Math 226

Introduction to Numerical Mathematics

Evolving notes

Simon Foucart

Foreword

Contents

I Preliminaries **9**

1 Introductory remarks **10**

1.1 Taylor series and ‘big-O’ notations 10

1.2 Evaluating a polynomial 11

1.3 Exercises 12

2 Programming suggestions **14**

2.1 Programming and coding advice 14

2.2 Case studies 16

2.3 Exercises 18

3 Non-decimal bases **19**

3.1 Base-β numbers 19

3.2 Conversions 20

3.3 Conversion $2 \leftrightarrow 8 \leftrightarrow 16$ 21

3.4 Exercises 21

4 Computer arithmetic **23**

4.1 Floating-point representation 23

4.2 Computer errors . . . 23

4.3 Loss of significance . . . 24

4.4 Exercises . . . 25

5 Linear algebra review **26**

5.1 All you need to know . . . 26

5.2 Exercises . . . 27

II Solving linear equations **28**

6 Gaussian elimination **29**

6.1 Linear systems . . . 29

6.2 Computational cost . . . 31

6.3 Gaussian elimination . . . 32

6.4 Exercises . . . 35

7 The LU and Cholesky factorizations **36**

7.1 LU-factorization . . . 36

7.2 Gaussian elimination revisited . . . 39

7.3 Cholesky factorization . . . 41

7.4 Exercises . . . 41

8 Iterative methods **44**

8.1 Description of two basic schemes . . . 44

8.2 Conditions for convergence . . . 46

8.3 Exercises . . . 48

9 Steepest descent and conjugate gradient methods **50**

9.1 Steepest descent algorithm 51

9.2 Conjugate gradient method 51

9.3 Exercises 54

10 The QR factorization **55**

10.1 The Gram–Schmidt orthogonalization process 55

10.2 Other methods 57

10.3 Exercises 59

11 Linear least-squares problems **60**

11.1 Statement of the problem 60

11.2 Solution of the problem 61

11.3 Exercises 61

III Solving nonlinear equations **62**

12 Bisection method **63**

12.1 Description of the method 63

12.2 Convergence analysis 64

12.3 Exercises 65

13 Newton's method **66**

13.1 Description of the method 66

13.2 Convergence analysis 68

13.3 Generalized setting 70

13.4 Exercises 70

14 Secant method **72**

14.1 Description of the method 72

14.2 Convergence analysis 73

14.3 Exercises 74

IV Approximation of functions **75**

15 Polynomial interpolation **76**

15.1 The interpolation problem 76

15.2 The Lagrange form of the polynomial interpolant 77

15.3 The error in polynomial interpolation 77

15.4 Exercises 79

16 Divided differences **80**

16.1 A definition 80

16.2 Recurrence relation 81

16.3 The Newton form of the polynomial interpolant 82

16.4 Exercises 84

17 Orthogonal polynomials **85**

17.1 Inner product 85

17.2 Orthogonal polynomials for a general weight 86

17.3 Three-term recurrence relation 87

17.4 Least-squares polynomial approximation 89

17.5 Exercises 90

18 Trigonometric interpolation and FFT **91**

18.1 Approximation 91

18.2 Interpolation 93

18.3 Fast Fourier Transform 96

18.4 Exercises 97

V Numerical differentiation and integration **98**

19 Estimating derivatives: Richardson extrapolation **99**

19.1 Numerical differentiation 99

19.2 Richardson extrapolation 100

19.3 Exercises 102

20 Numerical integration based on interpolation **104**

20.1 General framework 104

20.2 Trapezoidal rule 105

20.3 Simpson's rule 106

20.4 Exercises 107

21 Romberg integration **109**

21.1 Recursive trapezoidal rule 109

21.2 Romberg algorithm 110

21.3 Exercises 111

22 Adaptive quadrature **112**

22.1 Description of the method 112
22.2 The pseudocode 113
22.3 Exercises 115

23 Gaussian quadrature **116**
23.1 The main result 116
23.2 Examples 117
23.3 Error analysis 118
23.4 Exercises 119

VI Solving ordinary differential equations **120**

24 Difference equations **121**
24.1 Exercises 123

25 Euler's method for initial-value problems **124**
25.1 Existence and uniqueness of solutions 125
25.2 Euler's method 125
25.3 Exercises 127
25.4 Taylor series methods 127

26 Runge–Kutta methods **128**
26.1 Taylor series for $f(x, y)$ 128
26.2 Runge–Kutta method of order two 129
26.3 Runge–Kutta method of order four 130
26.4 Exercises 131

27 Multistep methods **132**

27.1 Description of the method 132

27.2 A catalog of multistep methods 133

27.3 Predictor–corrector method 135

27.4 Exercises 135

28 Systems of higher-order ODEs **136**

28.1 Reduction to first-order ordinary differential equations 136

28.2 Extensions of the methods 138

28.3 Exercises 139

29 Boundary value problems **140**

29.1 Shooting method 141

29.2 Finite-difference methods 142

29.3 Collocation method 144

29.4 Exercises 145

Part I

Preliminaries

Chapter 1

Introductory remarks

1.1 Taylor series and 'big-O' notations

Let us strart by recalling Taylor's theorem.

Theorem 1.1. If the function f possesses continuous derivatives of order $0, 1, \dots, n+1$ on a closed interval $I = [a, b]$, then for any $x, x_0 \in I$,

$$f(x) = \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k \; + \; E_{n+1},$$

where the **error term** , or **remainder**, E_{n+1} [which also depends on x and x_0] can be represented by

$$E_{n+1} = \begin{cases} \displaystyle\int_{x_0}^{x} \frac{f^{(n+1)}(t)}{n!}(x - t)^n dt, & \\ \displaystyle\frac{f^{(n+1)}(c)}{(n+1)!}(x - x_0)^{n+1}, & \text{for some } c \text{ between } x \text{ and } x_0. \end{cases}$$

Note that taking $n = 0$ together with the second form of the remainder yields the mean value theorem, itself implying Rolle's theorem. Let us write $x = x_0 + h$. Suppose for example that $\dfrac{f^{(n+1)}(c)}{(n+1)!}$ can be bounded by some constant K, then one has $|E_{n+1}| \leq K|h|^{n+1}$, which is abbreviated by the '**big-O**' notation

$$E_{n+1} = \mathcal{O}(h^{n+1}).$$

It means that, as h converges to zero, E_{n+1} converges to zero with (at least) essentially the same rapidity as $|h|^{n+1}$. In such circumstances, we have at hand a way to compute

(approximately) the value of $f(x)$: take the **Taylor polynomial** of degree n for f centered at x_0 and evaluate it at x. But beware that, when n grows, a **Taylor series** converges rapidly near the point of expansion but slowly (or not at all) at more remote points. For example, the Taylor series for $\ln(1+x)$ truncated to eight terms gives

$$\ln 2 \approx 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \frac{1}{6} + \frac{1}{7} - \frac{1}{8} = 0.63452...,$$

a rather poor approximation to $\ln 2 = 0.693147...$, while the Taylor series for $\ln\left(\frac{1+x}{1-x}\right)$ truncated to four terms gives

$$\ln 2 = 2\left[\frac{1}{3} + \frac{1}{3^3 \cdot 3} + \frac{1}{3^5 \cdot 5} + \frac{1}{3^7 \cdot 7}\right] = 0.69313...$$

As another example, consider the Taylor series of the exponential function centered at 0, that is

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}.$$

Even though the remainder, for a given x, is of the type $\mathcal{O}(1/n!)$, the determination of e^x via the Taylor series truncated to seven terms is quite bad for $x = 8$. However, when $x = 1/2$ this produces very good results, as compared e.g. to the approximation $e^{1/2} \approx \left(1 + \frac{1}{2n}\right)^n$, at least when **speed of convergence** is concerned. But is this the only criterion that matters? It seems that less operations are required to compute the latter...

1.2 Evaluating a polynomial

Given coefficients $a_0, \ldots, a_n$, we wonder how one should evaluate the polynomial $p(x) := a_n + a_{n-1}x + \cdots + a_0 x^n$ efficiently at a given point x. The first instinct is to calculate the terms $a_{n-i} \times \underbrace{x \times x \times \cdots \times x}_{i \text{ times}}$ for each $i \in [\![0, n]\!]$ and to sum them all. This requires n additions and $1 + 2 + \cdots + n = \frac{n(n+1)}{2}$ multiplications. Obviously, we have done too much work here since we have not used the result of the computaion of x^i to evaluate x^{i+1}. If we do, the calculation of the powers of the input number x only requires $n-1$ multiplications. Hence the whole process now requires n additions and $2n-1$ multiplications. This is much better, but can still be improved via **Horner's method**, also called **nested multiplication**. It is

based on a simple observation, namely

$$\begin{aligned} p(x) &= a_n + a_{n-1}x + \cdots + a_0 x^n \\ &= a_n + x(a_{n-1} + a_{n-2}x + \cdots + a_0 x^{n-1}) \\ &= \cdots = a_n + x(a_{n-1} + x(a_{n-2} + \cdots + x(a_1 + x a_0)\cdots)). \end{aligned}$$

This calls for the following piece of pseudocode:

$p \leftarrow a_0$
for $i = 1$ to n
do $p \leftarrow a_i + x * p$ **end do**

Here we notice that one addition and one multiplication are required at each step, hence the cost of the whole algorithm is n additions and n multiplications. This implies that the execution time is roughly divided by a factor 2.

1.3 Exercises

From the textbook: 8 p 14, 18 p 15.

1. What is the Taylor series for $\ln(1+x)$? for $\ln((1+x)/(1-x))$?

2. Verify the Taylor series

$$\ln(x+e) = 1 + \frac{x}{e} - \frac{x^2}{2e^2} + \cdots + \frac{(-1)^{k-1}}{k}\left(\frac{x}{e}\right)^k + \cdots$$

3. Rewrite the following polynomials in nested form and evaluate at $x = -1/2$:

   - $p(x) = 6x^3 - 2x^2 - 3x + 7$
   - $p(x) = 8x^5 - x^4 - 3x^3 + x^2 - 3x + 1$

4. Show how to evaluate the following polynomials efficiently:

   - $p(x) = x^{32}$
   - $p(x) = 3(x-1)^5 + 7(x-1)^9$
   - $p(x) = 6(x+2)^3 + 9(x+2)^7 + 3(x+2)^{15} - (x+2)^{31}$
   - $p(x) = x^{127} - x^{37} + 10x^{17} - 3x^7$

5. Using a computer algebra system (Maple, Mathematica,...), print 200 decimal digits of $\sqrt{10}$.

6. Express in mathematical notation without parentheses the final value of z in the following pseudocode:
   input $n, (b_i), z$
   $z \leftarrow b_n + 1$
   for $k = 1$ to $n - 2$
       do $z \leftarrow z * b_{n-k} + 1$ **end do**

7. How many multiplications occur in executing the following pseudocode?
   input $n, (a_{i,j}), (b_{i,j}), x$
   for $j = 1$ to n **do**
       for $i = 1$ to j **do**
           $x \leftarrow x + a_{i,j} * b_{i,j}$
       end do
   end do

8. For the pair (x_n, α_n), is it true that $x_n = \mathcal{O}(\alpha_n)$ as $n \to \infty$?

   - $x_n = 5n^2 + 9n^3 + 1$, $\alpha_n = n^2$
   - $x_n = \sqrt{n+3}$, $\alpha_n = 1/n$

Optional problems

1. Prove that the series of Exercise 2 converges on the interval $-e < x \leq e$.

2. Prove that if f possesses continous derivatives of order $0, 1, \ldots, n$ and if $f(x_0) = f(x_1) = \cdots = f(x_n)$ for $x_0 < x_1 < \cdots < x_n$, then $f^{(n)}(c) = 0$ for some $c \in (x_0, x_n)$.

From the textbook: 25 p 16.

Chapter 2

Programming suggestions

You may use any software you are comfortable with, however I will use pseudocode as the 'programming language' throughout the notes. This is only loosely defined, nevertheless, as a bridge between mathematics and computer programming, it serves our purpose well. The course does not focus on programming, but on understanding and testing the various numerical methods underlying scientific computing. Still, some words of advice can do no harm. The following suggestions are not intended to be complete and should be considered in context. I just want to highlight some consideration of efficiency, economy, readability, and roundoff errors.

2.1 Programming and coding advice

Use pseudocode before beginning the coding. It should contain sufficient detail so that the implementation is straightforward. It should also be easily read and understood by a person unfamiliar with the code.

Check and double check. It is common to write programs that may work on simple test but not on more complicated ones. Spend time checking the code before running it to avoid executing the program, showing the output, discovering an error, and repeating the process again and again.

Use test cases. Check and trace through the pseudocode using pencil-and-paper calculations. These sample tests can then be used as test cases on the computer.

Modularize code. Build a program in steps by writing and testing a series of self-contained subtasks (subprograms, procedures, or functions) as separate routines. It makes reading and debugging easier.

Generalize slightly. It is worth the extra effort to write the code to handle a slightly more general situation. For example, only a few additional statements are required to write a program with an arbitrary step size compared with a program in which the step size is fixed numerically. However, be careful not to introduce too much generality because it can make a simple task overly complicated.

Show intermediate results. Print out or display intermediate results and diagnostic messages to assist in debugging and understanding the program's operation. Unless impractical to do so, echo-print the input data.

Include warning messages. A robust program always warns the user of a situation that it is not designed to handle.

Use meaningful variable names. This is often helpful because they have greater mnemonic value than single-letter variables.

Declare all variables. All variables should be listed in type declarations in each program or program segment.

Include comments. Comments within a routine are helpful for recalling at some later time what the program does. It is recommended to include a preface to each program or program segment explaining the purpose, the input and output variable, etc. Provide a few comments between the major segments of the code; indent each block of code a consistent number of spaces; insert blank comment lines and blank spaces – it greatly improves the readability.

Use clean loops. Do not put unnecessary statements within loops: move expressions and variables outside a loop from inside a loop if they do not depend on it. Indenting loops, particularly for nested loops, can add to the readability of the code.

Use appropriate data structure.

Use built-in functions. In scientific programming languages, functions such as $\sin$, $\ln$, $\exp$, $\arcsin$ are available. Numeric functions such as integer, real, complex are also usually available for type conversion. One should use these as much as possible. Some of these intrinsic functions accept arguments of more that one type and return an output of the corresponding type. They are called generic functions, for they represent an entire family

of related functions.

Use program libraries. In preference to one you might write yourself, a routine from a program library should be used: such routines are usually state-of-the-art software, well tested and completely debugged.

Do not overoptimize. The primary concern is to write readable code that correctly computes the desired results. Tricks of the trade to make your code run faster and more efficiently are to be saved for later use in your programming career.

2.2 Case studies

Computing sums. When a long list of floating-point numbers is added, there will generally be less roundoff error if the numbers are added in order of increasing magnitude.

Mathematical constants. In many programming languages, the computer does not automatically know the values of common mathematical constants such as π or e. As it is easy to mistype a long sequence of digits, it is better to use simple calculations involving mathematical functions, e.g $\pi \leftarrow 4.0 \arctan(1.0)$. Problems will also occur if one uses a short approximation such as $\pi \leftarrow 3.14159$ on a computer with limited precision approximation and than later moves the code to another computer.

Exponents. The function x^y is generally computed as $\exp(y \ln x)$ if y is not an integer. Hence use preferentially the exponent 5 instead of 5.0; the exponents $1/2$ or 0.5 are not recommended since one can use the built-in function sqrt instead.

Avoid mixed mode. Mixed expressions are formulas in which variables and constants of different types appear together. Use the intrinsic type conversion functions. For example, $1/m$ should be coded as $1.0/\text{real}(m)$.

Precision. In the usual mode, i.e. single precision, one word of storage is used for each number. With double precision (also called extended precision), each number is allotted two or more words of storage. This is more time consuming, but is to be used when more accuracy is needed. In particular, on computers with limited precision, roundoff errors can quickly accumulate in long computations and reduce the accuracy to only three or four decimal places.

Memory fetches. When using loops, write the code so that fetches are made from adjacent words in memory. Suppose you want to store values in a two-dimensional array $(a_{i,j})$ in

which the elements of each column are stored in consecutive memory locations, you would then use i and j loops with the i loop as the innermost one to process elements down the columns.

When to avoid arrays. Although a sequence of values is computed, it is often possible to avoid arrays. For example, Newton's method (to be studied later) involves the recursion $x_{n+1} = x_n - f(x_n)/f'(x_n)$, but we are only interested in the final value of x_n, hence we can replace at each step an old x with the new numerical value $x - f(x)/f'(x)$. In other words, the pseudocode
for $n = 1$ to 10 **do** $x \leftarrow x - f(x)/f'(x)$ **end do**
will produce the result of ten iterations.

Limit iterations. To avoid endless cycling, limit the number of permissible steps by the use of a loop with a control variable. Hence, instead of

input x, d
$d \leftarrow f(x)/f'(x)$
while $|d| > \varepsilon$
    do $x \leftarrow x - d$, **output** x, $d \leftarrow f(x)/f'(x)$ **end do**

it is better to write

for $n = 1$ to n_{max}
    do $d \leftarrow f(x)/f'(x)$, $x \leftarrow x - d$, **output** n, x,
    if $|d| > \varepsilon$ **then** exit loop **end if**
    end do

Floating-point equality. The sequence of steps in a routine should not depend on whether two floating-point numbers are equal. Instead, reasonable tolerance should be permitted. Thus write
if $|x - y| < \varepsilon$ **then** ...
or even better
if $|x - y| < \varepsilon \max(|x|, |y|)$ **then** ...

Equal floating-point steps. In some situations, a variable t assumes a succession of values equally spaced a distance h apart along a real line. One way of coding this is
$t \leftarrow t_0$,   **for** $i = 1$ to n **do** $t \leftarrow t + h$ **end do**
Another way would be
for $i = 0$ to n **do** $t \leftarrow t_0 + \text{real}(i) * h$ **end do**
What is best depends on the situation at hand. In the first pseudocode, n additions occur,

each with a possible round of error. In the second, this is avoided at an added cost of n multiplications.

2.3 Exercises

1. The numbers $p_n = \int_0^1 x^n e^x dx$ satisfy the inequalities $p_1 > p_2 > p_3 > \cdots > 0$. Establish this fact. Using integration by parts, show that $p_{n+1} = e - (n+1)p_n$ and that $p_1 = 1$. Use the recurrence relation to generate the first twenty values of p_n on a computer and explain why the inequalities above are violated.

2. The following pieces of code are supposedly equivalent, with $\alpha \leftrightarrow \varepsilon$. Do they both produce endless cycles? Implement them on a computer [be sure to know how to abort a computation before starting].

   input α
   $\alpha \leftarrow 2$
   while $\alpha > 1$ **do** $\alpha \leftarrow (\alpha + 1)/2$ **end do**

   input ε
   $\varepsilon \leftarrow 1$
   while $\varepsilon > 0$ **do** $\varepsilon \leftarrow \varepsilon/2$ **end do**

Chapter 3

Non-decimal bases

3.1 Base-β numbers

Computers usually use base-2, base-8 or base-16 arithmetic, instead of the familiar base-10 arithmetic – which is not more natural than any other system, by the way, we are just more used to it. Our purpose here is to discuss the relations between the representations in different bases. First of all, observe for example that 3781.725 represents, in base 10, the number

$$3 \times 10^3 + 7 \times 10^2 + 8 \times 10 + 1 \quad + \quad 7 \times 10^{-1} + 2 \times 10^{-2} + 5 \times 10^{-3}.$$

More generally, in base β, we would write

$$(a_n a_{n-1} \cdots a_1 a_0 \,.\, b_1 b_2 b_3 \cdots)_\beta = \sum_{k=0}^{n} a_k \beta^k + \sum_{k=1}^{\infty} b_k \beta^{-k},$$

where the a_i's and b_i's are to be chosen among the **digits** $0, 1, \ldots, \beta - 1$. The first sum in this expansion is called the **integer part** and the second sum is called the **fractional part**. The separator is called the **radix point** – decimal point being reserved for base-10 numbers. The systems using base 2, base 8, and base 16 are called **binary**, **octal**, and **hexadecimal**, respectively. In the latter, the digits are $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f$.

3.2 Conversions

Let us convert $N = (101101001)_2$ to decimal form. We write

$$N = 1 \times 2^8 + 0 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2 + 1.$$

Then we perform the calculation in base-10 arithmetic, thus

$$N = 256 + 64 + 32 + 8 + 1 = 361.$$

To convert $(N)_\alpha$ to base β, the same procedure [expand in powers of α and carry out the arithmetic in base-β] can be used for any α and β, except that base-β arithmetic is tedious for $\beta \neq 10$. In fact, when $\alpha > \beta$, another technique is preferred.

Suppose for example that we want to find the binary representation of a number x, e.g. $x = (3781.725)_{10}$. Let us write

$$x = (a_n a_{n-1} \cdots a_1 a_0 \, . \, b_1 b_2 b_3 \cdots)_2.$$

The digit a_0 is easy to find: it is the remainder in the division by 2 of the integer part of x. Now, what about the digits a_1 and b_1? Well, we notice that

$$\begin{aligned} x/2 &= a_n a_{n-1} \cdots a_1 \, . \, a_0 b_1 b_2 b_3 \cdots \\ 2x &= a_n a_{n-1} \cdots a_1 a_0 b_1 \, . \, b_2 b_3 \cdots \,, \end{aligned}$$

so that a_1 and b_1 are obtained from the previous observation applied to $x/2$ and $2x$. The process can be continued. It yields an algorithm for the conversion of the integer part an another one for the conversion of the fractional part.

Integer part: Divide successively by 2 and store the remainder of the division as a digit until hitting a quotient equal to zero.

$3781 \to 1890;\ 1 \to 945;\ 01 \to 472;\ 101 \to 236;\ 0101 \to 118;\ 00101 \to 59;\ 000101 \to 29;\ 1000101 \to 14;\ 11000101 \to 7;\ 011000101 \to 3;\ 1011000101 \to 1;\ 11011000101 \to 0;\ 111011000101.$

In conclusion, we have obtained $(3781)_{10} = (111011000101)_2$.

Fractional part: Multiply by 2, store the integer part as a digit and start again with the new fractional part. Remark that the process might not stop.

$$\begin{aligned} 0.725 &\rightarrow 1;\ 0.45 \rightarrow 10;\ 0.9 \rightarrow 101;\ \mathbf{0.8} \\ &\rightarrow 1011;\ 0.6 \rightarrow 10111;\ 0.2 \rightarrow 101110;\ 0.4 \rightarrow 1011100;\ \mathbf{0.8} \\ &\rightarrow \text{ repeat the previous line.} \end{aligned}$$

In conclusion, we have obtained $(0.725)_{10} = (0.101\,1100\,1100\,1100\cdots)_2$.

3.3 Conversion $2 \leftrightarrow 8 \leftrightarrow 16$

The octal system is useful as an intermediate step when converting between decimal and binary systems by hand. The conversion $10 \leftrightarrow 8$ proceeds according to the previous principles, and the conversion $8 \leftrightarrow 2$ handles groups of three binary digits according to the table

Octal	0	1	2	3	4	5	6	7
Binary	000	001	010	011	100	101	110	111

Hence, the previous example gives

$$3781 \rightarrow 472;\ 5 \rightarrow 59;\ 05; \rightarrow 7; 305 \rightarrow 0;\ 7305,$$

so that $(3781)_{10} = (7305)_8 = (111\,011\,000\,101)_2$.

Most computers use the binary system for their internal representation. On computers whose word lengths are multiples of four, the hexadecimal system is used. Conversion between these two systems is also very simple, in view of the table

1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

3.4 Exercises

1. Convert $(110111001.101011101)_2$ and $(1001110101.01101)_2$ to hexadecimal, to octal, and then to decimal.

2. Convert the following numbers and then convert the results back to the original bases:

(a) $(47)_{10} = (\quad)_8 = (\quad)_2$

(b) $(0.782)_{10} = (\quad)_8 = (\quad)_2$

(c) $(110011.1110101101101)_2 = (\quad)_8 = (\quad)_{10}$

(d) $(361.4)_8 = (\quad)_2 = (\quad)_{10}$

3. Prove that a number has a finite binary representation if it has the form $\pm m/2^n$, where m and n are non-negative integers. Prove that any number having a finite binary representation also has a finite decimal representation.

4. Find the first twelve digits in the octal representation of $e\pi/4$. Use built-in mathematical functions to obtain the values of e and π.

5. With the help of a computer, establish that the number $e^{\pi\sqrt{163}}$ is very close to an 18-digits decimal integer and convert this integer to base 16. How many times is this integer dividable by 2?

Optional problems

1. Convert [by hand] the decimal number 4225.324 to base 7.

2. Write and test a routine for converting base-3 integers to binary integers.

Chapter 4

Computer arithmetic

4.1 Floating-point representation

Consider a computer that stores numbers in 64-bit words. Then a **single-precision** number is represented by three different parts: first, 1 bit is allotted to the sign, then 11 bits are allotted to an exponent c, called the **characteristic**, and finally 52 bits are allotted to a fractional part f, called the **mantissa**. More precisely, these numbers have the form

$$(-1)^s \times 2^{c-1023} \times (1+f).$$

Since c is represented by 11 binary digits, it belongs to the integer interval $[\![0, 2^{11}-1]\!] = [\![0, 2047]\!]$, and the range for the exponent $c-1023$ is $[\![-1023, 1024]\!]$. As for f, it lies in the interval $[0, 1-(1/2)^{52}]$. Therefore, the smallest and largest positive **machine numbers** are $2^{-1022} \times (1+0)$ and $2^{1023} \times (2-(1/2)^{52})$. Outside this range, we say that an **underflow** or an **overflow** has occured. The latter causes the computation to stop, the former produces zero [which is represented by $c=0$, $f=0$ and $s=0$ or $s=1$].

4.2 Computer errors

If a number has the form $x = (-1)^s \times 2^{c-1023} \times 1.d_1 d_2 \cdots d_{52} d_{53} \cdots$, with $c \in [\![-1023, 1024]\!]$, the process of replacing it by the number $x_c := (-1)^s \times 2^{c-1023} \times 1.d_1 \cdots d_{52}$ is called **chopping**, while the process of replacing it by the nearest machine number x_r is called **rounding**. The error involved is called the **roundoff error**. For the chopping process, the **relative**

error – not to be confused with the **absolute error** $|x - x_c|$ – satisfies

$$\frac{|x - x_c|}{|x|} = \frac{2^{c-1023} \times 0.0 \cdots 0 d_{53} d_{54} \cdots}{2^{c-1023} \times 1.d_1 d_2 \cdots} = 2^{-52} \times \frac{0.d_{53} d_{54} \cdots}{1.d_1 d_2 \cdots} \leq 2^{-52} \times \frac{1}{1} = 2^{-52}.$$

The number $\varepsilon := 2^{-52}$ is called the **unit roundoff error**. For the rounding process, we would obtain the better bound on the relative error

$$\frac{|x - x_r|}{|x|} \leq \frac{\varepsilon}{2}.$$

The value of the unit roundoff error, also known as **machine epsilon**, varies with the computer [it depends on the word length, the arithmetic base used, the rounding method employed]. As the smallest positive number ε such that $1 + \varepsilon \neq 1$ in the machine, it can be computed according to the following pseudocode:

input s, t
$s \leftarrow 1., t \leftarrow 1. + s$
while $t > 1$ **do** $s \leftarrow s/2., t \leftarrow 1 + s$ **end do**
output $2s$

Even though it can be sensible to assume that the relative error in any single basic arithmetic operation is bounded by the machine epsilon, the accumulation of these errors can have some significant effect. Consider the sequence (p_n) defined by $p_1 = 1$ and $p_{n+1} = e - (n+1)p_n$ [cf exercise 2.3.1], and let $(\widetilde{p}_n)$ be the *computed* sequence. The error $\delta_n := |p_n - \widetilde{p}_n|$ obeys the rule $\delta_{n+1} \approx (n+1)\delta_n$ [of course, this is not an equality], so that $\delta_n \approx n!\, \delta_1$ blows out of proportion. When small errors made at some stage are magnified in subsequent stages, like here, we say that a numerical process is **unstable**.

4.3 Loss of significance

A large relative error can occur as a result of a single operation, one of the principal causes being the subtraction of nearly equal numbers. As a straightforward example, suppose we subtract, on a hypothetical 5-decimal-digit computer, the numbers $x = 1.234512345$ and $y = 1.2344$. The difference should be $x - y = 0.000112345$, but the computer stores the value $(x - y)_s = 0.0001$. Almost all **significant digits** have been lost, and the relative error is large, precisely

$$\frac{|(x - y) - (x - y)_s|}{|x - y|} = \frac{1.2345 \times 10^{-5}}{1.12345 \times 10^{-4}} \approx 11\%.$$

Contrary to the roundoff errors which are inevitable, these sort of errors can be kept under control and the programmer must be alert to such situations. The simplest remedy is to carry out part of a computation in double-precision arithmetic, but this is costly and might not even help. A slight change in the formulas is often the answer. The assignment $y \leftarrow \sqrt{1+x^2}-1$ involves loss of significance for small x, but the difficulty can be avoided by writing $y \leftarrow \dfrac{x^2}{\sqrt{1+x^2}+1}$ instead [apply this to the solutions of a quadratic equation]. Likewise, the assignment $y \leftarrow \cos^2 x - \sin^2 x$ involves loss of significance near $\pi/4$. The assignment $y \leftarrow \cos(2x)$ should be used instead.

For completeness, we give a quantitative answer to the question: 'exactly how many significant binary bits are lost in the subtraction $x-y$ when x is close to y?'.

Theorem 4.1 (Loss of precision)**.** If two binary machine numbers x and y, with $x > y > 0$, satisfy $2^{-q} \le 1 - y/x \le 2^{-p}$ for some positive integers p and q, then at most q and at least p significant binary digits are lost in the subtraction $x-y$.

4.4 Exercises

From the textbook: 1.h p 26, 4 p 26, 15.a.b. p 27, 24 p 28.

1. There will be subtractive cancellation in computing $1+\cos x$ for some values of x. What are these values and how can the loss of precision be averted?
2. Show by an example that in computer arithmetic $(x \times y) \times z$ may differ from $x \times (y \times z)$.
3. Write and execute a program to compute $5 - \sqrt{25+x^2}$ and $x^2/(5+\sqrt{25+x^2})$ for x from 1 downto 0 with steps 0.01. Which results are reliable?

Optional problems

From the textbook: 25 p 29.

1. The inverse hyperbolic sine is given by $f(x) = \ln(x+\sqrt{x^2+1})$. Show how to avoid loss of significance in computing $f(x)$ when x is negative. Hint: exploit a relation between $f(-x)$ and $f(x)$.

Chapter 5

Linear algebra review

5.1 All you need to know

For a linear map $f : V \longrightarrow W$, the **kernel** and the **image** of f are defined by

$$\begin{aligned} \mathrm{Ker} f &= \{x \in V : \ f(x) = 0\}, && \text{this is a linear subspace of } V, \\ \mathrm{Im} f &= \{f(x), \ x \in V\}, && \text{this is a linear subspace of } W. \end{aligned}$$

From the equivalences [f injective] $\iff$ [$\mathrm{Ker} f = \{0\}$] and [f surjective] $\iff$ [$\mathrm{Im} f = W$], one derives

$$[f \text{ invertible}] \iff [\mathrm{Ker} f = \{0\} \text{ and } \mathrm{Im} f = W].$$

Using the **rank-nullity** theorem, reading

$$\underbrace{\dim \mathrm{Im} f}_{\text{rank}} + \underbrace{\dim \mathrm{Ker} f}_{\text{nullity}} = \dim V,$$

one obtain the equivalences

$$[f \text{ invertible}] \iff [\mathrm{Ker} f = \{0\} \text{ and } \dim V = \dim W] \iff [\mathrm{Im} f = W \text{ and } \dim V = \dim W].$$

An $m \times n$ matrix M defines the linear map $x \in \mathbb{R}^n \longrightarrow Ax \in \mathbb{R}^m$. Note that an $n \times n$ matrix M has at most one left (or right) inverse. Consequently, if a square matrix M admits a left (or right) inverse, it is a both-sided inverse. In other words, one has $AM = I \Rightarrow MA = I$. The **transpose** of a matrix M is the matrix $M^\top$ whose entries are $(M^\top)_{i,j} = M_{j,i}$. A square matrix M is called **symmetric** if $M^\top = M$ and **skew-symmetric** if $M^\top = -M$. An

orthogonal matrix is a square matrix P satisfying $PP^\top = I$ $(= P^\top P)$. In fact, a necessary and sufficient condition for a matrix to be orthogonal is that its columns (or rows) form an orthonormal system. An orthogonal matrix is also characterized by the fact that it preserves the norm [i.e. $\|Px\| = \|x\|$ for all x] – or the inner product [i.e. $\langle Px, Py\rangle = \langle x, y\rangle$ for all x and y].

A number λ is an **eigenvalue** of a $n \times n$ matrix M if $Mx = \lambda x$ for some nonzero vector x, called **eigenvector** associated to λ. The eigenvalues of A are the roots of the characteristic polynomial $P_M(\lambda) := \det(M - \lambda I)$, hence M admits at most n eigenvalues. Note that A is invertible (or **nonsingular**) if and only if 0 is not an eigenvalue of M, i.e. if $\det(M) \neq 0$. The Cayley–Hamilton theorem states that $P_M(M) = 0$. The $n \times n$ matrix M is said to be **diagonalizable** if there exists a system of eigenvectors of M which form a basis of $\mathbb{R}^n$, or equivalently if there exist a diagonal matrice D and an invertible matrix P such that $M = PDP^{-1}$. Note that the the columns of P are the eigenvectors of M and that the diagonal elements of D are its eigenvalues. Since the trace and the determinant of a matrix are invariant under the transformation $M \mapsto P^{-1}MP$, we see that $\mathrm{tr}(M) =$(sum of eigenvalues) and $\det(M) =$(product of eigenvalues). Remember that a (real) symmetric matrix is always diagonalizable via a orthogonal matrix, i.e. there exist a diagonal matrice D and an orthogonal matrix P such that $M = PDP^\top$; in other words, one can find an orthonormal basis of eigenvectors of M. A symmetric matrix M is called **positive definite** if $x^\top Mx = \langle x, Mx\rangle > 0$ for all nonzero vector x. Note that a symmetric matrix is positive definite if and only if all its eigenvalues are positive.

