Applied Numerical Analysis (AE2220-I)

R. Klees and R.P. Dwight

February 2020

Contents

Chapter 1

Preliminaries: Motivation, Computer arithmetic, Taylor series

1.1 Numerical Analysis Motivation

Numerical analysis is a toolbox of methods to find solutions of analysis problems by purely arithmetic operations, that is $+$, $-$, $\times$, and $\div$. For example, imagine you are performing some structural analysis problem, that requires you to evaluate the definite integral:

$$\tilde{I} = \int_0^\pi x \sin x \, \mathrm{d}x. \tag{1.1}$$

You can solve this exactly by applying the chain-rule, to discover $I = \pi$.

One numerical approach to the same problem is known as the Trapezoidal rule: divide the interval $[0, \pi]$ into n smaller intervals, and approximate the area under the curve in each interval by the area of a trapezoid, see Figure 1.1.

Writing this symbolically we have

$$\begin{aligned} I_n &= \sum_{i=0}^{n-1} h \cdot \frac{f(x_i + h) + f(x_i)}{2}, \\ x_i &= \frac{i\pi}{n}, \\ h &= \frac{\pi}{n} \end{aligned}$$

where $f(x) = x \sin x$, the integrand. The x_i define the edges of the subintervals, and h the width of each subinterval. The accuracy of the approximation $I_n \approx \tilde{I} = 3.14159265359\cdots$ depends on n:

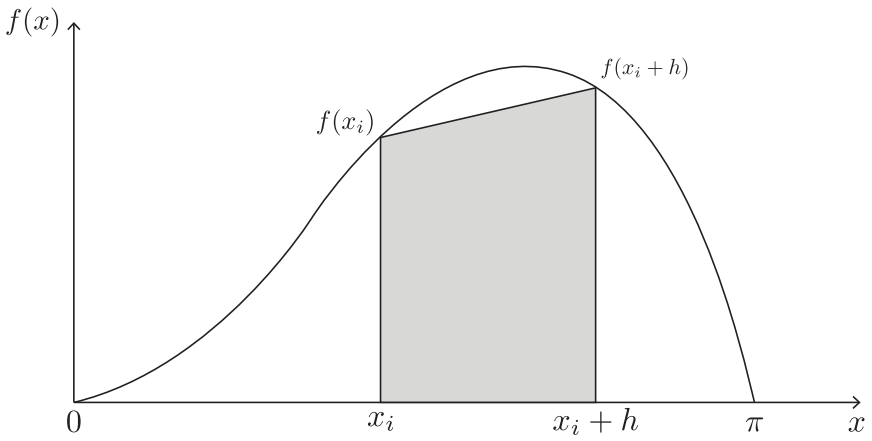


Figure 1.1: Trapezoidal rule for integral (1.1).

n	I_n	$\epsilon = \lvert I_n - \tilde{I}\rvert$
10	3.11	2.6×10^{-2}
100	3.1413	2.6×10^{-4}
1000	3.141590	2.6×10^{-6}
10000	3.14159262	2.6×10^{-8}

We want *efficient* methods, where the *error* $\epsilon \to 0$ rapidly as $n \to \infty$. It is often the case that evaluation of $f(x)$ is expensive, and then using $n = 10000$ might not be practical. In the above as n is increased by a factor of 10, h is reduced by a factor of 10, but the error ϵ is reduced by $10^2 = 100$. Because of the exponent 2 the method is said to be 2nd-order accurate.

In the above integral, an analytic solution was possible. Now what about:

$$\tilde{I} = \int_0^\pi \sqrt{1 + \cos^2 x}\, \mathrm{d}x?$$

With conventional analysis there exists no closed-form solution. With numerical analysis the procedure is exactly the same as before! In engineering practice integrands are substantially more complicated than this, and may have no closed-form expression themselves.

1.2 Computer Representation of Numbers

A fundamental question is how to represent infinite fields such as the integers $\mathbb{Z}$ and real numbers $\mathbb{R}$, with a finite and small number of bits. Consider that even individual numbers such as π can require an infinite decimal representation. How we describe real numbers in

the computer will have consequences for the accuracy of the numerical methods we develop in this course.

1.2.1 Integers

To represent integers we use a binary number system. Assume we have N bits

$$b = (b_0, b_1, \cdots, b_{N-1})$$

taking values 0 or 1. A given b represents the natural number

$$z = \sum_{i=0}^{N-1} b_i \cdot 2^i.$$

E.g. to represent the number 19:

i	6	5	4	3	2	1	0	
2^i	$2^6 = 64$	$2^5 = 32$	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$	
b_i	0	0	1	0	0	1	1	
z	$0 \times 64 +$	$0 \times 32 +$	$1 \times 16 +$	$0 \times 8 +$	$0 \times 4 +$	$1 \times 2 +$	1×1	$= 19$

Note that $z \geq 0$ and $z \leq \sum_{i=0}^{N-1} 2^i = 2^N - 1$, and we can represent every integer in that range with a choice of b. For 32-bit computers (CPU register size), $z < 2^{32} = 4294967296$. Not very large - more people in the world than this. For signed integers, 1 bit is used for the sign, $z < 2^{31} = 2147483648$. For 64-bit computers $z < 2^{64} \approx 1.8 \times 10^{19}$ - much better. Possible errors with integers:

- **Overflow** - trying to represent a number larger than z_{max}. E.g. typically for unsigned 32-bit integers $(2^{32} - 1) + 1 \to 0$, for signed 32-bit integers $(2^{31} - 1) + 1 \to -2^{31}$, depending on the exact system.

1.2.2 Real numbers - Fixed-point arithmetic

To represent real numbers, can use *fixed-point arithmetic*. One integer is assigned to each real number with a fixed interval h:

$$r = h \cdot \sum_{i=0}^{N-1} b_i \cdot 2^i.$$

E.g. with 32-bits, 1 bit to represent the sign, and a interval h of 1×10^{-4}, we can represent numbers between $\pm 2^{31} \cdot 1 \times 10^{-4} \approx \pm 200000$ with a resolution of 0.0001. This range and accuracy is obviously very limited. It is used primarily on embedded systems, for e.g. video/audio decoding where accuracy is not critical. Possible errors:

- **Overflow** - as for integers.

- **Accumulation of rounding error** - e.g. in the above system $\sum_{i=1}^{10} 0.00011$ gives 0.0010, rather than the exact 0.0011.

A real-life example of the last error is failures in the Patriot missile system. A 24-bit fixed-point number contained the current time in seconds, which was incremented every $\frac{1}{10}$th of a second. Key point - $\frac{1}{10} = 0.0001100110011...$ has an non-terminating expansion in binary, which was truncated after the 24th bit. So each increment we make an error of 9.5×10^{-8}s. After 100 hours cumulative error is $100 \times 60 \times 60 \times 10 \times 9.5 \times 10^{-8}\,\mathrm{s} = 0.34\,\mathrm{s}$ - in which time a target missile travels ≈ 0.5km. Quick fix: reboot every few days.

1.2.3 Real numbers - Floating-point arithmetic

Floating-point representations are the modern default for real-numbers. A real number x is written as a combination of a *mantissa* s and *exponent* e, given a fixed *base* b:

$$x = s \times b^e.$$

In particular:

- b - base, usually 2 or 10, fixed for system.
- s - significand (or mantissa), $1 \leq s < b$, with n-digits - a fixed-point number. E.g. in base 10 with 5-digits $1.0000 \leq s < 9.9999$. A
- e - exponent, an integer $e_{\min} \leq e \leq e_{\max}$.

For example, 5-digit mantissa, $b = 10$, $-8 \leq e \leq 8$. Then $10 \cdot \pi = 3.1416 \times 10^1$. The system as described does not contain zero, this is added explicitly. Negative numbers are defined using a sign bit, as for integers. The resulting sampling of the real-number line is shown in Figure 1.2.

Possible errors:

- **Overflow** - trying to represent a number larger/smaller than $\pm s_{\max} \times b^{e_{\max}}$. For example $9.9999 \times 10^8 + 0.0001 = \mathrm{inf}$. Special value inf.
- **Underflow** - trying to represent a number closer to zero than $1 \times b^{e_{\min}}$. For example $\left(1. \times 10^{-8}\right)^2 = 0$.
- **Undefined operation** - such as divide-by-zero, sqrt of -1. Special value, not-an-number nan.
- **Rounding error** - $1 + 1 \times 10^{-5} = 1$ in above system. We define the *machine epsilon*, $\epsilon_{\text{machine}}$, the smallest number when added to 1, gives a number distinct from 1. Also $\sqrt{1 + 1 \times 10^{-4}} = 1$. In the above system $\epsilon_{\text{machine}} = 0.0001$.

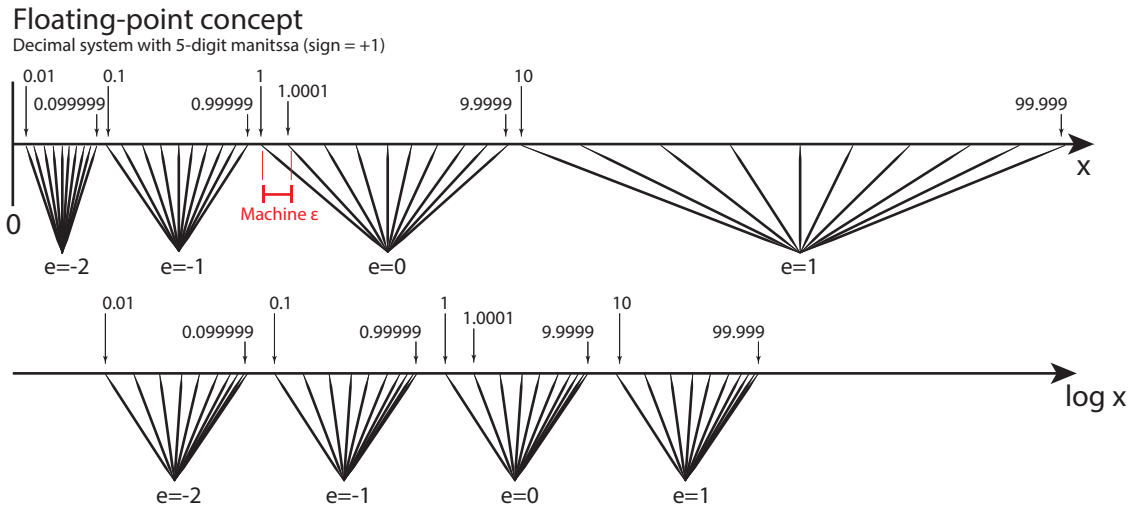


Figure 1.2: Graphical representation of a floating-point number system in base 10 (decimal), with a 5-digit mantissa $1.0000 \leq s \leq 9.9999$.

IEEE754 is a technical standard for floating-point arithmetic, defining not only the representation (see Figure 1.3), but also rounding behaviour under arithmetic operations.

- 64-bits, $b = 2$, 11-bit exponent, giving $e_{\max} = 1024$, $e_{\min} = -1023$, 53-bit mantissa (including 1 sign bit). This corresponds approximately to a decimal exponent between $-$ and 308 (since $10^{308} \approx 2^{1024}$), and about 16 decimal-digits (since $\left(\frac{1}{2}\right)^{53} \approx 1.1 \times 10^{-16}$.
- Overflow at $\approx \pm 1.7 \times 10^{308}$ (atoms in universe $\approx 10^{80}$).
- Underflow at $\approx \pm 2.2 \times 10^{-308}$ (Planck scale $\approx 1.6 \times 10^{-35}$m).
- Machine epsilon $\approx 1.1 \times 10^{-16}$.
- Smallest integer not in system $9007199254740993 = 2^{53} + 1$.

Note that floating-point with base-2 still doesn't represent $\frac{1}{10}$ exactly. This can be seen in e.g. Matlab by evaluating $0.3 - 0.1 - 0.1 - 0.1$. The answer under IEEE754 is something like $-2.7755575615628914 \times 10^{-17}$.

1.3 Taylor Series Review

The main idea of this course to approximate complex functions by a sequence of polynomials. Once we have a polynomial representation, differentiation, integration, interpolation etc. can

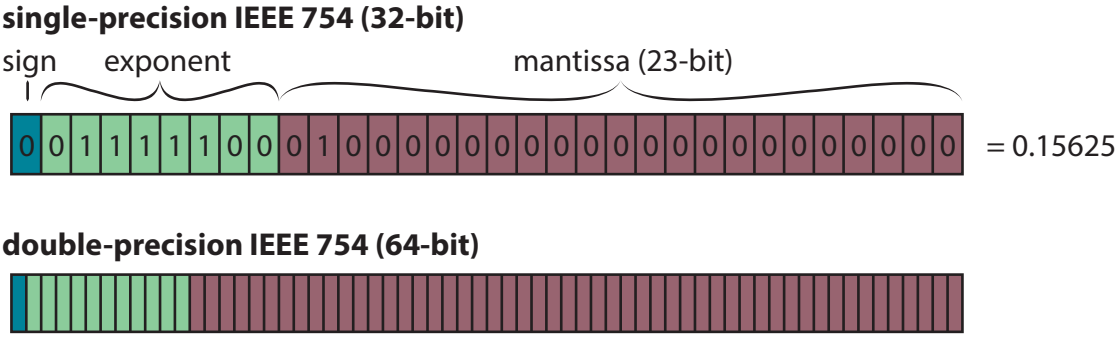


Figure 1.3: Bit layout in the IEEE754 standard.

be performed easily and exactly. So to find e.g.

$$\int_0^1 f(x)\,\mathrm{d}x,$$

we first approximate $f(x)$ by a polynomial $p(x)$, and then posit that the integral of $p(x)$ will be an approximation of the integral of $f(x)$:

$$f(x) \approx p(x),\ \forall x \in [0,1] \ \Rightarrow \int_0^1 f(x)\,\mathrm{d}x \approx \int_0^1 p(x)\,\mathrm{d}x.$$

One simple polynomial approximation that comes from basic calculus is Taylor's theorem:

Theorem 1.1 (Taylor's theorem with Lagrange remainder)
Let $f(x)$ be a $f : \mathbb{R} \to \mathbb{R}$ which is $N+1$-times continuously differentiable on the interval $[x_0, x]$ (the first $N+1$ derivatives exist and are continuous). Then the Taylor expansion of $f(x)$ about x_0 is

$$f(x) = \sum_{n=0}^{N} \frac{f^{(n)}(x_0)}{n!}(x-x_0)^n + \mathcal{O}(x-x_0)^{N+1}.$$

For small $(x - x_0)$ we expect the last term, the truncation error to be small, and therefore the sum to be a good approximation to $f(x)$. Note that the sum contains only powers of x - it is therefore a polynomial in x.

Furthermore we can write the truncation error in a specific form: there exists a $\xi \in [x_0, x]$ such that

$$f(x) = \sum_{n=0}^{N} \frac{f^{(n)}(x_0)}{n!}(x-x_0)^n + \frac{f^{(N+1)}(\xi)}{(N+1)!}(x-x_0)^{N+1}. \tag{1.2}$$

This is the Lagrange form of the remainder, and a generalization of the Mean-Value Theorem. It is important as it gives us an estimate for the error in the expansion, though in practice we never know ξ.

A convenient rewriting of (1.2) is obtained by defining the *step-size*

$$h = x - x_0,$$

given which

$$f(x_0 + h) = \sum_{n=0}^{N} \frac{f^{(n)}(x_0)}{n!} h^n + \frac{f^{(N+1)}(\xi)}{(N+1)!} h^{N+1}. \tag{1.3}$$

and the series is a good approximation for small h. The Taylor series including terms up to and including h^N is called an Nth-order Taylor expansion, or alternatively an $(N+1)th$-term Taylor expansion.

Example 1.2
Expand $f(x) = \cos(x)$ about $x_0 = 0$ in an 4th-order Taylor expansion, plus remainder.

$$\begin{aligned}
\cos(0) &= 1 \\
\cos'(0) &= -\sin(0) = 0 \\
\cos''(0) &= -\cos(0) = -1 \\
\cos'''(0) &= \sin(0) = 0 \\
\cos^{(4)}(0) &= \cos(0) = 1 \\
\cos^{(5)}(\xi) &= -\sin(\xi)
\end{aligned}$$

Substituting into (1.2) *gives*

$$\cos(x) = \left[1 - \frac{x^2}{2!} + \frac{x^4}{4!}\right] - \sin(\xi)\frac{x^5}{5!}$$

for some $\xi \in [0, x]$.

Example 1.3
Consider the case of expanding a polynomial $f(x) = ax^4 + bx^3 + cx^2 + dx + e$ as a Taylor series about 0:

$$\begin{aligned}
f(x) &= ax^4 + bx^3 + cx^2 + dx + e \\
f'(x) &= 4ax^3 + 3bx^2 + 2!cx + d \\
f''(x) &= 4 \cdot 3ax^2 + 3!bx + 2!c \\
f'''(x) &= 4!ax + 3!b \\
f''''(x) &= 4!a
\end{aligned}$$

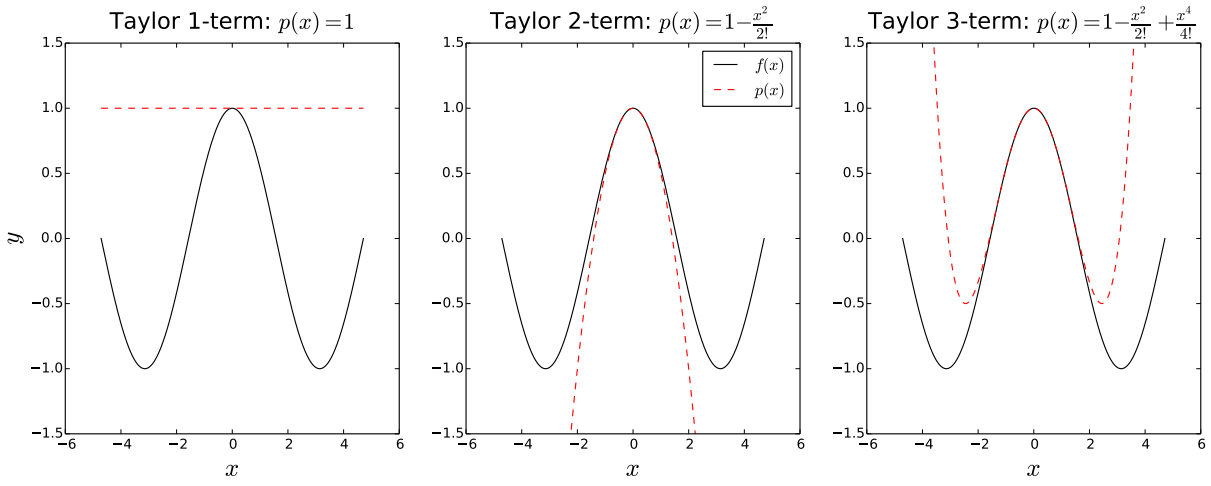


Figure 1.4: Successive approximations to $\cos(x)$ at 0 with a Taylor expansion.

then

$$\begin{aligned} f(0) &= 0!e \\ f'(0) &= 1!d \\ f''(0) &= 2!c \\ f'''(0) &= 3!b \\ f''''(0) &= 4!a \end{aligned}$$

and rearranging each term shows $a = \frac{f''''(0)}{4!}$ *etc. So an* N*-term Taylor series reproduces polynomials exactly, given sufficient terms. If you forget Taylor's expansion, you can derive it like this. Also for a general function* $f(x)$*, the Taylor series selects the polynomial with the same derivatives as* $f(x)$ *at* x_0*.*

We never know ξ. The important consequence of this remainder is that the rate at which the error goes to zero as $h \to 0$ is known: $\epsilon \propto h^{N+1}$. Using big-$\mathcal{O}$ notation $\epsilon = \mathcal{O}(h^{N+1})$. We do not know the exact error (if we did we would know the exact value of the function at x), but we know how *quickly* it gets smaller, and that we can make it as small as we like by reducing h.

Exercise 1.4
Is the condition on differentiability of $f(x)$ *important? Is the theorem still true if* $f(x)$ *is not continuously differentiable? Consider the case* $f(x) = |x|$ *expanded about* $x = 1$*.*

1.3.1 Truncation error versus Rounding error

Consider approximating $f'(x)$ at $x_0 = 0$. Start with the 1st-order Taylor expansion:

$$f(h) = f(0) + f'(0)h + \mathcal{O}(h^2),$$

and rearrange for $f'(0)$:

$$f'(0) = \frac{f(h) - f(0)}{h} + \mathcal{O}(h).$$

This is a simple way of finding an approximation to the derivative - this is the *forward difference scheme*. The truncation error of $f'(0)$ is $\mathcal{O}(h)$ - so it makes sense to choose h as small as possible to get the best approximation, e.g. $h = 1 \times 10^{-15}$, a number which can be represented by floating-point arithmetic. However we run into rounding error.

For example consider $f(x) = e^x$, so that $f'(0) = 1$. For $h = 1 \times 10^{-15}$ in floating-point arithmetic:

$$f'(0) \approx \frac{e^h - e^0}{h} = \frac{1.0000000000000011 - 1.}{1 \times 10^{-15}} = 1.1102230246251565.$$

The error is more than 10%! Now let $h = \sqrt{\epsilon_{\text{machine}}} \approx 1 \times 10^{-8}$, then

$$f'(0) \approx \frac{1.0000000099999999 - 1.}{1 \times 10^{-8}} = 0.99999999392252903.$$

Chapter 2

Iterative Solution of Non-linear Equations

One of the fundamental problems of mathematics is the solution of an equation in a single unknown

$$f(x) = 0.$$

Only for limited special cases can we solve the equation explicitly. For instance, if $f(x)$ is a simple algebraic polynomial of degree n, i.e.

$$f(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + \cdots a_n x^n,$$

then we can write down the solution of $f(x) = 0$ if the polynomial is of degree one or two. It is also possible to obtain the zeros of a cubic polynomial but the formula is already very complicated and hardly used in practice. For degree 5 and higher it can be shown that no such formula exists. Then, we have to resort to numerical approximation techniques. For functions $f(x)$ which are not polynomial in nature, numerical techniques are virtually the only approach to find the solution of the equation $f(x) = 0$. For instance, some elementary mathematical functions such as the square root and the reciprocal are evaluated on any computer or calculator by an equation-solving approach: $\sqrt{a}$ is computed by solving the equation $x^2 - a = 0$ and $\frac{1}{a}$ is computed by solving $\frac{1}{x} - a = 0$.

The algorithms we are going to study are all *iterative* in nature. That means we start our solution process with a guess at the solution being sought and then refine this guess following specific rules. In that way, a sequence of estimates of the solution is generated $x_0, x_1, \cdots x_N$ which should converge to the true solution $\tilde{x}$, meaning that

$$\lim_{N \to \infty} x_N = \tilde{x},$$

where $f(\tilde{x}) = 0$. However this is guaranteed only under certain conditions, depending on the algorithm.

Therefore simply describing the algorithm, and thereby the sequence, is not sufficient. We wish to know in advance:

1. Under what conditions the algorithm converges.
2. A bound on the error of the estimate x_N.
3. How rapidly the algorithm converges (the rate at which the error in x_N decreases).

2.1 Recursive Bisection

Let us first illustrate a very simple method to find a zero of the continuous function $f(x)$. We start with an interval $[a, b]$ in which we know a root exists. Then we half the interval, and choose the half which contains the root. We repeat the procedure on the new interval.

Algorithm 2.1 (Recursive Bisection)
Assume an interval $[a, b]$, and a continuous function $f(x)$, such that $f(a)$ and $f(b)$ have opposite sign. Then pseudocode for $N_{\max}$ iterations of recursive bisection is:

$a_0 \leftarrow a$; $b_0 \leftarrow b$
for $i = [0 : N_{\max}]$ ***do***
  $c_i \leftarrow \frac{1}{2}(a_i + b_i)$
  if $f(a_i) \cdot f(c_i) < 0$ ***then***
    $a_{i+1} \leftarrow a_i$; $b_{i+1} \leftarrow c_i$
  else
    $a_{i+1} \leftarrow c_i$; $b_{i+1} \leftarrow b_i$
  end if
end for
return c_i

Example 2.2
We apply this method to the polynomial

$$f(x) := x^3 - x - 1 = 0, \tag{2.1}$$

starting from the interval $[1, 2]$. We first note that $f(1) = -1$ and $f(2) = 5$, and since $f(x)$ is continuous on $[1, 2]$, it must vanish somewhere in the interval $[1, 2]$ by the intermediate value theorem *for continuous functions. Now, we take the midpoint of the interval $[1, 2]$ as the initial guess of the zero, i.e. $x_0 = 1.5$. The error in that guess $\epsilon_0 := |x_0 - \tilde{x}|$ is at most half of the length of the interval, i.e. ≤ 0.5. Since $f(1.5) = 0.875 > 0$ the zero must lye in the smaller interval $[1, 1.5]$. Again, we take the midpoint of that interval as the next guess of the solution, i.e. $x_1 = 1.25$, and the error of that guess is ≤ 0.25. We obtain $f(1.25) = -0.296 < 0$, thus a yet smaller interval where the solution must lie is $[1.25, 1.5]$, and the next guess of the solution is $x_2 = 1.375$; it has an error ≤ 0.125. See Figure 2.1.*

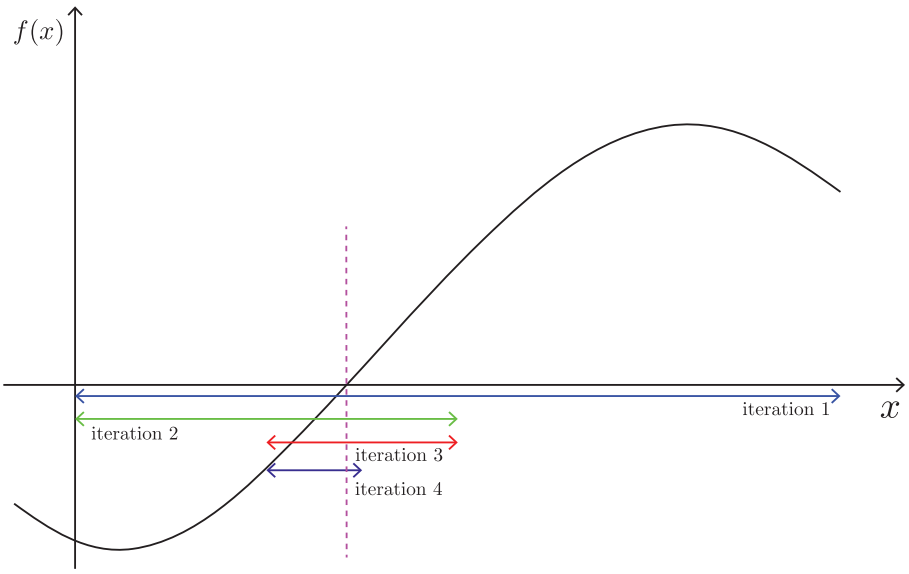


Figure 2.1: Progress of the Recursive Bisection Method.

