in Fig. 10.5. This filter has a time constant equal to RC, which means that the cut-off frequency ω_c is equal to $(RC)^{-1}$. It is at this frequency that the Fourier spectrum of the signal begins to fall off. The frequency is relative to the sampling frequency, here taken as unity. We return to the response spectrum of the filter below.

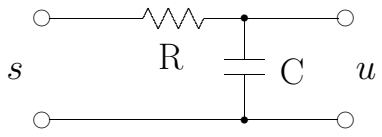

Figure 10.5: Analogue low-pass filter

Differentiating

Differentiation can also be done digitally on a discrete sequence. One way to do this is to differentiate a polynomial fitted to the sequence as described in Sect. 10.1, for example a least squares parabola as in (10.5). For a centred window we need the coefficients a_1, and from the equation set (10.8) we get, for a five-point parabola fit,

$$u_i' = \frac{1}{10}\big(-2\,y_1 - y_2 + y_4 - 2\,y_5\big) \tag{10.12}$$

(note that the centre point does not enter the approximation). For a simple three-point approximation we have

$$u_i' = \frac{1}{2}\big(-\,y_1 + y_3\big) \tag{10.13}$$

which can be regarded as the slope of a straight line drawn between points 1 and 3. There have been instruments which make use of an even simpler algorithm,

$$u_i' = -y_1 + y_2 \tag{10.14}$$

which is also a slope. This formula has the drawback that it actually refers to a point midway between the two, so that the resulting sequence is shifted in the time axis by half an interval.

Fig. 10.6 shows the differentiated curve in Fig. 10.1, for which a five-point least squares parabola has been used.

The article by Savitsky and Golay contains also tables of coefficients for differentiation by multipoint least squares polynomials.

10.2 Frequency spectra of filter response

Often we are interested in controlling the frequency or power spectrum of a signal to be filtered, or the spectrum of the response of the filter to be employed.

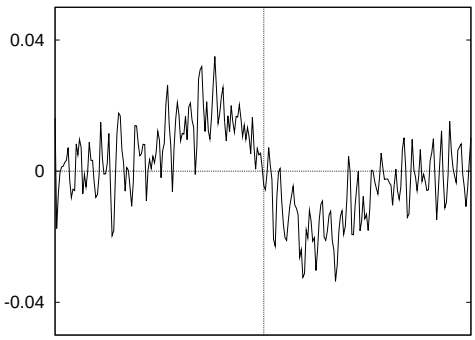

Figure 10.6: Five point least squares parabola differentiation

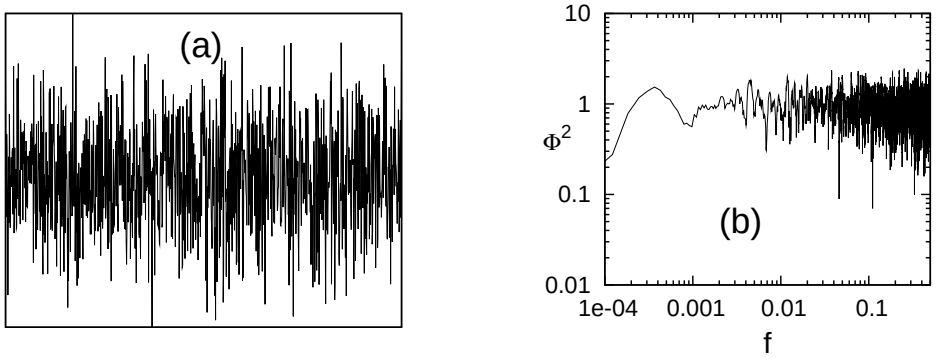

Figure 10.7: Random noise (a) and its power spectrum (b)