5.2 Exercises

From the textbook: read sections 6.3, 6.4 and 7.2; 5 p 380, 6 p 386, 11 p 386, 1.d-e-f p 435, 13 p 436.

Optional problems

From the textbook: 6 p 380, 7 p 380.

1. Prove the statement that a square matrix M has at most one left inverse. Deduce the statement that a left inverse is automatically both-sided. [Hint: if $AM = I$, multiply M on the left by $A + I - MA$.]

2. Establish that a symmetric matrix is positive definite iff all its eigenvalues are positive.

Part II

Solving linear equations

Chapter 6

Gaussian elimination

Many problems in economics, engineering and science involve differential equations which cannot be solved explicitely. Their numerical solution is obtained by discretizing the differential equation – see part VI – leading to a system of linear equations. One may already know that most of the time Gaussian elimination allows to solve this system. But the higher the required accuracy, the finer the discretization, and consequently the larger the system of linear equations. Gaussian elimination is not very effective in this case. One of the goals of this part is to devise better methods for solving linear systems.

6.1 Linear systems

A system of m linear equations in n unknowns $x_1, \ldots, x_n$ is written in the form

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + a_{1,3}x_3 + \cdots + a_{1,n}x_n &= b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + a_{2,3}x_3 + \cdots + a_{2,n}x_n &= b_2 \\ \vdots \quad &\quad \vdots \\ a_{m,1}x_1 + a_{m,2}x_2 + a_{m,3}x_3 + \cdots + a_{m,n}x_n &= b_m. \end{aligned}$$

In matrix notations, it can be written as [identify the different components]

$$Ax = b.$$

Geometrically, we look at the intersection of m hyperplanes of $\mathbb{R}^n$. Hence, if $n = m = 3$, we want to determine the intersection of 3 planes of the natural space $\mathbb{R}^3$. Although not totally necessary, we assume from now on that $m = n$, i.e. that A is a square matrix.

Simple cases

- **diagonal structure**: the system of equations has the form

$$\begin{bmatrix} a_{1,1} & 0 & \cdots & 0 \\ 0 & a_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

It has the solution $x = [b_1/a_{1,1}, b_2/a_{2,2}, \ldots, b_n/a_{n,n}]^\top$, so long as all the a_i's are nonzero. If $a_{i,i} = 0$ for some i, then either x_i can be chosen to be any number if $b_i = 0$, or the system is unsolvable.

- **lower triangular structure**: with $a_{i,j} = 0$ for $i < j$, the system reduces to

$$\begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ a_{2,1} & a_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

Assume that $a_{i,i} \neq 0$ for all i. The system can be solved by **forward substitutition**, in other words, get the value of x_1 from the first equation $a_{1,1}x_1 = b_1$, next substitute it in the second equation $a_{2,1}x_1 + a_{2,2}x_2 = b_2$ to find the value of x_2, next subsitute x_1 and x_2 in the third equation to find x_3, and so on. Based on the steps

$$x_i = (b_i - a_{i,1}x_1 - a_{i,2}x_2 - \cdots - a_{i,i-1}x_{i-1})/a_{i,i},$$

a formal algorithm will look like

input $n, (a_{i,j}), (b_i)$
for $i = 1$ to n **do**
  for $j = 1$ to $i - 1$ **do** $b_i \leftarrow b_i - a_{i,j}x_j$ **end do**
$x_i \leftarrow b_i/a_{i,i}$
end do
output (x_i)

- **upper triangular structure**: with $a_{i,j} = 0$ for $i > j$, the system reduces to

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ 0 & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

The system can be solved by **backward substitution**, i.e. get x_n first, then x_{n-1}, and so on until x_1. The pseudocode of the corresponding algorithm can be easily written.

6.2 Computational cost

We often make the sensible assumption that the execution time of an algorithm is roughly proportional to the number of operations being performed. We also usually neglect the number of additions/subtractions with respect to the number of multiplications/divisions, because the latter require more execution time than the former and because these numbers are often comparable in practice. Of course, if our algorithm does not contain any multiplication/division, this is no longer legitimate.
Looking back at the forward substitution algorithm, we observe that for a given i the **operation count** is i mult./div., hence the operation count for the whole algorithm is

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}.$$

It makes no sense to use this accurate expression, firstly because $n/2$ is small compared to $n^2/2$ when n is large and secondly because our starting assumptions are themselves not accurate. Instead, it is more appropriate to consider only $n^2/2$ and in fact only n^2 since the execution time is anyway obtained via a proportionality factor. We then use the 'big-O' notation to state that

the complexity of the forward/backward substitution algorithm is $\mathcal{O}(n^2)$.

For example, if a system of 50 equations in 50 unknowns is solved in 0.1 seconds on our computer, we can estimate the time needed to solve a 5000×5000 system by scaling by the factor $100^2 = 10^4$, so that the computation would take us $1000\,\mathrm{sec} \approx 15\,\mathrm{min}$.

6.3 Gaussian elimination

6.3.1 An example, to start with

Let us refresh our memory of this familiar concept with a particular case. We wish to solve the system

$$\begin{array}{rcrcrcl} 2x & + & y & - & z & = & 2 \\ 6x & + & 2y & - & 2z & = & 8 \\ 4x & + & 6y & - & 3z & = & 5. \end{array}$$

The strategy is to combine the second equation and the first one to form a new second equation where the unknown x does not appear, and likewise to form a new third equation where the unknown x does not appear either. Then we combine the two new equations into a new third one, getting rid of the unknown y as well. We are then able to solve by backward substitution, first determining z, then y and finally x. Here it how it unfolds, in so-called **tableau form**,

$$\left[\begin{array}{ccc|c} 2 & 1 & -1 & 2 \\ 6 & 2 & -2 & 8 \\ 4 & 6 & -3 & 5 \end{array}\right] \underset{R_3 \leftarrow R_3 - 2R_1}{\overset{R_2 \leftarrow R_2 - 3R_1}{\longrightarrow}} \left[\begin{array}{ccc|c} 2 & 1 & -1 & 2 \\ 0 & -1 & 1 & 2 \\ 0 & 4 & -1 & 1 \end{array}\right] \overset{R_3 \leftarrow R_3 + 4R_2}{\longrightarrow} \left[\begin{array}{ccc|c} 2 & 1 & -1 & 2 \\ 0 & -1 & 1 & 2 \\ 0 & 0 & 3 & 9 \end{array}\right].$$

The solution $z = 3$, $y = 1$, $x = 2$ follows.

Note that, when performing Gaussian elimination, we merely use some **elementary row operations** which leave the solutions of the system unchanged. Namely, these are

Op.1: multiply a row by a nonzero constant,

Op.2: add/subtract a row to another,

Op.3: swap one row for another.

6.3.2 The algorithm

Suppose we have already performed $j-1$ steps of the algorithm. The current situation is described by the tableau form

$$\left[\begin{array}{ccccccccc} a_{1,1}^{(j)} & a_{1,2}^{(j)} & \cdots & a_{1,j}^{(j)} & a_{1,j+1}^{(j)} & \cdots & a_{1,n}^{(j)} & | & b_1^{(j)} \\ 0 & a_{2,2}^{(j)} & \cdots & a_{2,j}^{(j)} & a_{2,j+1}^{(j)} & \cdots & a_{2,n}^{(j)} & | & b_2^{(j)} \\ 0 & 0 & \ddots & \vdots & \vdots & \cdots & \vdots & & \\ \vdots & \vdots & 0 & \boxed{a_{j,j}^{(j)}} & a_{j,j+1}^{(j)} & \cdots & a_{j,n}^{(j)} & | & b_j^{(j)} \\ 0 & \vdots & 0 & a_{j+1,j}^{(j)} & a_{j+1,j+1}^{(j)} & \cdots & a_{j+1,n}^{(j)} & | & b_{j+1}^{(j)} \\ 0 & \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & | & \vdots \\ 0 & \vdots & 0 & a_{n,j}^{(j)} & a_{n,j+1}^{(j)} & \cdots & a_{n,n}^{(j)} & | & b_n^{(j)} \end{array}\right],$$

and we perform the operations $R_{j+1} \leftarrow R_{j+1} - \frac{a_{j+1,j}^{(j)}}{a_{j,j}^{(j)}} R_j,\ \ldots,\ R_n \leftarrow R_n - \frac{a_{n,j}^{(j)}}{a_{j,j}^{(j)}} R_j$ to 'eliminate' $a_{j+1,j}^{(j)}, \ldots, a_{n,j}^{(j)}$. Remark that the numbers $a_{j,j}^{(j)}$, called **pivots**, must be nonzero otherwise the algorithm should halt. If $a_{j,j}^{(j)} = 0$ the remedy is to exchange the rows of the tableau to obtain, if possible, a nonzero pivot – see Op.3. This technique is called **partial pivoting**. In fact, one should make the exchange so that the element of largest magnitude in the lower part of the j-th column ends up in the pivotal position. This also reduces the chance of creating very large number, which might lead to *ill conditioning* and accumulation of *roundoff error*. We do not bother about partial pivoting in the following version of the Gaussian elimination algorithm. Here, the result of the algorithm is an upper triangular system of equations. The associated backward substitution subprogram is omitted.

input $n, (a_{i,j}), (b_i)$
for $j = 1$ to $n-1$
if $a_{j,j} = 0$ **then error** 'pivot j equals zero'
else for $i = j+1$ to n
  do $m \leftarrow a_{i,j}/a_{j,j}$, $b_i \leftarrow b_i - m * b_j$, $a_{j,j} \leftarrow 0$,
  for $k = j+1$ to n **do** $a_{i,k} \leftarrow a_{i,k} - m * a_{j,k}$ **end do**
  end do
end if
output $(a_{i,j})$

Remark. This portion of code has evolved from the primitive form

input $n, (a_{i,j}), (b_i)$
for $j = 1$ to $n - 1$
do eliminate column j
output $(a_{i,j})$

to the final one via the intermediary version

input $n, (a_{i,j}), (b_i)$
for $j = 1$ to $n - 1$
    for $i = j + 1$ to n **do** eliminate $a_{i,j}$ **end do**
end do
output $(a_{i,j})$

It is often a good idea to break up a code into several smaller parts – it facilitates its reading and the possible mistakes will be easier to identify.

6.3.3 Operation count

Examining the previous algorithm, we count $n - j + 2$ mult./div. for fixed j and i. Then, for a fixed j, the number of mult./div. is $(n - j)(n - j + 2) = (n - j + 1)^2 - 1$. Finally the total number of mult./div. is

$$\sum_{j=1}^{n-1} \left((n-j+1)^2 - 1\right) = \sum_{\ell=2}^{n} \ell^2 \; - (n-1) = \frac{n(n+1)(2n+1)}{6} - 1 - (n-1) = \frac{n^3}{3} + \cdots .$$

We observe that the elimination step is relatively expensive compared to the backward substitution step. We will remember that

the complexity of the Gaussian elimination algorithm is $\mathcal{O}(n^3)$.

Remark. When the $n \times n$ matrix A is invertible, the solution of the linear system $Ax = b$ obeys Cramer's rule, that is

$$x_i = \det \begin{bmatrix} a_{1,1} & \dots & a_{1,i-1} & b_1 & a_{1,i+1} & \dots & a_{1,n} \\ a_{2,1} & \dots & a_{2,i-1} & b_2 & a_{2,i+1} & \dots & a_{2,n} \\ \vdots & \dots & \vdots & \dots & \vdots & \dots & \vdots \\ a_{n,1} & \dots & a_{n,i-1} & b_n & a_{n,i+1} & \dots & a_{n,n} \end{bmatrix} / \det A, \qquad i \in [\![1, n]\!].$$

Unfortunately, the number of operations required to compute the two determinants grows like $\mathcal{O}(n!)$, therefore this method is totally unpractical.

6.4 Exercises

From the textbook: 4 p 356, 10 p 358, 12 p 358 [write the algorithm with the software of your choice and test it on the matrices of 5.c and 5.d p 357].

Optional problems

From the textbook: 19 p 359

Chapter 7

The LU and Cholesky factorizations

7.1 LU-factorization

7.1.1 Definition

We say that an $n \times n$ matrix A admits an LU**-decomposition** if

$$A = LU, \quad \text{where } L \text{ and } U \text{ are } n \times n \text{ lower and upper triangular matrices, respectively.}$$

This is of interest for several reasons, among which we can mention

- calculation of a determinant [$\det A = \det L \cdot \det U = \prod_{i=1}^{n} l_{i,i} \cdot \prod_{i=1}^{n} u_{i,i}$] and test for nonsingularity [A invertible iff $l_{i,i} \neq 0$ and $u_{i,i} \neq 0$ for all $i \in [\![1, n]\!]$].
- determination of an inverse [$A^{-1} = U^{-1}L^{-1}$].
- solution of a linear system:

$$[Ax = b] \iff [Ly = b, \quad y = Ux].$$

  Both latter systems are triangular and can easily be solved. The advantage of this decomposition, compared to the Gaussian elimination, is that one does not consider the right hand side b until the factorization is complete. Hence, when there are many right hand sides, it is unnecessary to repeat a $\mathcal{O}(n^3)$ procedure at each time, only $\mathcal{O}(n^2)$ operations will be required for each new b.

An LU-decomposition is never unique. This is not surprising, since $A = LU$ induces n^2 scalar equations [one for each entry of A] linking $n^2 + n$ unknowns [the entries of L and U]. As a matter of fact, if $D = \mathrm{Diag}[d_{1,1}, \ldots, d_{n,n}]$ is a diagonal matrix such that each diagonal element $d_{i,i}$ is nonzero and if $A = LU$, then there also holds

$$A = (LD)(D^{-1}U), \qquad \text{where } LD \text{ and } D^{-1}U \text{ are lower and upper triangular, respectively.}$$

Since the diagonal elements of LD and $D^{-1}U$ are $l_{i,i}d_{i,i}$ and $u_{i,i}/d_{i,i}$, $i \in [\![1, n]\!]$, we can most of the time make an arbitrary choice for the diagonal elements of L or U. The most common one imposes L to be **unit** lower triangular (i.e. $l_{i,i} = 1$, $i \in [\![1, n]\!]$), we refer to this as **Doolittle's factorization**. If U was unit upper triangular, we would talk about **Crout's factorization**.

Note that if A is nonsingular, then it admits at most one Doolittle's/Crout's factorization. Indeed, suppose e.g. that $A = LU = \widetilde{L}\widetilde{U}$, where $L, \widetilde{L}$ are unit lower triangular and $U, \widetilde{U}$ are upper triangular. Since A is nonsingular, one has $\det A = \det L \det U = \det U \neq 0$. Hence U is nonsingular. Remark also that $\widetilde{L}$ is nonsingular. Then, one can write $\widetilde{L}^{-1}L = \widetilde{U}U^{-1}$. This means that the upper matrix $\widetilde{U}U^{-1}$ is also unit lower triangular, therefore $\widetilde{U}U^{-1} = I$. It follows that $U = \widetilde{U}$ and $L = \widetilde{L}$.

7.1.2 The calculation of the LU-factorization

Suppose that A admits a Doolittle's factorization. Denote the columns of L by $\ell_1, \ldots, \ell_n$ and the rows of U by $u_1^\top, \ldots, u_n^\top$. Thus,

$$A = LU = \begin{bmatrix}\ell_1 & \cdots & \ell_n\end{bmatrix}\begin{bmatrix}u_1^\top \\ \vdots \\ u_n^\top\end{bmatrix} = \sum_{i=1}^{n} \ell_i u_i^\top.$$

Observe the specific form of each of the rank-one matrices $\ell_i u_i^\top$, which is

$$\begin{bmatrix}0\\ \vdots\\ 0\\ 1\\ \ell_{i+1,i}\\ \vdots\\ \ell_{n,i}\end{bmatrix}\begin{bmatrix}0 & \cdots & 0 & u_{i,i} & u_{i,i+1} & \cdots & u_{i,n}\end{bmatrix} = \left[\begin{array}{ccc|c|ccc} \ddots & & & \vdots & \ddots & & \\ & 0 & & 0 & & 0 & \\ & & \ddots & \vdots & & & \ddots \\ \hline \cdots & 0 & \cdots & u_{i,i} & u_{i,i+1} & \cdots & u_{i,n} \\ \hline \ddots & & & u_{i,i}\ell_{i+1,i} & \times & & \times \\ & 0 & & \vdots & \times & \ddots & \times \\ & & \ddots & u_{i,i}\ell_{n,i} & \times & & \times \end{array}\right].$$

This allows to simply read $u_1^\top =$ first row of A and $\ell_1 =$ first column of $A/u_{1,1}$. Then we consider $A_1 := A - \ell_1 u_1^\top$, and we read $u_2^\top =$ second row of A_1, $\ell_2 =$ second column of $A_1/u_{2,2}$, and so on. This leads to the following LU-algorithm:

Set $A_0 := A$; for $k = 1$ to n, set

$$u_k^\top = k\text{-th row of } A_{k-1}, \qquad \ell_k = k\text{-th column of } A_{k-1} \text{ scaled so that } \ell_{k,k} = 1,$$

and calculate $A_k = A_{k-1} - \ell_k u_k^\top$ before incrementing k.

Note that a necessary condition for the realization of this algorithm is that $u_{i,i} \neq 0$ for all $i \in [\![1, n]\!]$.

Let us now recall that the k-th leading **principal minor** of a matrix A is

$$A_k := \begin{bmatrix} a_{1,1} & \cdots & a_{1,k} \\ \vdots & \ddots & \vdots \\ a_{k,1} & \cdots & a_{k,k} \end{bmatrix}.$$

One can see that if a matrix A admits the LU-decomposition $A = LU$, then the matrix A_k admits the decomposition $A_k = L_k U_k$ by writing

$$A =: \left[\begin{array}{c|c} A_k & B \\ \hline C & D \end{array}\right] = \underbrace{\left[\begin{array}{c|c} L_k & 0 \\ \hline X & L' \end{array}\right]}_{:=L} \underbrace{\left[\begin{array}{c|c} U_k & Y \\ \hline 0 & U' \end{array}\right]}_{:=U} = \left[\begin{array}{c|c} L_k U_k & L_k Y \\ \hline X U_k & XY + L'U' \end{array}\right].$$

In particular, if $A = LU$ is a Doolittle's factorization, then

$$\det A_k = \det U_k = \prod_{i=1}^{k} u_{i,i}, \qquad k \in [\![1, n]\!],$$

and one obtains the diagonal entries of U as

$$u_{1,1} = \det A_1, \qquad u_{k,k} = \frac{\det A_k}{\det A_{k-1}}, \quad k \geq 2.$$

The necessary condition for the realization of the LU-algorithm can therefore be translated in terms of the matrix A as $\det A_k \neq 0$, $k \in [\![1, n]\!]$. It turns out that this condition is somewhat sufficient for a Doolittle's factorization to exist.

Theorem 7.1. For an $n \times n$ nonsigular matrix A, there holds the equivalence

$$\big[\det A_k \neq 0,\ k \in [\![1, n-1]\!]\big] \iff \big[A \text{ admits a (unique) Doolittle's factorization}\big].$$

Proof. We proceed by induction on n for the direct implication. Clearly the implication is true for $n = 1$. Now let A be an $n \times n$ invertible matrix such that $\det A_k \neq 0$ for all $k \in [\![1, n-1]\!]$. We can apply the induction hypothesis to A_{n-1} and write $A_{n-1} = \widetilde{L}\widetilde{U}$, where the unit lower triangular matrix $\widetilde{L}$ and the upper triangular matrix $\widetilde{U}$ are invertible. Then, with $A =: \left[\begin{array}{c|c} A_{n-1} & B \\ \hline C & d \end{array}\right]$, $X := C\widetilde{U}^{-1}$, $Y := \widetilde{L}^{-1}B$, and $\delta := d - XY$, one obtains

$$\left[\begin{array}{c|c} \widetilde{L} & 0 \\ \hline X & 1 \end{array}\right] \left[\begin{array}{c|c} \widetilde{U} & Y \\ \hline 0 & \delta \end{array}\right] = \left[\begin{array}{c|c} \widetilde{L}\widetilde{U} & \widetilde{L}Y \\ \hline X\widetilde{U} & XY + \delta \end{array}\right] = \left[\begin{array}{c|c} A_{n-1} & B \\ \hline C & d \end{array}\right],$$

which is a Doolittle's factorization for A.
For the converse implication, suppose that a nonsingular matrix admits the Doolittle's factorization $A = LU$. Then $\det A = \prod_{i=1}^{n} u_{i,i}$ is nonzero and it follows that $\det A_k = \prod_{i=1}^{k} u_{i,i}$ is nonzero, too. $\square$

7.2 Gaussian elimination revisited

The elementary row operations on a matrix A are obtained by multiplying A on the left by certain matrices, namely

$$\textbf{Op.1:} \quad \begin{bmatrix} \ddots & & {\scriptstyle i} & & \\ & 1 & \vdots & & \\ {\scriptstyle i} \quad \cdots & & \lambda & & \\ & & & 1 & \\ & & & & \ddots \end{bmatrix} \times A \quad \text{replaces the row } a_i^\top \text{ of } A \text{ by } \lambda a_i^\top,$$

$$\textbf{Op.2:} \quad \begin{bmatrix} \ddots & & & \\ {\scriptstyle i}\cdots & 1 & & \\ & \vdots & \ddots & \\ {\scriptstyle j}\cdots & \pm 1 & \cdots & 1 \\ & & & & \ddots \end{bmatrix} \times A \quad \text{replaces the row } a_j^\top \text{ of } A \text{ by } a_j^\top \pm a_i^\top,$$

$$\textbf{Op.3:} \quad \begin{bmatrix} 1 & & & & \\ {\scriptstyle i}\cdots & 0 & \cdots & 1 & \\ & \vdots & \ddots & \vdots & \\ {\scriptstyle j}\cdots & 1 & \cdots & 0 & \\ & & & & 1 \end{bmatrix} \times A \quad \text{swaps the rows } a_i^\top \text{ and } a_j^\top \text{ of } A.$$

Now suppose that the Gaussian elimination algorithm can be carried out without encountering any zero pivot. The algorithm, performed by means of row operations of the type $a_j^\top \leftarrow a_j^\top - \lambda_{j,i} a_i^\top$, $i < j$, produces an upper triangular matrix. But these row operations are obtained by left-multiplication with the unit lower triangular matrices

$$L_{i,j} = \begin{bmatrix} \ddots & & & & \\ i \cdots & 1 & & & \\ & \vdots & \ddots & & \\ j \cdots & -\lambda_{j,i} & \cdots & 1 & \\ & & & & \ddots \end{bmatrix}.$$

Hence, the algorithm can be expressed as

$$L_{n-1,n} \cdots L_{2,n} \cdots L_{2,3}\, L_{1,n} \cdots L_{1,2}\, A = U.$$

Since the inverse matrices

$$L_{i,j}^{-1} = \begin{bmatrix} \ddots & & & & \\ i \cdots & 1 & & & \\ & \vdots & \ddots & & \\ j \cdots & \lambda_{j,i} & \cdots & 1 & \\ & & & & \ddots \end{bmatrix}$$

are unit lower triangular, we get

$$A = LU, \qquad \text{where } L := L_{1,2}^{-1} \cdots L_{n-1,n}^{-1} \text{ is constructed throughout the algorithm.}$$

This is particularly transparent on an example, e.g. the one of section 6.3.1. Instead of inserting zeros in the lower part of the matrix, we store the coefficients $\lambda_{j,i}$.

$$\begin{bmatrix} 2 & 1 & -1 \\ 6 & 2 & -2 \\ 4 & 6 & -3 \end{bmatrix} \longrightarrow \begin{bmatrix} 2 & 1 & -1 \\ \hline 3 & -1 & 1 \\ 2 & 4 & -1 \end{bmatrix} \longrightarrow \begin{bmatrix} 2 & 1 & -1 \\ \hline 3 & -1 & 1 \\ \hline 2 & -4 & 3 \end{bmatrix}.$$

This translates into $\begin{bmatrix} 2 & 1 & -1 \\ 6 & 2 & -2 \\ 4 & 6 & -3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 2 & -4 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & -1 \\ 0 & -1 & 1 \\ 0 & 0 & 3 \end{bmatrix}.$

7.3 Cholesky factorization

This is just a fancy word for an LU-factorization applied to a symmetric positive definite matrix. Let us state the result straightaway.

Theorem 7.2. Let A be a real symmetric positive definite matrix. It admits a unique factorization $A = LL^\top$, where L is lower triangular with positive diagonal entries.

Proof. By considering $x^\top A x = x^\top A_k x > 0$ for nonzero vectors $x = [x_1, \ldots, x_k, 0, \ldots, 0]^\top$, we see that the leading principal minors of A are all positive definite, hence nonsingular. Thus A admits a Doolittle's factorization $A = LU$. From $LU = A = A^\top = U^\top L^\top$ and the nonsingularity of L and U, one derives $UL^{-\top} = L^{-1}U^\top =: D$. This latter matrix is simultaneously lower and upper triangular, i.e. it is diagonal. Its entries are positive, as seen from

$$d_{i,i} = e_i^\top D e_i = e_i^\top U L^{-\top} e_i = (L^{-\top} e_i)^\top LU (L^{-\top} e_i) =: x^\top A x > 0.$$

Hence the matrix $D^{1/2} := \mathrm{Diag}[\sqrt{d_{1,1}}, \ldots, \sqrt{d_{n,n}}]$ satisfies $D^{1/2} D^{1/2} = D$, and it remains to write

$$A = LU = LDL^\top = LD^{1/2}D^{1/2}L^\top = (LD^{1/2})(LD^{1/2})^\top.$$

The proof of uniqueness is left as an easy exercise. □

7.4 Exercises

From the textbook: 6.a-b. p 396.

1. Check that the matrix $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ has no LU-factorization.

2. Prove that the matrix

$$\begin{bmatrix} 1 & a & 1 \\ 1 & b & 1 \\ 1 & c & 1 \end{bmatrix}, \qquad a \neq b,$$

   admits a unique Doolittle's decomposition. Given an $n \times n$ matrix, what does it say about the converse of

$$[\det A_k \neq 0,\ k \in [\![1, n]\!]] \Longrightarrow [A \text{ admits a unique Doolitle's factorization}] \ ?$$

3. Calculate *all* Doolittle's factorization of the matrix

$$A = \begin{bmatrix} 10 & 6 & -2 & 1 \\ 10 & 10 & -5 & 0 \\ -2 & 2 & -2 & 1 \\ 1 & 3 & -2 & 3 \end{bmatrix}.$$

   By using one of these factorizations, find *all* solutions of the equation $Ax = b$, where $b = [-2, 0, 2, 1]^\top$.

4. Show that the LU-factorization algorithm requires $\mathcal{O}(n^3)$ mult./div.

5. True or false: If A has a Doolittle's factorization, then it has Crout's factorization? Give either a proof or a counterexample.

6. For a nonsingular matrix M, establish that the matrix $MM^\top$ is symmetric positive definite.

7. Implement first an algorithm for finding the inverse of an upper triangular matrix U [hint: if $C_1, \ldots, C_n$ denote the columns of U^{-1}, what is UC_i?]. Implement next the LU-factorization algorithm. Implement at last an algorithm for finding the inverse of a square matrix. Test your algorithms along the way.

8. Calculate the Cholesky factorization of the matrix

$$\begin{bmatrix} 1 & 1 & & & & \\ 1 & 2 & 1 & & & \\ & 1 & 3 & 1 & & \\ & & 1 & 4 & 1 & \\ & & & 1 & 5 & 1 \\ & & & & 1 & \lambda \end{bmatrix}.$$

   Deduce from the factorization the value of λ that makes the matrix singular. Also find this value of λ by seeking a vector – say, whose first component is one – in the null-space of the matrix .

Optional problems

1. A matrix $A = (a_{i,j})$ in which $a_{i,j} = 0$ when $j > i$ or $j < i - 1$ is called a Stieljes matrix. Devise an efficient algorithm for inverting such a matrix.

2. Find the inverse of the matrix

$$\begin{bmatrix} 1 & -1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & -1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & -1 & \cdots & 0 \\ \vdots & \vdots & & \ddots & \ddots & \vdots \\ \vdots & \vdots & & 0 & 1 & -1 \\ 0 & 0 & \cdots & \cdots & 0 & 1 \end{bmatrix}.$$

Chapter 8

Iterative methods

Direct methods [Gaussian algorithm, LU-factorization,...] for solving the problem $Ax = b$ are often computationally expensive. By contrast, if we drop the perfect accuracy requirement [which is unrealistic anyway], some **indirect methods** have decisive advantage over direct methods in terms of speed and demand on computer memory. These **iterative schemes** [producing a sequence of vectors that will approximate the solution] are furthermore very efficient for **sparse** systems [a large number of entries of A are zero] and generally stable.

8.1 Description of two basic schemes

Consider the linear system $Ax = b$, which we break down into n equations

$$\sum_{j=1}^{n} a_{i,j} x_j = b_i, \qquad i \in [\![1, n]\!].$$

These are rewritten by isolating the unknown x_i to give

$$a_{i,i} x_i = - \sum_{j=1,\, j \neq i}^{n} a_{i,j} x_j + b_i, \qquad i \in [\![1, n]\!].$$

In matrix form, this reduces to the equation $Dx = (L+U)x + b$, where we have decomposed A in a diagonal matrix D, a strictly lower triangular matrix $-L$, and a strictly upper triangular matrix $-U$. To solve this by iteration, we can construct [if D is invertible] a sequence of vectors $(x^{(k)})$ by setting

$$Dx^{(k+1)} = (L+U)x^{(k)} + b \tag{8.1}$$

and hope that $x^{(k)}$ will approach the exact solution x^*. This is known as the **Jacobi iteration**. In fact, if the sequence $(x^{(k)})$ is convergent, then its limit is necessarily x^*. The computations to be performed, somewhat hidden in (8.1), are

$$x_i^{(k+1)} = \frac{1}{a_{i,i}} \left[-\sum_{j<i} a_{i,j} x_j^{(k)} - \sum_{j>i} a_{i,j} x_j^{(k)} + b_i \right], \qquad i \in [\![1, n]\!].$$

At step i, it seems intuitively better to replace the coordinates $x_j^{(k)}$, $j < i$, by their updated values $x_j^{(k+1)}$, $j < i$, i.e. to perform the computations

$$x_i^{(k+1)} = \frac{1}{a_{i,i}} \left[-\sum_{j<i} a_{i,j} x_j^{(k+1)} - \sum_{j>i} a_{i,j} x_j^{(k)} + b_i \right], \qquad i \in [\![1, n]\!].$$

This translates into the matrix form $x^{(k+1)} = D^{-1} \left[Lx^{(k+1)} + Ux^{(k)} + b \right]$, or equivalently

$$(D - L)x^{(k+1)} = Ux^{(k)} + b.$$

This scheme is known as the **Gauss–Seidel iteration.**

The algorithms for these two schemes copy the following pieces of pseudocode.

Jacobi

input n, $a_{i,j}$, b_i, x_i, M, ε
for $k = 1$ to M
    do
    for $i = 1$ to n
        do $u_i \leftarrow \left(b_i - \sum_{j \neq i} a_{i,j} x_j \right) / a_{i,i}$ **end do**
    end for
    if $\max_j |u_j - x_j| < \varepsilon$ **then output** k, (x_i) **stop end if**
    for $i = 1$ to n
        do $x_i \leftarrow u_i$ **end do**
    end for
    end do
end for
output 'procedure unsuccessful', (x_i)

Gauss–Seidel

input n, $a_{i,j}$, b_i, x_i, M, ε
for $k = 1$ to M
    do

```
        for i = 1 to n
                do u_i ← (b_i − ∑_{j<i} a_{i,j} u_j − ∑_{j>i} a_{i,j} x_j)/a_{i,i} end do
        end for
        if max_j |u_j − x_j| < ε then output k, (x_i) stop end if
        for i = 1 to n
                do x_i ← u_i end do
        end for
        end do
end for
output 'procedure unsuccessful', (x_i)
```

If we drop the tolerance restriction, the latter can be shortened to

```
input n, a_{i,j}, b_i, x_i, M
for k = 1 to M
        do for i = 1 to n
                do x_i ← (b_i − ∑_{j≠i} a_{i,j} x_j) /a_{i,i} end do
        end for end do
end for
output k, (x_i)
```

8.2 Conditions for convergence

We assume that the matrix A is nonsingular. The equation $Ax = b$ can always be rewritten as $Qx = (Q - A)x + b$, where the **splitting matrix** Q should be nonsingular and the system $Qx = c$ should be easy to solve. Then we consider the iteration process $Qx^{(k+1)} = (Q - A)x^{(k)} + b$, which is rather rewritten, for the sake of theoretical analysis, as

$$x^{(k+1)} = (I - Q^{-1}A)x^{(k)} + Q^{-1}b. \tag{8.2}$$

We say that this iterative method **converges** if the sequence $(x^{(k)})$ converges [to the unique solution x^* of $Ax = b$] regardless of the initial vector $x^{(0)}$. Obviously, the method is convergent if and only if $\varepsilon^{(k)} := x^{(k)} - x^*$, the **error vector** in the k-th iterate, tends to zero, or equivalently if and only if the **residual vector** $r^{(k)} := Ax^{(k)} - b = A\varepsilon^{(k)}$ tends to zero. Subtracting $x^* = (I - Q^{-1}A)x^* + Q^{-1}b$ to (8.2) gives

$$\varepsilon^{(k+1)} = (I - Q^{-1}A)\varepsilon^{(k)}, \qquad \text{hence} \quad \varepsilon^{(k)} = (I - Q^{-1}A)^k\varepsilon^{(0)}.$$

The following result is now more or less immediate.