It is easy to prove that for any continuous function $f(\cdot)$ and initial end points satisfying $f(x_{\text{left}}) \cdot f(x_{\text{right}}) < 0$ the sequence of midpoints generated by the bisection method converges to the solution of $f(x) = 0$. Obviously, the midpoint of the corresponding interval differs from the solution by at most half the length of the interval. This gives a simple expression for the upper-bound on the error.

Let the error in the root after the ith iteration be

$$\epsilon_i := |x_i - \tilde{x}|.$$

On the first iteration we know that $\epsilon_0 \leq (b-a)/2$, and for every subsequent iteration the interval size halves, so

$$\epsilon_N \leq E_N = \frac{b-a}{2^{N+1}},$$

where E_N is the upper-bound on the error in x_N. Note that

$$E_{N+1} = \frac{1}{2} E_N,$$

so the error at each iteration is reduced by a constant factor of 0.5. This is an example of a *linear* rate of convergence. The name "linear" originates from the convergence curve when plotted on an iteration-log error graph, see Figure 2.2.

We are interested in the rate of convergence because of the common case where $f(x)$ is extremely expensive to evaluate. In N iterations of recursive bisection $f(\cdot)$ must be evaluated $N + 2$ times.

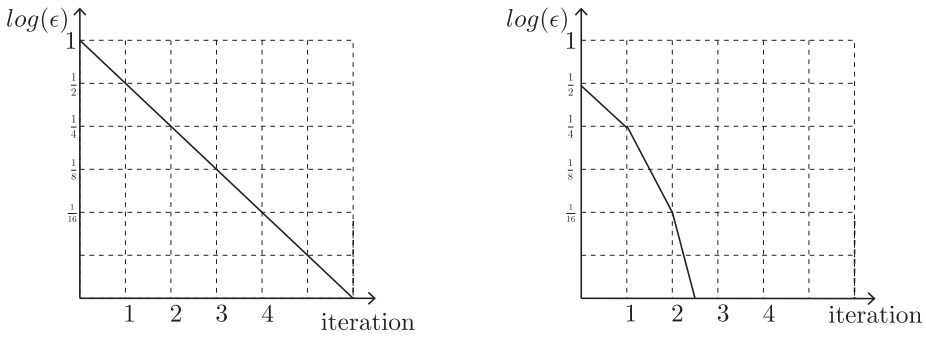


Figure 2.2: Convergence of recursive bisection (linear – left) and Newton (quadratic – right).

Recursive bisection is an excellent method: guaranteed to converge, and possessing a strong upper-bound on the error. The major limitation is that it applies in 1d only. It can not be generalized to the case where $f(\cdot)$ is a vector function of a vector variable:

$$f : \mathbb{R}^n \to \mathbb{R}^n.$$

The *fixed-point iteration* and *Newton* methods in the following can be generalized.

Exercise 2.3
Consider applying the recursive bisection method to a continuous function $f(x)$ with multiple roots in the initial interval.

1. *What restrictions exist on the number of roots in $[a, b]$ given that f is continuous and $f(a) < 0$, $f(b) > 0$?*
2. *If a continuous curve has 3 roots in the interval $[a, b]$, can recursive bisection converge to the middle root?*

Now consider the case that the function is not continuous - consider for example $f(x) = 1/x$ on the initial interval $[-1, 2]$. Does recursive bisection converge? Does it converge to a root?

2.2 Fixed-point iteration

The *fixed-point iteration* requires only that the original equation:

$$f(x) = 0 \tag{2.2}$$

be rewritten in an equivalent form

$$x = \varphi(x). \tag{2.3}$$

Provided that the two equations (2.2), (2.3) are equivalent, i.e. $x = \varphi(x) \Leftrightarrow f(x) = 0$, it follows that any solution of the original equation (2.2) is a solution of the second equation. Then, an initial guess x_0 generates a sequence of estimates of the fixed point by setting

$$x_n = \varphi(x_{n-1}), \quad n = 1, 2, \ldots. \tag{2.4}$$

Clearly if the exact solution $\tilde{x}$ is achieved on iteration n, $x_n = \tilde{x}$, then $x_{n+1} = \tilde{x}$. The exact solution is preserved, it is a *fixed point* of the iteration. See Figure 2.3. The algorithm can be summarized as follows:

Algorithm 2.4 (Fixed-Point Iteration)
Let the initial guess be x_0. Then the following performs a fixed-point iteration:

for $i = [0 : N_{\max}]$ ***do***
  $x_{i+1} \leftarrow \varphi(x_i)$
end for
return x_{i+1}

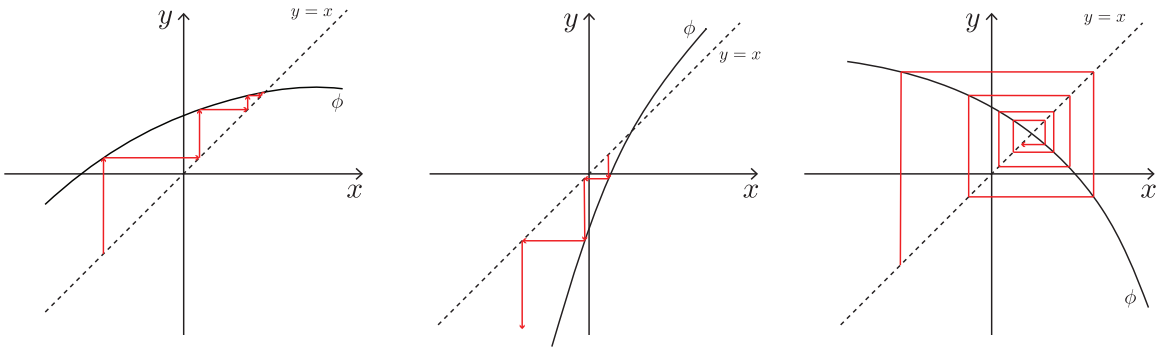


Figure 2.3: Progress of a fixed-point iteration for 3 different φ.

Note that the choice of $\varphi(\cdot)$ is not unique. As before, the questions we have to answer are (a) whether the iteration scheme converges, and if so (b) how the error behaves. Both these depend on the choice of $\varphi(\cdot)$.

Example 2.5 (Continuation of example 2.2)
Before answering these questions let us first illustrate the method using the example 2.2 given before. Usually, the iteration function φ is obtained by solving (2.2) *for x in some way. Equation* (2.1) *may be written as*

- $x = x^3 - 1 =: \varphi(x)$
- $x = (x + 1)^{1/3} =: \varphi(x)$

- $x = \frac{1}{x} + \frac{1}{x^2} =: \varphi(x), \quad x \neq 0$
- $x = \frac{1}{x^2-1} =: \varphi(x), \quad x \neq \{1, -1\}$

Let us take $\varphi(x) = (x+1)^{1/3}$. *As starting value we use* $x_0 = 1.5$. *Then, following 2.4 we obtain the scheme*

$$x_n = (x_{n-1} + 1)^{1/3}, \quad n = 1, 2, \ldots, \tag{2.5}$$

which generates the following values for x_1, x_2, x_3, *and* x_4:

$$x_1 = 1.3572, \quad x_2 = 1.3309, \quad x_3 = 1.3259, \quad x_4 = 1.3249.$$

When using $\varphi(x) = x^3 - 1$, *we obtain the scheme*

$$x_n = x_{n-1}^3 - 1, \quad n = 1, 2, \ldots, \tag{2.6}$$

which generates the values

$$x_1 = 2.375, \quad x_2 = 12.396, \quad x_3 = 1904.003, \quad x_4 = 6.902 \cdot 10^9.$$

Obviously, the latter sequence does not converge, since the solution of the given equation is $1.3247179\ldots$. *However, the first iteration scheme seems to converge, because* $x_4 = 1.3249$ *is a much better approximation than the starting value* $x_0 = 1.5$.

The example shows that it is necessary to investigate under what conditions the iteration scheme will converge and how fast the convergence will be. First we need some calculus:

Theorem 2.6 (Mean-Value Theorem)
If $f(x)$ *and* $f'(x)$ *are continuous, there exists a* $\xi \in [a, b]$ *such that*

$$f'(\xi) = \frac{f(b) - f(a)}{b - a}.$$

Note that this is a special case of Taylors theorem with Lagrange remainder. Visually the mean-value theorem is easy to verify, see Figure 2.4.

We now investigate the convergence of a fixed-point iteration. Let $\tilde{x}$ be the exact root of $f(x)$, and x_i the current approximation. Now we can derive a relationship between the error in x_i and x_{i+1} as follows:

$$\begin{aligned}
x_{i+1} &= \varphi(x_i) \\
x_{i+1} - \tilde{x} &= \varphi(x_i) - \tilde{x} \\
x_{i+1} - \tilde{x} &= \varphi(x_i) - \varphi(\tilde{x}) \\
x_{i+1} - \tilde{x} &= \frac{\varphi(x_i) - \varphi(\tilde{x})}{x_i - \tilde{x}}(x_i - \tilde{x}) \\
x_{i+1} - \tilde{x} &= \varphi'(\xi_i)(x_i - \tilde{x}),
\end{aligned}$$

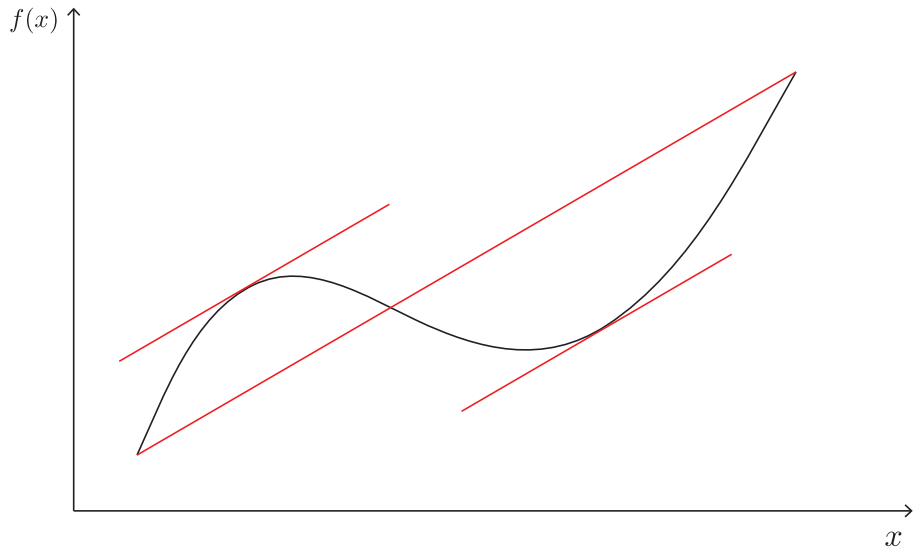


Figure 2.4: Graphical argument for the Mean-Value Theorem

where the mean-value theorem has been applied at the last stage. Note that the ξ in the MVT will be different at each iteration, hence ξ_i. By defining the error $\epsilon_i := |x_i - \tilde{x}|$, we have

$$\epsilon_{i+1} = \varphi'(\xi_i)\epsilon_i.$$

If we want the error to strictly drop at iteration i, then we require $|\varphi'(\xi_i)| < 1$. Otherwise error grows (divergence). If $-1 < \varphi'(\xi) < 0$ the error oscillates around the root, as we observed in Figure 2.3.

Assume that $|\varphi'(\xi_i)| < K < 1$, $\forall i$. Then we have an *error bound* after n iterations:

$$\epsilon_i < K\epsilon_{i-1} < K^2\epsilon_{i-2} < \cdots < K^i\epsilon_0.$$

Again we have linear convergence - error is reduced by a constant factor at each iteration.

Example 2.7 (Continuation of example 2.2)
We consider again the iteration function $\varphi(x) = x^3 - 1$ and investigate φ'. From $\varphi(x) = x^3 - 1$ it follows $\varphi'(x) = 3x^2$. For the initial guess $x = 1.5$, we obtain $\varphi'(1.5) = 6.75 > 1$, i.e. the condition $|\varphi'| < 1$ is already not fulfilled for the initial guess. Since the condition $|\varphi'| < 1$ is only fulfilled for $|x| < \sqrt{2/3}$, the iteration function $\varphi(x) = x^3 - 1$ is not suited for the iteration scheme.

Using the iteration function $\varphi(x) = (x+1)^{1/3}$, we obtain $\varphi'(x) = \frac{1}{3}(x+1)^{-2/3}$. Thus, $|\varphi'(x)| < 0.21$ for all $x \in [1, 2]$. Therefore, this iteration function leads to a sequence x_n of approximations which does converge to the solution ξ.

Because of the flexibility in choice of φ, fixed-point iterations encompass an enormous variety of iterative techniques. FPIs of some variety are used almost universally in physics simulation codes, including CFD. They can be very efficient for vector-valued $f(\cdot)$ and x, but convergence is rarely guaranteed, and often slow (think $K = 0.99$). In some cases we can improve on FPI with Newton's method.

2.3 Newton's method

The principle of Newton's method is to construct a *linear approximation* to $f(\cdot)$ at x_i. The next approximation of the root is where this linear approximation crosses the axis. Therefore in the trivial case that $f(x)$ is a straight-line the method will converge in 1 iteration (compare with an FPI in the same case).

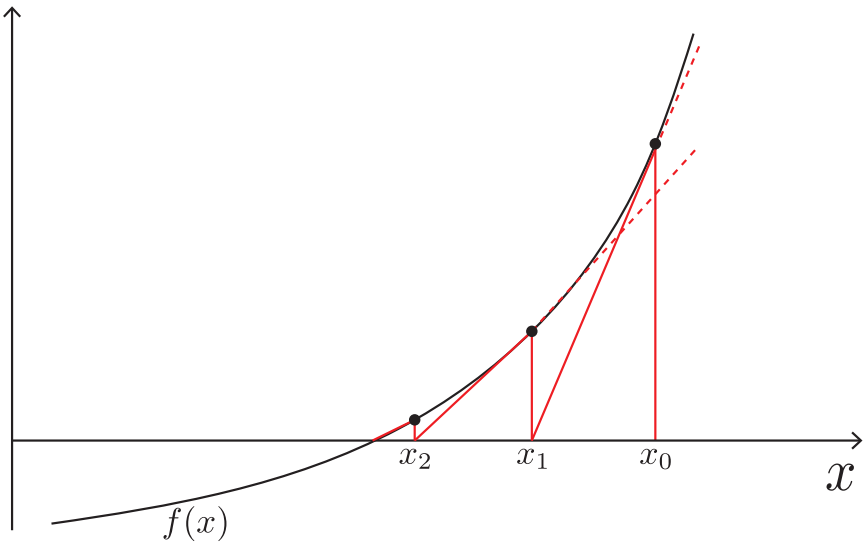


Figure 2.5: Progression of Newton's method

Assume that f is two times continuously differentiable in $[a, b]$ and that it has a *simple* zero $\tilde{x} \in [a, b]$, i.e. $f(\tilde{x}) = 0$ and $f'(\tilde{x}) \neq 0$. The Taylor series expansion of f at x_0 is

$$f(\tilde{x}) = f(x_0) + f'(x_0)(\tilde{x} - x_0) + f''(\xi)\frac{(\tilde{x} - x_0)^2}{2} = 0, \quad \xi \in [x_0, \tilde{x}].$$

If we are close to the solution, then $|\tilde{x} - x_0|$ is small, and we can neglect the remainder term. By neglecting the remainder we are invoking the linear approximation. We obtain

$$0 \approx f(x_0) + f'(x_0)(\tilde{x} - x_0),$$

rearranging for $\tilde{x}$ gives,

$$\tilde{x} \approx x_0 - \frac{f(x_0)}{f'(x_0)} =: x_1.$$

Thus we can establish the following iteration scheme:

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}, \quad n = 1, 2, \ldots$$

The method is called *Newton's method*. It is a special case of a FPI with iteration function

$$\varphi(x) = x - \frac{f(x)}{f'(x)}.$$

Stability and Convergence Again we ask the questions: (a) does the iteration converge?, and (b) how rapidly? To answer (a) we observe that

$$\varphi'(x) = \frac{f(x)f''(x)}{(f'(x))^2}.$$

So, $\varphi'(\tilde{x}) = 0$. Therefore, it follows that (provided φ' is continuous, which it is if f is twice continuously differentiable) $|\varphi'(x)| < 1$ for all x in some neighbourhood of the solution $\tilde{x}$. Therefore, Newton's method will converge provided that the initial guess x_0 is "close enough" to $\tilde{x}$. The convergence result demands a starting point x_0 which may need to be very close to the solution we look for. This is called *local convergence*. Such a starting point can often be found by first using several iterations of a FPI in the hope that a suitably small interval is obtained.

Something special happens in the convergence rate of Newton – which makes it unique. As before, let $\tilde{x}$ be a root of $f(x)$, and x_i the current approximation. By defining the current error:

$$e_i := x_i - \tilde{x}$$

which we expect to be *small* (at least close to the root), the iteration can be approximated with Taylor series about the exact solution:

$$\begin{aligned} x_{i+1} = \varphi(x_i) &= \varphi(\tilde{x} + e_i) && (2.7)\\ &= \varphi(\tilde{x}) + \varphi'(\tilde{x})e_i + \frac{\varphi''(\tilde{x})}{2}e_i^2 + \mathcal{O}(e_i^3) && (2.8)\\ &\approx \tilde{x} + 0 + \frac{\varphi''(\tilde{x})}{2}e_i^2, && (2.9) \end{aligned}$$

where the final approximation neglects only e_i^3 terms. Rearranging

$$e_{i+1} = \frac{\varphi''(\tilde{x})}{2}e_i^2.$$

The error is reduced by its square, each iteration. This is called *quadratic convergence*, see Figure 2.2.

Example 2.8 (Continuation of example 2.2)
We apply Newton's method to example 2.2 and obtain for the initial guess $x_0 = 1.5$ *the series*

n	x_n
0	1.5
1	1.348
2	1.3252
3	1.3247182
4	1.324717957

Exercise 2.9
Use Newton's method to compute $\sqrt{2}$. *Use the function* $f(x) = x^2 - 2$ *and the initial guess* $x_0 = 0.2$. *How many iterations are needed to get six decimal places? Perform the iterations.*

Chapter 3

Polynomial Interpolation in 1d

The process of constructing a smooth function which passes exactly though specified data points is called *interpolation*. Introducing some notation, the data points we denote

$$(x_i, f_i), \text{ for } i = 0, 1, \ldots, n,$$

which we imagine to come from some exact function $f(x)$ which is unknown, and which we wish to reconstruct on the interval $[x_0, x_n]$. We usually expect that $x_0 < x_1 < \cdots < x_n$, in particular that no two x_is are equal.

An interpolating function is called an *interpolant* and is a linear combination of prescribed *basis* functions. If the basis functions are:

$$\{\varphi_i(x) : i = 0, \ldots, n\}$$

then the interpolant will have the form

$$\phi(x) := a_0\varphi_0 + a_1\varphi_1 + \cdots + a_n\varphi_n = \sum_{i=0}^{n} a_i\varphi_i(x)$$

where a_i are the *interpolation coefficients*, and are constant (not a function of x), and must be choosen to force $\phi(x)$ to match the data (x_i, f_i).

Definition 3.1 (Interpolation of data)
Given $n+1$ pairs of numbers $(x_i, f_i), 0 \le i \le n$, with $x_i \neq x_j$ for all $i \neq j$. We are looking for a function

$$\phi(x) := \sum_{i=0}^{n} a_i\varphi_i,$$

satisfying the interpolation conditions

$$\phi(x_i) = f_i, \text{ for } i = 0, \ldots, n.$$

In the above definition note that we have $n+1$ interpolation conditions to satisfy, and also $n+1$ *degrees of freedom (DoFs)* the a_i that we can change to satsify the conditions. This suggests that the problem can be solved.

The functions $\varphi_i(x)$ might be polynomials (leading to *polynomial interpolation*), trigonometric functions (*Fourier interpolation*), rational functions (*rational interpolation*), or anything else. Then, the problem of linear interpolation can be formulated as follows:

If the element $\phi = \sum_{i=0}^{n} a_i \varphi_i$ is to satisfy the interpolation conditions $\phi(x_i) = f_i$ for $i = 0, \ldots, n$, then the coefficients must satisfy the linear system of equations:

$$\begin{aligned} a_0\varphi_0(x_0) + \cdots + a_n\varphi_n(x_0) &= f_0, \\ a_0\varphi_0(x_1) + \cdots + a_n\varphi_n(x_1) &= f_1, \\ \vdots \qquad\qquad\quad \vdots \qquad\quad & \vdots \\ a_0\varphi_0(x_n) + \cdots + a_n\varphi_n(x_n) &= f_n, \end{aligned}$$

or more concisely:

$$\sum_{i=0}^{n} a_i \varphi_i(x_j) = f_j, \text{ for } j = 0, \ldots, n. \tag{3.1}$$

This is a matrix equation. If $\mathbf{A}$ is the matrix with elements $a_{ij} = \phi_j(x_i)$, $\mathbf{a} = (a_0, \ldots, a_n)$, $\mathbf{f} = (f_0, \ldots, f_n)$, we write (3.1) as

$$\mathbf{A}\mathbf{a} = \mathbf{f}. \tag{3.2}$$

Equation (3.2) is a linear system of dimension $(n+1) \times (n+1)$. There exists a unique solution

$$\mathbf{a} = \mathbf{A}^{-1}\,\mathbf{f},$$

if

$$\det \mathbf{A} \neq 0.$$

The value of $\det \mathbf{A}$ depends on the chosen basis functions $\{\phi_j\}$ and on the data locations $\{x_i\}$, but not on the data values $\{f_i\}$. If $\det \mathbf{A} \neq 0$ for *every* selection of $n+1$ distinct data points, then the system of basis functions $\{\phi_j(x)\}$ is called *unisolvent* - a highly desirable property. Note that if $x_i = x_j$ for any $i \neq j$ then two rows of $\mathbf{A}$ will be identical and therefore $\det \mathbf{A} = 0$.

Initially we concentrate on the one-dimensional case (that is with only one independent variable x). This is called univariate interpolation.

3.1 The Monomial Basis

One natural basis for polynomials are the monomials:

$$\begin{aligned}\varphi_0(x) &= 1\\ \varphi_1(x) &= x\\ \varphi_2(x) &= x^2\\ \vdots &\quad \vdots\\ \varphi_n(x) &= x^n,\end{aligned}$$

denote the monomial basis

$$\mathbf{m}_n(x) := (1, x, x^2, \ldots, x^n).$$

Given which the interpolant has the form

$$\phi(x) := a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n = \sum_{i=0}^{n} a_i x^i.$$

It is clear that all polynomials of degree $\leq n$ can be written in this form. This is not the only possible basis, the Lagrange basis and the Newton basis will be discussed later.

As for all bases the coefficients $\{a_i\}$ are uniquely determined by the interpolation conditions

$$p_n(x_i) = f_i, \quad i = 0, \ldots, n,$$

these are $n+1$ conditions for $n+1$ unknowns. We write the conditions a linear system

$$\mathbf{V}\,\mathbf{a} = \mathbf{f}.$$

The particular form of the matrix that results with a monomial basis has a special name: the *Vandermonde matrix*

$$\mathbf{V} = \begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^{n-1} & x_0^n \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^{n-1} & x_n^n \end{pmatrix}. \tag{3.3}$$

The right-hand side is simply

$$\mathbf{f} = \begin{pmatrix} f_0 & f_1 & f_2 & \ldots & f_{n-1} & f_n \end{pmatrix}^T.$$

The interpolating polynomial is therefore

$$p_n(x) = \sum_{i=0}^{n} a_i \varphi_i(x) = \mathbf{a}^T\,\mathbf{m}_n(x).$$

Example 3.2 (Monomial interpolation of $\sin(x)$)
Construct a quadratic approximation of $f(x) = \sin(x)$ on the interval $[0, \pi]$. First choose the nodal locations - a quadratic approximation always requires 3 support points - we choose uniformly spaced points on the interval: $x = (0, \pi/2, \pi)$. Evaluating at these points gives $f = (0, 1, 0)$.

The Vandermonde matrix is

$$\mathbf{V} = \begin{pmatrix} 1 & x_0 & x_0^2 \\ 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \end{pmatrix} = \mathbf{V} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & \pi/2 & \pi^2/4 \\ 1 & \pi & \pi^2 \end{pmatrix}.$$

We therefore solve

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & \pi/2 & \pi^2/4 \\ 1 & \pi & \pi^2 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

for a, giving

$$a_0 = 0 \qquad a_1 = 4/\pi \qquad a_2 = -4/\pi^2.$$

The approximating function is therefore

$$p(x) = 1.27323954x - 0.40528473x^2,$$

plotted in Figure 3.1.