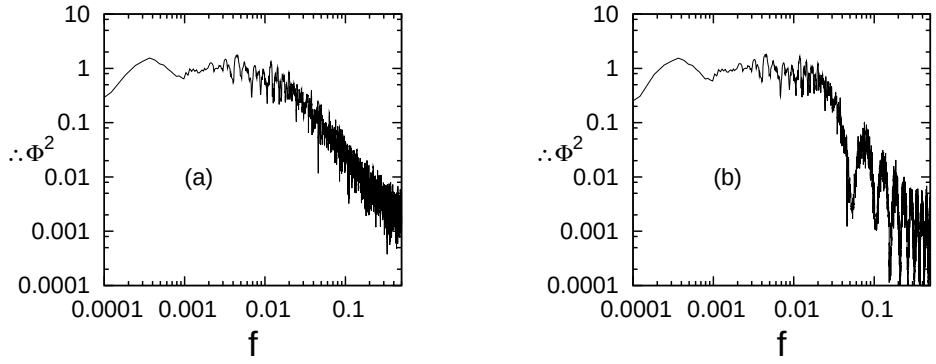

Figure 10.8: Power spectrum of (a) IIR-filtrered random noise for $\alpha = 0.9$, and (b) FIR-filtrered, 19-pt FIR parabola

The filter's response is plotted as the spectrum, and one aims to design a filter with a desired spectral response. What this response will be, can be computed from the approximation formula used. Fig. 10.7(a) shows a sequence of points randomly distributed with a Gaussian distribution. It is unfiltered, and is called "white noise"; it has an even power spectrum. Fig. 10.7(b) shows the power spectrum of this signal. It is usual to represent the spectrum on a log-log scale. The spectrum is close to even, as it should be. That is, the signal contains approximately the same intensities at all possible frequencies. When we filter the signal, we should see the effect of the filter response in the new spectrum. In Fig. 10.8(a) we see the effect of a simple IIR-filter (10.11) with α set to 0.9. It is clear that the spectrum begins to fall off at a frequency of 0.02 (relative to the sampling frequency), corresponding to a wave length of about 50 sampling intervals. The frequency at which the fall-off begins is called the "break frequency". In Fig. 10.8(b) we see the result of filtering the signal with an FIR filter consisting of a 19-point least squares parabola. The

break frequency is about the same as for the IIR filter.

With the aid of z-transform theory, which exceeds the bounds of this booklet, it is possible to predict the spectral response of any discrete filter. For the two above-mentioned simple IIR filters, the prediction is a break frequency at about $(1 - \alpha)/2\pi\alpha$. In the example, an α value of 0.9 was used, and the resulting prediction fits reasonably well with the observed 0.02 from 10.8(a). For a parabola fitted over a window of N points the prediction is about $0.5/N$.

Bibliography

[1] Løfstedt, Datamaskinens tal, Technical Report DAIMI FN-45, Daimi, Aarhus Universitet, 1990, avail. from Comp. Sci. Dept.

[2] O. Østerby, The error of the Crank-Nicolson method for linear parabolic equations with a derivative boundary condition, Report PB-534, DAIMI, Aarhus University, 1998, avail. from Comp. Sci. Dept.

[3] Abramowitz, Stegun (editors), Handbook of Mathematical Functions, Dover, New York, 1965, reprinted 1972; is in the Chem. library.

[4] Press, Teukolsky, Vetterling, Flannery, Numerical Recipes in Fortran, Cambridge University Press, Cambridge, 2 edition, 1986, is in the Chem. library.

[5] O. Østerby, Romberg integration, Report, DAIMI, Aarhus University, Nov. 17, 2005, avail. from Comp. Sci. Dept.

[6] Z. Kopal, Numerical Analysis, Chapman & Hall, London, 1955, is in the State Library.

[7] Fike, Computer Evaluations of Mathematical Functions, Prentice-Hall, NJ, 1968, is in the State Library.

[8] Brigham, The Fast Fourier Transform, Prentice-Hall, New Jersey, USA, 1974, is in the Chem. library.

[9] Runge, Z. Math. Phys. 48 (1903) 433.

[10] Danielson, Lanczos, J. Franklin Inst. 233 (1942) 435.

[11] A. Savitzky, M. J. E. Golay, Anal. Chem. 36 (1964) 1627.