Proposition 8.1. If $\delta := |||(I - Q^{-1}A)||| < 1$ for some natural matrix norm, then

1. the scheme (8.2) converges: $\lim_{k\to+\infty} \varepsilon^{(k)} = 0$,
2. the convergence is exponentially fast: $\|\varepsilon^{(k)}\| \leq \delta^k \|\varepsilon^{(0)}\|$,
3. the difference of consecutive terms controls the error: $\|\varepsilon^{(k)}\| \leq \dfrac{\delta}{1-\delta}\|x^{(k)} - x^{(k-1)}\|$.

Reminder: A norm $|||\bullet|||$on the space of $n \times n$ matrices is called a **natural** (or **induced**, or **subordinate**) matrix norm if it has the form $|||B||| = \max_{x\neq 0} \dfrac{\|Bx\|}{\|x\|}$ for some vector norm $\|\bullet\|$ on $\mathbb{R}^n$.

Proof. We get $\|\varepsilon^{(k)}\| \leq \delta^k \|\varepsilon^{(0)}\|$ directly from the expression of $\varepsilon^{(k)}$ in terms of $\varepsilon^{(0)}$, and in particular one has $\|\varepsilon^{(k)}\| \longrightarrow 0$. Furthermore, taking the limit $m \to +\infty$ in

$$\begin{aligned}
\|x^{(k+m)} - x^{(k)}\| &\leq \|x^{(k+m)} - x^{(k+m-1)}\| + \cdots + \|x^{(k+1)} - x^{(k)}\| \\
&= \|(I - Q^{-1}A)^m (x^{(k)} - x^{(k-1)})\| + \cdots + \|(I - Q^{-1}A)(x^{(k)} - x^{(k-1)})\| \\
&\leq (\delta^m + \cdots + \delta)\|x^{(k)} - x^{(k-1)}\| = \delta\frac{1-\delta^m}{1-\delta}\|x^{(k)} - x^{(k-1)}\|
\end{aligned}$$

yields the part 3. This actually insures that it is safe to halt the iterative process when $\|x^{(k)} - x^{(k-1)}\|$ is small. □

Corollary 8.2. If A is diagonally dominant, that is if

$$|a_{i,i}| > \sum_{j=1,\, j\neq i}^{n} |a_{i,j}|, \qquad i \in [\![1, n]\!],$$

then the Jacobi method is convergent.

Proof. For the Jacobi iteration, one takes $Q = D$. We consider the norm on $\mathbb{R}^n$ defined by $\|x\|_\infty := \max_{i\in[\![1,n]\!]} |x_i|$. We leave to the reader the task of checking every step in the inequality

$$|||I - D^{-1}A|||_\infty = \max_{i\in[\![1,n]\!]} \sum_{j=1}^{n} |(I - D^{-1}A)_{i,j}| = \max_{i\in[\![1,n]\!]} \sum_{j=1,\, j\neq i}^{n} \left|\frac{a_{i,j}}{a_{i,i}}\right| < 1.$$

□

As a matter of fact, Proposition 8.1 can be strengthen to give a necessary and sufficient condition for convergence. We recall first that the **spectral radius** of a matrix B is the maximum of the modulus of its eigenvalues, i.e.

$$\rho(B) := \max\{|\lambda| : \det(B - \lambda I) = 0\},$$

and we will use the fact that

$$\rho(B) = \inf\{|||B|||, \ |||\bullet||| \text{ is a natural matrix norm}\}.$$

Theorem 8.3. We have the equivalence

$$[\text{The iteration (8.2) converges}] \iff [\rho(I - Q^{-1}A) < 1].$$

Proof. The previous fact combined with Proposition 8.1 yields the $\Leftarrow$ part. As for the $\Rightarrow$ part, suppose that the iteration (8.2) converges, i.e. $B^k\varepsilon$ converges to zero for any $\varepsilon \in \mathbb{R}^n$, where we have set $B := I - Q^{-1}A$. Let us now assume that $r := \rho(B) \geq 1$, which should lead to a contradiction. There exist $x \in \mathbb{C}^n \setminus \{0\}$ and $\theta \in [0, 2\pi)$ such that $Bx = re^{i\theta}x$. It follows that $B^k x = r^k e^{ik\theta}x$. Writing $x = u + iv$ with $u, v \in \mathbb{R}^n$, one has $B^k x = B^k u + iB^k v \longrightarrow 0 + i \cdot 0 = 0$. But clearly $r^k e^{ik\theta}x \not\to 0$, hence the required contradiction. □

Corollary 8.4. If A is diagonally dominant, then the Gauss–Seidel method is convergent.

Proof. For the Gauss–Seidel iteration, one takes $Q = D - L = A + U$. According to the previous theorem, it suffices to prove that $|\lambda| < 1$ for any eigenvalue of $I - Q^{-1}A$. Let us then consider $x \in \mathbb{C}^n \setminus \{0\}$ such that $(I - Q^{-1}A)x = \lambda x$ for some $\lambda \in \mathbb{C}$ with $|\lambda| \leq 1$, and let us derive a contradiction. We have $(Q - A)x = \lambda Qx$, in other words $Ux = \lambda Dx - \lambda Lx$, or $\lambda Dx = Ux + \lambda Lx$. For the index i such that $|x_i| = \|x\|_\infty$, we get

$$\begin{aligned} |\lambda a_{i,i} x_i| &= \Big| -\sum_{j>i} a_{i,j}x_j - \lambda \sum_{j<i} a_{i,j}x_j \Big| \leq \sum_{j>i} |a_{i,j}||x_j| + |\lambda| \sum_{j<i} |a_{i,j}||x_j| \\ &\leq \Big[\sum_{j>i} |a_{i,j}|\Big] \cdot \|x\|_\infty + |\lambda| \Big[\sum_{j<i} |a_{i,j}|\Big] \cdot \|x\|_\infty \leq |\lambda| \Big[\sum_{j \neq i} |a_{i,j}|\Big] \cdot \|x\|_\infty < |\lambda||a_{i,i}|\|x\|_\infty, \end{aligned}$$

which results in a contradiction. □

8.3 Exercises

From the textbook: 2.a.c., 4.a.c, 5, 7 p 450; 17 p 451; 26 p 453.

1. Program the Gauss–Seidel method and test it on these examples

$$\begin{cases} 3x & + & y & + & z & = & 5 \\ x & + & 3y & - & z & = & 3 \\ 3x & + & y & - & 5z & = & -1 \end{cases}, \qquad \begin{cases} 3x & + & y & + & z & = & 5 \\ 3x & + & y & - & 5z & = & -1 \\ x & + & 3y & - & z & = & 3 \end{cases}.$$

Analyse what happens when these systems are solved by simple Gaussian elimination without pivoting.

2. The iteration $x^{(k+1)} = Hx^{(k)} + b$ is applied for $k = 0, 1, \ldots$, where H is the real 2×2 matrix

$$H = \begin{bmatrix} \alpha & \gamma \\ 0 & \beta \end{bmatrix},$$

with γ large and $|\alpha| < 1$, $|\beta| < 1$. Calculate the elements of H^k and show that they tend to zero as $k \to +\infty$. Further, establish the equation $x^{(k)} - x^* = H^k(x^{(0)} - x^*)$, where x^* is defined by $x^* = Hx^* + b$. Thus deduce that the sequence $(x^{(k)})$ converges to x^*.

3. For some choice of $x^{(0)}$, the iterative method

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} x^{(k+1)} + \begin{bmatrix} 0 & 0 & 0 \\ \xi & 0 & 0 \\ \eta & \zeta & 0 \end{bmatrix} x^{(k)} = b$$

is applied for $k = 0, 1, \ldots$ in order to solve the linear system

$$\begin{bmatrix} 1 & 1 & 1 \\ \xi & 1 & 1 \\ \eta & \zeta & 1 \end{bmatrix} x = b,$$

where ξ, η and ζ are constants. Find all values of the constants such that the sequence $(x^{(k)})$ converges for every $x^{(0)}$ and b. Give an example of nonconvergence when $\xi = \eta = \zeta = -1$. Is the solution always found in at most two iterations when $\xi = \zeta = 0$?

Chapter 9

Steepest descent and conjugate gradient methods

The goal here is once again to solve the system $Ax = b$, but with the additional assumption that A is a $n \times n$ symmetric positive definite matrix. Recall the inner product notation of two vectors x and y in $\mathbb{R}^n$:

$$\langle x, y \rangle := x^\top y = \sum_{i=1}^{n} x_i y_i.$$

Note that in general $\langle x, Ay \rangle = \langle A^\top x, y \rangle$, which becomes $\langle x, Ay \rangle = \langle Ax, y \rangle$ here, due to the symmetry of A. The following observation about the quadratic form $q(x) := \langle x, Ax \rangle - 2\langle x, b \rangle$ will be useful.

Lemma 9.1. For $x, v \in \mathbb{R}^n$, the minimun of q along the **ray through** x **parallel to** v is

$$\min_{t \in \mathbb{R}} q(x + tv) = q(x + t^* v) = q(x) - \frac{\langle v, b - Ax \rangle^2}{\langle v, Av \rangle}, \qquad \text{where} \quad t^* := \frac{\langle v, b - Ax \rangle}{\langle v, Av \rangle}.$$

Furthermore, the vector x^* minimizes q if and only if it satisfies $Ax^* = b$.

Proof. We calculate

$$\begin{aligned} q(x + tv) &= \langle x + tv, Ax + tAv - 2b \rangle = \langle x, Ax - 2b \rangle + t(\langle v, Ax - 2b \rangle + \langle x, Av \rangle) + t^2 \langle v, Av \rangle \\ &= q(x) + 2t\langle v, Ax - b \rangle + t^2 \langle v, Av \rangle. \end{aligned}$$

This quadratic polynomial is minimized at the zero of its derivative, i.e. at $t^* := \dfrac{\langle v, b - Ax \rangle}{\langle v, Av \rangle}$, where it takes the value given in the Lemma. The second part is now obtained as follow:

$$x^* \text{ minimizes } q \iff \forall\, v,\ q(x^*) \le \min_{t \in \mathbb{R}} q(x^* + tv) \iff \forall\, v,\ \langle v, b - Ax^* \rangle = 0 \iff b - Ax^* = 0.$$

□

Remark. Have we used the hypothesis that A is positive definite?

9.1 Steepest descent algorithm

According to the previous consideration, a solution of $Ax = b$ may be obtained as the limit of a sequence $(x^{(k)})$ for which $q(x^{(k)}) \to \min_{x \in \mathbb{R}^n} q(x)$. Here is the description of such an iterative scheme. Given $x^{(k-1)}$, pick a **search direction** $v^{(k)}$ [with $\|v^{(k)}\| = 1$, say], and let $x^{(k)}$ be the minimum of q along the ray through $x^{(k-1)}$ parallel to $v^{(k)}$, that is

$$x^{(k)} = x^{(k-1)} + t_k v^{(k)}, \qquad t_k = \frac{\langle v^{(k)}, b - Ax^{(k)} \rangle}{\langle v^{(k)}, Av^{(k)} \rangle}.$$

The search direction is taken to be the direction of greatest decrease of q – hence the name **steepest descent method**. This direction turns out to be

$$\nabla q(x^{(k-1)}) = \left[\frac{\partial q}{\partial x_1}(x^{(k-1)}), \ldots, \frac{\partial q}{\partial x_n}(x^{(k-1)}) \right]^\top = 2(Ax^{(k-1)} - b),$$

i.e. the direction of the residual. However, this method converges slowly, and is therefore rarely used for linear systems.

9.2 Conjugate gradient method

We still follow the basic strategy of minimizing the quadratic form q. The family of methods for which the search directions $(v^{(1)}, \ldots, v^{(n)})$ are chosen to form an A-orthogonal system constitutes the **conjugate direction methods**. The solution is given in a finite number of steps, precisely in n steps. Assuming exact arithmetic, that is; the picture is not so perfect if roundoff errors are taken into account.

Theorem 9.2. Let $(v^{(1)}, \ldots, v^{(n)})$ be a set of **A-orthogonal vectors**, i.e.

$$\langle v^{(i)}, Av^{(j)} \rangle = 0, \qquad \text{for } i, j \in [\![1, n]\!],\ i \neq j.$$

Choose the vector $x^{(0)}$ arbitrarily and define the sequence $(x^{(k)})$ by

$$x^{(k)} = x^{(k-1)} + \frac{\langle v^{(k)}, b - Ax^{(k-1)} \rangle}{\langle v^{(k)}, Av^{(k)} \rangle} v^{(k)}, \qquad k \in [\![1, n]\!]. \tag{9.1}$$

Then the resulting vector satisfies $Ax^{(n)} = b$.

Proof. We prove by induction on k that the residual $r^{(k)} = b - Ax^{(k)}$ is orthogonal to the system $(v^{(1)}, \ldots, v^{(k)})$, so that $r^{(n)}$ is orthogonal to the basis $(v^{(1)}, \ldots, v^{(n)})$, hence must be zero. For $k = 0$, there is nothing to establish, and the induction hypothesis holds trivially. Now suppose that it holds for $k-1$, that is $r^{(k-1)} \perp (v^{(1)}, \ldots, v^{(k-1)})$. Then apply $-A$ to (9.1) and add b to obtain

$$r^{(k)} = r^{(k-1)} - \frac{\langle v^{(k)}, r^{(k-1)} \rangle}{\langle v^{(k)}, Av^{(k)} \rangle} Av^{(k)}. \tag{9.2}$$

We clearly get $\langle r^{(k)}, v^{(j)} \rangle = 0$ for $j \in [\![1, k-1]\!]$ and we also observe that $\langle r^{(k)}, v^{(k)} \rangle = \langle r^{(k-1)}, v^{(k)} \rangle - \langle v^{(k)}, r^{(k-1)} \rangle = 0$. We have shown that $r^{(k)} \perp (v^{(1)}, \ldots, v^{(k)})$, i.e. that the induction hypothesis holds for k. This concludes the proof. □

To initiate this process, one may prescribe the A-orthogonal system at the beginning. This can be done using the Gram–Schmidt algorithm [see Chapter 10 – the positive definiteness of A insures that $\langle x, Ay \rangle$ defines an inner product]. One may also determine the search directions one at a time within the solution process. In the **conjugate gradient method**, the vectors $v^{(1)}, \ldots, v^{(n)}$ are chosen not only to be A-orthogonal, but also to induce an orthogonal system of residual vectors. Therefore, one should have $r^{(k)} \perp \mathrm{span}[r^{(0)}, \ldots, r^{(k-1)}]$, and the A-orthogonality of the search directions implies that $r^{(k)} \perp \mathrm{span}[v^{(1)}, \ldots, v^{(k)}]$, just like in the proof of Theorem 9.2. This suggests to impose $\mathrm{span}[r^{(0)}, \ldots, r^{(k-1)}] = \mathrm{span}[v^{(1)}, \ldots, v^{(k)}] =: U_k$ for each k. Then, since $r^{(k)} \in U_{k+1}$, one could write [renormalizing the search direction if necessary],

$$r^{(k+1)} = \alpha_{k+1} v^{(k+1)} + \alpha_k v^{(k)} + \cdots + \alpha_1 v^{(1)}, \qquad \text{with } \alpha_{k+1} = 1.$$

Using (9.2) and the A-orthogonality of $(v^{(1)}, \ldots, v^{(n)})$, one would derive that

$$\alpha_j = \frac{1}{\langle v^{(j)}, Av^{(j)} \rangle} \langle r^{(k)}, Av^{(j)} \rangle = \frac{1}{\langle v^{(j)}, r^{(j-1)} \rangle} \langle r^{(k)}, r^{(j-1)} - r^{(j)} \rangle.$$

We would have $\alpha_j = 0$ for $j \leq k-1$, $\alpha_k = -\dfrac{\langle r^{(k)}, r^{(k)} \rangle}{\langle v^{(k)}, r^{(k-1)} \rangle}$ and $1 = \alpha_{k+1} = \dfrac{\langle r^{(k)}, r^{(k)} \rangle}{\langle v^{(k+1)}, r^{(k)} \rangle}$. Note that the equality $\langle v^{(k+1)}, r^{(k)} \rangle = \langle r^{(k)}, r^{(k)} \rangle$ should also hold when k is replaced by $k-1$, so that $\alpha_k = -\dfrac{\langle r^{(k)}, r^{(k)} \rangle}{\langle r^{(k-1)}, r^{(k-1)} \rangle}$. Rearranging the expression for $r^{(k+1)}$, we would obtain $v^{(k+1)} = r^{(k)} + \dfrac{\langle r^{(k)}, r^{(k)} \rangle}{\langle r^{(k-1)}, r^{(k-1)} \rangle} v^{(k)}$. At this point, we have speculated on the possible recursive construction of the search directions. It is time to give a precise description of the process and to check rigorously that it yields the desired result.

Theorem 9.3. Choose the vector $x^{(0)}$ arbitrarily, and set $v^{(1)} = r^{(0)} = b - Ax^{(0)}$. Assuming that $x^{(0)}, \ldots, x^{(k-1)}$ and $v^{(1)}, \ldots, v^{(k)}$ have been constructed, set

$$\begin{aligned} x^{(k)} &= x^{(k-1)} + t_k\, v^{(k)}, & t_k &= \frac{\langle r^{(k-1)}, r^{(k-1)}\rangle}{\langle v^{(k)}, Av^{(k)}\rangle}, \\ v^{(k+1)} &= r^{(k)} + s_k\, v^{(k)}, & s_k &= \frac{\langle r^{(k)}, r^{(k)}\rangle}{\langle r^{(k-1)}, r^{(k-1)}\rangle}. \end{aligned}$$

Then the vector $x^{(n)}$ is a solution of $Ax = b$.

Remark. The residual vector should be determined according to $r^{(k)} = r^{(k-1)} - t_k Av^{(k)}$ rather than $r^{(k)} = b - Ax^{(k)}$. Besides, there seems to be a flaw in the definitions of t_k and s_k, as we divide e.g. by $\langle r^{(k-1)}, r^{(k-1)}\rangle$, which might equal zero. If this situation occurs, then $r^{(k-1)}$ is already a solution of $Ax = b$. In any case, if $r^{(k-1)} = 0$, then $t_k = 0$, hence $r^{(k)} = r^{(k-1)} = 0$, then $t_{k+1} = 0$, and so on. Updating the search direction becomes unnecessary.

Proof. The second equation implies that $\operatorname{span}[r^{(0)}, \ldots, r^{(k-1)}] = \operatorname{span}[v^{(1)}, \ldots, v^{(k)}] =: U_k$ for all k. We now establish by induction on k that $r^{(k)}$ and $Av^{(k+1)}$ are orthogonal to the space U_k. For $k = 0$, there is nothing to show. Assume now that $r^{(k-1)} \perp U_{k-1}$ and that $Av^{(k)} \perp U_{k-1}$. Then, we calculate, for $u \in U_{k-1}$,

$$\langle r^{(k)}, u\rangle = \langle r^{(k-1)}, u\rangle - t_k \langle Av^{(k)}, u\rangle = 0 + t_k \cdot 0 = 0,$$

and also

$$\begin{aligned} \langle r^{(k)}, v^{(k)}\rangle &= \langle r^{(k-1)}, v^{(k)}\rangle - t_k\langle Av^{(k)}, v^{(k)}\rangle = \langle r^{(k-1)}, v^{(k)}\rangle - \langle r^{(k-1)}, r^{(k-1)}\rangle \\ &= \langle r^{(k-1)}, v^{(k)} - r^{(k-1)}\rangle = s_{k-1}\langle r^{(k-1)}, v^{(k-1)}\rangle = 0. \end{aligned}$$

This proves that $r^{(k)} \perp U_k$, which is the first part of the induction hypothesis relative to $k+1$. As for the second part, we calculate

$$\begin{aligned} \langle Av^{(k+1)}, v^{(j)}\rangle &= \langle v^{(k+1)}, Av^{(j)}\rangle = \langle r^{(k)}, Av^{(j)}\rangle + s_k\langle v^{(k)}, Av^{(j)}\rangle \\ &= \frac{1}{t_k}\langle r^{(k)}, r^{(j-1)} - r^{(j)}\rangle + s_k\langle v^{(k)}, Av^{(j)}\rangle. \end{aligned}$$

Hence, for $j \leq k-1$, we readily check that $\langle Av^{(k+1)}, v^{(j)}\rangle = 0$, and for $j = k$, we observe that $\langle Av^{(k+1)}, v^{(k)}\rangle = -\langle r^{(k)}, r^{(k)}\rangle / t_k + s_k\langle v^{(k)}, Av^{(k)}\rangle = 0$. This proves that $Av^{(k+1)} \perp U_k$, which is the second part of the induction hypothesis relative to $k+1$. The inductive proof is now complete. In particular, we have shown that the system $(v^{(1)}, \ldots, v^{(n)})$ is A-orthogonal. Theorem 9.2 applies, and we conclude that $Ax^{(n)} = b$. □

A computer code for the conjugate gradient method is based on the following pseudocode.

input n, A, b, x, M, ε, δ
$r \leftarrow b - Ax, \quad v \leftarrow r, \quad c \leftarrow \langle r, r \rangle$
for $k = 1$ to M **do**
  if $\langle v, v \rangle < \delta$ **then stop**
  else $z \leftarrow Av, \quad t \leftarrow c/\langle v, z \rangle, \quad x \leftarrow x + tv, \quad r \leftarrow r - tz, \quad d \leftarrow \langle r, r \rangle$
  end if
  if $d < \varepsilon$ **then stop**
  else $v \leftarrow r + (d/c)v, \quad c \leftarrow d$
  end if
end do end for
output k, x, r

9.3 Exercises

From the textbook: 12, 13 p 479; 3, 4 p 477.

1. In the method of steepest descent, show that the vectors $v^{(k)}$ and $v^{(k+1)}$ are orthogonal and that $q(x^{(k+1)}) = q(x^{(k)}) - \dfrac{\langle r^{(k)}, r^{(k)} \rangle^2}{\langle r^{(k)}, Ar^{(k)} \rangle}$.
2. Program and test the conjugate gradient method. A good test case is the Hilbert matrix with a simple b-vector:

$$a_{i,j} = \frac{1}{i+j-1}, \qquad b_i = \frac{a_{i,1} + a_{i,2} + \cdots + a_{i,n}}{3}, \qquad i, j \in [\![1, n]\!].$$

Optional problems

1. Using Jacobi, Gauss–Seidel and the conjugate gradient methods with the initial vector $x^{(0)} = [0, 0, 0, 0, 0]^\top$, compute the solution of the system

$$\begin{bmatrix} 10 & 1 & 2 & 3 & 4 \\ 1 & 9 & -1 & 2 & -3 \\ 2 & -1 & 7 & 3 & -5 \\ 3 & 2 & 3 & 12 & -1 \\ 4 & -3 & -5 & -1 & 15 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 12 \\ -27 \\ 14 \\ -17 \\ 12 \end{bmatrix}.$$

Chapter 10

The QR factorization

By QR**-factorization** of an $m \times n$ matrix A, we understand either a decomposition of the form

$$A = QR, \qquad \text{where} \qquad \begin{array}{l} Q \text{ is an } m \times m \text{ orthogonal matrix,} \\ R \text{ is an } m \times n \text{ upper triangular matrix,} \end{array}$$

or a decomposition of the form

$$A = BT, \qquad \text{where} \qquad \begin{array}{l} \text{the columns of the } m \times n \text{ matrix } B \text{ are orthonormal,} \\ T \text{ is an } n \times n \text{ upper triangular matrix.} \end{array}$$

Most of the time, we will assume that $m \geq n$, in which case every matrix has a (non-unique) QR-factorization for either of these representations. In the case $m = n$, the factorization can be used to solve linear systems, according to

$$[Ax = b] \iff [Qy = b, \quad y = Rx].$$

The system $y = Rx$ is easy to solve [backward substitution], and so is the system $Qy = b$ [take $y = Q^\top b$]. Knowing a QR-factorization of A, one could also calculate its determinant [$\det A = \det Q \cdot \det R = \pm \prod_{i=1}^n r_{i,i}$] and find its inverse [$A^{-1} = R^{-1} Q^\top$].

10.1 The Gram–Schmidt orthogonalization process

10.1.1 The algorithm

Consider n linearly independent vectors $u_1, \ldots, u_n$ in $\mathbb{R}^m$. Observe that we necessarily have $m \geq n$. We wish to 'orthonormalize' them, i.e. to create vectors $v_1, \ldots, v_n$ such that

$$(v_1, \ldots, v_k) \text{ is an orthonormal basis for } V_k := \text{span}[u_1, \ldots, u_k], \qquad \text{each } k \in [\![1, n]\!].$$

It is always possible to find such vectors, and in fact they are uniquely determined if the additional condition $\langle v_k, u_k \rangle > 0$ is required. The step-by-step construction is based on the following scheme.

Suppose that $v_1, \ldots, v_{k-1}$ have been obtained; search in V_k for a vector

$$\widetilde{v}_k = u_k + \sum_{i=1}^{k-1} c_{k,i}\, v_i \quad \text{such that} \quad \widetilde{v}_k \perp V_{k-1};$$

the conditions $0 = \langle \widetilde{v}_k, v_i \rangle = \langle u_k, v_i \rangle + c_{k,i}$ impose the choice $c_{k,i} = -\langle u_k, v_i \rangle$;
now that $\widetilde{v}_k$ is completely determined, form the normalized vector $v_k = \dfrac{1}{\|\widetilde{v}_k\|}\widetilde{v}_k$.

The accompanying code parallels the pseudocode

input n, (u_k)
for $k = 1$ to n **do** $v_k \leftarrow u_k$,
  for $i = 1$ to $k-1$ **do**
  $c_{k,i} \leftarrow \langle u_k, v_i \rangle, \quad v_k \leftarrow v_k - c_{k,i} v_i$
end do end for
$v_k \leftarrow v_k / \|v_k\|$
end do end for
output (v_k)

For example, we write down explicitely all the steps in the orthonormalization process for the vectors

$$\underline{u_1 = [6,3,2]^\top}, \quad \underline{u_2 = [6,6,1]^\top}, \quad \underline{u_3 = [1,1,1]^\top}.$$

- $\widetilde{v}_1 = u_1, \quad \|\widetilde{v}_1\| = \sqrt{36+3+4} = 7, \quad \underline{v_1 = 1/7\,[6,3,2]^\top};$
- $\widetilde{v}_2 = u_2 + \alpha v_1, \quad 0 = \langle \widetilde{v}_2, v_1 \rangle \Rightarrow \alpha = -\langle u_2, v_1 \rangle = -(16+18+2)/7, \quad \underline{\alpha = -8},$
  $\widetilde{v}_2 = 1/7\,[7\cdot 6 - 8\cdot 6, 7\cdot 6 - 8\cdot 3, 7\cdot 1 - 8\cdot 2]^\top = 1/7\,[-6,18,-9]^\top = 3/7\,[-2,6,-3]^\top,$
  $\|\widetilde{v}_2\| = 3/7\,\sqrt{4+36+9} = 3, \quad \underline{v_2 = 1/7\,[-2,6,-3]^\top};$
- $\widetilde{v}_3 = u_3 + \beta v_2 + \gamma v_1, \quad \begin{matrix} 0 = \langle \widetilde{v}_3, v_2 \rangle, \\ 0 = \langle \widetilde{v}_3, v_1 \rangle, \end{matrix} \Rightarrow \begin{matrix} \beta = -\langle u_3, v_2 \rangle = -(-2+6-3)/7, & \underline{\beta = -1/7}, \\ \gamma = -\langle u_3, v_1 \rangle = -(6+3+2)/7, & \underline{\gamma = -11/7}, \end{matrix}$
  $\widetilde{v}_3 = 1/49\,[49+2-66, 49-6-33, 49+3-22]^\top = 1/49\,[-15,10,30]^\top = 5/49\,[-3,2,6]^\top,$
  $\|\widetilde{v}_3\| = 5/49\,\sqrt{9+4+36} = 5/7, \quad \underline{v_3 = 1/7\,[-3,2,6]^\top}.$

10.1.2 Matrix interpretation

Let A be a $m \times n$ matrix, with $m \geq n$, and let $u_1, \ldots, u_n \in \mathbb{R}^m$ denote its columns. The Gram–Schmidt algorithm produces orthonormal vectors $v_1, \ldots, v_n \in \mathbb{R}^m$ such that, for each $j \in [\![1, n]\!]$,

$$u_j = \sum_{k=1}^{j} t_{k,j}\, v_k = \sum_{k=1}^{n} t_{k,j}\, v_k, \tag{10.1}$$

with $t_{k,j} = 0$ for $k > j$, in other words, $T = [t_{i,j}]_{i,j=1}^{n}$ is an $n \times n$ upper triangular matrix. The n equations (10.1) reduce, in matrix form, to $A = BT$, where B is the $m \times n$ matrix whose columns are the orthonormal vectors $v_1, \ldots, v_n$. To explain the other factorization, let us complete $v_1, \ldots, v_n$ with $v_{m+1}, \ldots, v_n$ to form an orthonormal basis $(v_1, \ldots, v_m)$ of $\mathbb{R}^m$. The analogs of the equations (10.1), i.e. $u_j = \sum_{k=1}^{m} r_{k,j}\, v_k$ with $r_{k,j} = 0$ for $k > j$, read $A = QR$, where Q is the $m \times m$ orthogonal matrix with columns $v_1, \ldots, v_m$ and R is an $m \times n$ upper triangular matrix.

To illustrate this point, observe that the orthonormalization carried out in Section 10.1.1 translates into the factorization [identify all the entries]

$$\begin{bmatrix} 6 & 6 & 1 \\ 3 & 6 & 1 \\ 2 & 1 & 1 \end{bmatrix} = \underbrace{\frac{1}{7}\begin{bmatrix} 6 & -2 & -3 \\ 3 & 6 & 2 \\ 2 & -3 & 6 \end{bmatrix}}_{\text{orthogonal}} \underbrace{\begin{bmatrix} 7 & 8 & 11/7 \\ 0 & 3 & 1/7 \\ 0 & 0 & 5/7 \end{bmatrix}}_{\text{upper triangular}}.$$

10.2 Other methods

The Gram–Schmidt algorithm has the disadvantage that small imprecisions in the calculation of inner products accumulate quickly and lead to effective loss of orthogonality. Alternative ways to obtain a QR-factorization are presented below on some examples. They are based on the following idea and exploits the fact that the computed product of orthogonal matrices gives, with acceptable error, an orthogonal matrix.

Multiply the matrix A on the left by some orthogonal matrices Q_i which 'eliminate' some entries below the main 'diagonal', until the result is an upper triangular matrix R, thus

$$Q_k \cdots Q_2 Q_1 A = R \quad \text{yields} \quad A = QR, \quad \text{with} \quad Q = Q_1^\top Q_2^\top \cdots Q_k^\top.$$

We mainly know two types of orthogonal transformations, namely rotations and reflexions. Therefore, the two methods we describe are associated with **Givens rotations** [preferred when A is sparse] and **Householder reflections** [preferred when A is dense].

10.2.1 Givens rotations

The matrix

$$\Omega^{[i,j]} := \begin{bmatrix} \ddots & & i & & j & & \\ & 1 & \vdots & 0 & \vdots & 0 & \\ i & \cdots & \cos\theta & \cdots & \sin\theta & \cdots & \\ & 0 & \vdots & \ddots & \vdots & 0 & \\ j & \cdots & -\sin\theta & \cdots & \cos\theta & \cdots & \\ & 0 & \vdots & 0 & \vdots & 1 & \\ & & & & & & \ddots \end{bmatrix}$$

corresponds to a rotation along the two-dimensional space $\mathrm{span}[e_i, e_j]$. The rows of the matrix $\Omega^{[i,j]}A$ are the same as the rows of A, except for the i-th and j-th rows, which are linear combinations of the i-th and j-th rows of A. By choosing θ appropriately, we may introduce a zero at a prescribed position on one of these rows. Consider for example the matrix $A = \begin{bmatrix} 6 & 6 & 1 \\ 3 & 6 & 1 \\ 2 & 1 & 1 \end{bmatrix}$ of the end of Section 10.1.2. We pick $\Omega^{[1,2]}$ so that $\Omega^{[1,2]}A = \begin{bmatrix} 6 & 6 & 1 \\ 0 & \times & \times \\ \times & \times & \times \end{bmatrix}$. Then we pick $\Omega^{[1,3]}$ so that $\Omega^{[1,3]}\Omega^{[1,2]}A = \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \end{bmatrix}$. Finally, thanks to the leading zeros in the second and third rows, we can pick $\Omega^{[2,3]}$ so that $\Omega^{[2,3]}\Omega^{[1,3]}\Omega^{[1,2]}A = \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \end{bmatrix}$. The matrix $\left[\Omega^{[2,3]}\Omega^{[1,3]}\Omega^{[1,2]}\right]^\top$ is the orthogonal matrix required in the factorization of A.