Example 3.3 (Polynomial reconstruction with Taylor series)
A Taylor series can be regarded as an approximation to a function at a point. In the language of polynomial interpolation, a Taylor series approximation about $x_0 = 0$ is a kind of interpolation with a monomial basis. We don't need to solve a linear system for the coefficients, they are given by Taylor's theorem:

$$\begin{aligned} a_0 &= f(x_0) \\ a_1 &= f'(x_0) \\ a_2 &= \frac{1}{2!} f''(x_0) \\ \vdots \quad &\quad \vdots \\ a_n &= \frac{1}{n!} f^{(n)}(x_0). \end{aligned}$$

Thus

$$\phi(x) = f(x_0) + f'(x_0)x + \frac{1}{2!} f''(x_0)x^2 + \cdots + \frac{1}{n!} f^{(n)}(x_0)x^n,$$

is the polynomial reconstruction — and we already know an expression for the error (the Lagrange remainder). Note that it is not an interpolation, as it does not in general pass

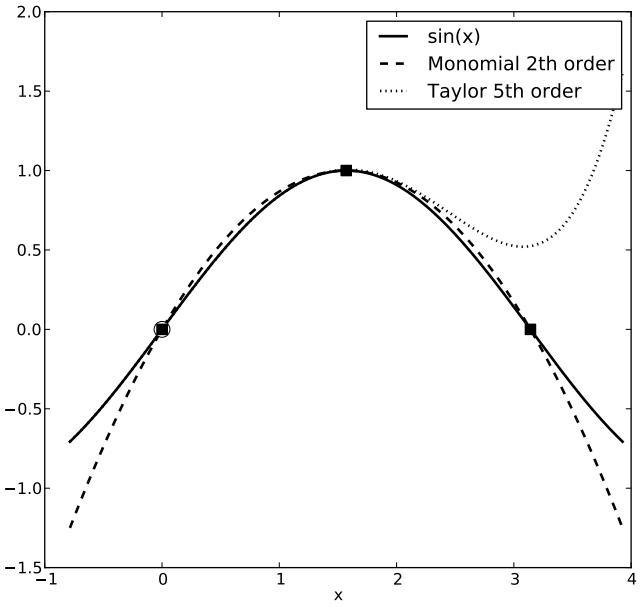


Figure 3.1: Approximating $\sin(x)$ with monomial interpolation and Taylor expansion.

through a prescribed set of points $\{(x_i, f_i)\}$. *Also the approximation is typically very good close to* x_0, *and deteriorates rapidly with distance. In interpolation we would like the error to be small on an entire interval. Finally in practice we often have a set of data points, but we very rarely have higher derivatives* $f^{(n)}(x_0)$ *of the function of interest, so we can not compute* a_i.

The Taylor expansion of $\sin(x)$ *about* $x_0 = 0$ *is plotted in Figure 3.1. The expansion is better than 2nd-order monomial interpolation close to* 0, *but the latter much better over the whole interval with a polynomial of much lower order.*

3.2 Why interpolation with polynomials?

Univariate interpolation is usually done using polynomials - this is an important case we consider in detail. The question is whether it is a good idea to approximate a smooth univariate function $f(x)$ by polynomials at all. The answer is 'yes' due to a theorem by Weierstrass (1885):

Theorem 3.4 (Weierstrass, 1885)
For any continuous function on the interval $[a, b]$ *and any* $\varepsilon > 0$, *there exists an algebraic polynomial* p *such that*[1]

$$\|f - p\|_\infty := \max |f(x) - p(x)| < \varepsilon. \tag{3.4}$$

The theorem tells us that any continuous function may be approximated as closely as we wish by a polynomial of suffciently high degree. The theorem does not tell us how to find that polynomial, and the remainder of this chapter is dedicated to that task.

Since a polynomial of degree n has $n + 1$ coefficients, there is a unique polynomial of degree $\leq n$ which agrees with a given function at $n + 1$ data points.

There are other reasons for using polynomials as interpolant:

- polynomials can be evaluated using the arithmetic operations $+, -, \times$ only (i.e. easy for a computer);
- derivatives and indefinite integrals of polynomials are easy to compute and are polynomials themselves;

[1] The ∞-norm is defined by

$$\|f\|_\infty := \max_{x \in [a,b]} |f(x)|,$$

and is one measure of the "size" of a function or distance between two functions. Another common measure is the L^2-norm (pronounced "L-two"),

$$\|f\|_2 := \sqrt{\int_a^b [f(x)]^2 \, \mathrm{d}x},$$

which is comparable to the Euclidian norm for vectors $|\mathbf{x}|_2 = \sqrt{x^2 + y^2 + z^2}$.

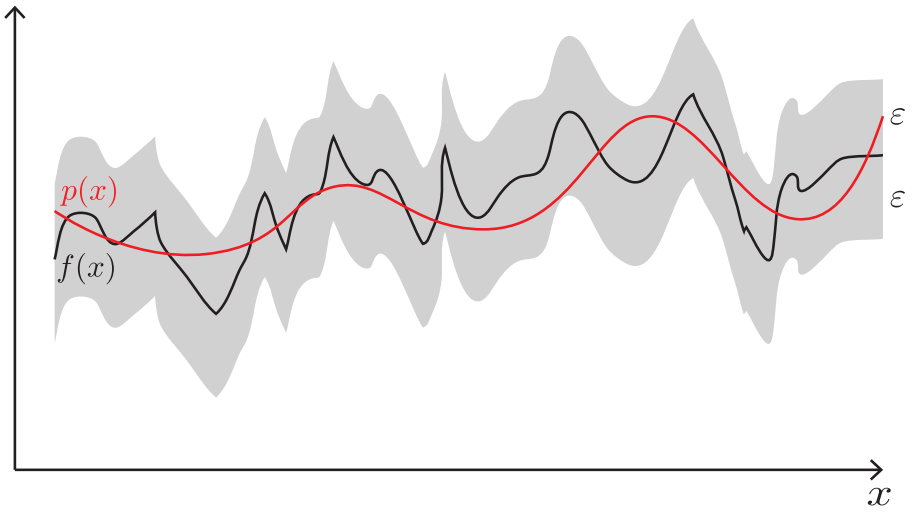


Figure 3.2: Graphical representation of the Weierstrass Approximation Theorm

- polynomials are always continuous and infinitely differentiable;
- univariate polynomial interpolation is always uniquely solvable (i.e. polynomial bases are unisolvent).

This last is very important: it means there is *exactly one* polynomial of degree $\leq n$ that passes through $n+1$ points:

Theorem 3.5 (Uniqueness of polynomial interpolation)
Given a continuous function $f(x)$ and a grid X of $n+1$ nodes $\{x_0, x_1, \ldots, x_n\}$ with $a \leq x_0 < x_1 < \ldots < x_n \leq b$. There exists a unique polynomial $p_n(x)$ of degree $\leq n$, such that

$$p_n(x_i) = f_i, \quad i = 0, 1, \ldots, n. \tag{3.5}$$

Note carefully that the polynomial p_n has degree n *or less*. Thus, if the $n+1$ data points lie on a straight line, the polynomial $\pi_n(x)$ will look like

$$a_0 + a_1 x + 0x^2 + \ldots + 0x^n.$$

3.3 Newton polynomial basis

In order to identify the interpolation coefficients using a monomial basis, the Vandermonde matrix $\mathbf{V}$ must be inverted, which could potentially be an ill-conditioned matrix. Also, every

time we add a new node to the data-set, we have to re-evaluate the values of all coeffients, which is inefficient. This can be avoided when using the *Newton basis*:

$$\pi_0(x) = 1, \quad \pi_k(x) = \prod_{j=0}^{k-1}(x - x_j), \quad k = 1, 2, \ldots, n. \tag{3.6}$$

Then as usual the interpolation polynomial can be written as a sum of basis functions:

$$p_n(x) = d_0\pi_0(x) + d_1\pi_1(x) + \ldots + d_n\pi_n(x) = \mathbf{d}^T\,\boldsymbol{\pi}(x). \tag{3.7}$$

Why is this a good choice of basis? Consider the interpolation conditions (3.1) for the Newton basis, and the linear system $\mathbf{Aa} = \mathbf{f}$ resulting from these. In general the matrix $\mathbf{A}$ has entries $a_{ij} = \phi_j(x_i)$, in the case of the Newton basis we have the matrix

$$\mathbf{U} = \begin{pmatrix} \pi_0(x_0) & \pi_1(x_0) & \ldots & \pi_n(x_0) \\ \vdots & \vdots & & \vdots \\ \pi_0(x_n) & \pi_1(x_n) & \ldots & \pi_n(x_n) \end{pmatrix}. \tag{3.8}$$

However, examining (3.6) shows that

$$\pi_k(x_j) = 0 \text{ if } j < k. \tag{3.9}$$

Therefore, the matrix $\mathbf{U}$ is a lower triangular matrix:

$$\mathbf{U} = \begin{pmatrix} 1 & 0 & 0 & \ldots & 0 \\ 1 & (x_1 - x_0) & 0 & \ldots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & (x_n - x_0) & (x_n - x_0)(x_n - x_1) & \ldots & \prod_{j=0}^{n-1}(x_n - x_j) \end{pmatrix}. \tag{3.10}$$

and the linear system

$$\mathbf{Ad} = \mathbf{f}$$

is particularly easy to solve.

Example 3.6 (Continuation of Example 3.2)
Construct a quadratic approximation of $f(x) = \sin(x)$ *on the interval* $[0, \pi]$ *with nodes at* $x = (0, \pi/2, \pi)$.

The Newton basis depends only on the nodes x_i*:*

$$\begin{aligned} \pi_0(x) &= 1 \\ \pi_1(x) &= (x - x_0) = x \\ \pi_2(x) &= (x - x_0)(x - x_1) = x(x - \pi/2) \end{aligned}$$

and the resulting linear system for the coefficients is

$$\mathbf{U} = \begin{pmatrix} \pi_0(x_0) & \pi_1(x_0) & \pi_2(x_0) \\ \pi_0(x_1) & \pi_1(x_1) & \pi_2(x_1) \\ \pi_0(x_2) & \pi_1(x_2) & \pi_2(x_2) \end{pmatrix} \begin{pmatrix} d_0 \\ d_1 \\ d_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & \pi/2 & 0 \\ 1 & \pi & \pi^2/2 \end{pmatrix} \begin{pmatrix} d_0 \\ d_1 \\ d_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}. \tag{3.11}$$

Clearly from the first row $d_0 = 0$. Substituting into the 2nd row gives directly $d_1 = 2/\pi$, and then the 3rd row becomes

$$\frac{2}{\pi}\pi + \frac{\pi^2}{2} d_2 = 0$$

so $d_2 = -4/\pi^2$ and the interpolating polynomial is

$$p(x) = \frac{2}{\pi}x - \frac{4}{\pi^2}x(x - \frac{\pi}{2}) = \frac{4}{\pi}x - \frac{4}{\pi^2}x^2,$$

i.e. exactly the same polynomial obtained in Example 3.2.

If you ever have to do polynomial interpolation on paper (e.g. in an exam) — this is usually the easiest way to do it

3.4 Lagrange polynomial basis

We would like to construct a polynomial basis with particularly simple form of the matrix $\mathbf{A}$ – this is the Lagrange basis. It is obtained in the following way: consider the reconstruction of a function using a monomial basis, and substitute in the expression for $\mathbf{a}$:

$$p_n(x) = \mathbf{a}^T \mathbf{m}_n(x) = \left(\mathbf{V}^{-1}\mathbf{f}\right)^T \mathbf{m}_n(x) = \mathbf{m}^T(x)\mathbf{V}^{-1}\mathbf{f}. \tag{3.12}$$

Suppose we define a new polynomial basis

$$\mathbf{l}(x) := \mathbf{m}(x)^T \mathbf{V}^{-1}. \tag{3.13}$$

Then, every element $l_i(x)$ of $\mathbf{l}$ is a polynomial in x of degree n, and the expression for the interpolating polynomial (3.12) becomes simply:

$$p_n(x) = \mathbf{l}(x)^T \mathbf{f} = \sum_{i=0}^{n} f_i\, l_i(x).$$

I.e. the interpolation matrix $\mathbf{A}$ has become the identity, and the interpolation coefficients $\mathbf{a}$ are just the function values $\mathbf{f}$.

From the interpolation condition $p_n(x_i) = f_i,\ i = 0, \ldots, n$, it follows that

$$l_i(x_j) = \delta_{ij},$$

where

$$\delta_{ij} := \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}.$$

Hence, the nodes $\{x_j : i \neq j\}$ are the *zeros* of the polynomial $l_i(x)$. In the next step we therefore construct polynomials which take the value 0 at all nodes except x_i, and take the value 1 at x_i:[2]

$$l_i(x) = \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{x - x_j}{x_i - x_j}, \quad i = 0, \ldots, n.$$

These polynomials of degree n, $l_i(x)$, are called *Lagrange polynomials* and are often denoted $l_i(x)$. This notation will be used in these notes:

$$l_i(x) = \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{x - x_j}{x_i - x_j}, \quad i = 0, \ldots, n.$$

When using the Lagrange polynomials, the interpolant can be written as

$$p_n(x) = \mathbf{f}^T \mathbf{l}(x), \tag{3.14}$$

which is the Lagrange representation of the interpolation polynomial. The advantage of the Lagrange representation is that the Vandermonde matrix (3.3) may become ill-conditioned if n is large and/or the distance between two interpolation nodes x_i and x_j is small (and two rows of $\mathbf{V}$ become similar, so $\det \mathbf{V} \to 0$. When using the Lagrange representation (3.14) it is possible to write down the interpolating polynomial $p_n(x)$ without solving a linear system to compute the coefficients; the coefficient of the Lagrange basis functions $l_i(x)$ is the function value f_i (this is useful on quizzes).

Example 3.7
Find the Lagrange interpolation polynomial which agrees with the following data. Use it to estimate the value of $f(2.5)$.

i	0	1	2	3
x_i	0	1	3	4
$f(x_i)$	3	2	1	0

[2] Alternatively we could invert $\mathbf{V}$ in (3.13) which gives the same answer, but in a less clean form.

The Lagrange polynomials for this case are

$$\begin{aligned} l_0(x) &= \frac{(x-1)(x-3)(x-4)}{(0-1)(0-3)(0-4)} = -\frac{1}{12}(x-1)(x-3)(x-4), \\ l_1(x) &= \frac{(x-0)(x-3)(x-4)}{(1-0)(1-3)(1-4)} = \frac{1}{6}x(x-3)(x-4), \\ l_2(x) &= \frac{(x-0)(x-1)(x-4)}{(3-0)(3-1)(3-4)} = -\frac{1}{6}x(x-1)(x-4), \\ l_3(x) &= \frac{(x-0)(x-1)(x-3)}{(4-0)(4-1)(4-3)} = \frac{1}{12}x(x-1)(x-3), \end{aligned}$$

and, therefore, the interpolation polynomial is

$$p(x) = 3l_0(x) + 2l_1(x) + 1l_2(x) + 0l_3(x) = \frac{-x^3 + 6x^2 - 17x + 36}{12}.$$

The estimated value of $f(2.5)$ *is* $p(2.5) = 1.28125$.

3.5 Interpolation Error

Intuitively we feel that as the number of data points and polynomial order increases, the accuracy of the interpolation should improve, i.e. the interpolation error (3.15) should become smaller and smaller — at least for smooth functions. This feeling is supported by the Weierstrass' theorem. However, what Weierstrass's theorem states is that there always exists arbitrarily exact polynomial approximations; what it does not say is that *any* interpolation process can be used to find them. In fact, within $[x_0, x_n]$ it is not always true that finer and finer samplings of the function will give better and better approximations through interpolation. This is the question of *convergence*. To find the answer to the question whether the interpolation error (3.15) goes to zero if the number of nodes goes to infinity is much more difficult then one might expect at first glance.

It turns out that the situation is particularly bad for equidistant nodes, i.e., if the interpolation points are uniformly spaced on the interval. Then, for instance, uniform convergence is not guaranteed even for $f(x) \in C^\infty$ (infinitely differentiable or *analytic* functions). One classical example is due to Carl Runge:

Example 3.8 (Non-convergence of polynomial interpolation)
Let f *be defined on the interval* $[-5, 5]$ *by* $f(x) = \frac{1}{1+x^2}$. *Calculate the interpolant using* $n = 11$ *and* $n = 21$ *equally spaced data points. The results are shown in figure 3.3. It can be shown that* $p_n(x) \not\to f(x)$ *as* $n \to \infty$ *for any* $|x| > 3.63$.

Hence, the interpolation error $\|f - p_n\|_\infty$ *grows without bound as* $n \to \infty$. *Another example is* $f(x) = |x|$ *on* $[-1, 1]$, *for which the interpolation polynomials do not even converge pointwise except at the 3 points* $-1, 0, 1$. *So, the answer to the question, whether there exists a single grid* X *for which the sequence of interpolation polynomials converge to any*

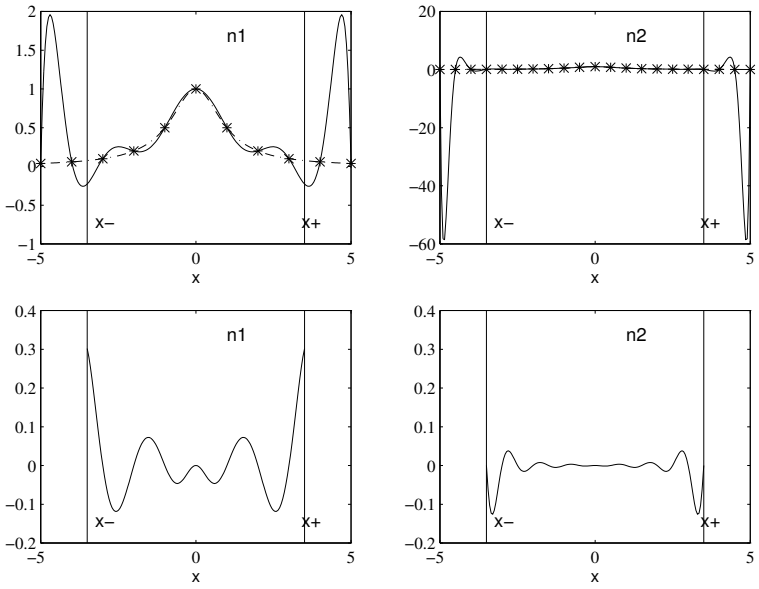


Figure 3.3: The top row shows $f(x)$ (dashed line) and the interpolant (solid line) using $n = 11$ and $n = 21$ equally spaced data points (stars) respectively. We also indicated the lines $x = \pm 3.5$. At the bottom row the difference between f and its interpolant are shown for $x \in [-3.5, 3.5]$.

continuous function f is 'no'. However, Weierstrass' theorem tells us that such a grid exists for every continuous function! Hence, the statement made before implies that we would need to determine this grid for each individual continuous function newly.

We need a way of estimating the error in an interpolation. Up to now, it has been irrelevant whether the data values f_i are related to each other in some way or not. However, if we want so say something about how the interpolant behaves between the data points, this question becomes important. This question is answered by a theorem due to Cauchy

Theorem 3.9 (Interpolation error (Cauchy))
If $f \in C^{n+1}([a,b])$, then for any grid X of $n+1$ nodes and for any $x \in [a,b]$, the interpolation error at x is

$$R_n(f;x) := f(x) - p_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}\,\omega_{n+1}(x), \tag{3.15}$$

where $\xi = \xi(x) \in (\min_k(x_k,x), \max_k(x_k,x)) \subset [a,b]$ and $\omega_{n+1}(x)$ is the nodal polynomial associated with the grid X, i.e.,

$$\omega_{n+1}(x) = \prod_{i=0}^{n}(x - x_i).$$

The nodal polynomial is unique, a polynomial of degree $n+1$, and has leading coefficient 1. The latter means that the coefficient of the highest power of x, x^{n+1}, is equal to 1. Moreover, all nodes x_i are the zeros of the nodal polynomial.

In most instances we don't know ξ. Then, we may use the following estimate:

$$|R_n(f;x)| \le \max_{x\in[a,b]} |f^{n+1}(x)| \frac{|\omega_{n+1}(x)|}{(n+1)!}. \tag{3.16}$$

In (3.16), we have no control over $\max_{x\in[a,b]} |f^{n+1}(x)|$, which depends on the function, and which can be very large. Take for instance

$$f(x) = \frac{1}{1+\alpha^2 x^2} \Rightarrow \|f^{n+1}\|_\infty = (n+1)!\,\alpha^{n+1}.$$

However, we may have control on the grid X. Hence, we may reduce the interpolation error if we choose the grid so that $\|\omega_{n+1}\|_\infty$ is small. The question is therefore: what is the grid for which this is minimized? The answer is given by the following theorem:

Theorem 3.10 (Control of $\|\omega_{n+1}\|_\infty$)
Among all the polynomials of degree $n+1$ and leading coefficient 1, the unique polynomial which has the smallest uniform norm on $[-1,1]$ is the Chebychev polynomial of degree $n+1$ divided by 2^n:

$$\frac{T_{n+1}(x)}{2^n}.$$

3.5.1 Chebychev polynomials

Since it can be shown that $\|T_{n+1}\|_\infty = 1$, we conclude that if we choose the grid X to be the $n+1$ zeros of the Chebychev polynomial $T_{n+1}(x)$, it is

$$\omega_{n+1}(x) = \frac{T_{n+1}(x)}{2^n},$$

and

$$\|\omega_{n+1}\|_\infty = \frac{1}{2^n}$$

and this is the smallest possible value! The grid X such that the x_i's are the $n+1$ zeros of the Chebychev polynomial of degree $n+1$, $T_{n+1}(x)$, is called the *Chebychev-Gauss* grid. Then, Eq. (3.16) tells us that if $f^{n+1} < C$, for some constant C the convergence of the interpolation polynomial towards f for $n \to \infty$ is extremely fast. Hence, the Chebyshev-Gauss grid has much better interpolation properties than any other grid, in particular the uniform one.

What is left is to define briefly the Chebyshev polynomials. The Chebyshev polynomial of degree n for $x \in [-1, 1]$ is defined by

$$T_n(x) := \cos(n \arccos(x)),$$

or equivalently

$$T_n(\cos(x)) = \cos(nx).$$

Note that the Chebyshev polynomials are only defined on the interval $[-1, 1]$.

Chebyshev polynomials can also be defined recursively. The first two are

$$T_0(x) = 1, \quad T_1(x) = x,$$

and the higher-degree Chebyshev polynomials can be obtained by applying

$$T_{n+1}(x) = 2x\,T_n(x) - T_{n-1}(x), \quad n = 1, 2, \ldots. \tag{3.17}$$

For instance, $T_2(x) = 2x^2 - 1$, $T_3(x) = 4x^3 - 3x$ etc. Note that $T_n(x)$ is an polynomial in x of degree n. It has n distinct zeros, which are all located inside the interval $[-1, 1]$. These zeros are given by

$$\xi_i = \cos\Big(\frac{2i-1}{2n}\,\pi\Big), \quad i = 1, \ldots, n. \tag{3.18}$$

The first few Chebyshev polynomials are plotted in Figure 3.4 - note their roots, and take maximum and minimum values of ± 1. The coeffcient of x^n in T_n is 2^{n-1}, which can be deduced from examining the recurrence relation (3.17). Therefore the maximum value of the monic polynomial

$$\tilde{T}_n(x) := \frac{T_n(x)}{2^{n+1}},$$

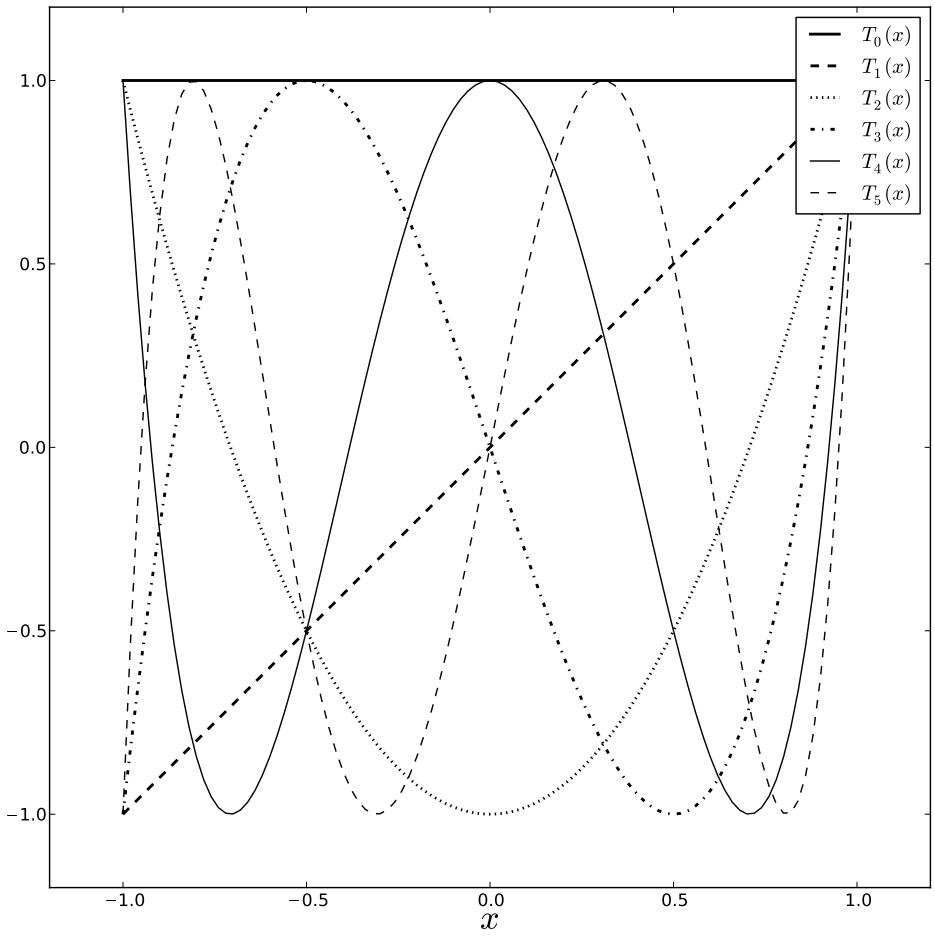


Figure 3.4: The first 6 Chebyshev polynomials.

is $1/2^{n+1}$.