10.2.2 Householder reflections

The reflection in the direction of a vector v transformes v into $-v$ while leaving the space $v^\top$ unchanged. It can therefore be expressed through the symmetric orthogonal matrix

$$H_v := I - \frac{2}{\|v\|^2} vv^\top.$$

Consider the matrix $A = \begin{bmatrix} 6 & 6 & 1 \\ 3 & 6 & 1 \\ 2 & 1 & 1 \end{bmatrix}$ once again. We may transform $u_1 = [6,3,2]^\top$ into $7e_1 = [7,0,0]^\top$ by way of the reflection in the direction $v_1 = u_1 - 7e_1 = [-1,3,2]^\top$. The latter is represented by the matrix

$$H_{v_1} = I - \frac{2}{\|v_1\|^2} v_1 v_1^\top = I - \frac{1}{7} \begin{bmatrix} 1 & -3 & -2 \\ -3 & 9 & 6 \\ -2 & 6 & 4 \end{bmatrix} = \frac{1}{7} \begin{bmatrix} 6 & 3 & 2 \\ 3 & -2 & -6 \\ 2 & -6 & 3 \end{bmatrix}.$$

Then the matrix $H_{v_1} A$ has the form $\begin{bmatrix} 7 & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \end{bmatrix}$, where the precise expression for the second column is

$$H_{v_1} u_2 = u_2 - \frac{\langle v_1, u_2 \rangle}{7} v_1 = u_2 - 2v_1 = \begin{bmatrix} 8 \\ 0 \\ -3 \end{bmatrix}.$$

To cut the argument short, we may observe at this point that the multiplication of $H_{v_1} A$ on the left by the permutation matrix $P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$ [which can be interpreted as $H_{e_2 - e_3}$] exchanges the second and third rows, thus gives an upper triangular matrix. In conclusion, the orthogonal matrix Q has been obtained as

$$H_{v_1}^\top P^\top = H_{v_1} P = \begin{bmatrix} 6 & 2 & 3 \\ 3 & -6 & -2 \\ 2 & 3 & -6 \end{bmatrix}.$$

10.3 Exercises

1. Fill in the numerical details in Section 10.2.1.

2. Implement and test the code for the Gram–Schmidt process. Based on this, implement and test a code producing the QR-factorization of a square matrix.

Chapter 11

Linear least-squares problems

11.1 Statement of the problem

Consider the linear system $Ax = b$, where A is an $m \times n$ matrix and b is a vector in $\mathbb{R}^m$. If $m < n$, we could add some artificial equations to make the system square. We shall therefore suppose from now on that $m \geq n$, and we also assume for simplicity that $\mathrm{rk}A = n$. When $m > n$, the equation $Ax = b$ has in general no solution, nonetheless we will try to 'make Ax as close to b as possible'. Hence we are looking at the minimization of $\|b - Ax\|$, the Euclidean norm of the residual vector. This is the **least-squares problem**. There is a simple characterization of the minimizing vector.

Theorem 11.1. The following equivalence holds

$$[\|b - Ax^*\| = \min_{x \in \mathbb{R}^n} \|b - Ax\|] \iff [A^\top (Ax^* - b) = 0].$$

Proof. According to the next lemma, used with $V = \mathrm{Im}A$, we see that x^* is characterized by $b - Ax^* \perp \mathrm{Im}A$, i.e. $0 = \langle b - Ax^*, Ax \rangle = \langle A^\top (b - Ax^*), x \rangle$ for all $x \in \mathbb{R}^m$, implying that $A^\top (b - Ax^*) = 0$. □

Lemma 11.2. If V is a subspace of $\mathbb{R}^d$ and if $c \in \mathbb{R}^d$, then

$$[\|c - v^*\| = \min_{v \in V} \|c - v\|] \iff c - v^* \perp V.$$

As a picture is worth thousands words, no formal proof is given.

11.2 Solution of the problem

It is possible to find the optimal x^* by solving the so-called **normal equations**, that is $A^\top Ax^* = A^\top b$. We could use Cholesky factorization for instance, or even the conjugate gradient method [check this last claim]. Another option is to use a QR-factorization of A. Remember that the $m \times m$ orthogonal matrix Q preserves the Euclidean norm, hence

$$\|b - Ax\| = \|QQ^\top b - QRx\| = \|c - Rx\|, \qquad \text{where} \quad c := Q^\top b.$$

We may write

$$c = \left[\begin{array}{c} c_1 \\ \hline c_2 \end{array}\right] \quad \text{and} \quad Rx = \left[\begin{array}{c} U \\ \hline 0 \end{array}\right] x = \left[\begin{array}{c} Ux \\ \hline 0 \end{array}\right], \qquad \text{so that} \qquad c - Rx = \left[\begin{array}{c} c_1 - Ux \\ \hline c_2 \end{array}\right].$$

Observe that $\mathrm{rk}U = \mathrm{rk}R = \mathrm{rk}A = n$, meaning that U is nonsingular, which clearly yields the value

$$\min_{x \in \mathbb{R}^n} \|c - Rx\| = \|c_2\|, \qquad \text{achieved for} \quad x^* = U^{-1}c_1.$$

Note that x^* can easily be computed by backward substitution.

11.3 Exercises

1. Find the least-squares solution to the system

$$\begin{bmatrix} 3 & 2 \\ 2 & 3 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 3 \\ 0 \\ 1 \end{bmatrix}.$$

2. Suppose that the solution to the previous problem is given to you as $[29/21, -2/3]^\top$. How can this be verified without solving for x and y.

Part III

Solving nonlinear equations

Chapter 12

Bisection method

We now turn our attention to the determination of the roots of a [nonlinear] equation, or, to say things differently, to the determination of the zeros of a function. Note that any equation $g(x) = h(x)$ can indeed be written in the form $f(x) = 0$, with $f := g - h$. In the theory of diffraction of light, for instance, one has to deal with the equation $x = \tan x$; in the calculation of planetary orbits, we need the roots of Kepler's equation $x - a \sin x = b$ for various values of a and b. Usually, an explicit expression for the solution cannot be obtained and one has to resort to numerical computations. We describe here how the **bisection method** achieves this specific goal.

12.1 Description of the method

The bisection method may also be referred to as the **method of interval halving**, since it follows the simple strategy:

For a function f changing sign on the interval $[a, b]$, choose between the left and right half-intervals one that supplies a change of sign for f [hence a zero], and repeat the process with this new interval.

Note that each of the two half-intervals may contain a zero of f, and that in general the function f have several zeros, but our process only produces one of them. Remark also that the method implicitely requires the function f to be continuous [forcing us to be cautious with the equation $x = \tan x$]. Since $\text{sgn}f(a) \neq \text{sgn}f(b)$, we must have either $\text{sgn}f(c) \neq \text{sgn}f(a)$ or $\text{sgn}f(c) \neq \text{sgn}f(b)$, where c denotes e.g. the midpoint of $[a, b]$, thus it is the **intermediate value theorem** which insures the existence of a zero of f in $[a, c]$ or in $[c, b]$.

To avoid unnecessary evaluations of functions, any value needed in the code should be stored rather than recomputed. Besides, to save multiplications, the test $\mathrm{sgn} f(a) \neq \mathrm{sgn} f(c)$ is preferable to the test $f(a)f(b) < 0$. A primitive pseudocode can be written as

input f, a, b, $Nmax$, ε
$fa \leftarrow f(a)$, $fb \leftarrow f(b)$, $e \leftarrow b - a$
if $\mathrm{sgn} fa = \mathrm{sgn} fb$ **then output** a, fa, b, fb, 'same sign for the function at a and b' **end if**
for $n = 0$ to $Nmax$
  do $e \leftarrow e/2$, $c \leftarrow a + e$, $fc \leftarrow f(c)$
  if $fc < \varepsilon$ **then output** n, c, fc, e **end if**
  if $\mathrm{sgn} fa \neq \mathrm{sgn} fc$ **then** $b \leftarrow c$, $fb \leftarrow fc$
    else $a \leftarrow c$, $fa \leftarrow fc$ **end if**
end for

12.2 Convergence analysis

We now investigate the accuracy of the bisection method by estimating how close the final point c is to a zero of the function f [evaluating how close $f(c)$ is to zero constitutes a different matter]. Let us denote by $[a_0, b_0]$, $[a_1, b_1]$, and so on, the intervals arising in the bisection process. Clearly, the sequence (a_n) is increasing and bounded above by b, and the sequence (b_n) is decreasing and bounded below by a, so both sequences must converge. Furthermore, the lengths of the successive intervals satisfy $b_n - a_n = (b_{n-1} - a_{n-1})/2$, hence $b_n - a_n = (b - a)/2^n$, and the limits of a_n and b_n must have the same value, say r. Then the inequalities $f(a_n)f(b_n) \leq 0$ and the continuity of f imply that $f(r)^2 \leq 0$, thus $f(r) = 0$, i.e r is a zero of f. For each n, the point r lies in the interval $[a_n, b_n]$, so that

$$|r - c_n| \leq \frac{1}{2}(b_n - a_n) \leq \frac{1}{2^{n+1}}(b - a), \qquad \text{where} \qquad c_n := \frac{a_n + b_n}{2}.$$

The following summary of the situation may be formulated.

Theorem 12.1. If the bisection algorithm is applied to a continuous function on an interval $[a, b]$ where $f(a)f(b) < 0$, then an approximate root is computed after n steps with error at most $(b - a)/2^{n+1}$.

If the error tolerance is prescribed at the start, we can determine in advance how many steps suffice to achieve this tolerance. Indeed, to insure that $|r - c_n| < \varepsilon$, it is enough to impose $(b - a)/2^{n+1} < \varepsilon$, i.e. $n > \log_2 \left(\frac{b-a}{2\varepsilon} \right) = \ln \left(\frac{b-a}{2\varepsilon} \right) / \ln 2$.

12.3 Exercises

From the textbook: 5, 10 p 51; 13, 15, 19 p 51.

- Write a program to find a zero of a function f in the following way: at each step, an interval $[a, b]$ is given and $f(a)f(b) < 0$; next c is computed as the zero of the linear function that agrees with f at a and b; then either $[a, c]$ or $[c, b]$ is retained, depending on whether $f(a)f(c) < 0$ or $f(c)f(b) < 0$. Test the program on several functions.

Chapter 13

Newton's method

Newton's method is another very popular numerical root-finding technique. It can be applied in many diverse situations. When specialized to real function of a real variable, it is often called **Newton–Raphson iteration**.

13.1 Description of the method

We start with some preliminary guess work. Suppose that r is a zero of a function f and that $x = r + h$ is an approximation to r. When x is close to r, Taylor's theorem allows us to write

$$0 = f(r) = f(x-h) = f(x) - hf'(x) + \mathcal{O}(h^2) \approx f(x) - hf'(x).$$

Solving in h gives $h \approx f(x)/f'(x)$. We therefore expect $x - f(x)/f'(x)$ to be a better approximation to r than x was. This is the core of Newton's method:

Construct a sequence (x_n) recursively, starting with an initial estimate x_0 for a zero of f, and apply the iteration formula

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{13.1}$$

Assuming convergence of this sequence to a number r, we obtain $r = r - f(r)/f'(r)$ by taking the limit as $n \to \infty$, thus $f(r) = 0$. As hoped for, the algorithm produces a zero of f. Note that a few precautions have to be taken for this conclusion to be legitimate: we need f and f' to be continuous, as well as $f'(r) \neq 0$. Theorem 13.3 has a slighty stronger set of hypotheses that insures the convergence of the scheme, provided that we start with x_0 'close enough' to a zero of f. This latter restriction is one of the weak point of the

method and it cannot be lifted. As a matter of fact, from the graphical interpretation of the method, it is not too hard to devise cases where Newton iteration will fail. For instance, the function $f(x) = \mathrm{sgn}(x)\sqrt{|x|}$ yields a cycle of period 2 for any choice of $x_0 \neq 0$ [i.e. $x_0 = x_2 = x_4 = \cdots$ and $x_1 = x_3 = x_5 = \cdots$]. The above mentioned graphical interpretation of the method is nothing more than the observation that x_{n+1} is obtained as the intersection of the x-axis with the tangent line to the f-curve at x_n. Indeed, (13.1) may be rewritten as $f(x_n) + f'(x_n)(x_{n+1} - x_n) = 0$.

As an example, here is a method often used to compute the square root of a number $y > 0$. We want to solve $f(x) := x^2 - y = 0$, so we perform the iterations

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \qquad \text{that is} \qquad x_{n+1} = \frac{1}{2}\left(x_n + \frac{y}{x_n}\right).$$

This formula is very old [dated between 100 B.C. and 100 A.D., credited to Heron, a Greek engineer and architect], yet very efficient. The value given for $\sqrt{17}$ after 4 iterations, starting with $x_0 = 4$, is correct to 28 figures.

We end this section with a pseudocode which includes stopping criteria. Note that we would need subprograms for $f(x)$ and $f'(x)$.

input a, M, δ, ε
$v \leftarrow f(a)$
output $0, a, v$
if $|v| < \varepsilon$ **then stop end if**
for $k = 1$ to M **do**
    $b \leftarrow a - v/f'(a), \quad v \leftarrow f(b)$
    output k, b, v
    if $|b - a| < \delta$ or $|v| < \varepsilon$ **then stop end if**
    $a \leftarrow b$
end do end for

13.2 Convergence analysis

The analysis will be carried out in the framework of **discrete dynamical systems**. This merely involves iterations of the type

$$x_{n+1} = F(x_n), \qquad \text{where } F \text{ is a continuous function mapping its domain into itself.}$$

Since $x_1 = F(x_0)$, $x_2 = F(x_1) = F(F(x_0))$, and so on, we often use the convenient notation $x_n = F^n(x_0)$. One simply take $F(x) = x - f(x)/f'(x)$ for Newton's method, in which case a zero of f is recognized as a **fixed point** of F, i.e. a point r for which $F(r) = r$. Here are some general convergence theorems in this setting.

Theorem 13.1. Suppose that F is a continuous function from $[a, b]$ into $[a, b]$. Then F has a fixed point in $[a, b]$. Suppose in addition that F is continuously differentiable on $[a, b]$ – in short, that $F \in \mathcal{C}^1[a, b]$. If there is a constant $0 \le c < 1$ such that $|F'(x)| \le c$ for all $x \in [a, b]$, then the fixed point is unique. Moreover, the iterates $x_n = F^n(x_0)$ converge towards this unique fixed point r and the error $e_n = x_n - r$ satisfies $|e_{n+1}| \le c\,|e_n|$.

Proof. Consider the function G defined on $[a, b]$ by $G(x) := F(x) - x$. We observe that $G(a) = F(a) - a \ge a - a = 0$ and that $G(b) = F(b) - b \le b - b = 0$. Thus, by the intermediate value theorem, there is a point $r \in [a, b]$ such that $G(r) = 0$, that is $F(r) = r$. Suppose now that $F \in \mathcal{C}^1[a, b]$ and that $|F'(x)| \le c < 1$ for all $x \in [a, b]$. Assume that there are points r and s such that $F(r) = r$ and $F(s) = s$. We then use the mean value theorem to obtain

$$|r - s| = |F(r) - F(s)| \le c|r - s|.$$

This is only possible if $|r - s| = 0$, i.e. $r = s$. Uniqueness of the fixed point r is established. We conclude the proof by writing

$$|e_{n+1}| = |x_{n+1} - r| = |F(x_n) - F(r)| \le c|x_n - r| = c\,|e_n|.$$

In particular, we get $|e_n| \le c^n\,|e_0|$, thus $e_n \to 0$, i.e. $x_n \to r$. □

Theorem 13.2. Let r be a fixed point of the function $F \in \mathcal{C}^1[a, b]$. If $|F'(r)| < 1$, then r is an **attractive** fixed point, in the sense that there exists $\delta > 0$ such that the sequence $(F^n(x_0))$ converges to r for all x_0 chosen in $[r - \delta, r + \delta]$. Moreover, the convergence is **linear**, meaning that $|x_{n+1} - r| \le c\,|x_n - r|$ for some constant $0 \le c < 1$.

Proof. Suppose that $|F'(r)| < 1$. Then, by continuity of F' there exists $\delta > 0$ such that $|F'(x)| \le (|F'(r)| + 1)/2 =: c < 1$ for all $x \in [r - \delta, r + \delta]$. Note that F maps $[r - \delta, r + \delta]$ into itself, so that the previous theorem applies. □

It is instructive to illustrate this phenomenon on the graph of the function F.

At this stage, one could already show that Newton's method converges at a linear rate, provided that the starting point is 'close enough' to a zero r of f. There is actually more to it, which comes as no surprise if $F'(r)$ is explicitely computed. We get

$$F'(r) = \frac{d}{dx}\left[x - \frac{f(x)}{f'(x)}\right]_{|x=r} = \left[1 - \frac{f'(x)f'(x) - f''(x)f(x)}{f'(x)^2}\right]_{|x=r} = 1 - (1-0) = 0.$$

Theorem 13.3. Suppose that f belongs to $\mathcal{C}^2[a,b]$ and that r is a **simple zero** of f [i.e. $f(r) = 0$, $f'(r) \neq 0$]. Then there exists $\delta > 0$ such that, for any $x_0 \in [r-\delta, r+\delta]$, the sequence defined by (13.1) converges to r. Moreover, the convergence is **quadratic**, in the sense that the error $e_n = x_n - r$ satisfies $|e_{n+1}| \leq C\, e_n^2$ for some constant C.

Proof. We already know that the sequence (x_n) lies in $[r-\delta, r+\delta]$ for some $\delta > 0$. We may, if necessary, decrease the value of δ to insure the continuity of f'' and $1/f'$ on $[r-\delta, r+\delta]$, and we set $D := \max_{x \in [r-\delta, r+\delta]} |f''(x)|$ and $d := \min_{x \in [r-\delta, r+\delta]} |f'(x)|$. We may also redude δ so that $C := \frac{D}{2d} < \frac{1}{\delta}$. We then write

$$\begin{aligned} e_{n+1} &= x_{n+1} - r = x_n - \frac{f(x_n)}{f'(x_n)} - r = -\frac{1}{f'(x_n)}\big[f(x_n) + (r - x_n)f'(x_n)\big] \\ &= -\frac{1}{f'(x_n)}\big[f(r) - \frac{1}{2}f''(y_n)(r-x_n)^2\big] \qquad \text{for some } y_n \text{ between } r \text{ and } x_n, \end{aligned}$$

and we obtain

$$|e_{n+1}| = \frac{1}{2}\frac{|f''(y_n)|}{|f'(x_n)|}\, e_n^2 \leq \frac{D}{2d}\, e_n^2 = C\, e_n^2.$$

This establishes the quadratic convergence. □

13.3 Generalized setting

The same strategy [linearize and solve] may be used to find numerical solutions of systems of nonlinear equations. Let us illustrate this point with the system

$$\begin{cases} f_1(x_1,x_2) & = & 0, \\ f_2(x_1,x_2) & = & 0. \end{cases}$$

If an approximation $(x_1,x_2) = (r_1+h_1, r_2+h_2)$ is close to a solution (r_1,r_2) of this system, Taylor expansion in two variables reads

$$\begin{cases} 0 & = & f_1(x_1-h_1,x_2-h_2) & \approx & f_1(x_1,x_2) - h_1\dfrac{\partial f_1}{\partial x_1}(x_1,x_2) - h_2\dfrac{\partial f_1}{\partial x_2}(x_1,x_2), \\ & & & & \\ 0 & = & f_2(x_1-h_1,x_2-h_2) & \approx & f_2(x_1,x_2) - h_1\dfrac{\partial f_2}{\partial x_1}(x_1,x_2) - h_2\dfrac{\partial f_2}{\partial x_2}(x_1,x_2). \end{cases}$$

We solve this 2×2 system of linear equations with unknowns (h_1,h_2), in the expectation that (x_1-h_1,x_2-h_2) provides a better approximation to (r_1,r_2). Introducing the **Jacobian matrix** of f_1 and f_2,

$$J := \begin{bmatrix} \dfrac{\partial f_1}{\partial x_1} & \dfrac{\partial f_1}{\partial x_2} \\ & \\ \dfrac{\partial f_2}{\partial x_1} & \dfrac{\partial f_2}{\partial x_2} \end{bmatrix},$$

Newton's method is merely the construction of a sequence $(x_1^{(n)}, x_2^{(n)})$ according to

$$\begin{bmatrix} x_1^{(n+1)} \\ x_2^{(n+1)} \end{bmatrix} = \begin{bmatrix} x_1^{(n)} \\ x_2^{(n)} \end{bmatrix} - \left[J(x_1^{(n)}, x_2^{(n)})\right]^{-1} \begin{bmatrix} f_1(x_1^{(n)}, x_2^{(n)}) \\ f_2(x_1^{(n)}, x_2^{(n)}) \end{bmatrix}.$$

Clearly, one can deal with larger systems following this very model. It will be convenient to use a matrix-vector formalism in this situation: express the system as $F(X) = 0$, with $F = [f_1,\ldots,f_n]^\top$ and $X = [x_1,\ldots,x_n]^\top$; linearize in the form $F(X+H) \approx F(X) + F'(X)H$, where $F'(X)$ represents an $n\times n$ Jacobian matrix. Of course, the $n\times n$ linear systems will be solved by using the methods already studied, not by determining inverses.

13.4 Exercises

From the textbook: 7 p 61; 18, 19 p 62; 2, 6.b.d.f p 71; 25, 27 p 73.

1. Fill in the details of the proofs in Section 13.2.
2. Let Newton's method be used on $f(x) = x^2 - y$, $y > 0$. Show that if x_n has k correct digits after the decimal point, then x_{n+1} will have at least $2k - 1$ correct digits after the decimal point, provided that $y > 0.0006$.
3. What is the purpose of the iteration $x_{n+1} = 2x_n - x_n^2 y$? Identify it as the Newton iteration of a certain function.
4. The polynomial $x^3 + 94x^2 - 389x + 294$ has zeros at 1, 3, and -98. The point $x_0 = 2$ should be a good starting point for computing one of the small zeros by Newton iteration. Is it really the case?

Optional problems

From the textbook: 34 p 74.

1. **Steffensen's method** follows the iteration formula

$$x_{n+1} = x_n - \frac{f(x_n)}{g(x_n)}. \qquad \text{where} \qquad g(x) := \frac{f(x + f(x)) - f(x)}{f(x)}.$$

   Show that it is quadratically convergent, under suitable hypotheses. Program this method. Compare it with Newton's method, e.g. to solve the equation $x^3 + 3x = 5x^2 + 7$ in 10 steps, starting at $x_0 = 5$.

Chapter 14

Secant method

14.1 Description of the method

One of the drawbacks of Newton's method is that it necessitates the computation of the derivative of a function. To overcome this difficulty, we can approximate $f'(x_n)$ appearing in the iteration formula by a quantity which is easier to compute. We can think of using

$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}.$$

The resulting algorithm is called the **secant method**. It is based on the iteration formula

$$x_{n+1} = x_n - \frac{f(x_n)\,(x_n - x_{n-1})}{f(x_n) - f(x_{n-1})},$$

which requires two initial points x_0 and x_1 to be set off. However, only one new evaluation of f is necessary at each step. The graphical interpretation for the secant method does not differ much from the one for Newton's method – simply replace 'tangent line' by 'secant line'.

The pseudocode for the secant method follows the one for Newton's method without too much modifications.

input a, b, M, δ, ε
$u \leftarrow f(a)$, **output** 0, a, u
if $|u| < \varepsilon$ **then stop end if**
$v \leftarrow f(b)$, **output** 1, b, v
if $|v| < \varepsilon$ **then stop end if**
for $k = 2$ to M **do**
    $s \leftarrow (b-a)/(v-u), \quad a \leftarrow b, \quad u \leftarrow v,$
    $b \leftarrow b - vs, \quad v \leftarrow f(b)$
    output k, b, v
    if $|a-b| < \delta$ or $|v| < \varepsilon$ **then stop end if**
end do end for

14.2 Convergence analysis

The error analysis that we present, although somehow lacking rigor, provides the rate of convergence for the secant method. The error at the $(n+1)$-st step is estimated by

$$\begin{aligned} e_{n+1} &= x_{n+1} - r = x_n - \frac{f(x_n)(x_n - x_{n-1})}{f(x_n) - f(x_{n-1})} - r \\ &= \frac{f(x_n)(x_{n-1} - r) - f(x_{n-1})(x_n - r)}{f(x_n) - f(x_{n-1})} = e_n\, e_{n-1} \frac{f(x_n)/e_n - f(x_{n-1})/e_{n-1}}{f(x_n) - f(x_{n-1})} \\ &\approx e_n\, e_{n-1} \frac{[f(r) + f'(r)e_n + f''(r)/2\; e_n^2]/e_n - [f(r) + f'(r)e_{n-1} + f''(r)/2\; e_{n-1}^2]/e_{n-1}}{[f(r) + f'(r)e_n] - [f(r) + f'(r)e_{n-1}]} \\ &\approx e_n\, e_{n-1} \frac{f''(r)/2\;(e_n - e_{n-1})}{f'(r)(e_n - e_{n-1})}. \end{aligned}$$

Thus, we have obtained

$$e_{n+1} \approx C\, e_n\, e_{n-1}, \qquad \text{where} \quad C := \frac{f''(r)}{2f'(r)}.$$

We postulate [informed guess, see Chapter 24] the relation $|e_{n+1}| \sim A|e_n|^\alpha$. This can also be written as $|e_n| \sim A^{-1/\alpha}|e_{n+1}|^{1/\alpha}$, which we use with n replaced by $n-1$ to derive

$$A|e_n|^\alpha \sim C A^{-1/\alpha} |e_n|^{1+1/\alpha}.$$

We should therefore have $\alpha = 1 + 1/\alpha$, or $\alpha = \dfrac{1+\sqrt{5}}{2} \approx 1.62$. We should also have $A = C^{1/(1+1/\alpha)} = C^{1/\alpha} = C^{\alpha-1} \approx \left[\dfrac{f''(r)}{2f'(r)}\right]^{0.62}$. Finally, with A thus given, we have established

$$|e_{n+1}| \approx A\,|e_n|^{(1+\sqrt{5})/2}.$$

The rate of convergence of the secant method is seen to be **superlinear**, which is better than the bisection method [linear rate], but not as good of Newton's method [quadratic rate]. However, each step of the secant method requires only one new function evaluation, instead of two for Newton's method. Hence, it is more appropriate to compare a pair of steps of the former with one step of the latter. In this case, we get

$$|e_{n+2}| \sim A|e_{n+1}|^{\alpha} \sim A^{1+\alpha}|e_n|^{\alpha^2} = C^{\alpha}|e_n|^{\alpha+1} \approx C^{1.62}|e_n|^{2.62},$$

which now appears better than the quadratic convergence of Newton's method.

14.3 Exercises

From the textbook: 8.b.d.f. p 71; 17.b.c. p 72.

1. If $x_{n+1} = x_n + (2 - e^{x_n})(x_n - x_{n-1})/(e^{x_n} - e^{x_{n-1}})$ with $x_0 = 0$ and $x_1 = 1$, what is $\lim_{n\to\infty} x_n$?

Part IV

Approximation of functions

Chapter 15

Polynomial interpolation

15.1 The interpolation problem

Consider some points $x_0 < x_1 < \cdots < x_n$ and some data values $y_0, y_1, \ldots, y_n$. We seek a function f, called an **interpolant** of $y_0, y_1, \ldots, y_n$ at $x_0, x_1, \ldots, x_n$, which satisfies

$$f(x_i) = y_i, \qquad i \in [\![0, n]\!].$$

This means that the f-curve intercept the $n+1$ points $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$. Since we prescribe $n+1$ conditions, we should allow $n+1$ degrees of freedom for the function f, and it should also be a 'simple' function. Hence, we will seek f in the linear space

$$\mathcal{P}_n := \{p : \ p \text{ is a polynomial of degree} \leq n\}.$$

In this situation, finding an interpolant is always possible.

Theorem 15.1. Fix $x_0 < x_1 < \cdots < x_n$. For any values $y_0, y_1, \ldots, y_n$, there exists a unique polynomial of degree at most n that interpolates $y_0, y_1, \ldots, y_n$ at $x_0, x_1, \ldots, x_n$ – in short,

$$\exists\,!\, p \in \mathcal{P}_n : \quad p(x_i) = y_i, \quad i \in [\![0, n]\!].$$

Proof. Define the linear map

$$T : p \in \mathcal{P}_n \mapsto \big(p(x_0), p(x_1), \ldots, p(x_n)\big) \in \mathbb{R}^{n+1}.$$

Our aim is to prove that T is bijective [check this claim]. Since $\dim \mathcal{P}_n = \dim \mathbb{R}^{n+1} = n+1$, it is enough to show either that T is surjective, or that T is injective. To prove the latter, consider $p \in \ker T$. The polynomial p, of degree at most n, possesses $n+1$ zeros, hence it must be the zero polynomial. This establishes the injectivity. $\square$

15.2 The Lagrange form of the polynomial interpolant

The previous argument is of no practical use, for it provides neither an explicit expression for the interpolant nor an algorithm to compute it. Note, however, that for each $i \in [\![0, n]\!]$, there is a unique polynomial $\ell_i \in \mathcal{P}_n$, called i-th **Lagrange cardinal polynomial** relative to $x_0, x_1, \ldots, x_n$, such that

$$\ell_i(x_j) = 0 \quad \text{for } i \neq j, \quad \text{and} \quad \ell_i(x_i) = 1.$$

One readily checks that the interpolant of $y_0, y_1, \ldots, y_n$ at $x_0, x_1, \ldots, x_n$ takes the form

$$p(x) = y_0\ell_0(x) + y_1\ell_1(x) + \cdots + y_n\ell_n(x) = \sum_{i=0}^{n} y_i\ell_i(x).$$

An explicit expression for the Lagrange cardinal polynomial is given by

$$\ell_i(x) = \frac{(x - x_0)\cdots(x - x_{i-1})(x - x_{i+1})\cdots(x - x_n)}{(x_i - x_0)\cdots(x_i - x_{i-1})(x_i - x_{i+1})\cdots(x_i - x_n)} = \prod_{j=0,\, j\neq i}^{n} \frac{x - x_j}{x_i - x_j}.$$

Given a function f, we will say that a polynomial p of degree at most n interpolates f at $x_0, x_1, \ldots, x_n$ if it interpolates the value $f(x_0), f(x_1), \ldots, f(x_n)$ at $x_0, x_1, \ldots, x_n$. Observe that it can be written as $p = \sum_i f(x_i)\ell_i$. In particular, since a polynomial of degree at most n is its own interpolant at $x_0, x_1, \ldots, x_n$, we have the representation

$$p = \sum_{i=0}^{n} p(x_i)\ell_i \qquad \text{for any } p \in \mathcal{P}_n.$$

As an example, we may write $2x^2 - 1 = \dfrac{x(x-1)}{2} + (x-1)(x+1) + \dfrac{(x+1)x}{2}$.

15.3 The error in polynomial interpolation

Suppose that only a sample $y_0 = f(x_0), y_1 = f(x_1), \ldots, y_n = f(x_n)$ of values of a function f are known. In order to manipulate f, we often approximate it by its polynomial interpolant. The interpolant, depending only on the finite data set, cannot provide a good approximant for all the functions that intercept the points $(x_0, y_0), \ldots, (x_n, y_n)$. Nevertheless, if the function f is 'nice enough', the error can be kept under control.

Theorem 15.2. Let $f \in \mathcal{C}^{n+1}[a, b]$ – f is $n + 1$ times differentiable and $f^{(n+1)}$ is continuous – and let $p \in \mathcal{P}_n$ be the interpolant of f at $x_0, \ldots, x_n \in [a, b]$. For each $x \in [a, b]$, there exists $\xi \in [a, b]$ – depending on x – such that

$$f(x) - p(x) = \frac{1}{(n+1)!} f^{(n+1)}(\xi) \prod_{i=0}^{n} (x - x_i). \tag{15.1}$$

Proof. Formula (15.1) is obvious if x is one of the x_i's. Suppose now that $x \in [a,b]$ is distinct from all the x_i's, and consider it as a fixed parameter. Then the function Φ defined by

$$\Phi(t) := [f(t) - p(t)] \prod_{i=0}^{n} (x - x_i) \; - \; [f(x) - p(x)] \prod_{i=0}^{n} (t - x_i)$$

vanishes [i.e. equals zero] at the $n+2$ distinct points $x, x_0, \ldots, x_n$. Then, using Rolle's theorem, we derive that Φ' vanishes at $n+1$ distinct points [at least]. Using Rolle's theorem once more, we see that Φ'' vanishes at n distinct points. Continuing in this fashion, we deduce that $\Phi^{(n+1)}$ vanishes at a point $\xi \in [a,b]$. Note that

$$0 = \Phi^{(n+1)}(\xi) = [f^{(n+1)}(\xi) - p^{(n+1)}(\xi)] \prod_{i=0}^{n} (x - x_i) \; - \; [f(x) - p(x)] \, \frac{d^{n+1}}{dt^{n+1}} \left(\prod_{i=0}^{n} (t - x_i) \right)_{|t=\xi}.$$

In view of $p^{(n+1)} \equiv 0$ and of $\frac{d^{n+1}}{dt^{n+1}} \Big(\prod_{i=0}^{n} (t - x_i) \Big) = \frac{d^{n+1}}{dt^{n+1}} \Big(t^{n+1} + \{\text{degree} \le n\} \Big) = (n+1)!$, this implies that

$$0 = f^{(n+1)}(\xi) \prod_{i=0}^{n} (x - x_i) \; - \; [f(x) - p(x)] \, (n+1)!,$$

which is just another way of writing (15.1). This proof is now complete. Another one will be given in the next chapter. □

A natural way to represent a polynomial of degree at most n is

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n.$$

The interpolation conditions $p(x_i) = y_i$, $i \in [\![0, n]\!]$, translates into the linear system

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & & \cdots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix},$$

whose unknowns are $a_0, a_1, \ldots, a_n$. This system is always solvable, as we have seen, hence the coefficient matrix – called a **Vandermonde** matrix – is nonsingular. Its determinant is therefore nonzero [provided that the x_i's are distinct] and can in fact be determined explicitely. However, finding a polynomial interpolant using this approach is not recommended.

15.4 Exercises

From the textbook: 12, 13 p 116.