When we want to interpolate a function $f(x)$ on the interval $[a, b]$ using a polynomial of degree n on a Chebyshev-Gauss grid, the grid points $a < x_0 < x_1 < \ldots < x_n < b$ can be computed by a simple linear transformation, which maps $[-1, 1]$ on $[a, b]$. Taking also into account that the first node of the grid, x_0, has index 0 and the zero closest to -1 of the Chebyshev polynomial $T_{n+1}(x)$ is ξ_{n+1}, the nodes on $[a, b]$ can be computed by

$$x_{n-i} = \frac{b+a}{2} + \frac{b-a}{2}\,\xi_i, \quad i = 1, \ldots, n. \tag{3.19}$$

The $\{x_i\}$ are the nodes of the grid X we would choose for interplation on the interval $[a, b]$.

Example 3.11 (Nodes of a Chebyshev-Gauss grid)
We want to interpolate a function $f(x)$ on the interval $[6, 10]$ by a degree-4 univariate polynomial using a Chebyshev-Gauss grid. Compute the grid nodes. Solution: we need 5 nodes, i.e., we take the zeros of $T_5(x)$, which can be computed as

$$\xi_i = \cos\Big(\frac{2i-1}{10}\,\pi\Big), \quad i = 1, \ldots, 5. \tag{3.20}$$

These zeros are transformed onto $[6, 10]$ using the transformation

$$x_{5-i} = 8 + 2\,\xi_i, \quad i = 1, \ldots, 5. \tag{3.21}$$

The results are shown in table 3.1 Hence, the interpolation nodes are $x_0 = 6.098$, $x_1 = 6.824$,

i	ξ_i	x_{5-i}
1	0.951	9.902
2	0.588	9.176
3	0.000	8.000
4	-0.588	6.824
5	-0.951	6.098

Table 3.1: 5-point Chebyshev-Gauss grid on $[6, 10]$.

$x_2 = 8.000$, $x_3 = 9.176$, and $x_4 = 9.902$.

Though the choice of the Chebyshev-Gauss grid for interpolation has advantages over a uniform grid, there are also some penalties.

1. Extrapolation using the interpolant based on data measured at Chebyshev-Gauss points is even more disastrous than extrapolation based on the same number of equally spaced data points.

2. In practice it may be difficult to obtain the data f_i measured at the Chebyshev points. Therefore, if for some reason it is not possible to choose the Chebyshev-Gauss grid, choose the grid so that there are more nodes towards the endpoints of the interval $[a, b]$.

Example 3.12 (Continuation of example 3.8)
Figure 3.5 shows the result of applying interpolation to the data points (3.19) *for* $n = 11$ *and* $n = 21$, *distributed over the interval* $[-5, 5]$. *In the case of* 11 *data points*, 6 *of them are placed in the previously troublesome region* $|x| > 3.5$.

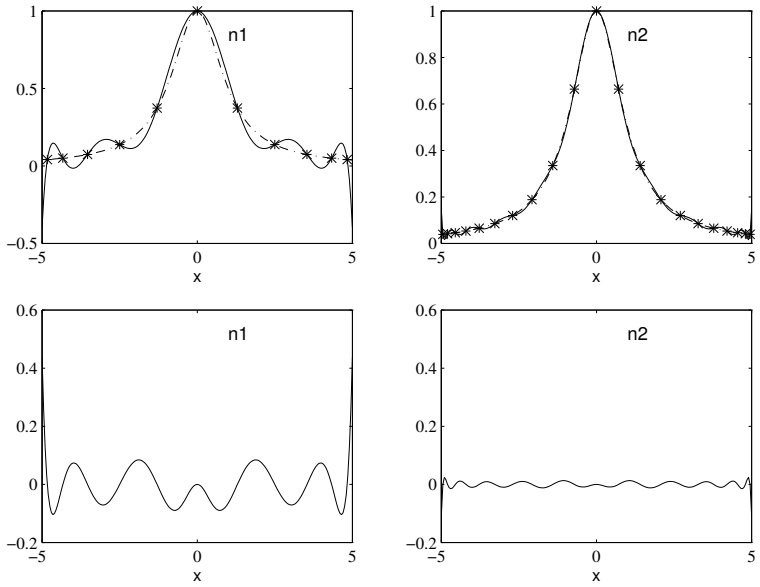


Figure 3.5: Here we show f (dashed line) and its interpolant (solid line) using unequally spaced data points (stars) distributed over $[-5, 5]$ according to (3.19). The bottom row shows again the difference between f and its interpolant.

Chapter 4

Advanced Interpolation: Splines, Multi-dimensions and Radial Bases

4.1 Spline interpolation

Standard polynomial interpolation through $n+1$ given data points produces a single polynomial of degree $\leq n$, which passes exactly through every data point. This can be unstable if the number of points is large. Further, if the number of data is large, the solution of the linear system may be expensive. Finally, the Runge phenomenon implies large oscillations around the boundaries of the grid. A solution to the latter problem is the choice of a Chebyshev-Gauss grid. However, as has been noticed already before, this may not always be possible in practice. Therefore, we want to consider another technique designed to reduce the problems that arise when data are interpolated by a *single* polynomial. The basic idea is to use a *collection of low-degree polynomials* to interpolate the give data rather than a single high-degree polynomial. That this may be beneficial is seen for instance in the estimation of the interpolation error. Remember the estimate

$$\|R(f;x)\|_\infty \leq \frac{\|f^{n+1}(\xi)\|_\infty}{(n+1)!}\,|\omega_{n+1}(x)|, \tag{4.1}$$

where

$$\omega_{n+1}(x) = (x-x_0)(x-x_1)\cdots(x-x_n) \tag{4.2}$$

is the nodal polynomial of degree $n+1$. Since $|\omega_{n+1}(x)|$ is is the product of $n+1$ linear factors $|x-x_i|$, each the distance between the two points that both lie in $[a,b]$, we have $|x-x_i| \leq b-a$ and so

$$|\omega_{n+1}(x)| \leq (b-a)^{n+1}. \tag{4.3}$$

Hence, the maximum absolute interpolation error is

$$\|R(f;x)\|_\infty \leq (b-a)^{n+1}\,\frac{\|f^{n+1}\|_\infty}{(n+1)!}. \tag{4.4}$$

This large error bound suggests that we can make the interpolation error as small as we wish by freezing the value of n and then reducing the size of $b-a$. We still need an approximation over the original interval $[a, b]$, so we use a *piecewise polynomial approximation*: the original interval is divided into non-overlapping subintervals and a different polynomial fit of the data is used on each subinterval.

A simple piecewise polynomial fit is obtained in the following way: for data $\{(x_i, f_i) : i = 0, \ldots, n\}$, where $a \leq x_0 < x_1 < \cdots < x_n \leq b$, we take the straight line connection of two neighbouring data points x_i and x_{i+1} as the interpolant on the interval $[x_i, x_{i+1}]$. Hence, globally the interpolant is the unique polygon obtained by joining the data points together. Instead of using linear polynomials to interpolate between two neighboring data points we may use quadratic polynomials to interpolate between three neighboring data points, cubic polynomials to interpolate between four neighboring data points etc. In this way we may improve the performance but at the expense of smoothness in the approximating function. Globally the interpolant will be at best continuous, but it will not be differentiable at the data points. For many applications a higher degree of smoothness at the data points is required. This additional smoothness can be achieved by using low-degree polynomials on each interval $[x_i, x_{i+1}]$ while imposing some smoothness conditions at the data points to ensure that the overall interpolating function has globally as high a degree of continuity as possible. The corresponding functions are called *splines*.

Definition 4.1 (Spline of degree *d*)
Let $a = x_0 < x_1 < \cdots < x_n = b$ be an increasing sequence of data points. The function s is a spline of degree d if

(a) s is a polynomial of degree at most d on each of the subintervals $[x_i, x_{i+1}]$,

(b) $s, s', \ldots, s^{(d-1)}$ are all continuous on $[a, b]$.

4.1.1 Linear Splines (d=1)

The connection of points by straight lines as explained above is an example of a spline of degree 1, i.e. a piecewise linear function which is continuous at the data points. This spline is called a *linear spline*. It is given as

$$s(x) = \begin{cases} s_0(x) & x \in [x_0, x_1] \\ s_1(x) & x \in [x_1, x_2] \\ \vdots & \vdots \\ s_{n-1}(x) & x \in [x_{n-1}, x_n] \end{cases}, \tag{4.5}$$

where

$$s_i(x) = f_i + \frac{f_{i+1} - f_i}{x_{i+1} - x_i}(x - x_i), \quad i = 0, 1, \ldots, n-1. \tag{4.6}$$

Hence, if we are given $n+1$ data points, the linear spline will consists of n degree-1 polynomials each of which holds between a pair of consecutive data points. With $h := \max_i |x_{i+1} - x_i|$, we obtain an upper bound for the interpolation error for $x \in [x_i, x_{i+1}]$ from (4.1) with $n = 1$ as follows: define

$$\hat{x} := \frac{x_i + x_{i+1}}{2},$$

the interval midpoint. Then for the nodal polynomial in this case:

$$|\omega_2(x)| = |(x - x_i)(x - x_{i+1})| \leq |(\hat{x} - x_i)(\hat{x} - x_{i+1})| = \frac{x_{i+1} - x_i}{2} \frac{x_{i+1} - x_i}{2},$$

where the 1st inequality follows from the fact that the maximum or minimum of a parabola is located at the midpoint of the two roots. Therefore by (4.1), an upper bound on the interpolation error on the interval is:

$$\|R(f; x)\|_\infty \leq \frac{|x_{i+1} - x_i|^2}{8} \|f^{(2)}\|_\infty \leq \frac{h^2}{8} \|f^{(2)}\|_\infty. \tag{4.7}$$

Note that this bound is smaller, by a factor of 4, than the general upper bound given in (4.4). It is therefore *tighter* (i.e. better). If an upper bound is the best possible upper bound in a given situation, it is called *tight*.

The most important feature of the above error bound is that it behaves like h^2. Suppose that the nodes are chosen equally spaced in $[a, b]$, so that $x_i = a + ih$, $i = 0, \ldots, N$, where $h = \frac{b-a}{N}$. As the number of data points $N + 1$ increases, the error using a linear spline as an approximation to $f(x)$ tends to zero like $\frac{1}{N^2}$. In general when a polynomial approximation uses polynomials of degree d, we expect the error to behave like h^{d+1}.

4.1.2 Cubic Splines (d=3)

Linear splines suffer from a major limitation: the derivative of a linear spline is generally discontinuous at each interior node x_i. Hence, the linear spline is a continuous function on $[a, b]$, but not smoother. To derive a piecewise polynomial approximation with a continuous derivative requires that we use polynomial pieces of higher degree and constrain the pieces to make the interpolant smoother. By far the most commonly used spline for interpolation purposes is the cubic spline ($d = 3$), which is in $C^2([a, b])$, i.e., the first and second derivatives at the interior nodes x_i are still continuous functions. We will restrict our attention on them.

Let us denote the subinterval $[x_i, x_{i+1}]$ by I_i and the spline restricted to I_i by s_i, i.e.

$$s(x) = s_i(x) \quad \text{for } x \in I_i, \quad i = 0, \ldots, n - 1.$$

The conditions which s must satisfy are that s interpolates f at the data points $x_0, \ldots, x_n$ and that s' and s'' must be continuous at the interior data points $x_1, \ldots, x_{n-1}$. Let us determine the spline interpolant s from these conditions.

Since s_i is a cubic polynomial, s'' is linear. Let us denote the yet unknown values of s'' at the data points x_i and x_{i+1} by M_i and M_{i+1}, respectively, i.e. $s''(x_i) = M_i$ and $s''(x_{i+1}) = M_{i+1}$.

$s_i(x)$ is a cubic polynomial, hence it can be written as

$$s_i(x) = a_i(x - x_i)^3 + b_i(x - x_i)^2 + c_i(x - x_i) + d_1, \quad i = 0, \ldots, n-1. \tag{4.8}$$

Then,

$$\begin{aligned} s_i'(x) &= 3a_i(x - x_i)^2 + 2b_i(x - x_i) + c_i \\ s_i''(x) &= 6a_i(x - x_i) + 2b_i. \end{aligned}$$

hence,

$$\begin{aligned} s_i''(x_i) = M_i = 2b_i &\Rightarrow b_i = \frac{M_i}{2} \\ s_i''(x_{i+1}) = 6a_i h_i + 2b_i &\Rightarrow a_i = \frac{M_{i+1} - M_i}{6h_i}, \end{aligned}$$

where we have defined $h_i := x_{i+1} - x_i$. We insert the results for a_i and b_i into the equation of the spline $s_i(x)$ and find

$$s_i(x) = \frac{M_{i+1} - M_i}{6h_i}(x - x_i)^3 + \frac{M_i}{2}(x - x_i)^2 + c_i(x - x_i) + d_i. \tag{4.9}$$

The interpolation conditions yield $s_i(x_i) = f_i$ and $s_i(x_{i+1}) = f_{i+1}$, hence,

$$\begin{aligned} s_i(x_i) = f_i = d_i &\Rightarrow d_i = f_i \\ s_i(x_{i+1}) = f_{i+1} = \frac{M_{i+1} - M_i}{6h_i}h_i^3 + \frac{M_i}{2}h_i^2 + ch_i + f_i& \\ \Rightarrow c_i = \frac{f_{i+1} - f_i}{h_i} - \frac{h_i}{3}M_i - \frac{h_i}{6}M_{i+1}.& \end{aligned}$$

We insert the results for d_i and c_i into the equation for $s_i(x)$ and find

$$s_i(x) = \frac{M_{i+1} - M_i}{6h_i}(x - x_i)^3 + \frac{M_i}{2}(x - x_i)^2 + \Big(\frac{f_{i+1} - f_i}{h_i} - \frac{h_i}{3}M_i - \frac{h_i}{6}M_{i+1}\Big)(x - x_i) + f_i. \tag{4.10}$$

So far, we have not used the conditions

$$s_i'(x_i) = s_{i-1}'(x_i), \quad i = 1, \ldots, n-1, \tag{4.11}$$

which provide $n - 1$ equations. It is

$$s_i'(x) = \frac{M_{i+1} - M_i}{2h_i}(x - x_i)^2 + M_i(x - x_i) + \Big(\frac{f_{i+1} - f_i}{h_i} - \frac{h_i}{3}M_i - \frac{h_i}{6}M_{i+1}\Big), \tag{4.12}$$

hence,

$$s_i'(x_i) = \Big(\frac{f_{i+1} - f_i}{h_i} - \frac{h_i}{3}\, M_i - \frac{h_i}{6}\, M_{i+1}\Big). \tag{4.13}$$

In the same way, we find

$$s_{i-1}'(x) = \frac{M_i - M_{i-1}}{2h_{i-1}}\,(x - x_{i-1})^2 + M_{i-1}\,(x - x_{i-1}) + \Big(\frac{f_i - f_{i-1}}{h_{i-1}} - \frac{h_{i-1}}{3}\, M_{i-1} - \frac{h_{i-1}}{6}\, M_i\Big), \tag{4.14}$$

hence,

$$s_{i-1}'(x_i) = \frac{h_{i-1}}{3}\, M_i + \frac{h_{i-1}}{6}\, M_{i-1} + \frac{f_i - f_{i-1}}{h_{i-1}}. \tag{4.15}$$

Therefore, the conditions $s_i'(x_i) = s_{i-1}'(x_i)$, $i = 1, \ldots, n-1$ yield the following system of $n-1$ equations for the $n+1$ unknowns $M_0, M_1, \ldots, M_{n-1}, M_n$:

$$\frac{h_{i-1}}{6} M_{i-1} + \frac{(h_{i-1} + h_i)}{3} M_i + \frac{h_i}{6} M_{i+1} = \frac{f_{i+1} - f_i}{h_i} - \frac{f_i - f_{i-1}}{h_{i-1}}, \quad i = 1, \ldots, n-1. \tag{4.16}$$

This system has infinitely many solutions. A unique solution can only be obtained if additional constraints are imposed. There are many constraints we could choose. The simplest constraints are

$$M_0 = M_n = 0. \tag{4.17}$$

When making this choice, the cubic spline is called *natural cubic spline*. The natural cubic spline may deliver no accurate approximation of the underlying function at the ends of the interval $[x_0, x_n]$. This may be anticipated from the fact that we are forcing a zero value on the second derivative when this is not necessarily the value of the second derivative of the function which the data measures. For instance, a natural cubic spline is built up from cubic polynomials, so it is reasonable to expect that if the data is measured from a cubic polynomial then the natural cubic spline will reproduce the cubic polynomial. However, if the data are measured from, e.g., the function $f(x) = x^2$, then the natural cubic spline $s(x) \neq f(x)$. The function $f(x) = x^2$ has nonzero second derivatives at the nodes x_0 and x_n where the value of the second derivative of the natural cubic spline is zero by definition.

To clear up the inaccuracy problem associated with the natural spline conditions, we could replace them with the correct second derivatives values

$$s''(x_0) = f''(x_0), \quad s''(x_n) = f''(x_n). \tag{4.18}$$

These second derivatives of the data are not usually available, but they can be replaced by reasonable approximations. Anyway, if the exact values or sufficiently accurate approximations are used then the resulting spline will be as accurate as possible for a cubic spline. Such approximations may be obtained by using polynomial interpolation to sufficient data values separately near each end of the interval $[x_0, x_n]$. Then, the two interpolating polynomials are each twice differentiated and the resulting twice differentiated polynomials are evaluated at the corresponding end points to approximate the f'' there.

A simpler, and usually sufficiently accurate spline may be determined as follows: on the first two and the last two intervals we define each a cubic polynomial, hence x_1 and x_{n-1} are in fact no nodes. This fixes in fact the two unknown second derivatives M_1 and M_{n-1} uniquely and we are left with the solution of a linear system of dimension $n-1$ for the $n-1$ unknowns $M_0, M_2, M_3, \ldots, M_{n-2}, M_n$. The corresponding cubic spline is sometimes called a *not-a-knot* spline ('knot' is an alternative notation for 'node').

For each way of supplying the additional constraints that is discussed before, the cubic spline is unique. From the error bound for polynomial interpolation, for a cubic polynomial interpolating at data points in the interval $[a, b]$, we have

$$\|R(f;x)\|_\infty \leq Ch^4 \|f^{(4)}\|_\infty, \tag{4.19}$$

where C is a constant and $h = \max_i(x_{i+1} - x_i)$. Therefore, we might anticipate that the error associated with a cubic spline interpolant behaves like h^4 for h small. However, the maximum absolute error associated with a *natural* cubic spline behaves like h^2 as $h \to 0$. In contrast, the maximum absolute error for a cubic spline based on correct endpoint second derivatives or on the not-a-knot conditions behaves like h^4. Unlike the natural cubic spline, the correct second derivative value and not-a-knot cubic splines reproduce cubic polynomials.

Example 4.2
Find the natural cubic spline which interpolates the data

x_i	0.0	0.1	0.3	0.6
f_i	0.0000	0.2624	0.6419	1.0296

With $h_0 = 0.1, h_1 = 0.2$, and $h_2 = 0.3$ we obtain the following linear system of equations for the unknowns M_1 and M_2:

$$\begin{aligned} \frac{0.3}{3} M_1 + \frac{0.2}{6} M_2 &= 1.8975 - 2.6240 \\ \frac{0.2}{6} M_1 + \frac{0.5}{3} M_2 &= 1.2923 - 1.8975 \end{aligned}$$

It has the solution $M_1 = -6.4871, M_2 = -2.3336$. Then, the natural cubic spline is

$$s(x) = \begin{cases} -10.812\,x^3 + 2.7321\,x & x \in [0, 0.1] \\ 3.4613(x-0.1)^3 - 3.2436(x-0.1)^2 + 2.4078(x-0.1) + 0.2624 & x \in [0.1, 0.3] \\ 1.2964(x-0.3)^3 - 1.1668(x-0.3)^2 + 1.5257(x-0.3) + 0.6419 & x \in [0.3, 0.6] \end{cases}$$

We can simplify substantially the rather time-consuming computations if the data points are equally spaced so that $x_i = x_0 + ih, i = 1, \ldots, n$. Then we obtain the following linear system

of equations for the coefficients $M_i, i = 1, \ldots, n-1$ of the natural cubic spline:

$$\begin{pmatrix} 4 & 1 & 0 & \ldots & \ldots & 0 \\ 1 & 4 & 1 & 0 & \ldots & 0 \\ \vdots & & & & & \vdots \\ 0 & \ldots & 0 & 1 & 4 & 1 \\ 0 & \ldots & \ldots & 0 & 1 & 4 \end{pmatrix} \begin{pmatrix} M_1 \\ M_2 \\ \vdots \\ M_{n-1} \end{pmatrix} = \frac{6}{h} \begin{pmatrix} f[x_0, x_1, x_2] \\ f[x_1, x_2, x_3] \\ \vdots \\ f[x_{n-2}, x_{n-1}, x_n] \end{pmatrix} \tag{4.20}$$

Example 4.3
Find the natural cubic spline interpolant to the function $f(x) = \frac{1}{1+x^2}$ *from the following table:*

x_i	-1	-0.5	0	0.5	1
f_i	0.5	0.8	1	0.8	0.5

Compute $s(0.8)$ *and the error* $f(0.8) - s(0.8)$.

It is $n = 4, h = \frac{1}{2}$. *Moreover, for a natural spline we have* $M_0 = M_4 = 0$. *Further, we obtain* $f[x_0, x_1, x_2] = -0.2$, $f[x_1, x_2, x_3] = -0.8$, $f[x_2, x_3, x_4] = -0.2$. *This gives the linear system of equations*

$$\begin{pmatrix} 4 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 4 \end{pmatrix} \begin{pmatrix} M_1 \\ M_2 \\ M_3 \end{pmatrix} = \begin{pmatrix} -2.4 \\ -9.6 \\ -2.4 \end{pmatrix}$$

with the solution $M_1 = 0, M_2 = -2.4, M_3 = 0$. *Thus, the natural cubic spline interpolant to the data is:*

$$s(x) = \begin{cases} s_0(x) = 0.5 + 0.6(x+1) & x \in [-1, -0.5] \\ s_1(x) = 0.8 + 0.6(x+0.5) - 0.8(x+0.5)^3 & x \in [-0.5, 0] \\ s_2(x) = 1 - 1.2x^2 + 0.8x^3 & x \in [0, 0.5] \\ s_3(x) = 0.8 - 0.6(x-0.5) & x \in [0.5, 1] \end{cases}$$

It is $s(0.8) = s_3(0.8) = 0.8 - 0.6 \cdot 0.3 = 0.62$ *and the absolute error is* $|f(0.8) - s(0.8)| = |0.61 - 0.62| = 0.01$.

Exercise 4.4
Find the natural cubic spline which interpolates the values $\ln(1+2x)$ *at* $x = 0, 0.1, 0.2, 0.3, 0.4,$ *and* 0.5. *Use this spline to estimate the values of* $\ln(1.1)$, $\ln(1.3)$, $\ln(1.5)$, $\ln(1.7)$, *and* $\ln(1.9)$. *Compare the results with those using the linear spline.*

4.2 Bivariate interpolation

The interpolation problem can be generalized to two dimensions (*bivariate interpolation*):

Definition 4.5 (Bivariate interpolation)
Given data points $\{\mathbf{x}_i : i = 1, \ldots, n\}$ *with* $\mathbf{x} \in \bar{D} \subset \mathbb{R}^2$ *and function values* $\{f_i : i = 1, \ldots, n\}$, *find a (continuous) function* p *such that*

$$p(\mathbf{x}_i) = f_i, \quad i = 1, \ldots, n. \tag{4.21}$$

Geometrically, we can interpret this as finding an approximation of a surface in $\mathbb{R}^3$. The generalization to N-dimensions follows similarly.

In many cases the domain $\bar{D}$ is a rectangle and the data points lie on a rectangular grid. It may also happen that $\bar{D}$ is of unusual shape and the data points are irregularly scattered throughout $\bar{D}$.

In the following we will discuss bivariate interpolation methods for regular and scattered data. To distinguish between regular and scattered data is also interesting for so-called two-stage processes in which we first construct an interpolation function based on scattered data and then use this interpolation function to generate data on a grid for the construction of another, perhaps smoother or more convenient interpolation function.

A convenient and common approach to the construction of a two-dimensional interpolation function is the following: assume p is a linear combination of certain basis functions b_k, i.e.,

$$p(\mathbf{x}) = \sum_{k=1}^{n} c_k \, b_k(\mathbf{x}), \quad \mathbf{x} \in \mathbb{R}^2. \tag{4.22}$$

Solving the interpolation problem under this assumption leads to a system of linear equations of the form

$$\mathbf{A}\,\mathbf{c} = \mathbf{f}, \tag{4.23}$$

where the entries of the matrix $\mathbf{A}$ are

$$a_{ij} = b_j(\mathbf{x}_i), \quad i, j = 1, \ldots, n. \tag{4.24}$$

A unique solution of the interpolation problem will exist if and only if $\det(\mathbf{A}) \neq 0$. In one-dimensional interpolation, it is well-known that one can interpolate to *arbitrary* data at n distinct points using a polynomial of degree $n-1$. However, it can be shown that $\det(\mathbf{A}) \neq 0$ does not hold for arbitrary distributions of distinct data points in two (or more) dimensions. Hence, it is not possible to perform a unique interpolation with bivariate polynomials of a certain degree for data given at arbitrary locations in $\mathbb{R}^2$. If we want to have a well-posed (i.e. uniquely solvable) bivariate interpolation problem for scattered data, then the basis needs to depend on the data (what this means will be explained later). There are a very few exceptions of this rule, for specific data distributions, choice of basis functions, and choice of the orientation of the Cartesian coordinate system.