1. Prove that $\sum_{i=0}^{n} \ell_i(x) = 1$ for all x.

2. Suppose that the function values $f(0)$, $f(1)$, $f(2)$ and $f(3)$ are given. We wish to estimate $f(6)$, $f'(0)$ and $\int_0^3 f(x)dx$ by employing the approximants $p(6)$, $p'(0)$ and $\int_0^3 p(x)dx$, where p is the cubic polynomial interpolating f at 0, 1, 2, and 3. Deduce from the Lagrange formula that each approximant is a linear combination of the four data with coefficients independent of f. Calculate the numerical values of the coefficients. Verify your work by showing that the approximants are exact when f is an arbitrary cubic polynomial.

3. Let f be a function in $\mathcal{C}^4[0,1]$ and let p be a cubic polynomial satisfying $p(0) = f(0)$, $p'(0) = f'(0)$, $p(1) = f(1)$, and $p'(1) = f'(1)$. Deduce from the Rolle's theorem that for every $x \in [0,1]$, there exists $\xi \in [0,1]$ such that

$$f(x) - p(x) = \frac{1}{24} x^2 (x-1)^2 f^{(4)}(\xi).$$

Optional problems

From the textbook: 33 p118.

1. Let a, b and c be distinct real numbers [not necessarily in ascending order], and let $f(a)$, $f(b)$, $f'(a)$, $f'(b)$ and $f'(c)$ be given. Because there are five data, one might try to approximate f by a polynomial of degree at most four that interpolates the data. Prove by a general argument that this interpolation problem has a solution and the solution is unique if and only if there is no nonzero polynomial $p \in \mathcal{P}_4$ that satisfies $p(a) = p(b) = p'(a) = p'(b) = p'(c) = 0$. Hence, given a and b, show that there exists a unique value $c \neq a, b$ such that there is no unique solution.

2. Try to find the explicit expression for the determinant of a Vandermonde matrix.

Chapter 16

Divided differences

16.1 A definition

Given distinct points $x_0,\, x_1,\, \ldots,\, x_n$ and given a function f, the **divided difference** of f at $x_0,\, x_1,\, \ldots,\, x_n$ is defined to be the coefficient of x^n in the polynomial $p \in \mathcal{P}_n$ interpolating f at $x_0,\, x_1,\, \ldots,\, x_n$. It is denoted by $[x_0, \ldots, x_n]f$ – or by $f[x_0, \ldots, x_n]$ in many texts. Lagrange formula can be called upon to derive the representation [check it]

$$[x_0, \ldots, x_n]f = \sum_{i=0}^{n} f(x_i) \prod_{j=0,\, j \neq i}^{n} \frac{1}{x_i - x_j}.$$

However, this expression is rarely used in practice, since divided differences can be calculated in a more efficient way – see Section 16.2. We may already observe that

$$\begin{aligned} [x_0]f &= f(x_0) \\ [x_0, x_1]f &= \frac{f(x_1) - f(x_0)}{x_1 - x_0}. \end{aligned}$$

Note that, when x_1 is close to x_0, the divided difference $[x_0, x_1]f$ approximate $f'(x_0)$. In fact, we show that higher derivatives can also be approximated by divided differences, which present the advantage of being easily computed – see Section 16.2.

Theorem 16.1. Let $f \in \mathcal{C}^n[a,b]$ and let $x_0,\, x_1,\, \ldots,\, x_n$ be distinct points in $[a,b]$. There exists a point ξ between the smallest and the largest of the x_i's such that

$$[x_0, \ldots, x_n]f = \frac{1}{n!} f^{(n)}(\xi).$$

Proof. Let $p \in \mathcal{P}_n$ be the polynomial interpolating f at the points $x_0,\, x_1,\, \ldots,\, x_n$. The error $f - p$ has $n+1$ zeros in $[\min_i x_i, \max_i x_i]$. We apply Rolle's theorem n times to deduce that

$f^{(n)} - p^{(n)}$ has a zero ξ in $[\min_i x_i, \max_i x_i]$, thus $f^{(n)}(\xi) = p^{(n)}(\xi)$. It remains to remark that $p^{(n)}(\xi) = n!\,[x_0, \ldots, x_n]f$, since $p(x) = [x_0, \ldots, x_n]f\, x^n + \{\text{degree} < n\}$. □

If we combine this theorem with the next one, we obtain the alternative proof of Theorem 15.2 previously mentioned.

Theorem 16.2. For a function f, let $p \in \mathcal{P}_n$ be the interpolant of f at the distinct points $x_0, x_1, \ldots, x_n$. If x is not one of the x_i's, then

$$f(x) - p(x) = [x_0, \ldots, x_n, x]f \prod_{i=0}^{n}(x - x_i).$$

Proof. Let $q \in \mathcal{P}_{n+1}$ be the interpolant of f at the points $x_0, x_1, \ldots, x_n$, and x. Note that the difference $q - p$ is a polynomial of degree at most $n+1$ which vanishes at $x_0, x_1, \ldots, x_n$. It is therefore of the type

$$q(t) - p(t) = c \prod_{i=0}^{n}(t - x_i), \qquad \text{for some constant } c. \tag{16.1}$$

Identifying the coefficients of t^{n+1}, we obtain $c = [x_0, \ldots, x_n, x]f$. It now remains to write (16.1) for $t = x$, keeping in mind that $q(x) = f(x)$. □

16.2 Recurrence relation

The recursive method we are about describe allows for fast computation of divided differences [and also accounts for their names].

Theorem 16.3. Given distinct points $x_0, x_1, \ldots, x_k, x_{k+1}$, there holds

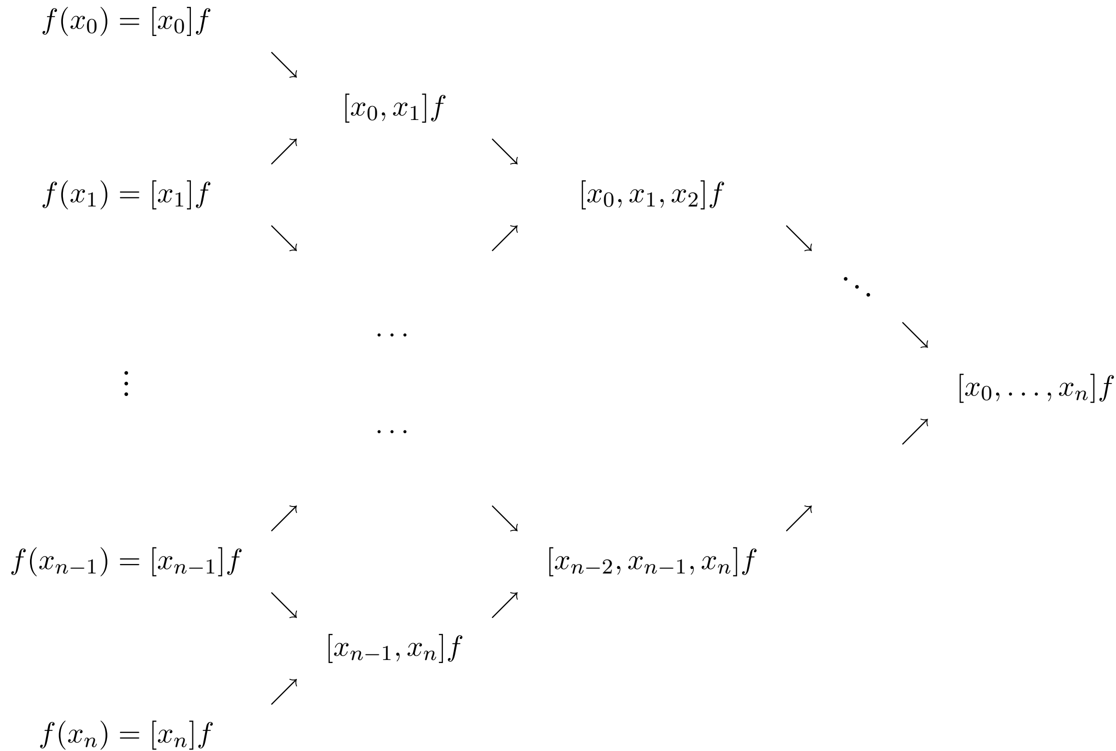
$$[x_0, x_1, \ldots, x_k, x_{k+1}]f = \frac{[x_1, \ldots, x_k, x_{k+1}]f - [x_0, x_1, \ldots, x_k]f}{x_{k+1} - x_0}.$$

Proof. Let $p, q \in \mathcal{P}_k$ be the interpolant of f at $x_0, x_1, \ldots, x_k$ and at $x_1, \ldots, x_k, x_{k+1}$, respectively. Define the polynomial $r \in \mathcal{P}_{k+1}$ by

$$r(x) := \frac{1}{x_{k+1} - x_0}\big((x - x_0)q(x) + (x_{k+1} - x)p(x)\big). \tag{16.2}$$

It is easy to check that $r(x_i) = f(x_i)$ for all $i \in [\![0, k+1]\!]$. Thus, the coefficient of x^{k+1} in r equals $[x_0, \ldots, x_{k+1}]f$. But, looking at the right-hand side of (16.2), we see that it also equals $\dfrac{1}{x_{k+1} - x_0}\big([x_1, \ldots, x_k, x_{k+1}]f - [x_0, x_1, \ldots, x_k]f\big)$. □

The evaluation of the **divided difference table** follows the diagram



16.3 The Newton form of the polynomial interpolant

To prove the existence of a polynomial $p \in \mathcal{P}_n$ interpolating f at $x_0, x_1, \ldots, x_n$, we could have proceeded inductively. Indeed, if we have already computed a polynomial $p \in \mathcal{P}_k$ interpolating f at $x_0, x_1, \ldots, x_k$, we can seek for a polynomial $p \in \mathcal{P}_{k+1}$ interpolating f at $x_0, x_1, \ldots, x_{k+1}$ in the form $p_{k+1}(x) = p_k(x) + c\,(x - x_0) \cdots (x - x_k)$. The constant c is uniquely determined by the condition $p_{k+1}(x_{k+1}) = f(x_{k+1})$. It can be expressed in terms of divided differences. This is the process that lies beneath the Newton interpolation formula, which is computationally more efficient than the Lagrange formula.

Theorem 16.4. The polynomial $p \in \mathcal{P}_n$ interpolating a function f at the distinct points $x_0, x_1, \ldots, x_n$ can be written as

$$\begin{aligned} p(x) &= [x_0]f + [x_0, x_1]f \cdot (x - x_0) + \cdots + [x_0, \ldots, x_n]f \cdot (x - x_0) \cdots (x - x_{n-1}) \\ &= \sum_{k=0}^{n} [x_0, \ldots, x_k]f \prod_{i=0}^{k-1} (x - x_i). \end{aligned}$$

Proof. For each $k \in [\![0, n]\!]$, let $p_k \in \mathcal{P}_k$ be the interpolant of f at $x_0, x_1, \ldots, x_k$. Remark, as in (16.1), that

$$p_{k+1}(x) - p_k(x) = c_k\,(x - x_0)\cdots(x - x_k), \qquad \text{for some constant } c_k.$$

Looking at the coefficient of x^{k+1}, we see that $c_k = [x_0, \ldots, x_{k+1}]f$. We then get

$$\begin{aligned} p(x) &= p_0(x) + \big(p_1(x) - p_0(x)\big) + \cdots + \big(p_{n-1}(x) - p_{n-2}(x)\big) + \big(p_n(x) - p_{n-1}(x)\big) \\ &= f(x_0) + c_0\,(x - x_0) + \cdots + c_{n-1}\,(x - x_0)\cdots(x - x_{n-1}) \\ &= [x_0]f + [x_0, x_1]f \cdot (x - x_0) + \cdots + [x_0, \ldots, x_n]f \cdot (x - x_0)\cdots(x - x_{n-1}), \end{aligned}$$

which is the required result. □

The evaluation of $p(x)$ should be performed using nested multiplication, i.e. according to the scheme implied by the writing

$$p(x) = [x_0]f + (x - x_0)\Big\{[x_0, x_1]f + (x - x_1)\Big\{[x_0, x_1, x_2]f + \cdots + (x - x_{n-1})\Big\{[x_0, \ldots, x_n]f\Big\}\cdots\Big\}\Big\}.$$

Observe that we only need the divided differences obtained in the top south–east diagonal of the table. They can be computed via the following algorithm designed to use few storage space. Start with a vector $d = (d_0, d_1, \ldots, d_n)$ containing the values $f(x_0), f(x_1), \ldots, f(x_n)$. Note that d_0 already provides the first desired coefficient. Then compute the second column of the table, putting the corresponding divided differences in positions of $d_1, \ldots, d_n$, so that d_1 provides the second desired coefficient. Continue in this pattern, being careful to store the new elements in the bottom part of the vector d without disturbing its top part. Here is the algorithm.

for $k = 0$ to n **do** $d_k \leftarrow f(x_k)$ **end do end for**
for $j = 1$ to n **do**
    for $i = n$ downto j **do**
    $d_i \leftarrow (d_i - d_{i-1})/(x_i - x_{i-j})$
    end do end for
end do end for

Once this is done, we can use the following version of Horner's algorithm to compute the interpolant at the point x.

$u \leftarrow d_n$
for $i = n - 1$ downto 0 **do**
$u \leftarrow (x - x_i)u + d_i$
end do end for

16.4 Exercises

From the textbook: 17, 19 p 129.

1. Prove that if f is a polynomial of degree k, then $[x_0, \ldots, x_n]f = 0$ for all $n > k$.
2. Find the Newton form of the cubic polynomial interpolating the data 63, 11, 7, and 28 at the points 4, 2, 0, and 3.

Optional problem

1. Prove the Leibniz formula for divided differences, which reads

$$[x_0, x_1, \ldots, x_n](fg) = \sum_{k=0}^{n} [x_0, \ldots, x_k]f \cdot [x_k, \ldots, x_n]g.$$

Chapter 17

Orthogonal polynomials

17.1 Inner product

Let w be a **weight function** on $[-1,1]$, say, i.e.

w integrable on $[-1,1]$, continuous on $(-1,1)$, $w(x)>0$ for all $x \in [-1,1] \setminus \mathcal{Z}$, $\mathcal{Z}$ a finite set.

The general expression

$$\langle f, g \rangle := \int_{-1}^{1} f(x) g(x) w(x) dx, \qquad f, g \in \mathcal{C}[-1,1],$$

defines an **inner product** on $\mathcal{C}[-1,1]$, which means that it shares the key properties of the usual inner product on $\mathbb{R}^n$, namely

1. symmetry: for $f, g \in \mathcal{C}[-1,1]$, there holds $\langle f, g \rangle = \langle g, f \rangle$,
2. linearity: for $f_1, f_2, g \in \mathcal{C}[-1,1],\ a, b \in \mathbb{R}$, there holds $\langle a f_1 + b f_2, g \rangle = a\langle f_1, g \rangle + b\langle f_2, g \rangle$,
3. positivity: for $f \in \mathcal{C}[-1,1]$, there holds $\langle f, f \rangle \geq 0$, with equality if and only if $f = 0$.

Let us justify the last statement, i.e. let us show that $f = 0$ as soon as $f \in \mathcal{C}[-1,1]$ satisfies $\langle f, f \rangle = 0$. The continuous function $g := f^2 w$ is nonnegative and has an integral equal to zero, hence must be identically zero. Therefore $f(x) = 0$ for all $x \in [-1,1] \setminus \mathcal{Z}$. But then, since $\mathcal{Z}$ is a finite set, the continuity of f forces f to vanish on $\mathcal{Z}$ as well.

Note that the expression $\|f\| := \sqrt{\langle f, f \rangle}$ defines a norm on $\mathcal{C}[-1,1]$.

17.2 Orthogonal polynomials for a general weight

By analogy with more familiar inner products, we will say that two functions f and g are **orthogonal** if $\langle f, g\rangle = 0$. We are interested in a sequence (p_n) of orthogonal polynomials of degree n, in the sense that we require

$$\deg p_n = n, \qquad \langle p_n, p_m\rangle = 0, \quad n \neq m.$$

Note that p_n should be orthogonal to all p_m with $m < n$, hence to every $p \in \mathcal{P}_{n-1}$. We call p_n 'the' n**-th orthogonal polynomial**. It is only defined up to a multiplicative constant. To define it uniquely, we need a normalization condition, e.g. we may require p_n to be **monic**, i.e. to have a leading coefficient equal to one.

Theorem 17.1. For every weight function w on $[-1, 1]$, there exists a unique n-th monic orthogonal polynomial.

Proof. Let us start by the uniqueness part. Consider two monic polynomials p and q which are orthogonal to the space $\mathcal{P}_{n-1}$. Then the difference $p - q$ is also orthogonal to $\mathcal{P}_{n-1}$. It also belongs to $\mathcal{P}_{n-1}$, which implies that $p - q = 0$, or $p = q$.
The existence part is proved by induction on n. For $n = 0$, we take $p_0 = 1$. Then, for $n \geq 1$, suppose that $p_0, p_1, \ldots, p_{n-1}$ have been constructed, and let us construct p_n. Inspired by the Gram–Schmidt algorithm, we define

$$p_n = q - \sum_{k=0}^{n-1} \frac{\langle q, p_k\rangle}{\langle p_k, p_k\rangle} p_k, \qquad \text{where } q(x) := x^n.$$

It is readily checked that $p_n \in \mathcal{P}_n$ is monic and that $\langle p_n, p_m\rangle = 0$ for all $m < n$. Hence p_n is indeed the n-th monic orthogonal polynomial. This concludes the induction. □

As an example, we may consider the **Chebyshev polynomials** T_n defined by the relation [in fact, that T_n is a polynomial of degree n is not apparent here]

$$T_n(\cos\theta) = \cos(n\theta), \quad \theta \in [0, \pi], \qquad \text{or equivalently} \quad T_n(x) = \cos(n \arccos(x)), \quad x \in [-1, 1].$$

They are orthogonal [but not monic] on $[-1, 1]$ with respect to the weight $w(x) = \dfrac{1}{\sqrt{1-x^2}}$, as seen from

$$\begin{aligned}\int_{-1}^{1} T_n(x) T_m(x) \frac{dx}{\sqrt{1-x^2}} \underset{x=\cos\theta}{=} & \int_0^\pi T_n(\cos\theta) T_m(\cos\theta) \frac{\sin\theta d\theta}{\sqrt{1-\cos^2\theta}} = \int_0^\pi \cos(n\theta)\cos(m\theta) d\theta \\ = & \frac{1}{2}\int_0^\pi [\cos((n+m)\theta) + \cos((n-m)\theta)] = 0 \text{ if } n \neq m.\end{aligned}$$

If one is adverse to trigonometric formulae, one can also integrate $I := \int_0^\pi \cos(n\theta)\cos(m\theta)d\theta$ by parts twice to obtain $I = m^2 I/n^2$, thus $I = 0$ when $n \neq m$. Note that T_n has n zeros in $(-1,1)$ and $n+1$ equioscillation points in $[-1,1]$, precisely

$$T_n\Big(\cos\Big(\frac{(2n+1-2k)\pi}{2n}\Big)\Big) = 0, \quad k \in [\![1,n]\!], \qquad T_n\Big(\cos\Big(\frac{(n-k)\pi}{n}\Big)\Big) = (-1)^{n-k}, \quad k \in [\![0,n]\!].$$

Other important examples of orthogonal polynomials include the **Legendre polynomials**, which are orthogonal on $[-1,1]$ with respect to the weight $w(x) = 1$. They are defined by

$$P_n(x) = \frac{1}{2^n\, n!}\frac{d^n}{dx^n}\big[(x^2-1)^n\big].$$

One can check that this formula yields a polynomial of degree n, which is not monic but normalized so that $P_n(1) = 1$. A direct calculation would give e.g. $P_2(x) = \frac{3}{2}x^2 - \frac{1}{2}$. It can also be seen [Rolle's theorem being involved] that P_n has n zeros in $(-1,1)$. This property is shared by all n-th orthogomal polynomials, regardless of the weight w.

Theorem 17.2. Any n-th orthogonal polynomial has n distinct zeros inside $(-1,1)$.

Proof. Let p be a polynomial of degree n which is orthogonal to the space $\mathcal{P}_{n-1}$. Denote by $x_1, \ldots, x_m$ the zeros of p where p changes sign – hence the endpoints -1 and 1 are not taken into account – and define $q(x) = (x - x_1)\cdots(x - x_m)$. Then the function $p(x)q(x)w(x)$ does not change sign on $[-1,1]$, so its integral cannot vanish. Thus q cannot be a polynomial of degree $\leq n-1$, i.e. $m = \deg q \geq n$. We have shown that p has a least m zeros in $(-1,1)$, hence it has exactly n zeros. □

17.3 Three-term recurrence relation

For orthogonal polynomial to have any practical use, we need to be able to compute them. The Gram–Schmidt process provides a way, of course, but we know that it is prone to loss of accuracy. It turns out that a considerably better procedure is available.

Theorem 17.3. Monic orthogonal polynomials can be constucted recursively, according to

$$p_{-1}(x) = 0, \quad p_0(x) = 1, \qquad \text{and for } n \geq 0,$$

$$p_{n+1}(x) = (x - \alpha_n)p_n(x) - \beta_n p_{n-1}(x), \quad \text{with } \alpha_n = \frac{\langle p_n, xp_n\rangle}{\langle p_n, p_n\rangle} \quad \text{and } \beta_n = \frac{\langle p_n, p_n\rangle}{\langle p_{n-1}, p_{n-1}\rangle}.$$

Proof. The proper proof would be done by induction. Here we simply determine what the recurrence relation must be like. Since the polynomial $xp_n(x)$ belongs to $\mathcal{P}_{n+1}$, it has [should have] an expansion of the form

$$xp_n = a_{n+1}p_{n+1} + a_n p_n + a_{n-1}p_{n-1} + \cdots + a_0 p_0. \tag{17.1}$$

Note that a_{n+1} must equal one for the leading coefficients on both sides to be the same. If we take the inner product with p_k for $k < n-1$, we obtain

$$a_k \langle p_k, p_k \rangle = \langle xp_n, p_k \rangle = \langle p_n, \underbrace{xp_k}_{\in \mathcal{P}_{n-1}} \rangle = 0,$$

hence $a_k = 0$, as expected. Now, taking the inner product with p_n, we get

$$a_n \langle p_n, p_n \rangle = \langle xp_n, p_n \rangle, \qquad \text{or} \qquad a_n = \frac{\langle xp_n, p_n \rangle}{\langle p_n, p_n \rangle}.$$

Finally, taking the inner product with p_{n-1}, we derive

$$a_{n-1} \langle p_{n-1}, p_{n-1} \rangle = \langle xp_n, p_{n-1} \rangle, \qquad \text{or} \qquad a_{n-1} = \frac{\langle xp_n, p_{n-1} \rangle}{\langle p_{n-1}, p_{n-1} \rangle} = \frac{\langle p_n, p_n \rangle}{\langle p_{n-1}, p_{n-1} \rangle}.$$

This very last step follows from the expansion (17.1), when n is replaced by $n-1$, by observing that $\langle xp_n, p_{n-1} \rangle = \langle p_n, xp_{n-1} \rangle = \langle p_n, p_n \rangle$. The announced recurrence relation is now simply a rewritting of the expansion of xp_n. □

For instance, the recurrence relation for Chebyshev polynomials reads

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x).$$

It can actually be directly deduced from

$$T_{n+1}(\cos\theta) + T_{n-1}(\cos\theta) = \cos((n+1)\theta) + \cos((n-1)\theta) = 2\cos(\theta)\cos(n\theta).$$

By immediate induction, we then see that T_n is indeed a polynomial of degree n, whose leading coefficient is 2^{n-1}. The importance of Chebyshev polynomials becomes clear in connection with Theorems 15.2 and 16.2 on the error in polynomial interpolation: the interpolation points are somehow best chosen at the zeros of T_{n+1}.

Theorem 17.4. The monic polynomial $\widetilde{T}_n := \frac{1}{2^{n-1}} T_n$ is the minimizer of the quantity $\max_{x \in [-1,1]} |p(x)|$ over all monic polynomials $p \in \mathcal{P}_n$.

Proof. Assume on the contrary that there is a monic polynomial $p \in \mathcal{P}_n$ such that

$$\max_{x\in[-1,1]} |p(x)| < \max_{x\in[-1,1]} |\widetilde{T}_n(x)| = \frac{1}{2^{n-1}}.$$

In particular, at the points $x_k := \cos\left(\frac{(n-k)\pi}{n}\right)$, $k \in [\![0,n]\!]$, one has

$$(-1)^{n-k} p(x_k) \leq |p(x_k)| < \frac{1}{2^{n-1}} = \frac{(-1)^{n-k} T_n(x_k)}{2^{n-1}} = (-1)^{n-k} \widetilde{T}_n(x_k).$$

Thus, the polynomial $p - \widetilde{T}_n$ changes sign at the $n+1$ points $x_0, x_1, \ldots, x_n$. By Rolle's theorem, we deduce that $p - \widetilde{T}_n$ has n zeros. Since $p - \widetilde{T}_n$ is of degree at most $n-1$, this is of course impossible. $\square$

17.4 Least-squares polynomial approximation

Consider a function $f \in \mathcal{C}[-1,1]$. Instead of approximating it by its polynomial interpolant, we may look for the **best approximation** to f from $\mathcal{P}_n$, i.e. we aim to minimize $\|f - p\|$ over all polynomials $p \in \mathcal{P}_n$. As in Chapter 11, geometric intuition tells us how to choose p: it should be the orthogonal projection of f on $\mathcal{P}_n$.

Theorem 17.5. Let $p_0, p_1, \ldots, p_n$, $\deg p_k = k$, be orthogonal polynomials with respect to a weight w. For a function $f \in \mathcal{C}[-1,1]$, the polynomial minimizing $\|f - p\|$ over all $p \in \mathcal{P}_n$ is

$$p_f := \sum_{k=0}^{n} \frac{\langle f, p_k \rangle}{\langle p_k, p_k \rangle} p_k. \tag{17.2}$$

Proof. Observe that, with p_f thus defined, we have $\langle p_f, p_i \rangle = \langle f, p_i \rangle$ for all $i \in [\![0,n]\!]$, hence $f - p_f \perp \mathcal{P}_n$. Let now p be an arbitrary polynomial in $\mathcal{P}_n$. We have

$$\begin{aligned} \|f-p\|^2 &= \|(f-p_f)+(p_f-p)\|^2 = \langle (f-p_f)+(p_f-p), (f-p_f)+(p_f-p) \rangle \\ &= \|f-p_f\|^2 + \|p_f-p\|^2 + 2\langle f-p_f, p_f-p \rangle = \|f-p_f\|^2 + \|p_f-p\|^2 \geq \|f-p_f\|^2, \end{aligned}$$

with equality if and only if $\|p_f - p\| = 0$, i.e. $p = p_f$. $\square$

Note that the coefficient $\frac{\langle f, p_k \rangle}{\langle p_k, p_k \rangle}$ is computed independently of n. In practice, we continue to add terms in the expansion (17.2) until $\|f - p\|^2$ is below a specified tolerance ε.

Remark. The arguments work so well here because the norm is euclidean, i.e. it is derived from an inner product. The situation would be quite different with another norm.

17.5 Exercises

From the textbook: 1.a.c.d., 3.a.c.d., 5.a.c.d. p 502; 8, 9 p 512.

1. Establish that T_n is odd, respectively even, whenever n is odd, respectively even.
2. Determine the three-term recurrence relation for the monic polynomials orthogonal on $[-1, 1]$ with respect to the weight $w(x) = \dfrac{1}{\sqrt{1-x^2}}$.
3. Let $\widetilde{P}_n$ be the Legendre polynomial of degree n, renormalized to be monic. Give an expression for $\widetilde{P}_n$ and prove that $\widetilde{P}_n$ is the minimizer of $\displaystyle\int_{-1}^{1} p(x)^2 dx$ over all monic polynomials $p \in \mathcal{P}_n$.

Optional problems

1. The Chebyshev polynomial U_n of the second kind satisfy

$$U_n(\cos(\theta)) = \frac{\sin((n+1)\theta)}{\sin(\theta)}, \qquad \theta \in [0, \pi].$$

   Find a three-term recurrence relation and use it to justify that U_n is a polynomial of degree n. Prove that the system (U_n) is orthogonal on $[-1, 1]$ with respect to the weight $w(x) = \sqrt{1-x^2}$. Finally, find a relation between U_n and T_{n+1}.

Chapter 18

Trigonometric interpolation and FFT

18.1 Approximation

To represent periodic phenomena, trigonometric functions will of course be more appropriate than algebraic polynomials. We will suppose for simplicity that the function to be approximated or interpolated is 2π-periodic [meaning that 2π is a period, but not necessarily the smallest one] and continuous – more briefly, $f \in \mathcal{C}_{2\pi}$. We denote by $\mathcal{T}_n$ the space of **trigonometric polynomials** of degree at most n, i.e. the linear combinations of 1,$\cos(x), \sin(x)$,$\cos(2x), \sin(2x), \ldots, \cos(nx), \sin(nx)$. Note that $\mathcal{T}_n$ is a subspace of $\mathcal{C}_{2\pi}$, i.e that trigonometric polynomials are 2π-periodic continuous functions. Let us equip the space $\mathcal{C}_{2\pi}$ with the inner product [check that it is indeed a inner product]

$$\langle f, g \rangle = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(x)g(x)dx, \qquad f, g \in \mathcal{C}_{2\pi}.$$

Observe that the intergration could be carried out over any interval of length 2π, without changing the value of the integral. We are on the look out for orthogonal bases for the space $\mathcal{T}_n$, and the most natural basis turns out to be one.

Theorem 18.1. With the [unusual] notations C_k and S_k for the functions

$$C_k(x) := \cos(kx), \qquad S_k(x) := \sin(kx),$$

the system $[1, C_1, S_1, \ldots, C_n, S_n]$ forms an orthogonal basis for $\mathcal{T}_n$, and for $k > 0$, one has

$$\langle C_k, C_k \rangle = \langle S_k, S_k \rangle = \frac{1}{2}.$$

Proof. It is clear that $\langle 1, C_k \rangle = \langle 1, S_k \rangle = 0$ for $k > 0$. Furthemore, for indices $k, h \in [\![1, n]\!]$, we observe that $\langle C_k, S_k \rangle = 0$, since the integrand $C_k S_k$ is an odd function. Besides, integrations by part yield

$$\int_{-\pi}^{\pi} \cos(kx)\cos(hx)dx = \frac{h}{k}\int_{-\pi}^{\pi} \sin(kx)\sin(hx)dx = \frac{h^2}{k^2}\int_{-\pi}^{\pi} \cos(kx)\cos(hx)dx,$$

hence, for $k \neq h$, there holds $\int_{-\pi}^{\pi} \cos(kx)\cos(hx)dx = 0$ and $\int_{-\pi}^{\pi} \sin(kx)\sin(hx)dx = 0$, thus $\langle C_k, C_h \rangle = 0$ and $\langle S_k, S_h \rangle = 0$. Finally, with $k = h$, we have

$$\int_{-\pi}^{\pi} \cos^2(kx)dx = \int_{-\pi}^{\pi} \sin^2(kx)dx,$$

and we get twice the value of the integral by summing its two expressions and using the identity $\cos^2 + \sin^2 = 1$. It follows that $\langle C_k, C_k \rangle = \langle S_k, S_k \rangle = \frac{1}{2}$. □

As usual, we may now express the best approximation to a function f from the space $\mathcal{T}_n$ as

$$S_n(f) = \frac{\langle f, 1 \rangle}{\langle 1, 1 \rangle} 1 + \sum_{k=1}^{n} \Big(\frac{\langle f, C_k \rangle}{\langle C_k, C_k \rangle} C_k + \frac{\langle f, S_k \rangle}{\langle S_k, S_k \rangle} S_k \Big). \tag{18.1}$$

In other words, with the **Fourier coefficients** of f defined by

$$a_k := \frac{1}{\pi}\int_{-\pi}^{\pi} f(t)\cos(kt)dt, \qquad b_k := \frac{1}{\pi}\int_{-\pi}^{\pi} f(t)\sin(kt)dt,$$

one has

$$S_n(f)(x) = \frac{a_0}{2} + \sum_{k=1}^{n} \big(a_k \cos(kx) + b_k \sin(kx) \big).$$

This is the partial sum of the Fourier series of f. One of the basic theorems from Fourier analysis states that, for a 2π-periodic continuously differentiable function f, its **Fourier series**

$$\frac{a_0}{2} + \sum_{k=1}^{\infty} \big(a_k \cos(kx) + b_k \sin(kx) \big)$$

converges uniformely to f, which means that

$$\max_{x \in [-\pi, \pi]} |f(x) - S_n(f)(x)| \longrightarrow 0 \quad \text{as } n \to \infty.$$

In the case of a 2π-periodic function made of continuously differentiable pieces, the convergence is weaker, in the sense that $S_n(f)(x) \to \dfrac{f(x^-) + f(x^+)}{2}$ for all x.

18.2 Interpolation

Remark first of all that any trigonometric polynomial of the form

$$p(x) = \frac{a_0}{2} + \sum_{k=1}^{n} \big(a_k \cos(kx) + b_k \sin(kx)\big)$$

can be alternatively represented as

$$p(x) = \sum_{k=-n}^{n} c_k e^{ikx}, \quad \text{where} \quad c_k := \frac{1}{2}(a_k - ib_k), \quad c_{-k} := \overline{c_k}, \quad k \geq 0.$$

It would be possible [and somewhat more elegant] to work with the latter form and express the theory only in terms of complex exponentials – we will not do it, however. To justify the second representation, we may write

$$\begin{aligned} p(x) &= \frac{a_0}{2} + \sum_{k=1}^{n} \left(a_k \frac{e^{ikx} + e^{-ikx}}{2} + b_k \frac{e^{ikx} - e^{-ikx}}{2i}\right) \\ &= \frac{a_0}{2} + \sum_{k=1}^{n} \frac{1}{2}(a_k - ib_k)e^{ikx} + \sum_{k=1}^{n} \frac{1}{2}(a_k + ib_k)e^{-ikx} = \sum_{k=-n}^{n} c_k e^{ikx}. \end{aligned}$$

The following lemma can now be establish using the previous remark.

Lemma 18.2. Any nonzero trigonometric polynomial in $\mathcal{T}_n$ has at most $2n$ zeros in $[-\pi, \pi)$.

Proof. Suppose that $2n+1$ points $x_0 < x_1 < \cdots < x_{2n}$ in $[-\pi, \pi)$ are zeros of a trigonometric polynomial $p(x) = \sum_{k=-n}^{n} c_k e^{ikx} = e^{-inx} \sum_{k=0}^{2n} c_k e^{ikx}$. It follows that the $2n+1$ complex numbers $e^{ix_0}, \ldots, e^{ix_{2n}}$ are distinct zeros of the [algebraic] polynomial $q(z) = \sum_{k=0}^{2n} c_k z^k$. Since that latter is of degree at most $2n$, this implies that $q = 0$, and in turn that $p = 0$. □

An important consequence regarding trigonometric interpolation can now be derived: provided that the number of conditions matches the number of degrees of freedom, interpolation by trigonometric polynomials is always possible. The proof, left as an exercise, follows the same lines as the proof of Theorem 15.1.

Theorem 18.3. Fix $x_0 < x_1 < \cdots < x_{2n}$ in $[-\pi, \pi)$. For any values $y_0, y_1, \ldots, y_{2n}$, there exists a unique trigonometric polynomial of degree at most n that interpolates $y_0, y_1, \ldots, y_{2n}$ at the points $x_0, x_1, \ldots, x_{2n}$ – in short,

$$\exists\,!\, p \in \mathcal{T}_n : \quad p(x_i) = y_i, \quad i \in [\![0, 2n]\!].$$

In general, simple representations for the trigonometric interpolant, matching Lagrange or Newton forms, are not available. However, if the interpolation points are assumed to be the equidistant, e.g

$$\tau_k := \frac{k\,2\pi}{2n+1}, \qquad k \in [\![-n, n]\!],$$

then the trigonometric interpolant admits a nice expression. It involves the pseudo-inner product

$$\langle f, g\rangle_{2n+1} := \frac{1}{2n+1} \sum_{k=-n}^{n} f(\tau_k) g(\tau_k), \qquad f, g \in \mathcal{C}_{2\pi},$$

or more generally, with $N \geq 2n+1$ and $\sigma_k := \dfrac{k\,2\pi}{N}$, $k \in [\![1, N]\!]$,

$$\langle f, g\rangle_N := \frac{1}{N} \sum_{k=1}^{N} f(\sigma_k) g(\sigma_k), \qquad f, g \in \mathcal{C}_{2\pi}.$$

These are termed a pseudo-inner products because the positivity condition is not fulfilled. But they would be genuine inner products if we restricted them to the space $\mathcal{T}_n$ [use Lemma 18.2]. As a matter of fact, it would agree with the usual inner product on this space.

Lemma 18.4. With $N \geq 2n+1$, one has, for any $p, q \in \mathcal{T}_n$,

$$\langle p, q\rangle_N = \langle p, q\rangle.$$

Proof. It is enough to verify that

$$\frac{1}{N} \sum_{k=1}^{N} P(\sigma_k) = \frac{1}{2\pi} \int_{-\pi}^{\pi} P(x) dx$$

for any trigonometric polynomial $P \in \mathcal{T}_{2n}$, since $pq \in \mathcal{T}_{2n}$ for any $p, q \in \mathcal{T}_n$ [check this]. In fact, it is enough to verify this for $P = 1, \mathbb{C}_1, S_1, \ldots, C_{2n}, S_{2n}$. This is immediate for $P = 1$. As for $P = C_h$, $h \in [\![1, 2n]\!]$, one has

$$\begin{aligned}
\frac{1}{2\pi} \int_{-\pi}^{\pi} \cos(hx) dx &= 0, \\
\frac{1}{N} \sum_{k=1}^{N} \cos(h\sigma_k) &= \frac{1}{N} \Re\Big(\sum_{k=1}^{N} e^{ih\sigma_k} \Big) = \frac{1}{N} \Re\Big(\sum_{k=1}^{N} e^{ihk2\pi/N} \Big) \\
&= \frac{1}{N} \Re\Big(e^{ih2\pi/N} \sum_{k=0}^{N-1} (e^{ih2\pi/N})^k \Big) \\
&= \frac{1}{N} \Re\Big(e^{ih2\pi/N}\, \frac{1 - (e^{ih2\pi/N})^N}{1 - e^{ih2\pi/N}} \Big) = 0,
\end{aligned}$$

hence the expected result. The same holds for $P = S_h$, $h \in [\![1, 2n]\!]$ [replace $\Re$ by $\Im$]. □

This aside tells us that the system $[1, C_1, S_1, \ldots, C_n, S_n]$ is also orthogonal with respect to $\langle \cdot, \cdot \rangle_N$. Let us now give the expression for the trigonometric interpolant. One should note the analogy with (18.1).

Theorem 18.5. The trigonometric polynomial $p \in \mathcal{T}_n$ that interpolates the function $f \in \mathcal{C}_{2\pi}$ at the equidistant points $\tau_{-n}, \ldots, \tau_n$ is given by

$$p = \frac{\langle f, 1 \rangle_{2n+1}}{\langle 1, 1 \rangle_{2n+1}} 1 + \sum_{k=1}^{n} \Big(\frac{\langle f, C_k \rangle_{2n+1}}{\langle C_k, C_k \rangle_{2n+1}} C_k + \frac{\langle f, S_k \rangle_{2n+1}}{\langle S_k, S_k \rangle_{2n+1}} S_k \Big). \tag{18.2}$$

Proof. Let us write p in the form $\alpha_0 + \sum_{k=1}^{n} (\alpha_k C_k + \beta_k S_k)$. By the $\langle \cdot, \cdot \rangle_{2n+1}$-orthogonality of the system $[1, C_1, S_1, \ldots, C_n, S_n]$, we deduce e.g. that

$$\alpha_k \langle C_h, C_h \rangle_{2n+1} = \langle p, C_h \rangle_{2n+1} = \frac{1}{2n+1} \sum_{k=-n}^{n} p(\tau_k) C_h(\tau_k) = \frac{1}{2n+1} \sum_{k=-n}^{n} f(\tau_k) C_h(\tau_k) = \langle f, C_h \rangle_{2n+1}.$$

□

In the same flavor, a slightly more general result holds.

Proposition 18.6. Given a function $f \in \mathcal{C}_{2\pi}$ and $N \geq 2n+1$, the trigonometric polynomial $p \in \mathcal{P}_n$ that best approximates f in the least-squares sense on the equidistant points $\sigma_1, \ldots, \sigma_N$ – i.e. that minimizes $\frac{1}{N} \sum_{k=1}^{N} (q(\sigma_k) - f(\sigma_k))^2$ over all $q \in \mathcal{T}_n$ – is given by

$$p = \frac{\langle f, 1 \rangle_N}{\langle 1, 1 \rangle_N} 1 + \sum_{k=1}^{n} \Big(\frac{\langle f, C_k \rangle_N}{\langle C_k, C_k \rangle_N} C_k + \frac{\langle f, S_k \rangle_N}{\langle S_k, S_k \rangle_N} S_k \Big).$$

Proof. Let P be a trigonometric polynomial in $\mathcal{T}_{\lfloor N/2 \rfloor}$ interpolating f at $\sigma_1, \ldots, \sigma_N$ [beware of the case N even]. Note that, for $q \in \mathcal{T}_n$,

$$\frac{1}{N} \sum_{k=1}^{N} (q(\sigma_k) - f(\sigma_k))^2 = \frac{1}{N} \sum_{k=1}^{N} (q(\sigma_k) - P(\sigma_k))^2 = \langle q - P, q - P \rangle_N,$$

so we are looking for the best approximation to P from $\mathcal{T}_n \subseteq \mathcal{T}_{\lfloor N/2 \rfloor}$ relatively to a genuine inner product on $\mathcal{T}_{\lfloor N/2 \rfloor}$. Thus we must have

$$p = \frac{\langle P, 1 \rangle_N}{\langle 1, 1 \rangle_N} 1 + \sum_{k=1}^{n} \Big(\frac{\langle P, C_k \rangle_N}{\langle C_k, C_k \rangle_N} C_k + \frac{\langle P, S_k \rangle_N}{\langle S_k, S_k \rangle_N} S_k \Big) = \frac{\langle f, 1 \rangle_N}{\langle 1, 1 \rangle_N} 1 + \sum_{k=1}^{n} \Big(\frac{\langle f, C_k \rangle_N}{\langle C_k, C_k \rangle_N} C_k + \frac{\langle f, S_k \rangle_N}{\langle S_k, S_k \rangle_N} S_k \Big).$$

□

18.3 Fast Fourier Transform

When N is taken large enough, the best least-squares approximation on $\sigma_1, \ldots, \sigma_N$ from $\mathcal{T}_n$ almost agrees with the n-th partial Fourier sum. This short section is devoted to the computational aspects of this least-squares approximation. In particular, an algorithm for the efficient determination of the coefficients in (18.2), called the **fast Fourier transform** and shortened as **FFT**, is described. To make things easier, we will determine the complex coefficients

$$c_h := \langle f, C_h \rangle_N + i \langle f, S_h \rangle_N = \frac{1}{N} \sum_{k=1}^{N} f(\sigma_k) e^{ih\sigma_k} =: \sum_{k=1}^{N} y_k e^{ih\sigma_k} \tag{18.3}$$

In a straightforward approach, each coefficient c_h necessitates $N+1$ multiplications [we assume that the $\cos(h\sigma_k)$ have been computed and stored once and for all] and since there are N of these coefficients, the number γ_N of multiplications required to determine the coefficients $c_1, \ldots, c_N$ is at most $(N+1)N = \mathcal{O}(N^2)$. With the FFT algorithm, the cost is reduced to $\mathcal{O}(N \ln N)$. This is a significant improvement, for example, when $N \approx 3 \cdot 10^4$, one has $N^2 \approx 9 \cdot 10^8$ while $N \ln N \approx 3 \cdot 10^5$. We will simply establish this when $N = 2^p$, in which case the cost of the FFT algorithm is $\mathcal{O}(p 2^p)$. Observe that, for $h \in [\![1, N/2]\!]$, one has

$$\begin{aligned}
c_h + c_{h+N/2} &= \frac{1}{N} \sum_{k=1}^{N} y_k (e^{ih\sigma_k} + e^{i(h+N/2)\sigma_k}) = \frac{1}{N} \sum_{k=1}^{N} y_k e^{ih\sigma_k} (1 + e^{ik\pi}) = \frac{2}{N} \sum_{k=1,\, k \text{ even}}^{N} y_k e^{ih\sigma_k} \\
&= \frac{1}{N/2} \sum_{k=1}^{N/2} y_{2k}\, e^{ihk2\pi/(N/2)}, \\
c_h - c_{h+N/2} &= \frac{1}{N} \sum_{k=1}^{N} y_k (e^{ih\sigma_k} - e^{i(h+N/2)\sigma_k}) = \frac{1}{N} \sum_{k=1}^{N} y_k e^{ih\sigma_k} (1 - e^{ik\pi}) = \frac{2}{N} \sum_{k=1,\, k \text{ odd}}^{N} y_k e^{ih\sigma_k} \\
&= \frac{e^{ih2\pi/N}}{N/2} \sum_{k=0}^{N/2} y_{2k+1}\, e^{ihk2\pi/(N/2)}.
\end{aligned}$$

These formulae have [almost] the same form as (18.3), N being replaced by $N/2$, and they allow for the determination of all the c_h, $h \in [\![1, N]\!]$, via the determination of all the $c_h + c_{h+N/2}$ and all the $c_h - c_{h+N/2}$, $h \in [\![1, N/2]\!]$. For the first ones, we perform $\gamma_{N/2}$ multiplications, and we perform $\gamma_{N/2} + N/2$ multiplications for the second ones [one extra multiplication for each coefficient]. Thus, we have

$$\gamma_N = 2\gamma_{N/2} + N/2, \qquad \text{or} \quad \gamma_{2^p} = 2\gamma_{2^{p-1}} + 2^{p-1}.$$

Of course, the process will now be repeated. With the assumption that $\gamma_{2^{p-1}} \leq (p-1)2^{p-1}$, we get

$$\gamma_{2^p} \leq (p-1)2^p + 2^{p-1} \leq (p-1)2^p + 2^p = p2^p.$$

We would deduce, with a proper induction, that $\gamma_{2^p} \leq p2^p$ for any p. This is the announced result.

18.4 Exercises

From the textbook: 13, 14 p 532; 2 p 531; 6, 12 p 532; 1, 4, 9 p 543.

1. Prove Theorem 18.3.
2. Determine the Fourier coefficients of the function $f \in \mathcal{C}_{2\pi}$ defined on $[-\pi, \pi]$ by $f(t) = t^2$. Write down the Fourier expansion of f and specify it for the points 0 and π to evaluate some infinite sums.

Part V

Numerical differentiation and integration

Chapter 19

Estimating derivatives: Richardson extrapolation

19.1 Numerical differentiation

In practical situations, a function f is rarely completely determined, but is available only through a finite numbers of values $f(x_0), \ldots, f(x_n)$, say. Can this information be used to estimate a derivative $f'(c)$ or an integral $\int_a^b f(x)dx$? Without further assumptions on f, this clearly appears impossible. It would be possible, though, if f was known to be a polynomial of degree at most n [see the theory of interpolation]. But the information at hand is not generally sufficient to fully recover f, and any numerical estimate for derivatives or integrals should be viewed skeptically unless accompanied by some bound on the errors involved.

To illustrate this point, consider the basic approximating formula for a derivative

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}.$$

To assess the error, one uses Taylor expansion

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2} f''(\xi), \qquad \xi \text{ between } x \text{ and } x+h,$$

which one rearranges as

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{h}{2} f''(\xi), \qquad \xi \text{ between } x \text{ and } x+h. \tag{19.1}$$

This is already more useful, since an error term comes with the numerical formula. Note the two parts in the error term: a factor involving some high-order derivative of f, forcing the function to belong to a certain smoothness class for the estimate to be valid, and a factor involving a power of h, indicating the speed of convergence as h approaches zero – the higher the power, the faster the convergence. The previous formula fares poorly, since the error behaves like $\mathcal{O}(h)$. One can obtain an improved formula using

$$\begin{aligned} f(x+h) &= f(x) + hf'(x) + \frac{h^2}{2} f''(x) + \frac{h^3}{6} f'''(\xi_1), \\ f(x-h) &= f(x) - hf'(x) + \frac{h^2}{2} f''(x) - \frac{h^3}{6} f'''(\xi_2), \end{aligned}$$

with ξ_1, ξ_2 between x and $x+h$, and between x and $x-h$, respectively. Subtracting and rearranging, one obtains

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{12}[f'''(\xi_1) + f'''(\xi_2)].$$

The error term is now a better $\mathcal{O}(h^2)$, provided that f is thrice, rather than merely twice, differentiable. Note that, if $f \in \mathcal{C}^3$, the error term can be rewritten [check it] as

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{6} f'''(\xi), \qquad \xi \text{ between } x-h \text{ and } x+h. \tag{19.2}$$

It should be pointed out that in both (19.1) and (19.2), there is a pronounced deterioration in accuracy as h approaches zero. This is due to subtractive cancellation: the values of $f(x-h)$, $f(x)$, and $f(x+h)$ are very close to each other – identical in the machine – and the computation of the difference yields severe loss of significant digits. Besides, one also has to be aware that numerical differentiation from empirical data is highly unstable and should be undertaken with great caution [or avoided]. Indeed, if the sampling points $x \pm h$ are accurately determined while the ordinates $f(x \pm h)$ are inaccurate, then the errors in the ordinates are magnified by the large factor $1/(2h)$.

19.2 Richardson extrapolation

The procedure known as **Richardson extrapolation** is used to obtain numerical formulae whose order of accuracy can be arbitrary high. Note that, in the same way as we obtained (19.2) but using the complete Taylor expansions of $f(x-h)$ and $f(x+h)$, we would obtain

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \Big[\frac{h^2}{3!} f^{(3)}(x) + \frac{h^4}{5!} f^{(5)}(x) + \frac{h^6}{7!} f^{(7)}(x) + \cdots\Big].$$

This equation takes the form

$$L = \phi^{[1]}(h) \ + a_2^{[1]} h^2 + a_4^{[1]} h^4 + a_6^{[1]} h^6 + \cdots . \tag{19.3}$$

Here L and $\phi^{[1]}(h)$ stand for $f'(x)$ and $[f(x+h) - f(x-h)]/(2h)$, but could very well stand for other quantities, so that the procedure is applicable to many numerical processes. As of now, the error term $a_2^{[1]} h^2 + a_4^{[1]} h^4 + \cdots$ is of order $\mathcal{O}(h^2)$. Our immediate aim is to produce a new formula whose error term is of order $\mathcal{O}(h^4)$. The trick is to consider (19.3) for $h/2$ instead of h, which yields

$$4L = 4\phi^{[1]}(h/2) \ + a_2^{[1]} h^2 + a_4^{[1]} h^4/4 + a_6^{[1]} h^6/16 + \cdots .$$

Combining the latter with (19.3), we obtain

$$\begin{aligned} L &= \frac{4}{3}\phi^{[1]}(h/2) - \frac{1}{3}\phi^{[1]}(h) \ - a_4^{[1]} h^4/4 - 5a_6^{[1]} h^6/16 - \cdots \\ &=: \phi^{[2]}(h) \ + a_4^{[2]} h^4 + a_6^{[2]} h^6 + a_8^{[2]} h^8 + \cdots . \end{aligned}$$

The numerical formula has already been improved by two orders of accuracy. But there is no reason to stop here: the same step can be carried out once more to get rid of the term in h^4, then once again to get rid of the term in h^6, and so on. We would get

$$\begin{aligned} L &= \phi^{[2]}(h) \quad + a_4^{[2]} h^4 + a_6^{[2]} h^6 + \cdots \\ 16L &= 16\phi^{[2]}(h/2) \ + a_4^{[2]} h^4 + a_6^{[2]} h^6/4 + \cdots , \end{aligned}$$

and consequently

$$L = \phi^{[3]}(h) \ + a_6^{[3]} h^6 + \cdots , \qquad \text{with} \quad \phi^{[3]}(h) := \frac{16}{15}\phi^{[2]}(h/2) - \frac{1}{15}\phi^{[2]}(h).$$

On the same model, we would get

$$\begin{aligned} L &= \phi^{[k]}(h) \quad + a_{2k}^{[k]} h^{2k} + \cdots \\ 4^k L &= 4^k \phi^{[k]}(h/2) \ + a_{2k}^{[k]} h^{2k} + \cdots , \end{aligned}$$

and consequently

$$L = \phi^{[k+1]}(h) \ + a_{2k+2}^{[k+1]} h^{2k+2} + \cdots , \qquad \text{with} \quad \phi^{[k+1]}(h) := \frac{4^k}{4^k - 1}\phi^{[k]}(h/2) - \frac{1}{4^k - 1}\phi^{[k]}(h).$$

Let us now give the algorithm allowing for M steps of Richardson extrapolation. Note that we prefer to avoid the recursive computation of the functions $\phi^{[k]}$, since we only need $\phi^{[M+1]}(h)$ for some particular h, say $h = 1$. Evaluating $\phi^{[M+1]}(h)$ involves $\phi^{[M]}(h/2)$ and $\phi^{[M]}(h)$, which in turn involve $\phi^{[M-1]}(h/4)$, $\phi^{[M-1]}(h/2)$, and $\phi^{[M-1]}(h)$. Hence, for any k,

the values $\phi^{[k]}(h/2^{M+1-k}), \ldots, \phi^{[k]}(h)$ are required, and the values $\phi^{[1]}(h/2^{M}), \ldots, \phi^{[1]}(h)$ are eventually required. Thus, we start by setting

$$D(n,1) := \phi^{[1]}(h/2^{n-1}), \qquad \text{for } n \in [\![1, M+1]\!],$$

and we compute the quantities $D(n,k)$ corresponding to $\phi^{[k]}(h/2^{n-k})$ by

$$D(n,k+1) = \frac{4^k}{4^k-1} D(n,k) - \frac{1}{4^k-1} D(n-1,k), \qquad \text{for } k \in [\![1, M]\!] \text{ and } n \in [\![k+1, M+1]\!].$$

This results in the construction of the triangular array

$$\begin{array}{ccccc}
D(1,1) & & & & \\
D(2,1) & D(2,2) & & & \\
D(3,1) & D(3,2) & D(3,3) & & \\
\vdots & \vdots & \vdots & \ddots & \\
D(M+1,1) & D(M+1,2) & D(M+1,3) & \cdots & D(M+1,M+1)
\end{array}$$

by way of the pseudocode

input x, h, M
for $n = 1$ to $M+1$ **do** $D(n,1) \leftarrow \Phi(x, h/2^{n-1})$ **end do end for**
for $k = 1$ to M **do**
  for $n = k+1$ to $M+1$ **do**
  $D(n,k+1) \leftarrow D(n,k) + [D(n,k) - D(n-1,k)]/(4^k - 1)$
  end do end for
end do end for
output D

The function Φ should have been made available separately here. Note that this algorithm too will end up producing meaningless results due to subtractive cancellation.

19.3 Exercises

From the textbook: 5 p 184, 9 p 185.

1. Establish the second derivative formula

$$f''(x) = \frac{f(x-h) - 2f(x) + f(x+h)}{h^2} - \frac{h^2}{12} f^{(4)}(\xi), \qquad \xi \text{ between } x-h \text{ and } x+h.$$

2. Derive the following two formulae, together with their error terms, for approximating the third derivative. Which one is more accurate?

$$
\begin{aligned}
f'''(x) &\approx \frac{1}{h^3}\big[f(x+3h) - 3f(x+2h) + 3f(x+h) - f(x)\big], \\
f'''(x) &\approx \frac{1}{2h^3}\big[f(x+2h) - 2f(x+h) + 2f(x-h) - f(x-2h)\big].
\end{aligned}
$$

3. Program the Richardson extrapolation algorithm given in the notes to estimate $f'(x)$. Test your program on

   - $\ln x$ at $x = 3$,
   - $\tan x$ at $x = \sin^{-1}(0.8)$,
   - $\sin(x^2 + x/3)$ at $x = 0$.

Optional problems

From the textbook: 15 p 186.

Chapter 20

Numerical integration based on interpolation

20.1 General framework

Suppose that we want to integrate a function f over an interval $[a, b]$. We select points $x_0, x_1, \ldots, x_n$ on $[a, b]$ and interpolate f at these points by a polynomial $p \in \mathcal{P}_n$. The Lagrange form of p is

$$p(x) = \sum_{i=0}^{n} f(x_i)\ell_i(x), \qquad \ell_i(x) := \prod_{j=0,\, j\neq i}^{n} \frac{x - x_j}{x_i - x_j}.$$

In the hope that p provides a good approximation to f, we expect that

$$\int_a^b f(x)dx \approx \int_a^b p(x)dx = \sum_{i=0}^{n} f(x_i) \int_a^b \ell_i(x)dx =: \sum_{i=0}^{n} A_i f(x_i).$$

Note that the coefficients $A_0, \ldots, A_n$ are independent of f and that the formula is **exact** for all polynomials of degree at most n. If the nodes $x_0, \ldots, x_n$ are equidistant, the latter formula is called **Newton–Cotes formula**. More generally, the approximation of the integral $\int_a^b f(x)dx$ by a sum of the type $\sum_{i=0}^{n} A_i f(x_i)$ is called a **numerical quadrature**. If we are given such a quadrature formula which is exact on $\mathcal{P}_n$, we can always retrieve the coeffiecients $A_0, \ldots, A_n$, using the Lagrange cardinal polynomials, by writing

$$\int_a^b \ell_j(x)dx = \sum_{i=0}^{n} A_i \ell_j(x_i) = A_j.$$

We could also retrieve the coefficients by solving the linear system [check its nonsingularity]

$$\frac{b^{k+1}-a^{k+1}}{k+1} = \int_a^b x^k dx = \sum_{i=0}^{k} A_i\, x_i^k, \qquad k \in [\![0,n]\!].$$

As mentioned in the previous chapter, a numerical estimate for an integral should come with a bound on the error. For this purpose, recall that the error in polynomial interpolation is

$$f(x) - p(x) = [x_0, \ldots, x_n, x]f \prod_{i=0}^{n}(x - x_i).$$

Integrating, we get

$$\int_a^b f(x)dx - \sum_{i=0}^{n} A_i f(x_i) = \int_a^b [x_0, \ldots, x_n, x]f \prod_{i=0}^{n}(x - x_i)dx.$$

Since $\big|[x_0, \ldots, x_n, x]f\big| \le \dfrac{M}{(n+1)!}$, where $M := \max\limits_{\xi\in[a,b]} |f^{(n+1)}(\xi)|$, we obtain the bound

$$\Big|\int_a^b f(x)dx - \sum_{i=0}^{n} A_i f(x_i)\Big| \le \frac{M}{(n+1)!}\int_a^b \prod_{i=0}^{n} |x - x_i|\, dx.$$

One could minimize the right-hand side over the nodes $x_0, \ldots, x_n$. In the case $[a,b] = [-1,1]$, this would lead to the zeros of U_{n+1} [Chebyshev polynomial of the second kind], that is to the choice $x_k = \cos\big((k+1)\pi/(n+2)\big)$, $k \in [\![0,n]\!]$. We called upon the fact that $2^{-n}U_n$ minimizes the quantity $\displaystyle\int_{-1}^{1} |p(x)|dx$ over all monic polynomials $p \in \mathcal{P}_n$.

20.2 Trapezoidal rule

Take $n = 1$, $x_0 = a$, and $x_1 = b$ to obtain [with the help of a picture] $A_0 = A_1 = \dfrac{b-a}{2}$. Hence the corresponding quadrature formula, exact for $p \in \mathcal{P}_1$, reads

$$\int_a^b f(x)dx = \frac{b-a}{2}[f(a) + f(b)].$$

The error term of this **trapezoidal rule** takes the form [check it]

$$\int_a^b f(x)dx - \frac{b-a}{2}[f(a) + f(b)] = -\frac{1}{12}(b-a)^3 f''(\xi), \qquad \xi \in [a,b].$$

Let us now partition the interval $[a,b]$ into n subintervals whose endpoints are located at $a = x_0 < x_1 < \cdots < x_n = b$. A **composite rule** is produced by applying an approximating

formula for the integration over each subinterval. In this case, we obtain the **composite trapezoidal rule**

$$\int_a^b f(x)dx = \sum_{i=1}^{n} \int_{x_{i-1}}^{x_i} f(x)dx \approx \sum_{i=1}^{n} \frac{x_i - x_{i-1}}{2}[f(x_{i-1}) + f(x_i)].$$

In other words, the integral of f is replaced by the integral of the **broken line** that interpolates f at $x_0, x_1, \ldots, x_n$. In particular, if $[a, b]$ is partitioned into n equal subintervals, i.e. if $x_i = a + ih$, where $h := \dfrac{b-a}{n}$ is the length of each subinterval, we have

$$\int_a^b f(x)dx \approx \frac{h}{2}\sum_{i=1}^{n}[f(a+(i-1)h) + f(a+ih)] = \frac{h}{2}\left[f(a) \ + \ 2\sum_{i=1}^{n-1} f(a+ih) \ + \ f(b)\right].$$

The latter expression is preferred for computations, since it avoids the unnecessary repetition of function evaluations. The error term takes the form $-\frac{1}{12}(b-a)h^2 f''(\xi)$ for some $\xi \in [a, b]$, and the verification of this statement is left as an exercise.

20.3 Simpson's rule

We now take $n = 2$, $x_0 = a$, $x_1 = \dfrac{a+b}{2}$, and $x_2 = b$. This leads to the formula

$$\int_a^b f(x)dx \approx \frac{b-a}{6}[f(a) + 4f((a+b)/2) + f(b)], \tag{20.1}$$

which is known as **Simpson's rule**. To determine the coefficients A_0, A_1, A_2, we can use e.g. the conditions $A_0 = A_2$ [symmetry], $A_0 + A_1 + A_2 = b - a$ [corresponding to $f(x) = 1$], and the condition corresponding to $f(x) = (x-a)^2$. Somehow surprisingly, Simpson's rule is not only exact on $\mathcal{P}_2$, but also on $\mathcal{P}_3$. Indeed, its error term can be written as

$$-\frac{1}{90}\left(\frac{b-a}{2}\right)^5 f^{(4)}(\xi), \qquad \xi \in [a, b].$$

To derive it, we denote by $2h$ and c the length and the midpoint of $[a, b]$. Thus, the left-hand side of (20.1) becomes

$$\begin{aligned}\int_{c-h}^{c+h} f(x)dx &= \int_{-h}^{h} f(c+u) = \int_{-h}^{h} \left(f(c) + f'(c)u + \frac{f''(c)}{2}u^2 + \frac{f'''(c)}{6}u^3 + \frac{f^{(4)}(c)}{24}u^4 + \cdots\right) \\ &= 2h\, f(c) + \frac{h^3}{3} f''(c) + \frac{h^5}{60} f^{(4)}(c) + \cdots ,\end{aligned}$$

while the right-hand side becomes

$$\begin{aligned}\frac{h}{3}[f(c-h)+4f(c)+f(c+h)] &= \frac{h}{3}\big[6f(c)+h^2f''(c)+\frac{h^4}{12}f^{(4)}(c)+\cdots\big] \\ &= 2h\,f(c)+\frac{h^3}{3}\,f''(c)+\frac{h^5}{36}\,f^{(4)}(c)+\cdots.\end{aligned}$$

The difference reduces to

$$\frac{h^5}{60}\,f^{(4)}(c)-\frac{h^5}{36}\,f^{(4)}(c)+\cdots=-\frac{h^5}{90}\,f^{(4)}(c)+\cdots.$$

The announced error formula would be obtained by expressing the Taylor remainders in a more careful way.

A **composite Simpson's rule** based on an even number n of subintervals is often used. With $x_i := a+ih$ and $h := \dfrac{b-a}{n}$, it reads

$$\begin{aligned}\int_a^b f(x)dx &= \sum_{i=1}^{n/2}\int_{x_{2i-2}}^{x_{2i}} f(x)dx \approx \frac{h}{3}\sum_{i=1}^{n/2}[f(x_{2i-2})+4f(x_{2i-1})+f(x_{2i})] \\ &= \frac{h}{3}\Big[f(x_0)\ +\ 2\sum_{i=1}^{n/2-1} f(x_{2i})\ +\ f(x_{2n})\ +\ 4\sum_{i=1}^{n/2} f(x_{2i-1})\Big].\end{aligned}$$

It can be established that the error term is of the form

$$-\frac{1}{180}(b-a)h^4 f^{(4)}(\xi), \qquad \xi\in[a,b].$$

20.4 Exercises

From the textbook: 3.c.d.f., 5.c.d.f., 13, 15 p195.

1. Establish the form given in the notes for the error in the composite trapezoidal rule .
2. Prove that Simpson's rule is exact on $\mathcal{P}_3$, without using the error formula.
3. Is there a formula of the form $\int_0^1 f(x)dx \approx \alpha[f(x_0)+f(x_1)]$ that correctly integrates all quadratic polynomials?
4. For n even, prove that if the formula

$$\int_{-1}^1 f(x)dx \approx \sum_{i=0}^n A_i f(x_i)$$

is exact for all polynomials in $\mathcal{P}_n$, and if the nodes are symmetrically placed about the origin, then the formula is exact for all polynomials in $\mathcal{P}_{n+1}$.

5. Write a function SimpsonUniform[f,a,b,n] to calculate $\int_a^b f(x)dx$ using the composite Simpson's rule with $2n$ equal subintervals. Use it to approximate π from an integral of the type $\int_a^b \frac{c}{1+x^2}dx$.

Optional problems

1. Prove that $2^{-n}U_n$ is the minimizer of $\int_{-1}^1 |p(x)|dx$ over all monic polynimials $p \in \mathcal{P}_n$ [Hint: prove first the orthogonality relations $\int_{-1}^1 U_m(x)\,\mathrm{sgn}[U_n(x)]dx = 0$, $m \in [\![0, n-1]\!]$, next integrate $p\,\mathrm{sgn}[U_n]$ for all monic polynomials $p = 2^{-n}U_n + a_{n-1}U_{n-1} + \cdots + a_0U_0$].

Chapter 21

Romberg integration

The **Romberg algorithm** combines the composite trapezoidal rule and the Richardson extrapolation process in order to approximate the integral $I := \int_a^b f(x)dx$.