In the following we will discuss bivariate interpolation. We will start with polynomial interpolation, which is suitable for gridded data. For scattered data, basis functions will be used that depend on the data, the so-called *radial basis functions*. Points in $\mathbb{R}^2$ are described using Cartesian coordinates, i.e., $\mathbf{x} = (x, y)^T$.

4.2.1 Tensor product polynomial interpolation

The bivariate interpolation problem is much more complicated than the univariate one even when restricting to polynomial interpolation. One reason is that it is not so easy to guarantee that the problem has a unique solution. For instance, the linear bivariate interpolation polynomial $p(x, y) = a_{00} + a_{10}x + a_{01}y$ may uniquely be determined by the function values at three data points $\{(x_i, y_i) : i = 1 \ldots, 3\}$. However, if the three data points lie on a line, then the problem does not have a unique solution. Therefore, to guarantee that a unique solution of the problem of polynomial interpolation exists, we have to impose some constraints on the location of the interpolation points and/or the domain.

Of particular practical interest is the case where the data points lie on a rectangular lattice in the (x, y)-coordinate plane and function values are assigned to each point. Thus, suppose there are $(n+1)(m+1)$ points $(x_i, y_j), 0 \le i \le n, 0 \le j \le m$ in the lattice, $n+1$ in x-direction and $m+1$ in y-direction (this automatically implies that the coordinate axes are supposed to be drawn parallel to the edges of the lattice; this pre-requisite is important to get a unique solution!). Then, each point is assigned coordinates (x_i, y_j), where $x_0 < x_1 < \ldots < x_{n-1} < x_n$ is the grid X and $y_0 < y_1 < \ldots < y_{m-1} < y_m$ is the grid Y. The interpolation problem then requires to determine a function $p = p(x, y)$ defined on the rectangle $[x_0, x_n] \times [y_0, y_m]$, for which $P(x_i, y_j) = f_{ij}$ for each i and j. If the x_i's and the y_j's are uniformly spaced, the lattice is called *uniform*.

The basic idea is to make use of the results of one-dimensional interpolation. Several representations of one-dimensional polynomials have been considered in the previous section: the monomial basis, the Lagrange basis, the Newton basis, and the basis of orthogonal polynomials. Now we consider products of these basis functions, i.e., we consider the set of $(n + 1) \cdot (m + 1)$ functions of the form

$$b_{ij}(x, y) = \varphi_i(x) \cdot \psi_j(y), \quad i = 0, \ldots, n, \;\; j = 0, \ldots, m. \tag{4.25}$$

For φ_i and ψ_j we may use any of the representations mentioned before. In particular, if we choose the Lagrange representation, it is

$$b_{ij}(x, y) = l_i^{(X)}(x) \cdot l_j^{(Y)}(y), \tag{4.26}$$

where

$$l_i^{(X)}(x) = \prod_{j=0, j\neq i}^{n} \frac{x - x_j}{x_i - x_j}, \quad l_j^{(Y)}(y) = \prod_{k=0, k\neq j}^{n} \frac{y - y_k}{y_j - y_k}, \tag{4.27}$$

and

$$b_{ij}(x_k, y_l) = l_i^{(X)}(x_k) \cdot l_j^{(Y)}(y_l) = \begin{cases} 1 & \text{if } k = i \text{ and } l = j \\ 0 & \text{otherwise} \end{cases}. \tag{4.28}$$

Hence, the functions $b_{ij}(x, y)$ behave like the univariate Lagrange basis functions, but are based now on the rectangular lattice. Then, the bivariate interpolation polynomial is given

by

$$p(x,y) = \sum_{i=0}^{n} \sum_{j=0}^{m} f_{ij} b_{ij}(x,y). \tag{4.29}$$

Defining a data matrix $\mathbf{F}$ with entries $F_{ij} = f_{ij}$ and vectors of basis functions $\mathbf{l}_X(x) = \left(l_0^{(X)}(x) \dots l_n^{(X)}(x)\right)^T$ and $\mathbf{l}_Y(y) = \left(l_0^{(Y)}(y) \dots l_m^{(Y)}(y)\right)^T$, we can write the bivariate interpolation polynomial also as

$$p(x,y) = \mathbf{l}_X^T(x)\Big)^T \mathbf{F}\, \mathbf{l}_Y(y). \tag{4.30}$$

Note that in general $l_k^{(X)}(x) \neq l_k^{(Y)}(y)$ unless $X = Y$.

Instead of using the Lagrange representation of the univariate polynomials in x and y, we could also use the Newton representation or the monomial representation. However, as already mentioned in the section on univariate interpolation, the Newton representation has to be preferred, because it is less computationally intensive than the other two representations. While in this manner quite acceptable interpolation surfaces result for small values of n and m, for larger values of n and m and/or certain configurations of data (i.e., equidistant data), the resulting surfaces have a very wavy appearance, due to strong oscillations along the boundary of the lattice. This phenomenon has already been discussed in the section on univariate interpolation, and all statements also apply to bivariate polynomial interpolation.

Therefore, for larger values of n and m it is better to use piecewise polynomials, e.g., splines. Howeover, the approach described above can only be used for Overhauser splines in x and y, but not for instance for the cubic spline. Theoretically, it would also be possible to use different basis functions for x and for y, though this must be justified by some a-priori information about the behavior of the function f in x and y.

The approach outlined before is called *tensor-product interpolation*. Existence and uniqueness of the tensor product interpolation is stated in the following theorem:

Theorem 4.6
Let $\varphi_0, \dots, \varphi_n$ be a set of functions and $x_0 < x_1 < \dots < x_n$ be a set of points with the property that, for any $f_0, f_1, \dots, f_n$,there exist unique numbers $\alpha_0, \dots, \alpha_n$ such that $\sum_{i=0}^{n} \alpha_i \varphi_i(x_k) = f_k$ for $k = 1, \dots, n$. Let $\psi_0, \dots, \psi_m$ have the corresponding property with respect to points $y_0 < y_1 < \dots < y_m$ and define

$$b_{ij}(x,y) = \varphi_i(x) \cdot \psi_j(y), \quad i = 0, \dots, n, \;\; j = 0, \dots, m. \tag{4.31}$$

Then, given any set of numbers f_{ij} there exists a unique corresponding set of numbers a_{ij} such that the function

$$p(x,y) = \sum_{i=0}^{n} \sum_{j=0}^{m} a_{ij}\, b_{ij}(x,y) \tag{4.32}$$

satisfies the interpolation conditions $p(x_k, y_l) = f_{kl}$ for each k and l.

Example 4.7
Find the bilinear interpolation polynomial from the data $f(0,0) = 1, f(1,0) = f(0,1) = f(1,1) = 0$.

The bilinear interpolation polynomial $p(x,y)$ *is given by:*

$$p(x,y) = \sum_{i=0}^{1} \sum_{j=0}^{1} f_{ij}\, l_i(x) l_j(y)$$

The Lagrange polynomials are:

$$l_0(x) = 1 - x, \quad l_1(x) = x, \quad l_0(y) = 1 - y, \quad l_1(y) = y$$

Therefore, we obtain

$$p(x,y) = (x-1)(y-1) = 1 - x - y + xy.$$

The surface $p(x,y) =$ constant *is called a "hyperbolic paraboloid".*

Exercise 4.8
Interpolate $f(x,y) = \sin(\pi x)\sin(\pi y)$ *on* $(x,y) \in [0,1] \times [0,1]$ *on a rectangular grid with the step size* $h_x = h_y = \frac{1}{2}$ *using a polynomial which is quadratic in* x *and* y.

4.2.2 Patch interpolation

A special case of polynomial interpolation is based on a triangulation of the interpolation domain. Normally rectangular or triangular elements are used for that. Whereas triangular elements can also be used for scattered data, the use of rectangular elements is restricted to gridded data.

Let us first consider a rectangular element, as in figure 4.1, with four nodes $P_1, \ldots, P_4$. If the function values are given at these four nodes, we can easily determine an interpolating function. Since 4 nodes determine uniquely an interpolation function with 4 parameters, a reasonable *'Ansatz'* might be:

$$p(x,y) = a_0 + a_1 x + a_2 y + a_3 xy$$

Let us assume that the points P_1, P_2, P_3, and P_4 have the *local* coordinates $(-1,1)$, $(1,1)$, $(-1,-1)$, and $(1,-1)$, respectively. Then, the interpolation condition reads

$$p(x_i, y_i) = f_i, \qquad i = 1, \ldots, 4$$

or

$$\begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{pmatrix} = \begin{pmatrix} 1 & -1 & 1 & -1 \\ 1 & 1 & 1 & 1 \\ 1 & -1 & -1 & 1 \\ 1 & 1 & -1 & -1 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix}$$

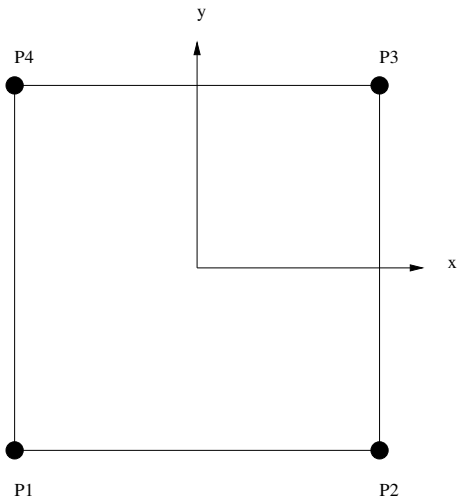


Figure 4.1: Rectangular patch.

This linear system of equations to determine the coefficients a_i can be written as

$$\mathbf{f} = \mathbf{A} \cdot \mathbf{a}$$

Formally,

$$\mathbf{a} = \mathbf{A}^{-1}\,\mathbf{f},$$

and we can write the interpolation polynomial on the rectangle as

$$p(x,y) = \mathbf{a}^T\mathbf{b}(x,y) = \mathbf{b}^T(x,y)\mathbf{a} = \mathbf{b}^T(x,y)A^{-1}\mathbf{f} =: \mathbf{S}(x,y)\mathbf{f}$$

The vector $\mathbf{S}(x,y)$ obtains the so-called *"shape functions"*. They can be calculated without knowing the function values at the nodes. In our case:

$$\mathbf{b}^T(x,y) = (1,x,y,xy),$$

and we obtain

$$\mathbf{S}(x,y) = \mathbf{b}^T(x,y)\,A^{-1} = \tfrac{1}{4}\begin{pmatrix} 1-x+y-xy \\ 1+x+y+xy \\ 1-x-y+xy \\ 1+x-y-xy \end{pmatrix} = \tfrac{1}{4}\begin{pmatrix} (1+y)(1-x) \\ (1+y)(1+x) \\ (1-y)(1-x) \\ (1-y)(1+x) \end{pmatrix} \tag{4.33}$$

Then, we can represent the interpolation function in the form

$$p(x,y) = \sum_{i=1}^{4} f_i\,S_i(x,y) \tag{4.34}$$

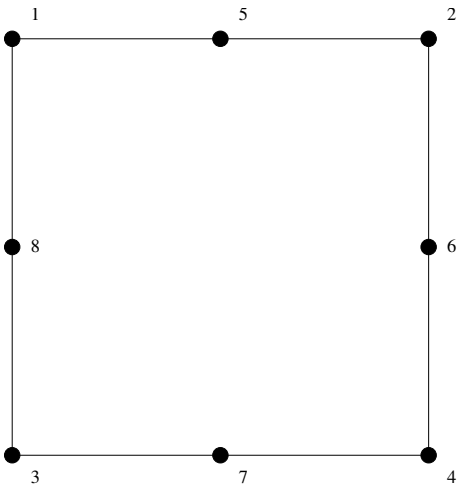


Figure 4.2: Quadratic Serendipity element.

Example 4.9 (Rectangular patch)
Given four data points $P_1 = (1.0, 0.5)$, $P_2 = (2.0, 0.5)$, $P_3 = (1.0, 0.2)$, $P_4 = (2.0, 0.2)$ *and the function values* $f_1 = 1.703$, $f_2 = 3.943$, $f_3 = 0.640$, $f_4 = 1.568$. *Calculate the bilinear interpolation polynomial.*

With respect to the standard rectangular patch, Fig 4.1, (coordinates (x, y)*), the bilinear interpolating polynomial is given by* (4.34) *with the basis functions given by* (4.33). *The mapping of the standard rectangular patch onto the given rectangular patch (coordinates* (ξ, η)*) is given by*

$$x = 2\xi - 3, \quad y = \frac{20}{3}\eta - \frac{7}{3}.$$

Therefore, in coordinates (ξ, η), *the basis functions are*

$$\tilde{\mathbf{S}}(\xi, \eta) = \mathbf{S}(x(\xi), y(\eta)) \ = \frac{1}{3} \begin{pmatrix} -2(5\eta - 1)(\xi - 2) \\ 2(5\eta - 1)(\xi - 1) \\ 5(2\eta - 1)(\xi - 2) \\ -5(2\eta - 1)(\xi - 1) \end{pmatrix},$$

and the bilinear interpolating polynomial is

$$p(\xi, \eta) = \sum_{i=1}^{4} f_i \, \tilde{S}_i(\xi, \eta).$$

If the function values at the mid-side nodes (as figure 4.2 shows) are known as well, we obtain with

$$\mathbf{b}^T(x, y) = (1, x, y, xy, x^2, y^2, x^2y, xy^2)$$

$$\mathbf{S}(x,y) = \tfrac{1}{4}\begin{pmatrix} -(y+1)(1-x)(1+x-y) \\ -(y+1)(1+x)(1-x-y) \\ -(1-x)(1-y)(1+x+y) \\ -(1+x)(1-y)(1-x+y) \\ 2(1-x)(1+x)(1+y) \\ 2(1+x)(1+y)(1-y) \\ 2(1+x)(1-x)(1-y) \\ 2(1+y)(1-x)(1-y) \end{pmatrix}$$

Exercise 4.10 (Quadratic Serendipity element)
Given 8 data points and corresponding function values. Calculate the quadratic polynomial of Serendipity-type, which interpolates the data.

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
ξ_i	*1.0*	*2.0*	*1.0*	*2.0*	*1.5*	*2.0*	*1.5*	*1.0*
η_i	*0.5*	*0.5*	*0.2*	*0.2*	*0.5*	*0.35*	*0.2*	*0.35*
f_i	*1.703*	*3.943*	*0.640*	*1.568*	*2.549*	*2.780*	*0.990*	*1.181*

We may also consider triangular elements, e.g. the standard triangle $(0,0)$, $(1,0)$, $(0,1)$, as shown in figure 4.3. If we know the function values at the vertices, we can uniquely determine a *linear* interpolator:

$$p(x,y) = a_0 + a_1 x + a_2 y$$

using the interpolation condition

$$p(x_i, y_i) = f_i \qquad i = 1,2,3.$$

This yields the linear system of equations

$$\begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix}$$

With $\mathbf{b}^T(x,y) = (1,x,y)$ we obtain

$$\mathbf{S}(x,y) = (1,x,y)\begin{pmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ -1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1-x-y \\ x \\ y \end{pmatrix}.$$

Thus, the linear interpolator is

$$p(x,y) = \sum_{i=1}^{3} f_i s_i(x,y) = f_1(1-x-y) + f_2 x + f_3 y$$

Following the same procedure, a quadratic interpolator

$$p(x, y) = a_0 + a_1 x + a_2 y + a_3 x^2 + a_4 xy + a_5 y^2$$

can be determined if in addition the function values at the three mid-sides are known, as is shown in figure 4.3.

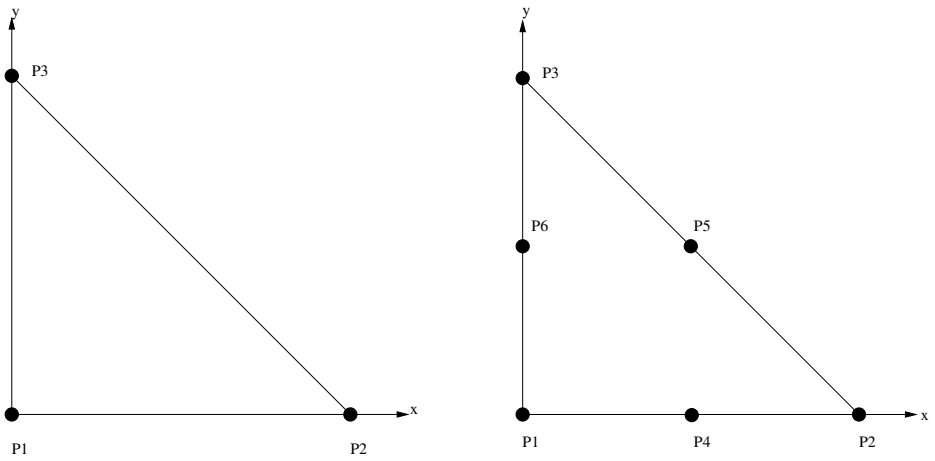


Figure 4.3: Linear triangular element (left) and quadratic triangular element (right).

With $\mathbf{b}^T(x, y) = (1, x, y, x^2, xy, y^2)$, we obtain

$$\begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & \frac{1}{2} & 0 & \frac{1}{4} & 0 & 0 \\ 1 & \frac{1}{2} & \frac{1}{2} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ 1 & 0 & \frac{1}{2} & 0 & 0 & \frac{1}{4} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{pmatrix}$$

and finally

$$\mathbf{S}(x, y) = \begin{pmatrix} (1 - x - y)(1 - 2x - 2y) \\ x(2x - 1) \\ y(2y - 1) \\ 4x(1 - x - y) \\ 4xy \\ 4y(1 - x - y) \end{pmatrix}.$$

If we have a triangle in general position as in figure 4.4, we perform simply a parameter

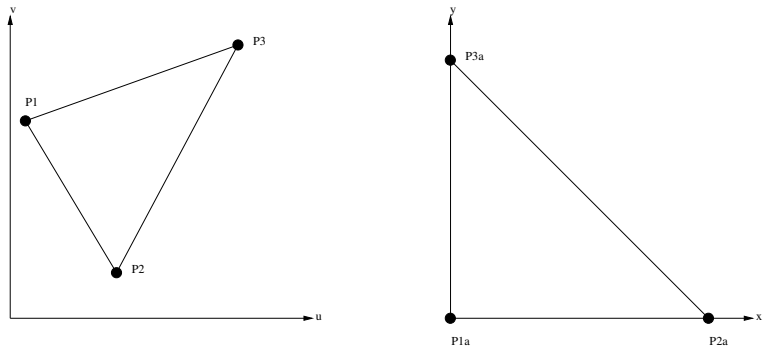


Figure 4.4: General triangle and transformed triangle.

transformation $(x, y) \to (u, v)$: if $P_i = (u_i, v_i)$ and $P_i' = (x_i, y_i)$, we have

$$\begin{cases} u &= u_1 + (u_2 - u_1)\, x + (u_3 - u_1)\, y \\ v &= v_1 + (v_2 - v_1)\, x + (v_3 - v_1)\, y \end{cases} \quad \Rightarrow$$

$$\begin{cases} x &= \left((v_1 - v_3)(u - u_1) + (u_3 - u_1)(v - v_1)\right)/N \\ y &= \left((v_2 - v_1)(u - u_1) + (u_1 - u_2)(v - v_1)\right)/N \\ N &:= u_2(v_1 - v_3) + u_1(v_3 - v_2) + u_3(v_2 - v_1) \end{cases}$$

Thus,

$$\mathbf{S}(x, y) = \mathbf{S}\left(x(u, v), y(u, v)\right) = \mathbf{S}^*(u, v) \tag{4.35}$$

Then

$$p^*(u, v) = \sum_i f_i\, S_i^*(u, v) = p(x, y) = \sum_i f_i\, S_i(x, y) \tag{4.36}$$

is the interpolant in (u, v)-coordinates.

In an analogous manner we can transform a general parallelogram onto the standard rectangle $[-1, 1] \times [-1, 1]$, as figure 4.5 shows.

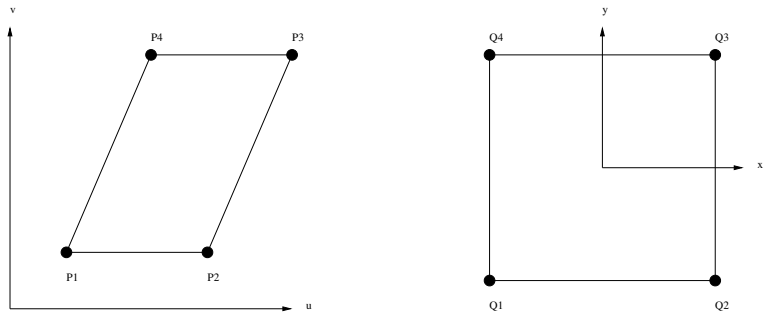


Figure 4.5: Parallelogram and standard rectangle.

4.2.3 Radial function interpolation

We mentioned already that the bivariate interpolation problem has in general no solution. Only for special arrangements of data points parallel to the coordinate axes of the Cartesian coordinate system we could find uniquely solvable interpolation problems. We also mentioned that uniqueness and existence of the bivariate interpolation problem for arbitrary distinct data points requires a different approach: instead of taking linear combinations of a set of basis functions that are independent of the data points, one takes a linear combination of translates of a single basis function that is radially symmetric about its centre. This approach is referred to as the radials basis function (RBF) method. This method is now one of the primary tools for interpolating multi-dimensional scattered data. Its simple form, and ability to accurately approximate an underlying function have made the method particularly popular.

Definition 4.11 (Radial function)
A function $\phi : \mathbb{R}^2 \to \mathbb{R}$ is called radial provided there exists a univariate function $\varphi : [0, \infty) \to \mathbb{R}$ such that

$$\Phi(\mathbf{x}) = \varphi(r), \quad r = \|\mathbf{x}\|, \tag{4.37}$$

and $\|\cdot\|$ is some norm on $\mathbb{R}^2$, usually the Euclidean norm. Hence,

$$\|\mathbf{x}_1\| = \|\mathbf{x}_2\| \Rightarrow \Phi(\mathbf{x}_1) = \Phi(\mathbf{x}_2). \tag{4.38}$$

The interpolation problem using radial basis functions (RBF's) can be formulated as follows:

Definition 4.12 (Basic RBF method)
Given a set of n distinct data points $\{\mathbf{x}_i : i = 1, \ldots, n\}$ and data values $\{f_i : i = 1, \ldots, n\}$, the basic RBF interpolant is given by

$$s(\mathbf{x}) = \sum_{j=1}^{n} a_j \, \Phi(\|\mathbf{x} - \mathbf{x}_j\|), \tag{4.39}$$

where $\Phi(r)$, $r \geq 0$ is some radial function. The coefficients a_j are determined from the interpolation conditions

$$s(\mathbf{x}_j) = f_j, \quad j = 1, \ldots, n, \tag{4.40}$$

which leads to the following symmetric linear system:

$$\mathbf{A}\,\mathbf{a} = \mathbf{f}. \tag{4.41}$$

The entries of the matrix $\mathbf{A}$ are given by

$$A_{ij} = \Phi(\|\mathbf{x}_i - \mathbf{x}_j\|). \tag{4.42}$$

Some common examples of the $\Phi(r)$ that lead to a uniquely solvable method (i.e., to a non-singular matrix $\mathbf{A}$) are:

1. **Gaussian**:
$$\Phi(r) = e^{-(\varepsilon r)^2}. \tag{4.43}$$

2. **Inverse Quadratic (IQ)**:
$$\Phi(r) = \frac{1}{1+(\varepsilon r)^2}. \tag{4.44}$$

3. **Inverse Multiquadric (IMQ)**
$$\Phi(r) = \frac{1}{\sqrt{1+(\varepsilon r)^2}}. \tag{4.45}$$

4. **Multiquadric (MQ)**:
$$\Phi(r) = \sqrt{1+(\varepsilon r)^2}. \tag{4.46}$$

5. **Linear**:
$$\Phi(r) = r. \tag{4.47}$$

In all cases, ε is a free parameter to be chosen appropriately. It controls the shape of the functions: as $\varepsilon \to 0$, radial functions become more flat. At this point, assume that it is some fixed non-zero real value. The cubic RBF and the very popular thin-plate spline RBF, defined by

6. **Cubic RBF**:
$$\Phi(r) = r^3. \tag{4.48}$$

7. **Thin-plate spline**:
$$\Phi(r) = r^2 \log r. \tag{4.49}$$

do not provide a regular matrix $\mathbf{A}$ unless some additional restrictions are met, which leads to the so-called *augmented RBF method*, which is discussed next. Before we define this method we need to make one more definition:

Definition 4.13
Let $\Pi_m(\mathbb{R}^2)$ be the space of all bivariate polynomials that have degree less than or equal to m. Furthermore, let M denote the dimension of $\Pi_m(\mathbb{R}^2)$, then
$$M = \frac{1}{2}(m+1)(m+2). \tag{4.50}$$

For instance, a basis of Π_1 comprises the polynomials 1, x, and y; this space has dimension $M = 3$; a basis of Π_2 comprises the polynomials 1, x, y, x^2, xy, and y^2; the dimension of the space is $M = 6$. In general, any function $f \in \Pi_m$ can be written as
$$f(x,y) = \sum_{0 \le i+j \le m} a_{ij} x^i y^j, \quad a_{ij} \in \mathbb{R}. \tag{4.51}$$

Now we are ready to define the *augmented RBF method*.

Definition 4.14 (Augmented RBF method)
Given a set of n distinct points $\{\mathbf{x}_i : i = 1, \ldots, n\}$ and data values $\{f_i : i = 1, \ldots, n\}$, the augmented RBF interpolant is given by

$$s(\mathbf{x}) = \sum_{j=1}^{n} \lambda_j \Phi(\|\mathbf{x} - \mathbf{x}_j\|) + \sum_{k=1}^{M} \gamma_k p_k(\mathbf{x}), \quad \mathbf{x} \in \mathbb{R}^2, \tag{4.52}$$

where $\{p_k(\mathbf{x}) : k = 1, \ldots, M\}$ is a basis of the space $\Pi_m(\mathbb{R}^2)$ and $\Phi(r)$, $r \geq 0$ is some radial function. To account for the additional polynomial terms, the following constraints are imposed:

$$\sum_{j=1}^{n} \lambda_j p_k(\mathbf{x}_j) = 0, \quad k = 1, \ldots, M. \tag{4.53}$$

The expansion coefficients λ_j and γ_k are then determined from the interpolation conditions and the constraints (4.53):

$$\begin{pmatrix} \mathbf{A} & \mathbf{P} \\ \mathbf{P}^T & \mathbf{0} \end{pmatrix} \cdot \begin{pmatrix} \boldsymbol{\lambda} \\ \boldsymbol{\gamma} \end{pmatrix} = \begin{pmatrix} \mathbf{f} \\ \mathbf{0} \end{pmatrix}. \tag{4.54}$$

$\mathbf{P}$ is the $n \times M$ matrix with entries $P_{ij} = p_j(\mathbf{x}_i)$ for $i = 1, \ldots, n$ and $j = 1, \ldots, M$.

It can be shown that the augmented RBF method is uniquely solvable for the cubic and thin-plate spline RBFs when $m = 1$ and the data points are such that the matrix $\mathbf{P}$ has $\text{rank}(\mathbf{P}) = M$. This is equivalent to saying that for a given basis $\{p_k(\mathbf{x}) : k = 1, \ldots, M\}$ for $\Pi_m(\mathbb{R}^2)$ the data points $\{\mathbf{x}_j : j = 1, \ldots, n\}$ must satisfy the condition

$$\sum_{k=1}^{M} \gamma_k \, p_k(\mathbf{x}_j) = 0 \Rightarrow \boldsymbol{\gamma} = \mathbf{0}, \tag{4.55}$$

for $j = 1, \ldots, n$. For $m = 1$, it is $M = 3$, and the RBF interpolant is augmented with a constant and linear bivariate polynomial, i.e., it is

$$p_1(\mathbf{x}) = 1, \;\; p_2(\mathbf{x}) = x, \;\; p_3(\mathbf{x}) = y, \tag{4.56}$$

and the RBF interpolant is given by

$$s(x, y) = \sum_{j=1}^{n} \lambda_j \Phi(\|\mathbf{x} - \mathbf{x}_j\|) + \gamma_0 + \gamma_1 \, x + \gamma_2 \, y. \tag{4.57}$$

Hence, we see that the augmented RBF method is much less restrictive than the basic RBF method on the functions $\Phi(r)$ that can be used; however, it is far more restrictive on the data points $\{\mathbf{x}_j : j = 1, \ldots, n\}$ that can be used. Remember that the only restriction on the data points for the basic method is that the points are distinct.

Let us make some remarks related to the effect of ε and n on the stability of the basic and/or augmented RBF method. For the infinitely smooth $\Phi(r)$ (i.e., for the Gaussian, the

inverse quadratic, the inverse multiquadric, and the multiquadric RBF), the accuracy and the stability depend on the number of data points n *and* the value of the shape parameter ε. For a fixed ε, as the number of data points increases, the RBF interpolant converges to the underlying (sufficiently smooth) function being interpolated at a spectral rate, i.e., $O(e^{-c/h})$, where c is a constant and h is a measure of the typical distance between data points. The Gaussian RBF exhibits even 'super-spectral' convergence, i.e., $O(e^{-c/h^2})$. In either case, the value of c in the estimates is effected by the value of ε. For a fixed number of data points, the accuracy of the RBF interpolant can often be significantly improved by decreasing the value of ε. However, decreasing ε or increasing the number n of data points has a severe effect on the stability of the linear system (4.41) and (4.54), respectively. For a fixed ε, the condition number of the matrix in the linear systems grows exponentially as the number of data points is increased. For a fixed number of data points, similar growth occurs as $\varepsilon \to 0$.

A very important feature of the RBF method is that its complexity does not increase as dimension of the interpolation increases. Their simple form makes implementing the methods extremely easy compared to, for example, a bicubic spline method. However, main computational challenges are that i) the matrix for determining the interpolation coefficients, (4.41) and (4.54) is dense, which makes the computational cost of the methods high; ii) the matrix is ill-conditioned when the number of data points is large; iii) for the infinitely smooth RBFs and a fixed number of data points, the matrix is also ill-conditioned when ε is small. There are techniques available, which address these problems. However, they are beyond the scope of this course.

4.2.4 Bicubic spline interpolation

In univariate interpolation, the cubic spline offers a good interpolant also for a large number of data points. The idea of a cubic spline can also be generalized to the bivariate case if the data are given on a lattice. The corresponding spline is called a *bicubic spline*, denoted $S(x, y)$. A bicubic spline on a lattice with $n+1$ points in x and $m+1$ points in y is defined by the following properties:

(1) S fulfills the interpolation property

$$S(x_i, y_j) = f_{ij}, \qquad i = 0, 1, \ldots, n, \quad j = 0, 1, \ldots, m.$$

(2) $S \in C^1(D)$, $\dfrac{\partial^2 S}{\partial x \partial y}$ continuous on D.

(3) S is a bicubic polynomial within each rectangle D_{ij}:

$$D_{ij} := \{(x, y) : x_i \le x \le x_{i+1},\ i = 0, \ldots, n-1; \quad y_j \le y \le y_{j+1},\ j = 0, \ldots, m-1\}.$$

(4) S fulfills certain conditions on the boundary of the domain $\bar{D}$ which still have to be defined.

Because of (3), the bicubic spline function $S(x,y)$ has on $(x,y) \in D_{ij}$ the representation

$$S(x,y)_{|D_{ij}} =: S_{ij} = \sum_{k=0}^{3}\sum_{l=0}^{3} a_{ijkl}\,(x-x_i)^k (y-y_j)^l,$$
$$(x,y) \in D_{ij},\ \ i = 0,\ldots,n-1,\ \ j = 0,\ldots,m-1 \quad (4.58)$$

The $16m \cdot n$ coefficients a_{ijkl} have to be determined such that the conditions (1) and (2) are fulfilled. To determine them uniquely we must formulate certain boundary conditions like in the one-dimensional case. One possibility is to prescribe the following partial derivatives of S:

$$\begin{aligned}
\frac{\partial S}{\partial x}(x_i,y_j) &=: p_{ij} = a_{ij10}, && i = 0,n,\ \ j = 0,1,\ldots,m \\
\frac{\partial S}{\partial y}(x_i,y_j) &=: q_{ij} = a_{ij01}, && i = 0,1,\ldots,n,\ \ j = 0,m \\
\frac{\partial^2 S}{\partial x \partial y}(x_i,y_j) &=: r_{ij} = a_{ij11}, && i = 0,n,\ \ j = 0,m
\end{aligned} \quad (4.59)$$

They can be calculated approximately using one-dimensional splines or other interpolation methods. For example one-dimensional splines through three points each and their derivatives can be used to approximate the boundary conditions. The following algorithm assumes that the boundary values (4.59) are given. It determines all coefficients a_{ijkl} in 9 steps:

Step 1: Calculation of $a_{ij10} = p_{ij}$ for $i = 1,\ldots,n-1$ and $j = 0,\ldots,m$:

$$\frac{1}{h_{i-1}}a_{i-1,j10} + 2\left(\frac{1}{h_{i-1}} + \frac{1}{h_i}\right)a_{ij10} + \frac{1}{h_i}a_{i+1,j10} =$$
$$= \frac{3}{h_{i-1}^2}(a_{ij00} - a_{i-1,j00}) + \frac{3}{h_i^2}(a_{i+1,j00} - a_{ij00}),$$
$$i = 1,\ldots,n-1, \quad j = 0,\ldots,m \quad (4.60)$$

where $h_\kappa = x_{\kappa+1} - x_\kappa,\ \ \kappa = 0,1,\ldots,n-1$. These are $m+1$ linear systems, each with $n-1$ equations for $n+1$ unknowns. By prescribing the $2(m+1)$ quantities $p_{ij} = a_{ij10},\ \ i = 0,n,\ \ j = 0,\ldots,m$, these systems are uniquely solvable.

Step 2: Calculation of $a_{ij01} = q_{ij}$ for $i = 0,\ldots,n$ and $j = 1,\ldots,m-1$ with

$$\frac{1}{h_{j-1}}a_{i,j-1,01} + 2\left(\frac{1}{h_{j-1}} + \frac{1}{h_j}\right)a_{ij01} + \frac{1}{h_j}a_{i,j+1,01} =$$
$$= \frac{3}{h_{j-1}^2}(a_{ij00} - a_{i,j-1,00}) + \frac{3}{h_j^2}(a_{i,j+1,00} - a_{ij00}),$$
$$i = 0,\ldots,n, \quad j = 1,\ldots,m-1 \quad (4.61)$$

where $h_\kappa = y_{\kappa+1} - y_\kappa,\ \ \kappa = 0,1,\ldots,m-1$. With the prescribed $2(n+1)$ boundary values $qij = a_{ij01},\ \ i = 0,\ldots,n,\ \ j = 0,m$, these systems are uniquely solvable.

Step 3: Calculation of $a_{ij11} = r_{ij}$ for $i = 1, 2, \ldots, n-1$ and $j = 0, m$ from the system

$$\frac{1}{h_{i-1}} a_{i-1,j11} + 2\left(\frac{1}{h_{i-1}} + \frac{1}{h_i}\right) a_{ij11} + \frac{1}{h_i} a_{i+1,j11} = \\ = \frac{3}{h_{i-1}^2}(a_{ij01} - a_{i-1,j01}) + \frac{3}{h_i^2}(a_{i+1,j01} - a_{ij01}), \\ i = 1, 2, \ldots, n-1, \quad j = 0, m \quad (4.62)$$

where $h_\kappa = x_{\kappa+1} - x_\kappa$. The values of the four corner points a_{0011}, a_{n011}, a_{0m11}, and a_{nm11} are given.

Step 4: Calculation of the derivatives $r_{ij} = a_{ij11}$ for $i = 0, 1, \ldots, n$ and $j = 1, 2, \ldots, m-1$ with

$$\frac{1}{h_{j-1}} a_{i,j-1,11} + 2\left(\frac{1}{h_{j-1}} + \frac{1}{h_j}\right) a_{ij11} + \frac{1}{h_j} a_{i,j+1,11} = \\ = \frac{3}{h_{j-1}^2}(a_{ij10} - a_{i,j-1,10}) + \frac{3}{h_j^2}(a_{i,j+1,10} - a_{ij10}), \\ i = 0, 1, \ldots, n, \quad j = 1, 2, \ldots, m-1 \quad (4.63)$$

where $h_\kappa = y_{\kappa+1} - y_\kappa$. The required values of the boundary points $a_{ij11},\ i = 1, \ldots, n-1,\ j = 0, m$, have been calculated in step 3. The corner points $a_{ij11},\ i = 0, n,\ j = 0, m$ are prescribed.

Step 5: Determination of the matrix $G(x_i)^{-1}$. Because

$$G(x_i) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & h_i & h_i^2 & h_i^3 \\ 0 & 1 & 2h_i & 3h_i^2 \end{pmatrix}$$

with $\det G(x_i) = h_i^4 \neq 0$, $h_i = x_{i+1} - x_i,\ i = 0, \ldots, n-1$, the inverse $G(x_i)^{-1}$ exists and can explicitly be calculated:

$$G(x_i)^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -\frac{3}{h_i^2} & -\frac{2}{h_i} & \frac{3}{h_i^2} & -\frac{1}{h_i} \\ \frac{2}{h_i^3} & \frac{1}{h_i^2} & -\frac{2}{h_i^3} & \frac{1}{h_i^2} \end{pmatrix}$$

Step 6: Determination of the matrix $\left(G(y_j)^T\right)^{-1}$. Because

$$G(y_j) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & h_j & h_j^2 & h_j^3 \\ 0 & 1 & 2h_j & 3h_j^2 \end{pmatrix}$$

with $\det G(y_j) = h_j^4 \neq 0$, $h_j = x_{j+1} - x_j$, $j = 0, \ldots, m-1$, the inverse $\left(G(y_j)^T\right)^{-1}$ exists:

$$\left(G(y_j)^T\right)^{-1} = \begin{pmatrix} 1 & 0 & -\frac{3}{h_j^2} & \frac{2}{h_j^3} \\ 0 & 1 & -\frac{2}{h_j} & \frac{1}{h_j^2} \\ 0 & 0 & \frac{3}{h_j^2} & -\frac{2}{h_j^3} \\ 0 & 0 & -\frac{1}{j} & \frac{1}{h_j^2} \end{pmatrix}$$

Step 7: Determination of the matrices $M_{ij}, i = 0, \ldots, n-1, j = 0, \ldots, m-1$ as

$$M_{ij} = \begin{pmatrix} a_{ij00} & a_{ij01} & a_{i,j+1,00} & a_{i,j+1,01} \\ a_{ij10} & a_{ij11} & a_{i,j+1,10} & a_{i,j+1,11} \\ a_{i+1,j00} & a_{i+1,j01} & a_{i+1,j+1,00} & a_{i+1,j+1,01} \\ a_{i+1,j10} & a_{i+1,j11} & a_{i+1,j+1,10} & a_{i+1,j+1,11} \end{pmatrix}$$

Step 8: Calculation of the coefficient matrices A_{ij}:

$$A_{ij} = \{a_{ijkl}\}_{k=0,\,l=0}^{3\quad 3} = G(x_i)^{-1}\, M_{ij}\, \left(G(y_j)^T\right)^{-1}$$

Step 9: Formation of the bicubic spline function $S_{ij}(x, y)$ for all rectangles D_{ij} with equation (4.58).

The boundary conditions can be determined for instance by fitting one-dimensional splines through three points each: Through the points $(x_i, f_{ij}), i = 0, 1, 2, n-2, n-1, n$ we fit for $j = 0, \ldots, m$ one-dimensional natural cubic splines and calculate the derivatives; they provide the p_{ij} at the boundary. Through the points $(y_j, f_{ij}), j = 0, 1, 2, m-2, m-1, m$ for $i = 0, \ldots, n$ we fit the same type of spline functions and calculate the derivatives; they provide the q_{ij} at the boundary. To calculate the $r_{ij} = a_{ij11}$ for $i = 0, n$, $j = 0, m$ we fit one-dimensional natural splines through (x_i, q_{ij}) for $i = 0, 1, 2, n-2, n-1, n$ and $j = 0, m$ and determine the derivatives.

Chapter 5

Least-squares Regression

In the chapters on interpolation we discussed the approximation of a given function by another function that exactly reproduced the given data values. We considered both univariate functions (i.e., functions which depend on one variable) and bivariate functions (i.e. functions which depend on two variables). The function that interpolates given data has been called "interpolant". By an "approximant" (or "approximating function") we mean a function which *approximates* the given data values as well as possible (in some sense we need to agree upon later) but does not necessarily *reproduce* the given data values exactly. That is, the graph of the approximant will not in general go through the data points, but will be close to them — this is called *regression*.

A justification for approximation rather than interpolation is the case of experimental or statistical data. The data from experiments are normally subject to errors. The interpolant would exactly reproduce the errors. The approximant, however, allows to adjust for these errors such that a smooth function results. In the presence of noise it would even be foolish and, indeed, inherently dangerous, to attempt to determine an interpolant, because it is very likely that the interpolant oscillates violently about the curve or surface which represents the true function. Another justification is that there may be so many data points that efficiency considerations force us to approximate from a space spanned by fewer basis functions that data points.

In regression, as for interpolation, we consider the case where a function has to be recovered from partial information, e.g. when we only know (possibly noisy) values of the function at a set of points.

5.1 Least-squares basis functions

The most commonly used classes of approximating functions are functions that can be written as linear combinations of basis functions $\varphi_i, i = 1, \ldots, M$, i.e., approximants of type:

$$\phi(\mathbf{x}) = \sum_{i=1}^{M} c_i \, \varphi_i(\mathbf{x}) = \mathbf{c}^T \boldsymbol{\varphi}(\mathbf{x}), \tag{5.1}$$

with

$$\mathbf{c} := \begin{pmatrix} c_1 & c_2 & \ldots & c_M \end{pmatrix}^T, \tag{5.2}$$

and

$$\boldsymbol{\varphi}(\mathbf{x}) := \begin{pmatrix} \varphi_1(\mathbf{x}) & \varphi_2(\mathbf{x}) & \ldots & \varphi_M(\mathbf{x}) \end{pmatrix}^T,$$

exactly the same as for interpolation. In the following we will restrict to this type of approximants. Moreover, we will assume that the function we want to approximate is at least continuous.

Where least-squares differs from interpolation is that the number of basis functions M is in general *less* than the number of data points N,

$$M \leq N,$$

which means we will have more constraints than free variables, and therefore we will not be able to satisfy all constraints exactly.

Commonly used classes of (univariate) basis functions are:

- algebraic polynomials:
$$\varphi_1 = 1, \; \varphi_2 = x, \; \varphi_3 = x^2, \ldots$$
- trigonometric polynomials:
$$\varphi_1 = 1, \; \varphi_2 = \cos x, \; \varphi_3 = \sin x, \; \varphi_4 = \cos 2x, \; \varphi_5 = \sin 2x, \ldots$$
- exponential functions:
$$\varphi_1 = 1, \; \varphi_2 = \mathrm{e}^{\alpha_1 x}, \; \varphi_3 = \mathrm{e}^{\alpha_2 x}, \ldots$$
- rational functions:
$$\varphi_1 = 1, \varphi_2 = \frac{1}{(x-\alpha_1)^{p_1}}, \varphi_3 = \frac{1}{(x-\alpha_2)^{p_2}}, \ldots, \quad p_i \in \mathbb{N}$$

.

For bivariate regression, radial functions are also popular; they have been introduced in the previous chapter.

5.2 Least-squares approximation - Example

In practical applications we are interested in getting "good" approximations, i.e. the approximant should not deviate "much" from the given data. However, what is the precise meaning of "good" and "much"? In other words, how do we measure the quality of the approximation or, equivalently, how do we measure the error in the approximation?

In order to answer this question let us first discuss a simple example. Suppose we are given the data in table 5.1. We have 3 data points. Let us look for a straight line, which best fits in some sense the given data. The equation of the straight line is

i	1	2	3
x_i	0	1	2
f_i	4.5	3.0	2.0

Table 5.1: Example data set.

$$\phi(x) = a_0 + a_1\, x,$$

with so far unknown coefficients a_0 and a_1. Intuitively, we would like the straight line to be as close as possible to the function $f(x)$ that generates the data. A measure of the 'closeness' between $\phi(x)$ and $f(x)$ could be based on the difference of the function values of f and ϕ at the given data points, i.e., on the quantities

$$r_i := f(x_i) - \phi(x_i) = f_i - (a_0 + a_1\, x_i), \quad i = 1, 2, 3.$$

The residuals are shown graphically in Figure 5.1 as the vertical red bars capturing the distance between samples of f at the nodes, and the approximation ϕ. The r_i's are called the *residuals*. The least-squares method finds among all possible coefficients a_0 and a_1 the pair that minimizes the *square sum of the residuals*,

$$\sum_{i=1}^{3} r_i^2,$$

i.e., that makes $\sum_{i=1}^{3} r_i^2$ as small as possible. This minimization principle is sometimes called the *(discrete) least-squares principle*. Of course, other choices also exists: we could, e.g., minimize the absolute sum of the residuals,

$$\sum_{i=1}^{N} |r_i|,$$

or we could minimize the largest absolute residual:

$$\max_i(|r_i|).$$

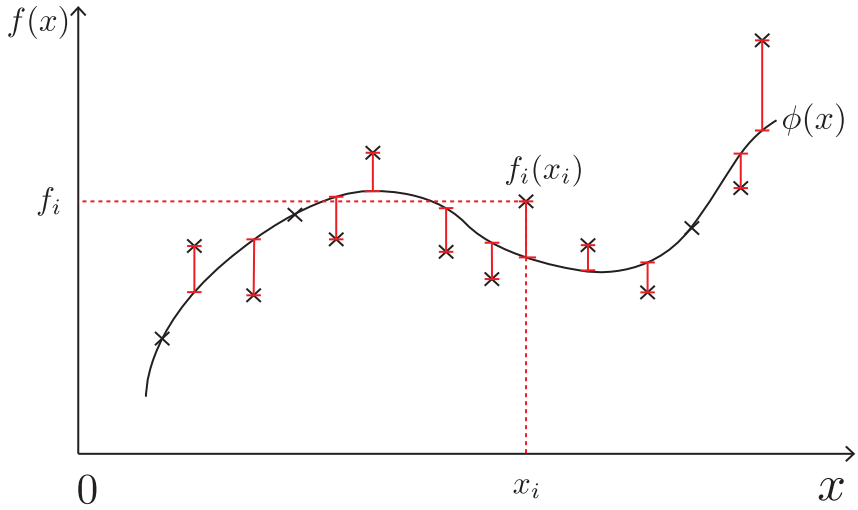


Figure 5.1: Example of regression of samples of a function $f(x)$ with the curve $\phi(x)$. Red bars show the *residuals*, which are minimized in order to solve for ϕ.

The advantage of the least-squares principle is that it is the only one among the three principles, which yields a *linear* system of equations for the unknown coefficients a_0 and a_1. That is the main reason why this principle has become so popular. Let us determine the coefficients a_0 and a_1 according to the least-squares principle. We define a function Φ, which is equal to the square sum of the residuals:

$$\Phi(a_0, a_1) := \sum_{i=1}^{3} r_i^2 = \sum_{i=1}^{3} (f_i - a_0 - a_1\, x_i)^2.$$

We have written $\Phi(a_0, a_1)$ to emphasize that the square sum of the residuals is seen as a function of the unknown coefficients a_0 and a_1. Hence, minimizing the square sum of the residuals means to look for the minimum of the function $\Phi(a_0, a_1)$. A necessary condition for Φ to attain a minimum is that the first derivatives with respect to a_0 and a_1 are equal to zero:

$$\frac{\partial \Phi}{\partial a_0} = -2 \sum_{i=1}^{3} (f_i - a_0 - a_1\, x_i) = 0,$$
$$\frac{\partial \Phi}{\partial a_1} = -2 \sum_{i=1}^{3} (f_i - a_0 - a_1\, x_i) x_i = 0.$$

This is a system of 2 equations for the 2 unknowns a_0 and a_1. It is called *normal equations*. The solution of the normal equations is sometimes called the *least-squares solution*, denoted $\hat{a}_0$ and $\hat{a}_1$. This is mostly done to emphasize that other solutions are possible, as well, as

outlined before. Adopting this notation for the least-squares solution, the normal equations are written as

$$\sum_{i=1}^{3}(\hat{a}_0 + \hat{a}_1\, x_i) = \sum_{i=1}^{3} f_i$$
$$\sum_{i=1}^{3}(\hat{a}_0\, x_i + \hat{a}_1\, x_i^2) = \sum_{i=1}^{3} f_i\, x_i,$$

and in matrix-vector notation

$$\begin{pmatrix} \sum_{i=1}^{3} 1 & \sum_{i=1}^{3} x_i \\ \sum_{i=1}^{3} x_i & \sum_{i=1}^{3} x_i^2 \end{pmatrix} \begin{pmatrix} \hat{a}_0 \\ \hat{a}_1 \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{3} f_i \\ \sum_{i=1}^{3} f_i\, x_i \end{pmatrix}. \tag{5.3}$$

Numerically, we find

$$\begin{pmatrix} 3 & 3 \\ 3 & 5 \end{pmatrix} \begin{pmatrix} \hat{a}_0 \\ \hat{a}_1 \end{pmatrix} = \begin{pmatrix} 9.5 \\ 7 \end{pmatrix}.$$

The solution is (to 4 decimal places) $\hat{a}_0 = 4.4167$ and $\hat{a}_1 = -1.2500$. Hence, the least-squares approximation of the given data by a straight line is

$$\hat{\phi}(x) = 4.4167 - 1.2500\, x.$$

The least-squares residuals are computed as $\hat{r}_i = f_i - \hat{\phi}_i$, which gives $\hat{r}_1 = 0.0833$, $\hat{r}_2 = -0.1667$, and $\hat{r}_3 = 0.0833$. The square sum of the (least-squares) residuals is $\hat{\Phi} = \sum_{i=1}^{3} \hat{r}_i^2 = 0.0417$. Notice that the numerical values of the coefficients found before is the choice which yields the smallest possible square sum of the residuals. No other pair of coefficients a_0, a_1 yields a smaller Φ (try it yourself!).