21.1 Recursive trapezoidal rule

Recall that the composite trapezoidal rule on $[a,b]$, using n subintervals of length $h = \frac{b-a}{n}$, provides an approximate value for I given by

$$I(n) = h \sum_{i=0}^{n}{}' f(a+ih),$$

where the prime on the summation symbol means that the first and last terms are to be halved. Note that, when computing $I(2n)$, we will need values of the form $f\Big(a+2i\frac{b-a}{2n}\Big) = f\Big(a+i\frac{b-a}{n}\Big)$. Since these are precisely the summands of $I(n)$, we should use the work done in the computation of $I(n)$ to compute $I(2n)$. Precisely, remark that

$$I(2n) = \frac{I(n)}{2} + \frac{b-a}{2n} \sum_{i=1}^{n} f\Big(a+(2i-1)\frac{b-a}{2n}\Big).$$

This allows to compute the sequence $I(1), I(2), \ldots, I(2^k), \ldots$ in a recursive way, without duplicate function evaluations. Let us now accept the validity of an error formula of the type

$$I = I(2^n) + c_2 h^2 + c_4 h^4 + \cdots, \qquad \text{where } h = \frac{b-a}{2^n}.$$

21.2 Romberg algorithm

Since the previous formula remains valid when h is halved, we may apply Richardson extrapolation to obtain more and more accurate error formulae [to use exactly the same formalism as in Chapter 19, we could express everything only in terms of h]. This would lead us to define, for $n \in [\![1, M+1]\!]$,

$$R(n,1) = I(2^{n-1}),$$

and then, for $k \in [\![1, M]\!]$ and $n \in [\![k+1, M+1]\!]$,

$$R(n,k+1) = R(n,k) + \frac{1}{4^k - 1}[R(n,k) - R(n-1,k)].$$

The calculation provides an array of the form

$$\begin{array}{ccccc}
R(1,1) & & & & \\
R(2,1) & R(2,2) & & & \\
R(3,1) & R(3,2) & R(3,3) & & \\
\vdots & \vdots & \vdots & \ddots & \\
R(M+1,1) & R(M+1,2) & R(M+1,3) & \cdots & R(M+1,M+1)
\end{array}$$

Note that the first column is computed according to the recursive process presented earlier. Usually, we only need a moderate value of M, since $2^M + 1$ function evaluations will be performed. The pseudocode is only marginally different from the one in Chapter 19, the difference being that the triangular array is computed row by row, instead of column by column.

input a, b, f, M
$h \leftarrow b - a, \quad R(1,1) \leftarrow \dfrac{b-a}{2}[f(a) + f(b)]$
for $n = 1$ to M **do**
    $h \leftarrow h/2, \quad R(n+1,1) = R(n,1)/2 + h\sum_{i=1}^{2^{n-1}} f\big(a + (2i-1)h\big)$
    for $k = 1$ to n **do**
        $R(n+1,k+1) = R(n+1,k) + [R(n+1,k) - R(n,k)]/(4^k - 1)$
    end do end for
end do end for
output R

21.3 Exercises

From the textbook: 9 p 211, 13 p 212, 2 p 211 [after having written a program to carry out the Romberg algorithm].

1. Calculate $\int_0^1 \frac{\sin x}{\sqrt{x}} dx$ by the Romberg algorithm [Hint: change of variable].

2. Assume that the first column of the Romberg array converges to $I := \int_a^b f(x)dx$ – i.e. that $\lim_{n\to\infty} R(n,1) = I$. Show that the second column also converges to I. Can one infer that $\lim_{n\to\infty} R(n,n) = I$?

Chapter 22

Adaptive quadrature

22.1 Description of the method

Adaptive quadrature methods are designed to take into account the behavior of the function f to be integrated on an interval $[a, b]$. To do so, sample points will be clustered in the regions of large variations of f. We present here a typical adaptive method based on Simpson's rule, where the user only supplies the function f, the interval $[a, b]$, and a desired accuracy ε.

We start by using Simpson's rule on the interval $[a, b]$. If the approximation is not accurate enough, then the interval $[a, b]$ is divided into two equal subintervals, on each of which Simpson's rule is used again. If one [or both] of the resulting approximations is not accurate enough, then a subdivision is applied once more. The repetition of this process constitutes the main idea behind the **adaptive Simpson's method**. Note that, at the end of the process, we will have obtained approximations $S_1, \ldots, S_n$ to the integrals of f on some intervals $[x_0, x_1], \ldots, [x_{n-1}, x_n]$. Denoting $e_1, \ldots, e_n$ the associated local errors, we get

$$\int_a^b f(x)dx = \sum_{i=1}^n \int_{x_{i-1}}^{x_i} f(x)dx = \sum_{i=1}^n S_i + \sum_{i=1}^n e_i.$$

Thus, to ensure that the total error is bounded by ε, we can impose

$$|e_i| \leq \varepsilon \, \frac{x_i - x_{i-1}}{b - a},$$

for in this case

$$\Big| \sum_{i=1}^n e_i \Big| \leq \sum_{i=1}^n |e_i| \leq \frac{\varepsilon}{b-a} \sum_{i=1}^n (x_i - x_{i-1}) = \frac{\varepsilon}{b-a}(b-a) = \varepsilon.$$

On each interval $[u, w]$ that will be considered, we write the basic Simpson's rule as

$$\begin{aligned}\int_u^w f(x)dx &= \frac{w-u}{6}\left[f(u) + 4f\big((u+w)/2\big) + f(w)\right] - \frac{1}{90}\left(\frac{w-u}{2}\right)^5 f^{(4)}(\zeta), \\ &=: S(u,w) - \frac{1}{90}\left(\frac{w-u}{2}\right)^5 f^{(4)}(\zeta), \qquad \zeta \in [u,w].\end{aligned}$$

Then, if $[u, w]$ is to be divided into two equal subintervals $[u, v]$ and $[v, w]$, a more accurate value of the integral can be computed according to

$$\begin{aligned}\int_u^w f(x)dx &= \int_u^v f(x)dx + \int_v^w f(x)dx \\ &= S(u,v) - \frac{1}{90}\left(\frac{v-u}{2}\right)^5 f^{(4)}(\xi') + S(v,w) - \frac{1}{90}\left(\frac{w-v}{2}\right)^5 f^{(4)}(\xi'') \\ &= S(u,v) + S(v,w) - \frac{1}{90}\left(\frac{w-u}{2}\right)^5 \frac{1}{2^5}\left[f^{(4)}(\xi') + f^{(4)}(\xi'')\right] \\ &= S(u,v) + S(v,w) - \frac{1}{2^4}\frac{1}{90}\left(\frac{w-u}{2}\right)^5 f^{(4)}(\xi),\end{aligned}$$

where ξ', ξ'', and ξ belong to $[u, v]$, $[v, w]$, and $[u, w]$, respectively. This is justified only if $f^{(4)}$ is continuous. As usual, the error term in such a formula cannot be bounded without some knowledge of $f^{(4)}$. For automatic computation, however, it is imperative to be able to estimate the magnitude of $f^{(4)}(\xi)$. So we have to come up with an additional assumption. Over small intervals, the function $f^{(4)}$ will be considered almost constant – in particular, we suppose that $f^{(4)}(\zeta) = f^{(4)}(\xi)$. This term can then be eliminated to obtain

$$\begin{aligned}15\int_u^w f(x)dx &= 16[S(u,v) + S(v,w)] - S(u,w), \\ \int_u^w f(x)dx &= [S(u,v) + S(v,w)] + \frac{1}{15}[S(u,v) + S(v,w) - S(u,w)].\end{aligned}$$

Observe that the error term has now been replaced by a quantity we are able to compute, hence an error tolerance condition can be tested.

22.2 The pseudocode

Let us complete the groundwork before writing down the final version of the algorithm. We start with $u = a$ and $w = b$. Throughout the algorithm, the data are stored in a vector

$$[u, h, f(u), f(u+h), f(u+2h), S],$$

where $2h = w - u$ is the length of $[u, w]$ and S is the Simpson's estimate on $[u, w]$, that is

$$S = S(u,w) = \frac{h}{3}\left[f(u) + 4f(u+h) + f(u+2h)\right].$$

Then one computes the midpoint $v = u+h$ and the estimates $S_1 := S(u,v)$ and $S_2 = S(v,w)$. To see whether $S_1 + S_2$ is a good enough approximation, we test the inequality

$$|S_1 + S_2 - S| \leq \varepsilon \frac{30h}{b-a}.$$

If the inequality holds, then the refined value $S_1+S_2+[S_1+S_2-S]/15$ is the accepted value of the integral of f on $[u,w]$. In this case, it is added to a variable Σ which should eventually receive the approximate value of the integral on $[a,b]$. If the inequality does not hold, then the interval $[u,w]$ is divided in two. The previous vector is discarded and replaced by the two vectors

$$[u, h/2, f(u), f(y), f(v), S_1], \qquad y := u + h/2,$$
$$[v, h/2, f(v), f(z), f(w), S_2], \qquad z := v + h/2.$$

The latter are added to an existing stack of such vectors. Note that only two new function evaluations have been required to compute S_1 and S_2. Note also that the user should prevent the size of the stack from exceeding a prescribed value n, in order to avoid an infinite algorithm. In view of the description we have made, we may now suggest the following version of the pseudocode. A recursive version would also be conceivable.

input f, a, b, ε, n
$\Delta \leftarrow b-a, \quad \Sigma \leftarrow 0, \quad h \leftarrow \Delta/2, \quad c \leftarrow (a+b)/2, \quad k \leftarrow 1,$
$fa \leftarrow f(a), \quad fb \leftarrow f(b), \quad fc \leftarrow f(c), \quad S \leftarrow [fa + 4fc + fb]h/3,$
$v^{[1]} \leftarrow [a, h, fa, fc, fb, S]$
while $1 \leq k \leq n$ **do**
    $h \leftarrow v_2^{[k]}/2,$
    $fy = f(v_1^{[k]} + h)\ , \qquad S_1 \leftarrow [v_3^{[k]} + 4fy + v_4^{[k]}]h/3,$
    $fz = f(v_1^{[k]} + 3h), \qquad S_2 \leftarrow [v_4^{[k]} + 4fz + v_5^{[k]}]h/3$
    if $|S_1 + S_2 - v_6^{[k]}| < 30\varepsilon h/\Delta$
        then $\Sigma \leftarrow \Sigma + S_1 + S_2 + [S_1 + S_2 - v_6^{[k]}]/15, \qquad k \leftarrow k-1,$
        if $k = 0$ **then output** Σ, **stop**
    else **if** $k = n$ **then output** failure, **stop**
        $fw \leftarrow v_5^{[k]},$
        $v^{[k]} \leftarrow [v_1^{[k]}, h, v_3^{[k]}, fy, v_4^{[k]}, S_1],$
        $k \leftarrow k+1,$
        $v^{[k]} \leftarrow [v_1^{[k-1]} + 2h, h, v_5^{[k-1]}, fz, fw, S_2]$
    end if
end do end while

22.3 Exercises

From the textbook: 1.b.d, 2.b.d. p 218; 5, 9 p 219.

1. Program the adaptive algorithm and test it on the integrals
   - $\int_0^1 x^{1/2}dx$,
   - $\int_0^1 (1-x)^{1/2}dx$,
   - $\int_0^1 (1-x)^{1/4}dx$.

Chapter 23

Gaussian quadrature

23.1 The main result

Consider the quadrature formula

$$\int_{-1}^{1} f(x)dx \approx \sum_{i=0}^{n} A_i f(x_i).$$

If based on interpolation, we know that the formula is exact on $\mathcal{P}_n$. We also have seen that the degree of accuracy might be larger than n for some choices of nodes $x_0, \ldots, x_n$. We may ask what is the largest degree of accuracy possible. Observe that the latter cannot exceed $2n+1$, for if the formula was exact on $\mathcal{P}_{2n+2}$, we would obtain a contradiction by considering $f(x) = (x-x_1)^2 \cdots (x-x_n)^2$. In fact, the maximal degree of accuracy is exactly $2n+1$. Indeed, a judicious choice of nodes leads to the following **Gaussian quadrature**, which is exact on $\mathcal{P}_{2n+1}$ [the dimension of the space where exactness occurs has doubled from a priori $n+1$ to $2n+2$].

Theorem 23.1. Let $x_0, \ldots, x_n$ be the zeros of the Legendre polynomial of degree $n+1$. Then the quadrature formula

$$\int_{-1}^{1} f(x)dx \approx \sum_{i=0}^{n} A_i f(x_i), \qquad A_i = \int_{-1}^{1} \prod_{j=0,\, j\neq i}^{n} \frac{x-x_j}{x_i-x_j}\, dx,$$

is exact on $\mathcal{P}_{2n+1}$.

Remark that the formula makes sense as long as $x_0, \ldots, x_n$ are distinct points in $(-1,1)$. This property is not specific to the zeros of the Legendre polynomial of degree $n+1$, but

is shared by the $(n+1)$-st orthogonal polynomials relative to all possible weight functions – see Theorem 17.2. As a matter of fact, the quadrature rule also remains valid in this context. We only need to establish this general result.

Theorem 23.2. Let $x_0, \ldots, x_n$ be the zeros of the $(n+1)$-st orthogonal polynomial relative to the weight function w. Then the quadrature formula

$$\int_{-1}^{1} f(x)w(x)dx \approx \sum_{i=0}^{n} A_i f(x_i), \qquad A_i = \int_{-1}^{1} \prod_{j=0,\, j\neq i}^{n} \frac{x-x_j}{x_i-x_j}\, w(x)\, dx,$$

is exact on $\mathcal{P}_{2n+1}$.

Proof. Let f be a polynomial of degree at most $2n+1$. Consider the division of f by p, the $(n+1)$-st orthogonal polynomial reative to w. We have

$$f = q\,p + r, \qquad \text{with} \quad \deg r \leq n.$$

Note that we must have $\deg q \leq n$ to ensure that $\deg f \leq 2n+1$. Then observe that

$$\begin{aligned}
\int_{-1}^{1} f(x)w(x)dx &= \int_{-1}^{1} q(x)p(x)w(x)dx + \int_{-1}^{1} r(x)w(x)dx = \int_{-1}^{1} r(x)w(x)dx, \\
\sum_{i=0}^{n} A_i f(x_i) &= \sum_{i=0}^{n} A_i[q(x_i)p(x_i) + r(x_i)] = \sum_{i=0}^{n} A_i r(x_i).
\end{aligned}$$

The equality of these two quantities for all $r \in \mathcal{P}_n$ is derived from the expansion of r on the Lagrange cardinal polynomials $\ell_0, \ldots, \ell_n$ associated to $x_0, \ldots, x_n$, just as in Chapter 20. □

It is important to notice that the **weights** $A_0, \ldots, A_n$ in the previous formula are always positive. To realize this, observe that, besides the expression $A_i = \int_{-1}^{1} \ell_i(x)w(x)dx$, there also holds $A_i = \int_{-1}^{1} \ell_i^2(x)w(x)dx$ [simply take $f = \ell_i^2$].

23.2 Examples

In theory, for any weight function w, we could devise the corresponding quadrature formula numerically: construct the $(n+1)$-st orthogonal polynomial numerically, find its zeros e.g. by using NSolve in Mathematica, and find the weights e.g. by solving a linear system. Fortunately, this work can be avoided for usual choices of w, since the nodes x_i and the weights A_i can be found in appropriate handbooks.

Let us consider the simple case $w = 1$ and $n = 1$. The zeros of the second orthogonal polynomial, i.e. of the Legendre polynomial $P_2(x) = 3x^2/2 - 1/2$ are $x_0 = -1/\sqrt{3}$ and $x_1 = 1/\sqrt{3}$. The weights A_0 and A_1 must be identical, by symmetry, and must sum to $\int_{-1}^{1} dx = 2$. Hence the Gaussian quadrature, which is exact on $\mathcal{P}_3$, takes the form

$$\int_{-1}^{1} f(x)dx \approx f\big(-1/\sqrt{3}\big) + f\big(1/\sqrt{3}\big).$$

With $w = 1$ and $n = 2$, the Gaussian quadrature would be

$$\int_{-1}^{1} f(x)dx \approx \frac{5}{9}\, f\big(-\sqrt{3/5}\big) + \frac{8}{9}\, f\big(0\big) + \frac{5}{9}\, f\big(\sqrt{3/5}\big).$$

If we now consider the weight function $w(x) = \dfrac{1}{\sqrt{1-x^2}}$, the $(n{+}1)$-st orthogonal polynomial reduces to the Chebyshev polynomial T_{n+1}, whose zeros are $\cos\big((2i{+}1)\pi/(2n{+}2)\big)$, $i \in [\![0, n]\!]$. It turns out that the weights A_i are independent of i, their common value being $\pi/(n+1)$. Hence, with $F(x) := \sqrt{1-x^2} f(x)$, we may write

$$\begin{aligned}
\int_{-1}^{1} f(x)dx &= \int_{-1}^{1} F(x)\frac{dx}{\sqrt{1-x^2}} \approx \frac{\pi}{n+1}\sum_{i=0}^{n} F\big(\cos\big((2i+1)\pi/(2n+2)\big)\big) \\
&= \frac{\pi}{n+1}\sum_{i=0}^{n} \sin\big((2i+1)\pi/(2n+2)\big) \cdot f\big(\cos\big((2i+1)\pi/(2n+2)\big)\big).
\end{aligned}$$

This is known as **Hermite's quadrature formula**. Note that it is not exact on $\mathcal{P}_{2n+1}$, but rather on $w\,\mathcal{P}_{2n+1} := \big\{w\,p,\ p \in \mathcal{P}_{2n+1}\big\}$.

23.3 Error analysis

For completeness, we state without proof the following error estimate.

Theorem 23.3. Let p be the $(n+1)$-st monic orthogonal polynomial relative to the weight function w, and let $x_0, \ldots, x_n$ be its zeros. Then, for any $f \in \mathcal{C}^{2n+2}[-1, 1]$, the error term in the Gaussian quadrature formula takes the form, for some $\xi \in (-1, 1)$,

$$\int_{-1}^{1} f(x)w(x)dx - \sum_{i=0}^{n} A_i f(x_i) = \frac{f^{(2n+2)}(\xi)}{(2n+2)!}\int_{-1}^{1} p^2(x)w(x)dx.$$

As expected, if f belongs to $\mathcal{P}_{2n+1}$, the error term vanishes.

23.4 Exercises

From the textbook: 1.a.c.d.h, 2.a.c.d.h, 4.a.c.d.h, 7 p 226.

1. Is is true that if
$$\int_a^b f(x)\, w(x)\, dx \approx \sum_{i=0}^n A_i f(x_i)$$
is exact on $\mathcal{P}_{2n+1}$, then the polynomial $(x - x_0) \cdots (x - x_n)$ is orthogonal to the space $\mathcal{P}_n$ on $[a, b]$ with respect to the weight function w?

2. Find a formula of the form
$$\int_0^1 x\, f(x)\, dx \approx \sum_{i=0}^n A_i f(x_i)$$
with $n = 1$ that is exact for all polynomials of degree at most 3. Repeat with $n = 2$, making the formula exact on $\mathcal{P}_5$.

Part VI

Solving ordinary differential equations

Chapter 24

Difference equations

Many numerical algorithms, in particular those derived from the discretization of differential equations, are designed to produce a sequence $x = (x_n)$ of numbers. Often, the sequence x obeys an $(m+1)$-term recurrence relation of the type

(24.1) $$x_{n+m} = a_{n,m-1}\, x_{n+m-1} + \cdots + a_{n,0}\, x_n, \qquad \text{all } n \in \mathbb{N}.$$

We will establish that the solutions of the **difference equation** (24.1) can be given in explicit form when the coefficients $a_{n,m-1}, \ldots, a_{n,0}$ are constant, i.e. independent of n. But observe first that the set $\mathcal{S}$ of all real sequences satisfying (24.1) is a linear subspace of the space $\mathbb{R}^{\mathbb{N}}$ of all real sequences. In fact, this linear subspace has dimension m, since the linear map

$$x \in \mathcal{S} \mapsto [x_1, x_2, \ldots, x_m]^\top \in \mathbb{R}^m$$

is a bijection [check this fact]. The latter simply says that a solution of (24.1) is uniquely determined by its first m terms.

If the coefficients of the recurrence relation are constant, we may rewrite (24.1) as

(24.2) $$x_{n+m} + c_{m-1}\, x_{n+m-1} + \ldots + c_1\, x_{n+1} + c_0\, x_n = 0, \qquad \text{all } n \in \mathbb{N}.$$

To this equation, we associate the **characteristic polynomial**

$$p(\lambda) := \lambda^m + c_{m-1}\lambda^{m-1} + \cdots + c_1\lambda + c_0.$$

Theorem 24.1. Suppose that the characteristic polynomial p has m simple zeros $\lambda_1, \ldots, \lambda_m$. Then the sequences

$$\begin{aligned} x^{[1]} &= (1, \lambda_1, \lambda_1^2, \ldots, \lambda_1^n, \ldots), \\ &\vdots \\ x^{[m]} &= (1, \lambda_m, \lambda_m^2, \ldots, \lambda_m^n, \ldots), \end{aligned}$$

form a basis for the vector space $\mathcal{S}$ of solutions of (24.2).

Proof. First of all, we need to check that each sequence x^i, $i \in [\![1, n]\!]$, is a solution of (24.2). To see this, we simply write

$$x^{[i]}_{n+m} + c_{m-1} x^{[i]}_{n+m-1} + \cdots + c_0 x^{[i]}_n = \lambda_i^{n+m-1} + c_{m-1}\lambda_i^{n+m-2} + \cdots + c_0 \lambda_i^{n-1} = \lambda_i^{n-1} \cdot p(\lambda_i) = 0.$$

Now, since the system $(x^{[1]}, \ldots, x^{[m]})$ has cardinality equal to the dimension of $\mathcal{S}$, we only need to verify that it is linearly independent. If it was not, then the system composed of $[1, \lambda_1, \ldots, \lambda_1^{m-1}]^\top, \ldots, [1, \lambda_m, \ldots, \lambda_m^{m-1}]^\top$ would be linearly dependent, which is impossible [remember Vandermonde matrices]. This completes the proof. □

Let us illustrate how this result is used in practice. Consider the **Fibonacci sequence** F defined by

$$F_1 = 1, \qquad F_2 = 1, \qquad F_n = F_{n-1} + F_{n-2} \quad \text{for } n \geq 3.$$

The first ten terms are 1, 1, 2, 3, 5, 8, 13, 21, 34, 55. We now emphasize a closed-form expression for the Fibonacci number F_n. According to Theorem 24.1, the two sequences $x^{[1]} = (\mu^{n-1})$ and $x^{[2]} = (\nu^{n-1})$ form a basis of the space $\{x \in \mathbb{R}^{\mathbb{N}} : \; x_n = x_{n-1} + x_{n-2}\}$, where μ and ν are the roots of the quadratic polynomial $\lambda^2 - \lambda - 1$, that is

$$\mu = \frac{1-\sqrt{5}}{2} \qquad \text{and} \qquad \nu = \frac{1+\sqrt{5}}{2} \quad \text{[golden ratio]}.$$

It follows that $F = \alpha \cdot x^{[1]} + \beta \cdot x^{[2]}$ for some constants α and β. These are determined by

$$\begin{cases} F_1 = \alpha \cdot x_1^{[1]} + \beta \cdot x_1^{[2]}, \\ F_2 = \alpha \cdot x_2^{[1]} + \beta \cdot x_2^{[2]}, \end{cases} \qquad \text{i.e.} \qquad \begin{cases} 1 = \alpha \cdot 1 + \beta \cdot 1, \\ 1 = \alpha \cdot \mu + \beta \cdot \nu. \end{cases}$$

We obtain $\alpha = -\dfrac{\mu}{\sqrt{5}}$ and $\beta = \dfrac{\nu}{\sqrt{5}}$. Finally, we conclude

$$F_n = \frac{1}{\sqrt{5}}[\nu^n - \mu^n] = \frac{1}{\sqrt{5}}[\nu^n - (1-\nu)^n].$$

For completeness, we now formulate a slight generalization of Theorem 24.1. Its proof is based on the one of Theorem 24.1, and is therefore omitted.

Theorem 24.2. Suppose that the polynomial p has k zeros of respective multiplicities $m_1, \ldots, m_k$, with $m_1 + \cdots + m_k = k$. Then the sequences

$$(1, \lambda, \ldots, \lambda^n, \ldots)_{|\lambda=\lambda_i}, \quad \frac{d}{d\lambda}(1, \lambda, \ldots, \lambda^n, \ldots)_{|\lambda=\lambda_i}, \quad \ldots \quad , \quad \frac{d^{m_i-1}}{d\lambda^{m_i-1}}(1, \lambda, \ldots, \lambda^n, \ldots)_{|\lambda=\lambda_i},$$

where i runs through $[\![1, k]\!]$, form a basis for the vector space $\mathcal{S}$ of solutions of (24.2).

24.1 Exercises

1. Give bases consisting of real sequences for the solution spaces of $x_{n+2} = 2x_{n+1} - 3x_n$ and $x_{n+3} = 3x_{n+2} - 4x_n$.
2. Consider the difference equation $x_{n+2} - 2x_{n+1} - 2x_n = 0$. One of its solutions is $x_n = (1-\sqrt{3})^{n-1}$. This solution oscillates in sign and converges to zero. Compute and print out the first 100 terms of this sequence by use of the equation $x_{n+2} = 2(x_{n+1} + x_n)$ starting with $x_1 = 1$ and $x_2 = 1 - \sqrt{3}$. Explain the curious phenomenon that occurs.
3. Setting $x(\lambda) := (1, \lambda, \ldots, \lambda^n, \ldots)$, explain in details why the system $\big(x(\lambda_1), \ldots, x(\lambda_m)\big)$ is linearly independent whenever the numbers $\lambda_1, \ldots, \lambda_m$ are distinct.
4. Consider the difference equation $4x_{n+2} - 8x_{n+1} + 3x_n = 0$. Find the general solution and determine whether the difference equation is **stable** – meaning that every solution x is bounded, that is to say $\sup_n |x_n| < \infty$. Assuming that $x_1 = 0$ and $x_2 = -2$, compute x_{101} using the most efficient method.

Chapter 25

Euler's method for initial-value problems

An **ordinary differential equation** – in short, an **ODE** – is an equation that involves one or more derivatives of an unknown univariate function. For example, y is understood to be a function of t in the ordinary differential equation $y' - y = \exp(t)$, which admits $y(t) = (t + C)\exp(t)$ as a solution, C being an arbitrary constant. This indicates that, without any further conditions, the solution of a differential equation is not unique. Hence we often impose some **initial conditions**, i.e. we specify the values of [the derivatives of] the unknown function at a given point, to make the solution unique – see next section. In dynamics, for instance, the motion of a particle, which is governed by a second order differential equation, is completely determined by its initial position and velocity. To describe such **initial-value problems**, we often adopt the formalism

$$\frac{dy}{dt} = f(t, y), \quad t \in [a, b], \quad y(a) = y_0.$$

Note that we are abusing the notations by writing $f(t, y)$ instead of $f\big(t, y(t)\big)$. In many situations, we lack tools to express the solution of this equation analytically, so we settle for a numerical solution. In any case, this is preferable to certain types of 'explicit' formulae. For example, even though a solution of the equation

$$\frac{dy}{dt} = g(t) \cdot y(t)$$

could be written as

$$y(t) = C \exp\Big(\int_a^t g(u)du\Big),$$

this is of no practical use if the integration cannot be carried out for the specific function g.

25.1 Existence and uniqueness of solutions

Before attempting to find approximations to the solution of a differential equation, it would be wise to ask oneself if a solution exists and is unique. Although the answer might be negative, existence and uniqueness are usually ensured under rather mild assumptions.

Theorem 25.1. If the function $f = f(t, y)$ is continuous on the strip $[a, b] \times (-\infty, \infty)$ and satisfies a **Lipschitz condition** in its second variable, that is

$$|f(t, y_1) - f(t, y_2)| \leq L\,|y_1 - y_2| \qquad \text{for some constant } L \text{ and all } t \in [a, b],\ y_1, y_2 \in (-\infty, \infty),$$

then the initial-value problem

$$\frac{dy}{dt} = f(t, y), \quad t \in [a, b], \quad y(a) = y_0.$$

has a unique solution in the interval $[a, b]$.

Let us mention that, in order to check that a Lipschitz condition is satisfied, we may verify that $\sup_{t,y} \left|\frac{\partial f}{\partial y}(t, y)\right| < \infty$ and use the mean value theorem. Note also that, under the same assumptions as in Theorem 25.1, we can even conclude that the initial-value problem is **well-posed**, in the sense that it has a unique solution y and that there exist constants $\varepsilon_0 > 0$ and $K > 0$ such that for any $\varepsilon \in (0, \varepsilon_0)$, any number γ with $|\gamma| < \varepsilon$, and any continuous function δ with $\sup_{t \in [a,b]} |\delta(t)| < \varepsilon$, the initial-value problem

$$\frac{dz}{dt} = f(t, z) + \delta(t), \quad t \in [a, b], \quad z(a) = y_0 + \gamma,$$

has a unique solution z, which further satisfies $\sup_{t \in [a,b]} |z(t) - y(t)| \leq K\,\varepsilon$. In other words, a small perturbation in the problem, e.g. due to roundoff errors in the representations of f and y_0, does not significantly perturb the solution.

25.2 Euler's method

Many numerical methods to solve the initial-value problem

$$y' = f(t, y), \quad t \in [a, b], \quad y(a) = y(0),$$

are more efficient than Euler's method, so that one rarely uses it for practical purposes. For theoretical purposes, however, it has the advantage of its simplicity. The output of

the method does not consist of an approximation of the solution function y, but rather of approximations y_i of the values $y(t_i)$ at some **mesh points** $a = t_0 < t_1 < \cdots < t_{n-1} < t_n = b$. These points are often taken to be equidistant, i.e. $t_i = a + i\,h$, where $h = \dfrac{b-a}{n}$ is the **step size**. The approximations y_i are constructed one-by-one via the difference equation

$$y_{i+1} = y_i + h\, f(t_i, y_i).$$

The simple approximation

$$f(t_i, y(t_i)) = y'(t_i) \approx \frac{y(t_{i+1}) - y(t_i)}{h}$$

was used here to get rid of the derivative. The associated pseudocode is trivial to write, therefore omitted. We concentrate instead on a convergence analysis of the method.

Theorem 25.2. Let us assume that the function $f = f(t, y)$ is continuous on the strip $[a, b] \times (-\infty, \infty)$ and satisfies a Lipschitz condition with constant L in its second variable. Let y be the unique solution of the initial-value problem

$$\frac{dy}{dt} = f(t, y), \quad t \in [a, b], \quad y(a) = y_0.$$

If y is twice differentiable and $\sup\limits_{t \in [a,b]} |y''(t)| =: M < \infty$, then, for each $i \in [\![0, n]\!]$, one has

$$|y(t_i) - y_i| \le \frac{hM}{2L}[\exp\big(L(t_i - a)\big) - 1],$$

where $y_0, \ldots, y_n$ are the approximations generated by Euler's method on the uniform mesh $t_0, \ldots, t_n$ with step size $h = \dfrac{b-a}{n}$.

Proof. Given $i \in [\![0, n-1]\!]$, we can write, for some $\xi_i \in (t_i, t_{i+1})$,

$$\begin{aligned}
|y(t_{i+1}) - y_{i+1}| &= |\big(y(t_i) + h\, y'(t_i) + \frac{h^2}{2} y''(\xi_i)\big) - \big(y_i + h\, f(t_i, y_i)\big)| \\
&= |y(t_i) - y_i + h\,(y'(t_i) - f(t_i, y_i)) + \frac{h^2}{2} y''(\xi_i)| \\
&= |y(t_i) - y_i + h\,(f(t_i, y(t_i)) - f(t_i, y_i)) + \frac{h^2}{2} y''(\xi_i)| \\
&\le |y(t_i) - y_i| + h\,|f(t_i, y(t_i)) - f(t_i, y_i)| + \frac{h^2}{2} |y''(\xi_i)| \\
&\le |y(t_i) - y_i| + h\,L\,|y(t_i) - y_i| + \frac{h^2}{2} M.
\end{aligned}$$

To simplify the notations, set $u_i := |y(t_i) - y_i|$, $a := hL$, and $b := h^2 M/2$. Then the previous inequality takes the form $u_{i+1} \le (1 + a)u_i + b$, which is equivalent [add a constant c to be

chosen smartly] to the inequality $u_{i+1} + b/a \leq (1+a)\big(u_i + b/a\big)$. It follows by immediate induction that

$$u_i + \frac{b}{a} \leq (1+a)^i \Big(u_0 + \frac{b}{a}\Big), \qquad \text{i.e.} \qquad u_i + \frac{hM}{2L} \leq (1+hL)^i \, \frac{hM}{2L}.$$

To obtain the announced result, remark that $(1+hL)^i \leq \exp(ihL) = \exp\big(L(t_i - a)\big)$. □

Note that, y'' being a priori unknown, we cannot give a precise evaluation for M. If f is 'well-behaved', however, we can differentiate the relation $y'(t) = f(t, y(t))$ and estimate M from there. It should also be noted that, because of roundoff errors, decreasing the step size h beyond a certain critical value will not improve the accuracy of the approximations.

25.3 Exercises

From the textbook: 1.a.c., 2.a.c. p 255; 8.a.c. p 256; 1.b.c. p 263; 9 p 264; 12 p 265.

Optional problems

From the textbook: 9 p 256; 16 p 265.

25.4 Taylor series methods

To be completed.

Chapter 26

Runge–Kutta methods

The methods named after Carl Runge and Wilhelm Kutta are designed to imitate the Taylor series methods without requiring analytic differentiation of the original differential equation. Performing preliminary work before writing the computer program can indeed be a serious obstacle. An ideal method should involve nothing more that writing a code to evaluate f, and the **Runge–Kutta methods** accomplish just that. We will present the Runge–Kutta method of order two, even though its low precision usually precludes its use in actual scientific computations – it does find application in real-time calculations on small computers, however. Then the Runge–Kutta method of order four will be given, without any form of proof.