In order to give the normal equations more 'structure', we can define the following *scalar product* of two functions given on a set of N points x_i:

$$\langle f, g\rangle := \sum_{i=1}^{N} f(x_i) g(x_i).$$

Obviously, $\langle f, g\rangle = \langle g, f\rangle$, i.e., the scalar product is symmetric. Using this scalar product, the normal equations (5.3) can be written as (try it yourself!):

$$\begin{pmatrix} < 1, 1 > & < 1, x > \\ < 1, x > & < x, x > \end{pmatrix} \begin{pmatrix} \hat{a}_0 \\ \hat{a}_1 \end{pmatrix} = \begin{pmatrix} < f, 1 > \\ < f, x > \end{pmatrix}.$$

5.3 Least-squares approximation - The general case

Let us now generalize this approach to an arbitrary number N of given data and an approximation function of type

$$\phi(\mathbf{x}) = \sum_{i=1}^{M} a_i\, \varphi_i(\mathbf{x}), \quad M \leq N,$$

for given basis functions $\{\varphi_i(\mathbf{x})\}$. Note that we allow for both univariate and bivariate data. In the univariate case, the location of a data point is uniquely described by 1 variable, denoted e.g., x; in the bivariate case, we need 2 variables to uniquely describe the location of a data point, e.g., the Cartesian coordinates (x, y). Please also notice that the number of basis functions, M, must not exceed the number of data points, N. In least-squares, usually $M \ll N$.
Then, the residuals are (cf. the example above),

$$r(\mathbf{x}_i) = f(\mathbf{x}_i) - \phi(\mathbf{x}_i), \quad i = 1, \ldots, N,$$

or, simply, $r_i = f_i - \phi_i$, and the least-squares principle is

$$\Phi(\mathbf{a}) = \sum_{i=1}^{N} r(\mathbf{x}_i)^2 = \sum_{i=1}^{N} r_i^2 = \langle r, r \rangle \stackrel{!}{=} \text{ minimum}, \tag{5.4}$$

where the vector $\mathbf{a}$ is defined as $\mathbf{a} := (a_1, \ldots, a_M)^T$. The advantage of the use of a *scalar product* becomes clear now, because the normal equations, which are the solution of the minimization problem (5.4), are

$$\begin{pmatrix} \langle \varphi_1, \varphi_1 \rangle & \cdots & \langle \varphi_1, \varphi_M \rangle \\ \vdots & & \vdots \\ \langle \varphi_M, \varphi_1 \rangle & \cdots & \langle \varphi_M, \varphi_M \rangle \end{pmatrix} \begin{pmatrix} \hat{a}_1 \\ \vdots \\ \hat{a}_M \end{pmatrix} = \begin{pmatrix} \langle f, \varphi_1 \rangle \\ \vdots \\ \langle f, \varphi_M \rangle \end{pmatrix}, \tag{5.5}$$

with

$$\langle \varphi_j, \varphi_k \rangle = \sum_{i=1}^{N} \varphi_j(\mathbf{x}_i) \varphi_k(\mathbf{x}_i). \tag{5.6}$$

Note that the normal equations are symmetric, because

$$\langle \varphi_k, \varphi_j \rangle = \langle \varphi_j, \varphi_k \rangle.$$

The solution of the normal equations yields the least-squares estimate of the coefficients $\mathbf{a}$, denoted $\hat{\mathbf{a}}$, and the discrete least-squares approximation is the function

$$\hat{\phi}(\mathbf{x}) = \sum_{i=1}^{M} \hat{a}_i \, \varphi_i(\mathbf{x}).$$

The smallness of the square sum of the residuals,

$$\langle \hat{r}, \hat{r} \rangle = \langle f - \hat{\phi}, f - \hat{\phi} \rangle$$

can be used as a criterion for the efficiency of the approximation. Alternatively, the so-called *root-mean-square (RMS) error* in the approximation is also used as a measure of the fit of the function $\hat{\phi}(\mathbf{x})$ to the given data. It is defined by

$$\sigma_{RMS} := \sqrt{\frac{\langle \hat{r}, \hat{r} \rangle}{N}},$$

where $\hat{r} = f - \hat{\phi}$, and

$$\langle \hat{r}, \hat{r} \rangle = \sum_{i=1}^{N} \Big(f(\mathbf{x}_i) - \hat{\phi}(\mathbf{x}_i) \Big)^2.$$

When *univariate algebraic polynomials* are used as basis functions it can be shown that $N \geq M$ together with the linear independence of the basis functions guarantee the *unique solvability* of the normal equations. For other types of univariate basis functions this is not guaranteed. For $N < M$ the uniqueness gets lost. For $N = M$ we have *interpolation* and $\hat{\phi}(x_i) = f(x_i)$ for $i = 1, \ldots, N$.

For the bivariate case using radial functions as basis functions we have the additional problem that we have less basis functions (namely M) than we have data sites (namely N). Hence, we cannot place below every data point a basis function. Therefore, we need to have a strategy of where to locate the radial functions. This already indicates that the question whether the normal equations can be solved in the bivariate case with radial functions is non-trivial. In fact, we can only guarantee unique solvability of the normal equations for certain radial functions (i.e., multiquadrics, Gaussian) and severe restrictions to the location of the centres of the radial functions (they must be sufficiently well distributed over D in some sense) and to the location of the data sites (they must be fairly evenly clustered about the centres of the radial functions with the diameter of the clusters being relatively small compared to the separation distance of the data sites). Least-squares approximation with radial basis functions is not subject of this course.

The method of (discrete) least squares has been developed by Gauss in 1794 for smoothing data in connection with geodetic and astronomical problems.

Example 5.1
Given 5 function values $f(x_i)$, $i = 1, \ldots, 5$, of the function $f(x) = (1 + x^2)^{-1}$ (see the table below). We look for the discrete least-squares approximation $\hat{\phi}$ among all quadratic polynomials ϕ.

Step 1: *the choice of the basis functions is prescribed by the task description:*

$$\varphi_1 = 1,\ \varphi_2 = x,\ \varphi_3 = x^2 \Rightarrow \phi(x) = \sum_{i=1}^{3} c_i \varphi_i(x).$$

Step 2: *because no other information is available, we choose $w_i = 1$, $i = 1, \ldots, 5$.*

Step 3: *the given values $f(x_i)$, $i = 1, \ldots, 5$*

i	1	2	3	4	5
x_i	-1	$-\frac{1}{2}$	0	$\frac{1}{2}$	1
$f(x_i)$	0.5	0.8	1	0.8	0.5

lead to the normal equations

$$\begin{pmatrix} 5 & 0 & 2.5 \\ 0 & 2.5 & 0 \\ 2.5 & 0 & 2.125 \end{pmatrix} \begin{pmatrix} \hat{c}_1 \\ \hat{c}_2 \\ \hat{c}_3 \end{pmatrix} = \begin{pmatrix} 3.6 \\ 0 \\ 1.4 \end{pmatrix}$$

with the solution (to 5 decimal places) $\hat{c}_1 = 0.94857$, $\hat{c}_2 = 0.00000$, $\hat{c}_3 = -0.45714$, *yielding* $\hat{\phi}(x) = 0.94857 - 0.45714\,x^2, \quad x \in [-1, 1]$. *Under all quadratic polynomials,* $\hat{\phi}(x)$ *is the best approximation of* f *in the discrete least squares sense. For* $x = 0.8$, *we obtain for instance* $\hat{\phi}(0.8) = 0.65600$. *The absolute error at* $x = 0.8$ *is* $|f(0.8) - \hat{\phi}(0.8)| = 4.6 \cdot 10^{-2}$. *The results are plotted in figure 5.2.*

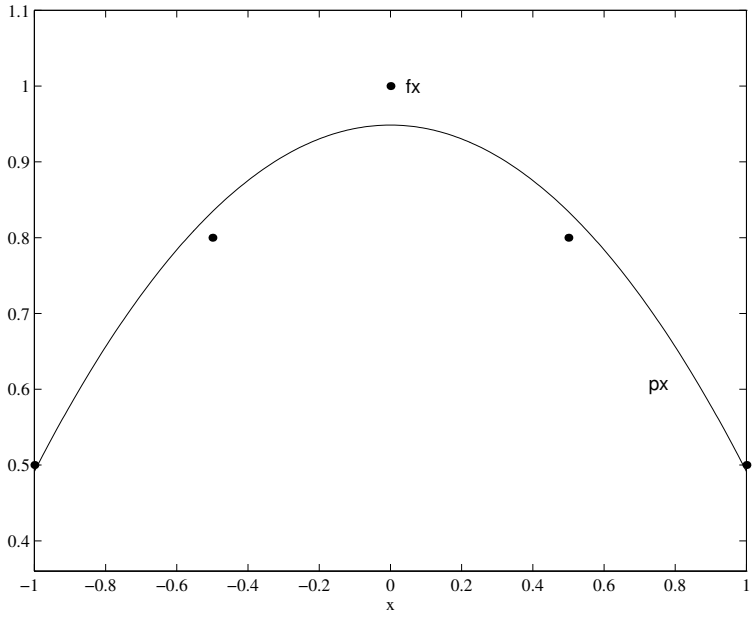


Figure 5.2: The 5 data points given by $f(x) = (1+x^2)^{-1}$ (dots) and the best approximation ϕ (solid line) of all quadratic polynomials in the discrete least squares sense.

Exercise 5.2

Given points $(x_i, f(x_i))$, $i = 1, \ldots, 4$.

i	1	2	3	4
x_i	0.02	0.10	0.50	1.00
$f(x_i)$	50	10	1	0

We look for the best approximation $\hat{\phi}$ *in the discrete least squares sense of all functions* $\phi(x) = \sum c_k \varphi_k(x)$ *and the following basis functions:*

1. $\varphi_1 = 1$, $\varphi_2 = x$
2. $\varphi_1 = 1$, $\varphi_2 = x$, $\varphi_3 = x^2$
3. $\varphi_1 = 1$, $\varphi_2 = x$, $\varphi_3 = x^2$, $\varphi_4 = x^3$
4. $\varphi_1 = 1$, $\varphi_2 = 1/x$

Give a graphical representation of the four different $\hat{\phi}$. What choice of the basis functions yields the "best" result?

5.4 Weighted least-squares

We can slightly generalize the least-squares method by assigning to each data value f_i a so-called *weight* $w_i > 0$. This may be justified, for instance, if the accuracy of the data values vary, i.e., if one data point, say, the one with index i, is more accurate than another one, say, with index j. If this is true, then it is natural to expect that $|f_i - \phi_i|$ is smaller than $|f_j - \phi_j|$. To achieve this, we have to assign the data point with index i a larger weight than the data point with index j. The corresponding method is called *weighted least-squares method*. It can be shown that the normal equations associated with the weighted least-squares method are formally identical to the normal equations associated with the classical least-squares method (which uses unit weights $w_i = 1$) if we slightly redefine the scalar product: instead of the definition (5.6), we use

$$\langle \varphi_j, \varphi_k \rangle := \sum_{i=1}^{N} w_i \, \varphi_j(\mathbf{x}_i) \varphi_k(\mathbf{x}_i). \tag{5.7}$$

The *weighted least-squares principle* is

$$\Phi(\mathbf{a}) = \langle r, r \rangle = \sum_{i=1}^{N} w_i (f_i - \phi_i)^2 \stackrel{!}{=} \text{ minimum}, \tag{5.8}$$

and the weighted least-squares solution is given by Eq. (5.5) when using the definition of the scalar product, Eq. (5.7). Note that according to Eq. (5.8), the weighted least-squares method does not minimize the square sum of the residuals, but the weighted square sum of the residuals. The RMS error in the weighted least-squares approximation is

$$\sigma_{RMS} = \sqrt{\frac{\langle \hat{r}, \hat{r} \rangle}{N}} = \sqrt{\frac{1}{N} \sum_{i=1}^{N} w_i \hat{r}_i^2}.$$

Chapter 6

Numerical Differentiation

6.1 Introduction

Numerical differentiation is the computation of derivatives of a function using a computer. The function may be given analytically or on a discrete set of points $x_0, \ldots, x_n$, the nodes, similarly to interpolation. The derivatives may be needed at arbitrary points or at the nodes $x_0, \ldots, x_n$.

The basic idea of numerical differentiation of a function $f(x)$ is to first interpolate $f(x)$ at the $n+1$ nodes $x_0, \ldots, x_n$ and then to use the analytical derivatives of the interpolating polynomial $\phi(x)$ as an approximation to the derivatives of the function $f(x)$.

If the interpolation error $E(f) = f(x) - \phi(x)$ is small it may be hoped that the result of differentiation, i.e., $\phi^{(k)}(x)$ will also satisfactorily approximate the corresponding derivative $f^{(k)}(x)$. However, if we visualize an interpolating polynomial $\phi(x)$ we often observe that it oscillates about the function $f(x)$, i.e., we may anticipate the fact that even though the deviation between the interpolating polynomial $\phi(x)$ and the function $f(x)$ (i.e., the interpolation error $E(x)$) be small throughout an interval, still the slopes of the two functions may differ significantly. Furthermore, roundoff errors or noise in the given data of alternating sign in consecutive nodes could effect the calculation of the derivative strongly if those nodes are closely spaced.

There are different procedures for deriving numerical difference formulae. In this course, we will discuss three of them: (i) using Taylor series, (ii) Richardson extrapolation, and (iii) using interpolating polynomials.

6.2 Numerical differentiation using Taylor series

Taylor series can be used to derive difference formulae for equidistant nodes if the derivative at a node is looked for. Suppose $f \in C^k([a,b])$ for some $k \geq 1$. For a given step size h, we

consider the Taylor series

$$f(x+h) = f(x) + f'(x)\,h + \frac{f''(x)}{2}\,h^2 + \ldots + \frac{f^{(k-1)}(x)}{(k-1)!}\,h^{k-1} + R_k(\xi), \tag{6.1}$$

where the Lagrange form of the remainder is

$$R_k(\xi) = \frac{f^{(k)}(\xi)}{k!}\,h^k, \quad x < \xi < x+h. \tag{6.2}$$

We want to determine an approximation of the first derivative $f'(x)$ for a fixed $x \in (a,b)$. If we choose $k = 2$ then

$$f(x+h) = f(x) + h\,f'(x) + \frac{1}{2}h^2\,f''(\xi), \quad x < \xi < x+h, \tag{6.3}$$

which can be rearranged to give

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{1}{2}\,h\,f''(\xi), \quad x < \xi < x+h. \tag{6.4}$$

Suppose $f(x)$ is given at a set of equidistant nodes $x_0 < x_1 \ldots < x_i < \ldots$ with distance h. Using the equation just derived we find for $f'(x_i)$

$$f'(x_i) = \frac{f_{i+1} - f_i}{h} + O(h). \tag{6.5}$$

This formula is referred to as the *forward difference formula*[1]. It has simple geometrical interpretation: we are trying to find the gradient of a curve by looking at the gradient of a short chord to that curve (cf. Fig 6.1).

Example 6.1 (Application of the forward difference formula)
Given the function $f(x) = x^x$ for $x > 0$. Approximate $f'(1)$ using the forward difference formula trying some different values of h (work with 10 significant digits): the solution is given in table 6.1.

We observe that as h is taken smaller and smaller, the forward difference result appears to converge to the correct value $f'(1) = 1$. We also observe that each time we divide h by a factor of 10, then the error decreases (roughly) by a factor of 10. Thus we might say the forward difference formula has error roughly proportional to h. This is exactly what the term $O(h)$ in Eq. (6.5) tells us; the error of the forward difference formula is on the order of $O(h)$, i.e., it scales proportional to h for sufficiently small h. Difference formulae, which

[1] Remember from real analysis that the derivative is defined as the limit

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}.$$

Compare this with the forward difference formula.

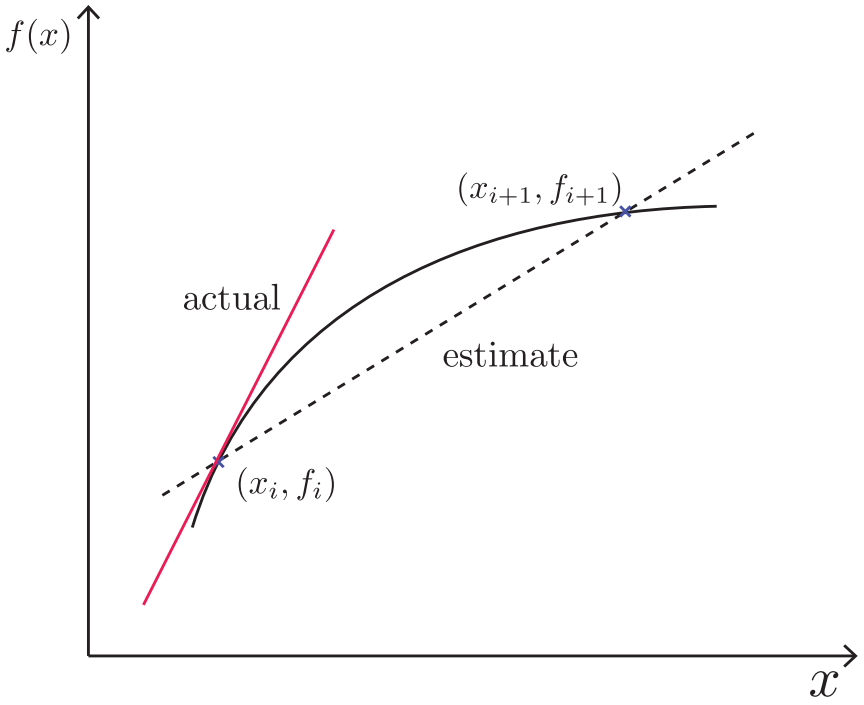


Figure 6.1: Geometric interpretation of the forward difference formula.

h	$(f(1+h)-f(1))/h$	error
0.1	1.105342410	-0.105342410
0.01	1.010050300	-0.010050300
0.001	1.001001000	-0.001001000
0.0001	1.000100000	-0.000100000

Table 6.1: Convergence of the (first order) forward difference formula.

have an error $O(h)$ are called *first order formulae*. Therefore, the forward difference formula is sometimes also called the *first order forward difference formula*. Note, however, that if h is taken much smaller, then rounding error becomes an issue in these calculations, see below.

Similarly,

$$f(x-h) = f(x) + f'(x)(-h) + \frac{f''(\eta)}{2}(-h)^2, \quad x-h < \eta < x. \tag{6.6}$$

Hence, for equidistant nodes, we find at $x = x_i$:

$$f'(x_i) = \frac{f_i - f_{i-1}}{h} + O(h), \tag{6.7}$$

which is called the *(first order) backward difference formula for the first derivative*. A geometrical interpretation of the backward difference formula is shown in Fig 6.2.

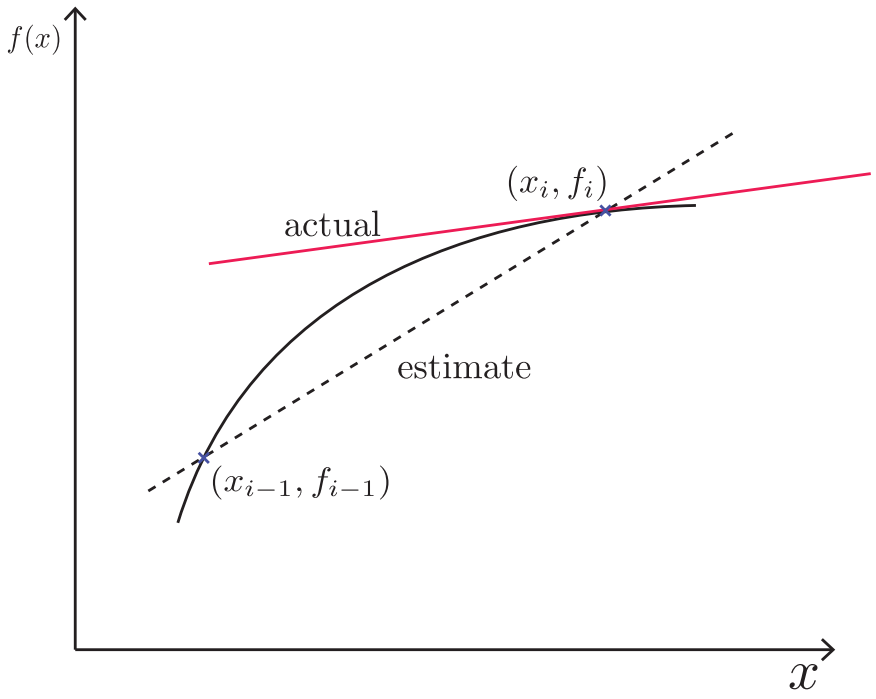


Figure 6.2: Geometric interpretation of the backward difference formula.

Higher order (i.e., more accurate) schemes can be derived by Taylor series of the function f at different points about the point x_i. For instance, assume that $f \in C^3([a,b])$. Taking the

difference between the two Taylor series

$$f(x+h) = f(x) + f'(x)\,h + \frac{f''(x)}{2}\,h^2 + \frac{f^{(3)}(\xi_1)}{6}\,h^3, \quad x < \xi_1 < x+h, \text{ and} \tag{6.8}$$

$$f(x-h) = f(x) - f'(x)\,h + \frac{f''(x)}{2}\,h^2 - \frac{f^{(3)}(\xi_2)}{6}\,h^3, \quad x-h < \xi_2 < x, \tag{6.9}$$

we find

$$f(x+h) - f(x-h) = 2f'(x)\,h + \frac{f^{(3)}(\xi_1) + f^{(3)}(\xi_2)}{6}\,h^3. \tag{6.10}$$

Since $f^{(3)}$ is continuous, the intermediate value theorem implies that there must exist some ξ between ξ_1 and ξ_2 such that

$$f^{(3)}(\xi) = \frac{f^{(3)}(\xi_1) + f^{(3)}(\xi_2)}{2}. \tag{6.11}$$

Thus,

$$f(x+h) - f(x-h) = 2f'(x)\,h + \frac{f^{(3)}(\xi)}{3}\,h^3, \quad x-h < \xi < x+h. \tag{6.12}$$

Solving for $f'(x)$,

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{f^{(3)}(\xi)}{6}\,h^2, \quad x-h < \xi < x+h \tag{6.13}$$

$$= \frac{f(x+h) - f(x-h)}{2h} + O(h^2) \tag{6.14}$$

For equidistant nodes, we find at $x = x_i$

$$f'(x_i) = \frac{f_{i+1} - f_{i-1}}{2h} + O(h^2). \tag{6.15}$$

The first term on the right-hand side of this equation is the *central difference formula for the first derivative*. Like the forward difference formula, it has a geometrical interpretation in terms of the gradient of a chord (cf. Fig 6.3). Beware that it might also suffer catastrophic cancellation problems when h is very small, see below.

Example 6.2 (Application of the central difference formula)
We apply the central difference formula to estimate $f'(1)$ when $f(x) = x^x$. To 10 significant digits, we find the results shown in table 6.2.

From table 6.2 we deduce that each time we decrease h by a factor of 10, the error is reduced (roughly) by a factor of 100. This is due to the fact that the error of the central difference formula is $O(h^2)$. We thus say that the central difference formula is a *second order formula* and therefore sometimes referred to as the *second order central difference formula*. It follows that the central difference formula is usually superior to the forward difference formula, because as h is decreased, the error for the central difference formula will converge to zero much faster than that for the forward difference formula. The superiority may be seen in the above example: for $h = 0.0001$, the forward difference formula is correct to 4 significant figures, whereas the central difference formula is correct to 8 significant figures.

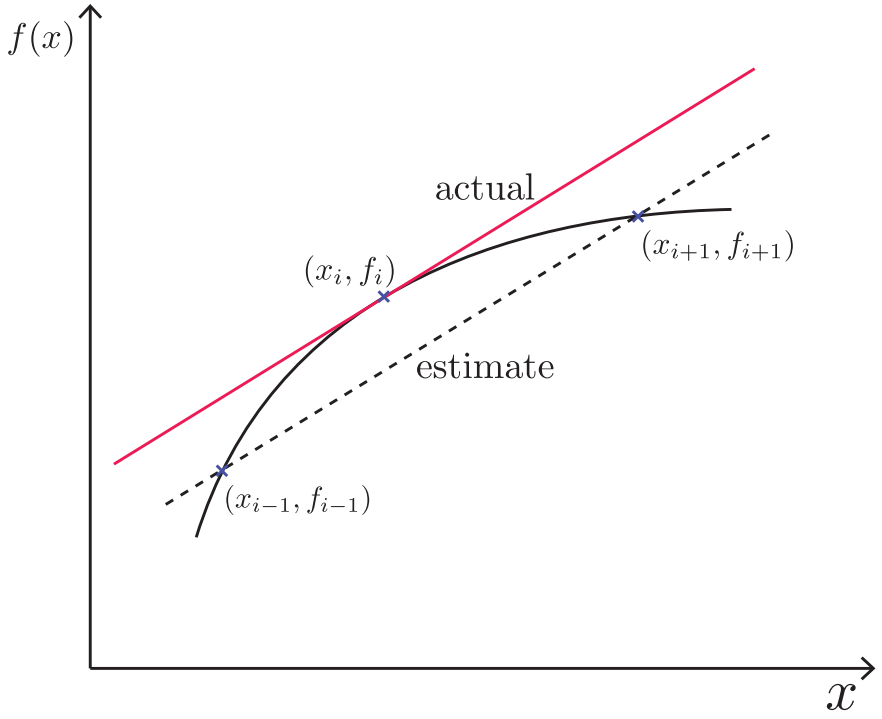


Figure 6.3: Geometric interpretation of the central difference formula.

h	$\frac{f(1+h)-f(1-h)}{2h}$	error
0.1	1.005008325	0.105342410
0.01	1.000050001	0.010050300
0.001	1.000000500	0.000000500
0.0001	1.000000005	0.000000005

Table 6.2: Example (second order) central difference formula for the first derivative.

.

Example 6.3 (Comparison of various difference formulae for the first derivative) *Use the data given in table 6.3 to estimate the acceleration at $t = 16$ s using the forward, backward, and central difference formula for various step sizes. The data have been generated with the formula*

$$v(t) = 2000 \ln\left[\frac{1.4 \cdot 10^5}{1.4 \cdot 10^5 - 2100\,t}\right] - 9.8\,t, \quad 0 \le t \le 30, \tag{6.16}$$

where v is given in [m/s] and t is given in [s]. The results are shown in table 6.3.

method	approximation [m/s^2]	absolute error [m/s^2]
forward	30.475	2.6967
backward	28.915	2.5584
central	29.695	0.0692

Table 6.3: Comparison of forward, backward, and central difference formula. Note that the central difference formula outperforms the forward and backward difference formulae.

In the same way, a fourth-order formula would look like

$$f'(x_i) = \frac{f_{i-2} - 8f_{i-1} + 8f_{i+1} - f_{i+2}}{12h} + O(h^4), \tag{6.17}$$

for $f \in C^5([a,b])$. One of the difficulties with higher order formulae occurs near the boundaries of the interval. They require functional values at points outside the interval, which are not available. Near the boundaries one usually resorts to lower order (backward and forward) formulae.