26.1 Taylor series for $f(x, y)$

The **bivariate Taylor series**, analogous to the usual univariate one, takes the form

$$f(x+h, y+k) = \sum_{i=0}^{\infty} \frac{1}{i!} \left(\left[h\frac{\partial}{\partial x} + k\frac{\partial}{\partial y} \right]^i f \right) (x, y),$$

where the operators appearing in this equation are

$$\begin{aligned}
\left[h\frac{\partial}{\partial x}+k\frac{\partial}{\partial y}\right]^0 f &= f\\
\left[h\frac{\partial}{\partial x}+k\frac{\partial}{\partial y}\right]^1 f &= h\frac{\partial f}{\partial x}+k\frac{\partial f}{\partial y}\\
\left[h\frac{\partial}{\partial x}+k\frac{\partial}{\partial y}\right]^2 f &= h^2\frac{\partial^2 f}{\partial x^2}+2hk\frac{\partial^2 f}{\partial x\partial y}+k^2\frac{\partial^2 f}{\partial x^2}\\
&\vdots
\end{aligned}$$

If the series is truncated, an error term is needed to restore equality. We would have

$$f(x+h,y+k)=\sum_{i=0}^{n-1}\frac{1}{i!}\left(\left[h\frac{\partial}{\partial x}+k\frac{\partial}{\partial y}\right]^i f\right)(x,y)+\frac{1}{n!}\left(\left[h\frac{\partial}{\partial x}+k\frac{\partial}{\partial y}\right]^n f\right)(\overline{x},\overline{y}),$$

where the point $(\overline{x},\overline{y})$ lies in the line segment joining (x,y) and $(x+h,y+k)$. It is customary to use subscripts to denote partial derivatives, for instance

$$f_x:=\frac{\partial f}{\partial x},\qquad f_y:=\frac{\partial f}{\partial y},\qquad f_{xy}=\frac{\partial^2 f}{\partial x\partial y}\left(=\frac{\partial^2 f}{\partial y\partial x}\right).$$

Hence we may write the Taylor expansion of f [check it on $f(x,y)=(x+y)^2$] as

$$f(x+h,y+k) = f(x,y)+hf_x(x,y)+kf_y(x,y)+\frac{1}{2}\Big(h^2f_{xx}(x,y)+2hkf_{xy}(x,y)+k^2f_{yy}(x,y)\Big)+\cdots$$

26.2 Runge–Kutta method of order two

In order to solve the initial-value problem

$$y'=f(t,y),\qquad t\in[a,b],\qquad y(a)=y_0,$$

we need a procedure for advancing the solution function one step at a time. For the Runge–Kutta method of order two, the formula for $y(t+h)$ in terms of known quantities is

(26.1) $$y(t+h)=y(t)+w_1K_1+w_2K_2,\qquad \text{with}\begin{cases}K_1 := hf(t,y),\\ K_2 := hf(t+\alpha h,y+\beta K_1).\end{cases}$$

The objective is to determine the constants w_1, w_2, α, and β so that (26.1) is as accurate as possible, i.e. reproduces as many terms as possible in the Taylor expansion

(26.2) $$y(t+h)=y(t)+y'(t)h+\frac{y''(t)}{2}h^2+\frac{y'''(t)}{6}h^3+\cdots.$$

One obvious way to force (26.1) and (26.2) to agree up through the term in h is by taking $w_1 = 1$ and $w_2 = 0$. This is Euler's method. One can improve this to obtain agreement up through the term in h^2. Let us apply the bivariate Taylor theorem in (26.1) to get

$$\begin{aligned} y(t+h) &= y(t) + w_1 h f(t,y) + w_2 h\big(f(t,y) + \alpha h f_t(t,y) + \beta h f(t,y) f_y(t,y) + \mathcal{O}(h^2)\big) \\ &= y(t) + (w_1 + w_2) f(t,y) h + \big(\alpha w_2 f_t(t,y) + \beta w_2 f(t,y) f_y(t,y)\big) h^2 + \mathcal{O}(h^3). \end{aligned}$$

Making use of $y''(t) = f_t(t,y) + f(t,y)f_y(t,y)$, Equation (26.2) can also be written as

$$y(t+h) = y(t) + f(t,y)h + \Big(\frac{1}{2} f_t(t,y) + \frac{1}{2} f(t,y) f_y(t,y)\Big) h^2 + \mathcal{O}(h^3).$$

Thus, we shall require

$$w_1 + w_2 = 1, \qquad \alpha w_2 = \frac{1}{2}, \qquad \beta w_2 = \frac{1}{2}.$$

Among the convenient solutions, there are the choice $\alpha = 1$, $\beta = 1$, $w_1 = 1/2$, $w_2 = 1/2$, resulting in the **second-order Runge–Kutta** method [also known as **Heun's method**]

$$y(t+h) = y(t) + \frac{h}{2} f(t,y) + \frac{h}{2} f\big(t+h, y + hf(t,y)\big),$$

and the choice $\alpha = 1/2$, $\beta = 1/2$, $w_1 = 0$, $w_2 = 1$, resulting in the **modified Euler's method**

$$y(t+h) = y(t) + hf\big(t + h/2, y + h/2\, f(t,y)\big).$$

26.3 Runge–Kutta method of order four

The classical **fourth-order Runge–Kutta method** is of common use when solving an initial-value problem. It is based on the approximation formula

$$y(t+h) = y(t) + \frac{1}{6}(K_1 + 2K_2 + 2K_3 + K_4), \qquad \text{with} \begin{cases} K_1 := hf(t,y), \\ K_2 := hf(t + h/2, y + K_1/2), \\ K_3 := hf(t + h/2, y + K_2/2), \\ K_4 := hf(t + h, y + K_3). \end{cases}$$

Despite its elegance, this formula is tedious to derive, and we shall not do so. The method is termed fourth-order because it reproduces the terms in the Taylor series up to and including the one involving h^4, so that the [local truncation] error is in h^5. Note that the solution at $y(t+h)$ is obtained at the expense of four function evaluations. Here is a possible pseudocode to be implemented.

input f, t, y, h, n
$t_0 \leftarrow t$, **output** 0, t, y
for $i = 1$ to n **do**
    $K_1 \leftarrow hf(t, y)$
    $K_2 \leftarrow hf(t + h/2, y + K_1/2)$
    $K_3 \leftarrow hf(t + h/2, y + K_2/2)$
    $K_4 \leftarrow hf(t + h, y + K_3)$
    $y \leftarrow y + (K_1 + 2K_2 + 2K_3 + K_4)/6$
    $t \leftarrow t_0 + ih$
    output i, t, y
end do end for

26.4 Exercises

From the textbook: 2.a.c., 14.a.c. p 281; 27, 28 p 282.

Optional problems

From the textbook: 31 p 283.

Chapter 27

Multistep methods

27.1 Description of the method

The Taylor series and the Runge–Kutta methods for solving an initial-value problem are **singlestep methods**, in the sense that they do not use any knowledge of prior values of y when the solution is being advanced from t to $t+h$. Indeed, if $t_0, t_1, \ldots, t_i, t_{i+1}$ are steps along the t-axis, then the approximation of y at t_{i+1} depends only on the approximation of y at t_i, while the approximations of y at $t_{i-1}, t_{i-2}, \ldots, t_0$ are not used. Taking these values into account should, however, provide more efficient procedures. We present the basic principle involved here on a particular case. Suppose we want to solve numerically the initial-value problem $y'(t) = f(t, y)$, with $y(t_0) = y_0$. We prescribe the [usually equidistant] mesh points $t_0, t_1, \ldots, t_n$. By integrating on $[t_i, t_{i+1}]$, we have

$$y(t_{i+1}) = y(t_i) + \int_{t_i}^{t_{i+1}} f(t, y(t))dt.$$

This integral can be approximated by a numerical quadrature scheme involving the values $f(t_j, y(t_j))$ for $j \leq i$. Thus we will define y_{i+1} by a formula of the type

$$y_{i+1} = y_i + af_i + bf_{i-1} + cf_{i-2} + \cdots ,$$

where $f_j := f(t_j, y_j)$. An equation of this type is called an **Adams–Bashforth formula**. Let us determine the coefficients appearing in this equation for a five-step method [meaning that one step involves the five preceding ones]. To be determined are the coefficients a, b, c, d, e in the quadrature formula

$$\int_0^1 F(x)dx \approx aF(0) + bF(-1) + cF(-2) + dF(-3) + eF(-4), \tag{27.1}$$

which we require to be exact on $\mathcal{P}_4$. We follow the **method of undetermined coefficients**, working with the basis of $\mathcal{P}_4$ consisting of

$$\begin{aligned} p_0(x) &= 1, \\ p_1(x) &= x, \\ p_2(x) &= x(x+1), \\ p_3(x) &= x(x+1)(x+2), \\ p_4(x) &= x(x+1)(x+2)(x+3). \end{aligned}$$

Substituting p_0, p_1, p_2, p_3, p_4 into (27.1) results in the system of equations

$$\begin{cases} a + b + c + d + e = 1 \\ \quad - b - 2c - 3d - 4e = 1/2 \\ \quad\quad 2c + 6d + 12e = 5/6 \\ \quad\quad\quad - 6d - 24e = 9/4 \\ \quad\quad\quad\quad 24e = 251/30 \end{cases}$$

It yields $e = 251/720$, $d = -1274/720$, $c = 2616/720$, $b = -2774/720$, $a = 1901/720$. Finally, the **Adams–Bashforth formula of order five** is

$$y_{i+1} = y_i + \frac{h}{720}\big[1901\ f_i - 2774\ f_{i-1} + 2616\ f_{i-2} - 1274\ f_{i-3} + 251\ f_{i-4}\big].$$

Observe that only y_0 is available at the start, while the procedure requires y_0, y_1, y_2, y_3, y_4 to be initiated. A Runge–Kutta method is ideal to obtain these values.

27.2 A catalog of multistep methods

In general, a $(m+1)$-step method is based on the format

(27.2) $$y_{i+1} = a_m\ y_i + \cdots + a_0\ y_{i-m}\ +\ h\ \big[b_{m+1}\ f_{i+1} + b_m\ f_i + \cdots + b_0\ f_{i-m}\big].$$

If the coefficient b_{m+1} is zero, then the method is said to be **explicit**, since y_{i+1} does not appear on the right-hand side, thus y_{i+1} can be computed directly from (27.2). If otherwise the coefficient b_{m+1} is nonzero, then y_{i+1} appears on the right-hand side of (27.2) by way of $f_{i+1} = f(t_{i+1}, y_{i+1})$, and Equation (27.2) determines y_{i+1} implicitly. The method is therefore said to be **implicit**.

We now list, for reference only, various Adams–Bashforth and **Adams–Moulton** formulae. The latter are derived just as the former would be – see Section 27.1 – except that the

quadrature formula for $\int_0^1 F(x)dx$ also incorporates the value $F(1)$. This implies that the Adams–Moulton methods are implicit, by opposition to the explicit Adams–Bashforth methods.

[Beware, the following are prone to typographical mistakes!]

- **Adams–Bashforth two-step method**

$$y_{i+1} = y_i \ + \ \frac{h}{2} \left[3\, f_i - f_{i-1}\right]$$

  Local truncation error: $\frac{5}{12}\, y^{(3)}(\xi_i)\, h^2$, for some $\xi_i \in (t_{i-1}, t_{i+1})$.

- **Adams–Bashforth three-step method**

$$y_{i+1} = y_i \ + \ \frac{h}{12} \left[23\, f_i - 16\, f_{i-1} + 5\, f_{i-2}\right]$$

  Local truncation error: $\frac{3}{8}\, y^{(4)}(\xi_i)\, h^3$, for some $\xi_i \in (t_{i-2}, t_{i+1})$.

- **Adams–Bashforth four-step method**

$$y_{i+1} = y_i \ + \ \frac{h}{24} \left[55\, f_i - 59\, f_{i-1} + 37\, f_{i-2} - 9\, f_{i-3}\right]$$

  Local truncation error: $\frac{251}{720}\, y^{(5)}(\xi_i)\, h^4$, for some $\xi_i \in (t_{i-3}, t_{i+1})$.

- **Adams–Bashforth five-step method**

$$y_{i+1} = y_i \ + \ \frac{h}{720} \left[1901\, f_i - 2774\, f_{i-1} + 2616\, f_{i-2} - 1274\, f_{i-3} + 251\, f_{i-4}\right]$$

  Local truncation error: $\frac{95}{288}\, y^{(6)}(\xi_i)\, h^5$, for some $\xi_i \in (t_{i-4}, t_{i+1})$.

- **Adams–Moulton two-step method**

$$y_{i+1} = y_i \ + \ \frac{h}{12} \left[5\, f_{i+1} + 8\, f_i - f_{i-1}\right]$$

  Local truncation error: $-\frac{1}{24}\, y^{(4)}(\xi_i)\, h^3$, for some $\xi_i \in (t_{i-1}, t_{i+1})$.

- **Adams–Moulton three-step method**

$$y_{i+1} = y_i \ + \ \frac{h}{24} \left[9\, f_{i+1} + 19\, f_i - 5\, f_{i-1} + f_{i-2}\right]$$

  Local truncation error: $-\frac{19}{720}\, y^{(5)}(\xi_i)\, h^4$, for some $\xi_i \in (t_{i-2}, t_{i+1})$.

- **Adams–Moulton four-step method**

$$y_{i+1} = y_i \; + \; \frac{h}{720} \left[251\, f_{i+1} + 646\, f_i - 264\, f_{i-1} + 106\, f_{i-2} - 19\, f_{i-3}\right]$$

  Local truncation error: $-\dfrac{3}{160}\, y^{(6)}(\xi_i)\, h^5$, for some $\xi_i \in (t_{i-3}, t_{i+1})$.

Note that the order of the local truncation error of the Adams–Moulton m-step method matches the one of the Adams–Bashforth $(m+1)$-step method, for the same number of function evaluations. Of course, this has a price, namely solving the equation in y_{i+1}.

27.3 Predictor–corrector method

In numerical practice, the Adams–Bashforth formulae are rarely used by themselves, but in conjunction with other formulae to enhance the precision. For instance, the Adams–Bashforth five-step formula can be employed to *predict* a tentative estimate for the value y_{i+1} given by the Adams–Moulton five-step formula, denote this estimate by y^*_{i+1}, and use it to replace $f_{i+1} = f(t_{i+1}, y_{i+1})$ by its approximation $f^*_{i+1} = f(t_{i+1}, y^*_{i+1})$ in the Adams–Moulton formula, thus yielding a *corrected* estimate for y_{i+1}. This general principle provides satisfactory algorithms, called **predictor–corrector methods**. In fact, remark that y_{i+1} should be interpreted as a fixed point of a certain mapping ϕ, and as such may be computed, under appropriate hypotheses, as the limit of the iterates $\phi^k(z)$, for suitable z. Note that the predictor–corrector simply estimates y_{i+1} by $\phi(y^*_{i+1})$. A better estimate would obviously be obtained by applying ϕ over again, but in practice only one or two further iterations will be performed. To accompany this method, we can suggest the following pseudocode.

27.4 Exercises

From the textbook: 3.b.d., 7.b.d., 8.b.d. p 301; 13, 14 p 302.

Chapter 28

Systems of higher-order ODEs

28.1 Reduction to first-order ordinary differential equations

A system of first-order ordinary differential equations takes the standard form

$$\begin{cases} y_1' &= f_1(t, y_1, \ldots, y_n), \\ y_2' &= f_2(t, y_1, \ldots, y_n), \\ &\vdots \\ y_n' &= f_n(t, y_1, \ldots, y_n). \end{cases} \tag{28.1}$$

As an example, we may consider a simple version of the **predator–prey model**, in which x stands for the number of preys and y for the number of predators. The equations are

$$\begin{cases} x' = a\,x - b\,x\,y \\ y' = c\,x\,y - d\,y \end{cases}, \qquad \text{for some nonnegative paramaters } a, b, c, d.$$

As another example, we may consider a **linear** system of differential equations, such as

$$\begin{cases} x' = x + 4y - e^t, \\ y' = x + y + 2e^t, \end{cases}$$

whose general solution is

$$\begin{aligned} x(t) &= 2ae^{3t} - 2be^{-t} - 2e^t, \\ y(t) &= ae^{3t} + be^{-t} + e^t/4, \end{aligned} \qquad a, b \text{ arbitrary constants.}$$

These constants are to be determined from a set of two conditions, e.g. the initial conditions $x(0) = 4$ and $y(0) = 5/4$. In general, a set of n initial conditions $y_i(t_0) = y_{0,i}$, $i \in [\![1, n]\!]$, will come with the system (28.1).

Observe now that this system can be rewritten using a condensed vector notation, setting

$$Y := \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \text{ – a mapping from } \mathbb{R} \text{ into } \mathbb{R}^n, \quad F := \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix} \text{ – a mapping from } \mathbb{R} \times \mathbb{R}^n \text{ into } \mathbb{R}^n,$$

thus yielding the more convenient form

$$Y' = F(t, Y).$$

The initial conditions then translates into the vector equality $Y(t_0) = Y_0$.

A higher-order differential equation can also be converted into this vector format, by transforming it first into a system of first-order ordinary differential equations. Indeed, for the differential equation

$$y^{(n)} = f(t, y, y', \ldots, y^{(n-1)}),$$

we may set $x_1 := y, x_2 := y', \ldots, x_n := y^{(n-1)}$, so that one has

$$\begin{cases} x_1' &= & x_2, \\ & \vdots & \\ x_{n-1}' &= & x_n, \\ x_n' &= & f(t, x_1, \ldots, x_n). \end{cases}$$

The same approach can of course be applied to systems of higher-order ordinary differential equations. To illustrate this point, one may check that the system

$$\begin{cases} (x'')^2 + t\, e^y + y' &= & x' - x, \\ y'\, y'' - \cos(x\, y) + \sin(t\, x\, y) &= & x, \end{cases}$$

can be rewritten as

$$\begin{cases} x_1' &= & x_2, \\ x_2' &= & [x_2 - x_1 - x_4 - t\, e^{x_3}]^{1/2}, \\ x_3' &= & x_4, \\ x_4' &= & [x_1 - \sin(t\, x_2\, x_3) + \cos(x_1\, x_3)]/x_4. \end{cases}$$

Remark also that any system of the form (28.1) can be transformed into an **autonomous** system, i.e. a system that does not contains t explicitely. It suffices for that to introduce the function $y_0(t) = t$, to define $Y = [y_0, y_1, \ldots, y_n]^\top$ and $F = [1, f_1, \ldots, f_n]^\top$, so that (28.1) is expressed as $Y' = F(Y)$.

28.2 Extensions of the methods

28.2.1 Taylor series method for systems

We write a truncated Taylor expansion for each of the unknown functions in (28.1) as

$$y_i(t+h) \approx y_i(t) + hy_i'(t) + \frac{h^2}{2}y_i''(t) + \cdots + \frac{h^n}{n!}y_i^{(n)}(t),$$

or, in the shorter vector notation,

$$Y(t+h) \approx Y(t) + hY'(t) + \frac{h^2}{2}Y''(t) + \cdots + \frac{h^n}{n!}Y^{(n)}(t).$$

The derivatives appearing here are to be computed from the differential equation before being included [in the right order] in the code. For instance, to solve the system

$$\begin{cases} x' &= \quad x + y^2 - t^3, & x(1) = 3, \\ y' &= \quad y + x^3 + \cos(t), & y(1) = 1, \end{cases}$$

numerically on the interval $[0, 1]$, we may follow the pseudocode

input a, b, α, β, n
$t \leftarrow b$, $x \leftarrow \alpha$, $y \leftarrow \beta$, $h \leftarrow (b-a)/n$
output t, x, y
for $k = 1$ to n **do**
    $x1 \leftarrow x + y^2 - t^3$
    $y1 \leftarrow y + x^3 + \cos(t)$
    $x2 \leftarrow x1 + 2yy1 - 3t^2$
    $y2 \leftarrow y1 + 3x^2x1 - \sin(t)$
    $x3 \leftarrow x2 + 2yy2 + 2(y1)^2 - 6t$
    $y3 \leftarrow y2 + 6x(x1)^2 + 3x^2x2 - \cos(t)$
    $x \leftarrow x - h(x1 - h/2(x2 - h/3(x3)))$
    $y \leftarrow y - h(y1 - h/2(y2 - h/3(y3)))$
    $t \leftarrow b - kh$
    output t, x, y
end do end for

28.2.2 Runge–Kutta method for systems

For an autonomous system, the classical fourth-order Runge–Kutta formula reads, in vector form,

$$Y(t+h) \approx Y(t) + \frac{1}{6}[F_1 + 2F_2 + 2F_3 + F_4], \qquad \text{with } \begin{cases} F_1 &= h\,F(Y(t)), \\ F_2 &= h\,F(Y(t) + F_1/2), \\ F_3 &= h\,F(Y(t) + F_2/2), \\ F_4 &= h\,F(Y(t) + F_3). \end{cases}$$

28.2.3 Multistep method for systems

Adapting the usual case turns out to be straightforward, too. The vector form of the Adams–Bashforth–Moulton predictor–corrector method of order five, for example, reads

$$\begin{aligned} Y^*(t+h) &\approx Y(t) + \frac{h}{720}\big[1901\,F(Y(t)) - 2774\,F(Y(t-h)) + 2616\,F(Y(t-2h)) \\ &\qquad -1274\,F(Y(t-3h)) + 251\,F(Y(t-4h))\big], \\ Y(t+h) &\approx Y(t) + \frac{h}{720}\big[251\,F(Y^*(t+h)) + 646\,F(Y(t)) - 264\,F(Y(t-h)) \\ &\qquad +106\,F(Y(t-2h)) - 19\,F(Y(t-3h))\big]. \end{aligned}$$

28.3 Exercises

From the textbook: 6, 9, 10 p 324.

1. Convert the system of second-order ordinary differential equations

$$\begin{cases} x'' - x'y &= 3y'x\log(t), \\ y'' - 2xy' &= 5x'y\sin(t), \end{cases}$$

   into an autonomous first-order differential equation in vector form.

Chapter 29

Boundary value problems

Consider a second-order differential equation of the type

$$y'' = f(t, y, y'), \qquad t \in [a, b],$$

where the two conditions on y and y' at $t = a$ are replaced by two conditions on y at $t = a$ and at $t = b$, i.e. we specify

$$y(a) = \alpha, \qquad y(b) = \beta.$$

This is an instance of a **two-point boundary-value** problem. It is usually harder to deal with than an initial-value problems, in particular it cannot be solved numerically by the step-by-step methods described earlier. The question of existence and uniqueness of solutions is not easily answered either. For example, the second-order differential equation $y'' + y = 0$ [whose general solution is $c_1 \sin(t) + c_2 \cos(t)$] will have infinitely many solutions if coupled with the boundary conditions $y(0) = 1$ and $y(\pi) = 1$, no solution if coupled with the conditions $y(0) = 1$ and $y(\pi) = 0$, and a unique solution if coupled with the conditions $y(0) = 1$ and $y(\pi/2) = -3$. Existence and uniqueness theorems do exist, however, and we simply mention one of them.

Theorem 29.1. If the functions f, f_y, and $f_{y'}$ are continuous on the domain $D := [a, b] \times (-\infty, \infty) \times (-\infty, \infty)$, and if

1. $f_y(t, y, y') > 0$ for all $(t, y, y') \in D$,
2. $\sup\limits_{(t,y,y') \in D} |f_{y'}(t, y, y')| < \infty$,

then the boundary-value problem

$$y'' = f(t, y, y'), \qquad t \in [a, b], \qquad y(a) = \alpha, \quad y(b) = \beta,$$

admits a unique solution.

As a consequence, we can derive the following result.

Corollary 29.2. The **linear boundary-value problem**

$$y''(t) = p(t)\,y'(t) + q(t)\,y(t) + r(t), \qquad t \in [a,b], \qquad y(a) = \alpha, \quad y(b) = \beta,$$

admits a unique solution as soon as the functions p, q, and r are continuous on $[a,b]$ and $q(t) > 0$ for all $t \in [a,b]$.

This applies e.g. to the equation $y'' = y$, but not to the equation $y'' = -y$, as expected from the previous discussion. It is instructive to attempt a direct proof of Corollary 29.2, and one is advised to do so. We now focus on our primary interest, namely we ask how to solve boundary-value problems numerically.

29.1 Shooting method

Since methods for initial-value problems are at our disposal, we could try and use them in this situation. Precisely, we will make a guess for the initial value $y'(a)$ and solve the corresponding initial-value problem, hoping that the resulting $y(b)$ agrees with β. If not, we alter our guess for $y'(a)$, compute the resulting $y(b)$, and compare it with β. The process is repeated as long as $y(b)$ differs too much from β. This scheme is called the **shooting method**. To formalize the procedure, consider the initial-value problem

$$y'' = f(t, y, y'), \qquad t \in [a,b], \qquad y(a) = \alpha, \quad y'(a) = z,$$

where only z is variable here. Then, assuming existence and uniqueness of the solution y, its value at the right endpoint depends on z. We may write

$$\varphi(z) := y(b), \qquad \text{or} \qquad \psi(z) := y(b) - \beta.$$

Our goal is to solve the equation $\psi(z) = 0$, for which we have already developed some techniques. Note that, in this approach, any algorithm for the initial-value problem can be combined with any algorithm for root finding. The choices are determined by the nature of the problem at hand. We choose here to use the secant method. Thus, we need two initial guesses z_1 and z_2, and we construct a sequence (z_n) from the iteration formula

$$z_{n+1} = z_n - \psi(z_n)\,\frac{z_n - z_{n-1}}{\psi(z_n) - \psi(z_{n-1})}, \qquad \text{i.e.} \quad z_{n+1} = z_n + (\beta - \varphi(z_n))\,\frac{z_n - z_{n-1}}{\varphi(z_n) - \varphi(z_{n-1})}.$$

We will monitor the value of $\psi(z) = \varphi(z) - \beta$ as we go along, and stop when it is sufficiently small. Note that the valuess of the approximations y_i are to be stored until better ones

are obtained. Remark also that the function φ is very expensive to compute, so that in the first stages of the shooting method, as high precision is essentially wasted, a large step size should be used. Let us now observe now that the secant method provides the exact solution in just one step if the function φ is linear. This will occur for linear boundary-value problems. Indeed, if y_1 and y_2 are solutions of

$$y'' = p\,y' + q\,y + r, \qquad t \in [a,b], \qquad y(a) = \alpha, \tag{29.1}$$

with $y_1'(a) = z_1$ and $y_2'(a) = z_2$, then for any λ the function $\widetilde{y} := (1-\lambda)y_1 + \lambda y_2$ is the solution of (29.1) with $\widetilde{y}\,'(a) = (1-\lambda)z_1 + \lambda z_2$. Considering the value of $\widetilde{y}$ at b, we get $\varphi((1-\lambda)z_1 + \lambda z_2) = (1-\lambda)\varphi(z_1) + \lambda\varphi(z_2)$, which characterizes φ as a linear function.

Let us finally recall that the secant method outputs z_{n+1} as the zero of the linear function interpolating ψ at z_{n-1} and z_n [think of the secant line]. An obvious way to refine the shooting method would therefore be to use higher-order interpolation. For example, we could form the cubic polynomial p that interpolates ψ at z_1, z_2, z_3, z_4, and obtain z_5 as the solution of $p(z_5) = 0$. Still with cubic interpolants, an even better refinement involves **inverse interpolation**. This means that we consider the cubic polynomial q that interpolates the data z_1, z_2, z_3, z_4 at the points $\psi(z_1)$, $\psi(z_2)$, $\psi(z_3)$, $\psi(z_4)$, and we obtain z_5 as $z_5 = q(0)$.

29.2 Finite-difference methods

Another approach consists of producing a discrete version of the boundary value problem

$$y'' = f(t, y, y'), \qquad t \in [a,b], \qquad y(a) = \alpha, \quad y(b) = \beta.$$

For this purpose, we partition the interval $[a,b]$ into n equal subintervals of length $h = (b-a)/n$ and we exploit the centered-difference formulae

$$\begin{aligned}
y'(t) &= \frac{1}{2h}\big[y(t+h) - y(t-h)\big] - \frac{h^2}{6}y^{(3)}(\xi), && \xi \text{ between } t-h \text{ and } t+h,\\
y''(t) &= \frac{1}{h^2}\big[y(t+h) - 2y(t) + y(t-h)\big] - \frac{h^2}{12}y^{(4)}(\zeta), && \zeta \text{ between } t-h \text{ and } t+h,
\end{aligned}$$

to construct approximations y_i of the solution at the mesh points $t_i = a + ih$, $i \in [\![0,n]\!]$. Neglecting the error terms, the original problem translates into the system

$$\begin{cases} y_0 = \alpha, \\ y_{i+1} - 2y_i + y_{i-1} = h^2 f\big(t_i, y_i, [y_{i+1} - y_{i-1}]/2h\big), \quad i \in [\![1, n-1]\!], \\ y_n = \beta. \end{cases} \tag{29.2}$$

In the linear case [i.e. for $f(t,y,y') = p(t)y' + q(t)y + r(t)$], the equations (29.2) for the unknowns $y_1, \ldots, y_{n-1}$ can be written in matrix form as

$$\begin{bmatrix} d_1 & b_1 & & & \\ a_2 & d_2 & b_2 & & \\ & \ddots & \ddots & \ddots & \\ & & a_{n-2} & d_{n-2} & b_{n-2} \\ & & & a_{n-1} & d_{n-1} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-2} \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} c_1 - a_0\alpha \\ c_2 \\ \vdots \\ c_{n-2} \\ c_{n-1} - b_{n-1}\beta \end{bmatrix}, \qquad \text{with} \quad \begin{cases} d_i = 2 + h^2 q(t_i), \\ a_i = -1 - hp(t_i)/2, \\ b_i = -1 + hp(t_i)/2, \\ c_i = -h^2 r(t_i). \end{cases}$$

Note that the system is tridiagonal, therefore can be solved with a special Gaussian algorithm. Observe also that, under the hypotheses of Corollary 29.2 and for h small enough, the matrix is diagonally dominant, hence nonsingular, since $|a_i| + |b_i| = 2 < |d_i|$. Under the same hypotheses, we can also show that the discrete solution converges towards the actual solution, in the sense that $\max_{i \in [\![0,n]\!]} |y_i - y(t_i)| \underset{h \to 0}{\longrightarrow} 0$.

In the nonlinear case, the equations (29.2) can be written as

$$F(Y) = 0, \qquad \text{with } Y := [y_1, \ldots, y_{n-1}]^\top, \quad F \text{ an appropriate vector of functions } f_i.$$

Newton's method for nonlinear system can be called upon in this setting [see Section 13.3]. We construct a sequence $(Y^{[k]})$ of vector according to the iteration formula

$$Y^{[k+1]} = Y^{[k]} - F'(Y^{[k]})^{-1} F(Y^{[k]}). \tag{29.3}$$

As in the unidimensional case, convergence is ensured provided that one starts with $Y^{[0]}$ close enough to the solution and that the Jacobian matrix $F'(Y^{[k]})$ is invertible. Recall that the Jacobian matrix displays the entries $\dfrac{\partial f_i}{\partial y_j}$, so that it takes the form

$$\begin{bmatrix} \ddots & \ddots & \ddots \\ -1 + hu_i/2 & 2 + h^2 v_i & -1 - hu_i/2 \\ \ddots & \ddots & \ddots \end{bmatrix}, \quad \text{with} \quad \begin{cases} u_i := f_{y'}(t_i, y_i, [y_{i+1} - y_{i-1}]/2h), \\ v_i := f_y\,(t_i, y_i, [y_{i+1} - y_{i-1}]/2h). \end{cases}$$

This matrix appears to be diagonally dominant, hence invertible, for h small enough provided that we make the assumptions that $f_{y'}$ is bounded and that f_y is positive – see the hypotheses of Theorem 29.1. Then the vector $Y^{[k+1]}$ has to be explicitely computed from (29.3), not by inverting the Jacobian matrix but rather by solving a linear system. This task is not too demanding, since the Jacobian matrix is tridiagonal.

29.3 Collocation method

Remark that the linear boundary-value problem can be expressed as

$$Ly = r, \tag{29.4}$$

where Ly is defined to be $y'' - py' - qy$. The **collocation method** we are about to present only relies on the fact that L is a linear operator. The idea is to try and find a solution of the equation (29.4) in the form

$$y = c_1\, v_1 + \cdots + c_n v_n,$$

where the functions $v_1, \ldots, v_n$ are chosen at the start. By linearity of L, the equation (29.4) becomes

$$\sum_{i=1}^{n} c_i\, Lv_i = r.$$

In general, there is no hope to solve this for the coefficients $c_1, \ldots, c_n$, since the solution y has no reason to be a linear combination of $v_1, \ldots, v_n$. Instead of equality everywhere, we will merely require that the function $\sum_i c_i\, Lv_i$ interpolates the function r at n prescribed points, say $\tau_1, \ldots, \tau_n$. Thus, we need to solve the $n \times n$ system

$$\sum_{i=1}^{n} c_i\, (Lv_i)(\tau_j) = r(\tau_j), \qquad j \in [\![1, n]\!]. \tag{29.5}$$

Let us suppose, for simplicity, that we work on the interval $[0, 1]$ and that $\alpha = 0$ and $\beta = 0$ [see how we can turn to this case anyway]. We select the functions $v_1, \ldots, v_n$ to be

$$v_{i,n} = t^i (1-t)^{n+1-i}, \qquad i \in [\![1, n]\!].$$

The calculation of $Lv_{i,n}$ does not present any difficulties and is in fact most efficient if we use the relations

$$\begin{aligned} v'_{i,n} &= i\, v_{i-1,n-1} - (n+1-i)\, v_{i,n-1} \\ v''_{i,n} &= i(i-1)\, v_{i-2,n-2} - 2i(n+i-1)\, v_{i-1,n-2} + (n+1-i)(n-i)\, v_{i,n-2}. \end{aligned}$$

It is not hard to realize [try it] that this choice of functions $v_1, \ldots, v_n$ leads to a nonsingular collocation system (29.5).

It is more popular, however, to consider some B-splines for the functions $v_1, \ldots, v_n$. A brief account on B-splines will be given separately to this set of notes.

29.4 Exercises

From Chapter 11 of the textbook, of your own liking.

XXXXXXXXXXXXXXXXXXXXXXXXX