6.2.1 Approximation of derivatives of 2nd degree

Similar formulae can be derived for second or higher order derivatives. For example, suppose $f(x) \in C^4([a,b])$. Then,

$$f(x+h) = f(x) + f'(x)\,h + \frac{f''(x)}{2}\,h^2 + \frac{f^{(3)}(x)}{6}\,h^3 + \frac{f^{(4)}(\xi_1)}{24}\,h^4, \quad x < \xi_1 < x+h, \tag{6.18}$$

and

$$f(x-h) = f(x) - f'(x)\,h + \frac{f''(x)}{2}\,h^2 - \frac{f^{(3)}(x)}{6}\,h^3 + \frac{f^{(4)}(\xi_2)}{24}\,h^4, \quad x-h < \xi_2 < x. \tag{6.19}$$

Adding the last two equations gives

$$f(x+h) + f(x-h) = 2f(x) + f''(x)\,h^2 + \frac{f^{(4)}(\xi_1) + f^{(4)}(\xi_2)}{2}. \tag{6.20}$$

Since $f^{(4)}$ is continuous, the intermediate value theorem implies that there must be some ξ between ξ_1 and ξ_2 such that

$$f^{(4)}(\xi) = \frac{f^{(4)}(\xi_1) + f^{(4)}(\xi_2)}{2}. \tag{6.21}$$

Therefore,

$$f(x+h) - 2f(x) + f(x-h) = f''(x)\,h^2 + \frac{f^{(4)}(\xi)}{12}\,h^4, \quad x-h < \xi < x+h. \tag{6.22}$$

Solving for $f''(x)$ gives

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} - \frac{f^{(4)}(\xi)}{12}\,h^2, \quad x-h < \xi < x+h. \tag{6.23}$$

For equidistant data, we obtain for the second derivative of the function f at the point x_i:

$$f''(x_i) = \frac{f_{i+1} - 2f_i + f_{i-1}}{h^2} + O(h^2). \tag{6.24}$$

This is the *central difference formula for the second derivative.*

Example 6.4 (Central difference formula for the second derivative)
The velocity of a rocket is given by

$$v(t) = 2000\,\ln\left[\frac{1.4\cdot 10^5}{1.4\cdot 10^5 - 2100\,t}\right] - 9.8\,t, \quad 0 \le t \le 30, \tag{6.25}$$

where v is given in [m/s] *and t is given in* [s]. *Use the forward difference formula for the second derivative of $v(t)$ to calculate the jerk at $t = 16$ s. Use a step size of $\Delta t = 2$ s. The forward difference approximation of the second derivative is*

$$\phi''(t_i) = \frac{v_{i+2} - 2v_{i+1} + v_i}{\Delta t^2}. \tag{6.26}$$

With $t_i = 16$ and $\Delta t = 2$, we find $t_{i+1} = t_i + \Delta t = 18$ and $t_{i+2} = t_i + 2\Delta t = 20$. Hence,

$$\phi''(16) = \frac{v(20) - 2v(18) + v(16)}{2^2} = \frac{517.35 - 2\cdot 453.02 + 392.07}{4} = 0.845\ [\mathrm{m/s^3}]. \tag{6.27}$$

The exact value, $v''(16)$ can be computed from

$$v''(t) = \frac{18000}{(-200+3t)^2}. \tag{6.28}$$

Hence, $v''(16) = 0.7791\mathrm{m/s^3}$. The absolute relative error is

$$|\varepsilon| = \left|\frac{0.7791 - 0.8452}{0.7791}\right| \cdot 100 = 8.5\%. \tag{6.29}$$

6.2.2 Balancing truncation error and rounding error

Note that all of these methods suffer from problems with rounding errors due to cancellation. This is because the terms in the numerator for each formula become close together as the step size h decreases. There is a point where the rounding error, which increases if h decreases, becomes larger than the truncation error, which decreases as a power of h. This can be illustrated using the first order forward difference formula

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{1}{2}\, h\, f''(\xi), \quad x < \xi < x + h. \tag{6.30}$$

Suppose that when evaluating $f(x + h)$ and $f(x)$, we encounter roundoff errors $\varepsilon(x + h)$ and $\varepsilon(x)$, respectively. Then, our computation uses in fact the values $F(x + h)$ and $F(x)$, where $F(x + h) = f(x + h) - \varepsilon(x + h)$ and $F(x) = f(x) - \varepsilon(x)$. Then, the total error in the approximation of $f'(x)$ by the forward difference formula is

$$f'(x) - \frac{F(x+h) - F(x)}{h} = \frac{\varepsilon(x+h) - \varepsilon(x)}{h} - \frac{h}{2}\, f''(\xi), \quad x < \xi < x + h. \tag{6.31}$$

If we assume that the roundoff errors $\varepsilon(x + h)$ and $\varepsilon(x)$ are bounded by some number $\varepsilon > 0$ and the second derivative of f is bounded by a number $M > 0$ in the interval $(x, x+h)$, then

$$\begin{aligned}
\left| f'(x) - \frac{F(x+h) - F(x)}{h} \right| &\le \left| \frac{\varepsilon(x+h) - \varepsilon(x)}{h} - \frac{h}{2}\, f''(\xi) \right| \\
&\le \left| \frac{\varepsilon(x+h)}{h} \right| + \left| \frac{\varepsilon(x)}{h} \right| + \left| \frac{h}{2}\, f''(\xi) \right| \\
&\le \frac{2\varepsilon}{h} + \frac{M}{2}\, h.
\end{aligned}$$

This is an upper bound of the total error of the first order forward difference scheme. It consists of two parts. The second term on the right-hand side results from the error of the forward difference scheme (*truncation error*). This term decreases when reducing the step size h. The first term is caused by roundoff errors in the function evaluation. Thereby, a small roundoff error ε is amplified in case of tiny step sizes h. This implies that when using the forward difference scheme, it does not make sense to make h arbitrarily small to obtain a more accurate estimate of the first derivative. If h is smaller than some optimal value, say, h_{opt}, roundoff errors will exceed the error of the forward difference scheme as they increase proportional to $1/h$. The optimal h, denoted h_{opt}, is the one which minimizes the upper bound of the total error: let $e(h)$ denote the upper bound of the total error, i.e., we define

$$e(h) := \frac{2\varepsilon}{h} + \frac{M}{2}\, h. \tag{6.32}$$

$e(h)$ attains a minimum if $\frac{de}{dh} = 0$, i.e. for a value h_{opt} given by

$$h_{opt} = 2\sqrt{\frac{\varepsilon}{M}}. \tag{6.33}$$

Similar equations can be derived for other difference schemes, as well.

6.3 Richardson extrapolation

Richardson extrapolation is a method of increasing the order of accuracy of any numerical scheme — here we show it by the example of numerical differentiation but it applies equally to numerical integration and numerical solution of ODEs. For numerical differentiation, we can always express the truncation error as a series of powers of the step size h. In general, let $F(h)$ be a difference formula with step-size h, approximating $f'(x)$:

$$F(h) = f'(x) + \alpha_1 h^{p_1} + \alpha_2 h^{p_2} + \alpha_3 h^{p_3} + \ldots, \tag{6.34}$$

where we assume that $p_1 < p_2 < p_3 < \ldots$. Here, α_i are constants,

$$f'(x) = F(h) - \alpha_1 h^{p_1} - \alpha_2 h^{p_2} - \ldots = F(h) + O(h^{p_1}). \tag{6.35}$$

Hence, if we take $F(h)$ as approximation to $f'(x)$, the error is $O(h^{p_1})$. If h goes to zero, $F(h) \to f'(x)$ — i.e. the method is consistent. Assume that we know the power p_1, which we can obtain by the analysis of the previous sections. If we compute $F(h)$ for two step sizes, h and $h/2$, we have

$$\begin{aligned} F(h) &= f'(x) + \alpha_1 h^{p_1} + \alpha_2 h^{p_2} + \alpha_3 h^{p_3} + \ldots \\ F(h/2) &= f'(x) + \alpha_1 \frac{h^{p_1}}{2^{p_1}} + \alpha_2 \frac{h^{p_2}}{2^{p_2}} + \alpha_3 \frac{h^{p_3}}{2^{p_3}} + \ldots \end{aligned}$$

By taking a linear combination of these two formulae, we can eliminate α_1, giving

$$f'(x) = \frac{2^{p_1} F(h/2) - F(h)}{2^{p_1} - 1} + \beta_2 h^{p_2} + \beta_3 h^{p_3} + \ldots = \frac{2^{p_1} F(h/2) - F(h)}{2^{p_1} - 1} + O(h^{p_2}). \tag{6.36}$$

If we take the first term on the right-hand side as new approximation to $f'(x)$, we immediately see the important point here: we have reduced the error from $O(h^{p_1})$ in the original approximation to $O(h^{p_2})$ in the new approximation. If $F(h)$ is known for several values of h, for example, h, $h/2$, $h/4$, $h/8$ etc, the extrapolation process above can be repeated to obtain still more accurate approximations.

Example 6.5 (Richardson extrapolation of 1st-order forward differences)
Let us apply this method to the 1st-order forward difference formula. The Taylor series

$$f(x+h) = f(x) + f'(x)\,h + \frac{f''(x)}{2}\,h^2 + \frac{f^{(3)}(x)}{6}\,h^3 + \ldots, \tag{6.37}$$

can be rearranged for $f'(x)$, giving

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{f''(x)}{2}\,h - \frac{f^{(3)}(x)}{6}\,h^2 - \ldots \tag{6.38}$$

The forward difference approximation of $f'(x)$ is therefore

$$F_1(h) := \frac{f(x+h) - f(x)}{h}, \tag{6.39}$$

and the truncation error can be written

$$F_1(h) = f'(x) + \frac{f''(x)}{2}\,h + \frac{f^{(3)}(x)}{6}\,h^2 + \ldots = f'(x) + O(h). \tag{6.40}$$

Applying Richardson extrapolation via equation (6.36) *gives the new difference formula (where we have used* $p_1 = 1$ *and* $p_2 = 2$*):*

$$F_2(h) = \frac{2F_1(h/2) - F_1(h)}{2^1 - 1} = 2F_1(h/2) - F_1(h). \tag{6.41}$$

Because $p_2 = 2$*,*

$$F_2(h) = f'(x) + O(h^2), \tag{6.42}$$

i.e., the error of the new approximation $F_2(h)$ *is* $O(h^2)$*.*

Given the formula $F_2(h)$*, we can apply Richardson extrapolation again by applying* (6.36) *to* $F_2(h)$*, where now* $p_1 = 2$ *and* $p_2 = 3$ *corresponding to the truncation error of* $F_2(h)$*:*

$$F_3(h) = \frac{2^2F_2(h/2) - F_2(h)}{2^2 - 1} = \frac{4F_2(h/2) - F_2(h)}{3} = f'(x) + O(h^3). \tag{6.43}$$

The truncation error of the $F_3(h)$ *is* $O(h^3)$*, and we can continue like this indefinitely, provided samples of* $f(x + ih/2^n)$*,* $i \in \{0, \ldots, 2^n\}$ *are available.*

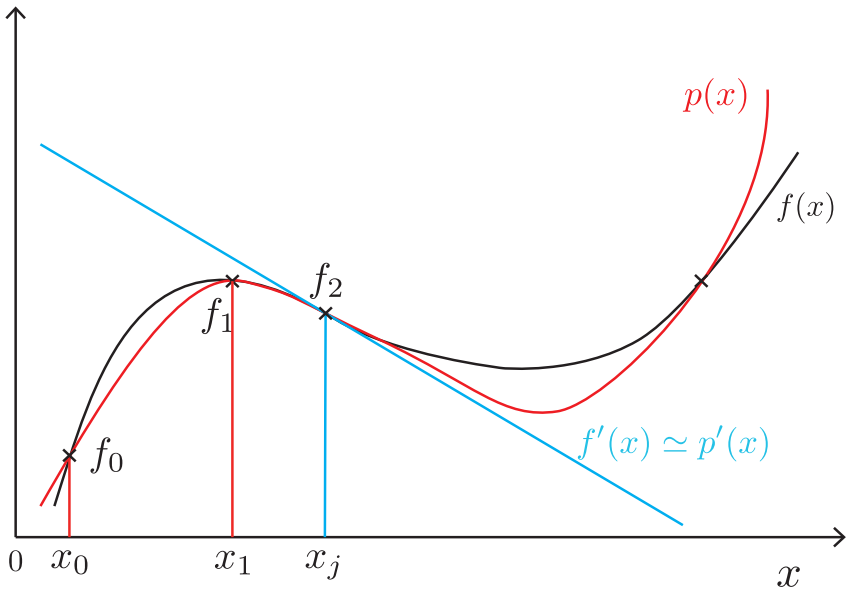
6.4 Difference formulae from interpolating polynomials

The Taylor series approach for approximating the derivatives of a function $f(x)$ requires the knowledge of values of the function at specific points in relation to where the derivative is to be approximated. For example, to approximate $f(x)$ by means of the central difference formula, one must know the value of $f(x)$ at $x + h$ and $x - h$ for some small value of h. This does not present a problem if we already know the function $f(x)$, but if $f(x)$ is only known through a finite collection of unequally spaced points, then the formulae we derived earlier will not, in general, have much value if we wish to find a derivative at a specific value of x. To be able to deal with this, we shall create a function that interpolates a finite number of points lying on the graph of $f(x)$. We shall then approximate $f^{(r)}(x)$ by differentiating this interpolating polynomial r times. The approach is shown graphically in Figure 6.4.

When we interpolate a function $f(x)$ at $n+1$ points $x_0, \ldots, x_n$, we have (cf. Chapters on interpolation), the function $f(x)$ can be written as:

$$f(x) = \sum_{i=0}^{n} l_i(x) f_i + \frac{f^{(n+1)}(\xi)}{(n+1)!}\,\omega_{n+1}(x), \quad x_0 < \xi < x_n, \tag{6.44}$$

where $l_i(x)$ are the Lagrange basis functions associated to the nodes $x_0 \ldots x_n$. Note that the first term on the right-hand side is the Lagrange representation of the interpolating



$$f'(x_j) = \sum_{i=0}^{n} l_i'(x_j) f_i + \omega_{n+1}'(x_j) \frac{f^{(n+1)}(\xi_1)}{(n+1)!}$$

Figure 6.4: Approximating derivatives of $f(x)$ (black), by first approximating with a interpolating polynomial $p(x)$ (red), and subsequently differentiating the polynomial (blue).

polynomial. Correspondingly, the second term on the right-hand side is the interpolation error. We differentiate this equation r times and obtain

$$f^{(r)}(x) = \sum_{i=0}^{n} l_i^{(r)}(x)\, f_i + \frac{1}{(n+1)!}\,\frac{d^r}{dx^r}\Big[\omega_{n+1}(x)\, f^{(n+1)}(\xi)\Big], \tag{6.45}$$

where $\xi = \xi(x)$, i.e., ξ is a function of x. Hence, we can compute an estimate of the r-th derivative $f^{(r)}(x)$ by computing the r-th derivative of the interpolating polynomial.

Example 6.6 (Differentiation using interpolating polynomials)
If $r = 1$, we obtain

$$f'(x) = \sum_{i=0}^{n} l_i'(x)\, f_i + \omega_{n+1}'(x)\frac{f^{(n+1)}(\xi_1)}{(n+1)!} + \omega_{n+1}(x)\frac{f^{(n+2)}(\xi_2)}{(n+2)!}, \tag{6.46}$$

where $\xi_1,\ \xi_2 \in (x_0, x_n)$. If we need to know f' at a node, say, x_j, we obtain

$$f'(x_j) = \sum_{i=0}^{n} l_i'(x_j)\, f_i + \omega_{n+1}'(x_j)\frac{f^{(n+1)}(\xi_1)}{(n+1)!}, \tag{6.47}$$

since $\omega_{n+1}(x)$ vanishes at $x = x_j$.

Example 6.7 (Differentiation using interpolating polynomials)
The 3 points $(0.1, e^{0.1})$, $(1, e)$, and $(2, e^2)$ lie on the graph of $f(x) = e^x$. Find an approximation of the value $f'(x)$ using the derivative of the interpolating polynomial through the 3 points. The (quadratic) interpolating polynomial is given by (test if yourself)

$$p(x) = 1.151496\, x^2 + 0.125887\, x + 1.077433, \tag{6.48}$$

to 6 decimal places. The first derivative of $p(x)$ is given by

$$p'(x) = 3.029925\, x + 0.125887, \tag{6.49}$$

while $f'(x) = e^x$. Table 6.4 compares values of $p'(x)$ with $f'(x)$ for various values of x (limited to 4 decimal places):

For instance, from quadratic polynomial interpolation through the data points (x_{i-1}, f_{i-1}), (x_i, f_i), and (x_{i+1}, f_{i+1}), we find

$$\begin{aligned}\phi(x) = {} & \frac{(x - x_i)(x - x_{i+1})}{(x_{i-1} - x_i)(x_{i-1} - x_{i+1})}\, f_{i-1} \\ & + \frac{(x - x_{i-1})(x - x_{i+1})}{(x_i - x_{i-1})(x_i - x_{i+1})}\, f_i \\ & + \frac{(x - x_{i-1})(x - x_i)}{(x_{i+1} - x_{i-1})(x_{i+1} - x_i)}\, f_{i+1},\end{aligned} \tag{6.50}$$

x	$p'(x)$	$f'(x)$	error
0.0	0.1259	1.0000	08741
0.1	0.4289	1.1052	0.6763
0.5	1.6408	1.6487	0.0078
1.0	3.1558	2.7183	-0.4375
1.5	4.6708	4.4817	-0.1891
2.0	6.1857	7.3891	1.2033
2.5	7.7007	12.1825	4.4818

Table 6.4: Differentiation using interpolating polynomials

for $x \in [x_i, x_{i+1}]$. We differentiate $\phi(x)$ analytically and can use the result to obtain an approximation of $f'(x)$ at any point $x \in [x_i, x_{i+1}]$. Note that you should not evaluate $\phi'(x)$ at a point x outside the interval $[x_i, x_{i+1}]$. If we want to know an approximation of $f'(x)$ at the node x_i, we evaluate $\phi'(x)$ at $x = x_i$:

$$\phi'(x_i) = -\frac{h_i}{h_{i-1}(h_{i-1}+h_i)}\, f_{i-1} - \frac{h_{i-1}-h_i}{h_{i-1}h_i}\, f_i + \frac{h_{i-1}}{h_i(h_{i-1}+h_i)}\, f_{i+1}, \tag{6.51}$$

where $h_k = x_{k+1} - x_k$. For equally spaced intervals it is $h_k = h =$ constant, and we find

$$\phi'(x_i) = \frac{f_{i+1} - f_{i-1}}{2h}. \tag{6.52}$$

This is the *central difference formula for the first derivative*, which has already been derived previously making use of Taylor series. If we differentiate $\phi(x)$, Eq. (6.50), twice we obtain a constant:

$$\phi''(x) = \frac{2}{h_{i-1}(h_{i-1}+h_i)}\, f_{i-1} - \frac{2}{h_{i-1}\, h_i}\, f_i + \frac{2}{h_i(h_{i-1}+h_i)}\, f_{i+1}, \quad x \in [x_i.x_{i+1}]. \tag{6.53}$$

This constant can be used as an approximation of $f''(x)$ at any point in the interval $[x_i, x_{i+1}]$. For equidistant points, it simplifies to

$$\phi''(x) = \frac{f_{i-1} - 2f_i + f_{i+1}}{h^2}. \tag{6.54}$$

When we use Eq. (6.54) as an approximation to $f''(x_i)$, we obtain the well-known *central difference formula for the second derivative*. Correspondingly, if we use Eq. (6.54) as an approximation to $f''(x_{i-1})$ and $f''(x_{i+1})$ it is called the *forward formula for the second derivative* and *backward formula for the second derivative*, respectively. Note that the backward formula for the second derivative is in fact based on an extrapolation.

Of course, additional differentiation formulae can be obtained if we fit an algebraic polynomial through 4 or more points and computing the derivative(s) analytically. Note that

the data points do not need to be equidistant and that the use of interpolating polynomials allows the computation of derivative(s) at any point inside the data interval. For instance, for equidistant nodes through 5 points, we obtain the approximation

$$f'(x_i) \approx \frac{1}{12h}\Big(f_{i-2} - 8f_{i-1} + 8f_{i+1} - f_{i+2}\Big) + \frac{h^4}{30}\, f^{(5)}(\xi), \quad x - 2h < \xi < x + 2h. \tag{6.55}$$

Example 6.8 (Differentiation using interpolating polynomials)
The upward velocity of a rocket is given as a function of time in table 6.5. Use a cubic interpolating polynomial to obtain an approximation of the acceleration of the rocket at $t = 16$ s. Since we want to find the velocity at $t = 16$ s, and we are using a cubic interpolating

time t [s]	velocity $v(t)$ [m/s]
0	0
10	227.04
15	362.78
20	517.35
22.5	602.97
30	901.67

Table 6.5: Example: approximate first derivatives from a cubic interpolating polynomial

polynomial, we need to choose the four points closest to $t = 16$ s and that also bracket $t = 16$ s to evaluate it. The four points are $t_0 = 10$, $t_1 = 15$, $t_2 = 20$, and $t_3 = 22.5$. We use the monomial representation of the cubic interpolating polynomial,

$$\phi(t) = a_0 + a_1\, t + a_2\, t^2 + a_3\, t^3, \tag{6.56}$$

and obtain from the interpolation condition the linear system

$$\begin{pmatrix} 1 & 10 & 100 & 1000 \\ 1 & 15 & 225 & 3375 \\ 1 & 20 & 400 & 8000 \\ 1 & 22.5 & 506.25 & 11391 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{pmatrix} 227.04 \\ 362.78 \\ 517.35 \\ 602.97 \end{pmatrix} \tag{6.57}$$

Solving the above system of equations gives

$$a_0 = -4.3810,\ a_1 = 21.289,\ a_2 = 0.13065,\ a_3 = 0.0054606. \tag{6.58}$$

The acceleration at $t = 16$ s is found by the evaluation of $\phi'(t) = a_1 + 2a_2\, t + 3a_3\, t^2$ at $t = 16$:

$$\phi'(t = 16) = 29.664\ \text{m/s}^2. \tag{6.59}$$

Chapter 7

Numerical Integration

7.1 Introduction

The numerical computation of definite integrals is one of the oldest problems in mathematics. In its earliest form the problem involved finding the area of a region bounded by curved lines, a problem which has been recognised long before the concept of integrals was developed in the 17th and 18th century.

Numerical integration is often referred to as *numerical quadrature*, a name which comes from the problem of computing the area of a circle by finding a square with the same area. The numerical computation of two- and higher-dimensional integrals is often referred to as *numerical cubature*. We will treat both topics in this chapter.

There are mainly three situations where it is necessary to calculate approximations to definite integrals. First of all it may be so that the antiderivative of the function to be integrated cannot be expressed in terms of elementary functions such as algebraic polynomials, logarithmic functions, or exponential functions. A typical example is $\int \mathrm{e}^{-x^2}\mathrm{d}x$. Secondly, it might be that the antiderivative function can be written down but is so complicated that its function values are better computed using numerical integration formulas. For instance, the number of computations that must be carried out to evaluate

$$\int_0^x \frac{\mathrm{d}t}{1+t^4} = \frac{1}{4\sqrt{2}} \log \frac{x^2+\sqrt{2}x+1}{x^2-\sqrt{2}x+1} + \frac{1}{2\sqrt{2}} \left(\arctan \frac{x}{\sqrt{2}-x} + \arctan \frac{x}{\sqrt{2}+x} \right)$$

using the 'exact' formula is substantial. A final reason for developing numerical integration formulas is that, in many instances, we are confronted with the problem of integrating experimental data, which also includes data generated by a computation. In that case the integrand is only given at discrete points and theoretical devices may be wholly inapplicable. This case also arises when quadrature methods are applied in the numerical treatment of differential equations. Most methods for discretizing such equations rest on numerical integration methods.

Let us begin by introducing some notations which will be used in this chapter. The idea of numerical integration of a definite integral $\int_a^b f(x)\mathrm{d}x$ is to look for an approximation of the form

$$\int_a^b f(x)\mathrm{d}x \approx \sum_{i=0}^{n} w_i f(x_i). \tag{7.1}$$

The coefficients w_i are called the "*weights*"; they depend on the location of the points x_i and on the integration domain $[a, b]$. The points x_i are called the "*nodes*" (or "*knots*"). The definite sum is called a "*quadrature formula*". All quadrate formulas look like (7.1); they only differ in the choice of the nodes and the weights. We will often write

$$I(f; a, b) := \int_a^b f(x)\mathrm{d}x \tag{7.2}$$

$$Q(f; a, b) := \sum_{i=0}^{n} w_i f(x_i) \tag{7.3}$$

The difference $I(f; a, b) - Q(f; a, b) =: E(f; a, b)$ is called the "*error of numerical integration*", or briefly "*quadrature error*". Thus,

$$I(f; a, b) = Q(f; a, b) + E(f; a, b) \tag{7.4}$$

Definition 7.1 (Concept of the degree d of precision (DOP))
A quadrature formula

$$Q(f; a, b) = \sum_{i=0}^{n} w_i\, f(x_i) \tag{7.5}$$

approximating the integral

$$I(f; a, b) = \int_a^b f(x)\,\mathrm{d}x \tag{7.6}$$

has DOP d if

$$\int_a^b f(x)\,\mathrm{d}x = \sum_{i=0}^{n} w_i\, f(x_i) \tag{7.7}$$

whenever $f(x)$ is a polynomial of degree at most d, but

$$\int_a^b f(x)\,\mathrm{d}x \neq \sum_{i=0}^{n} w_i\, f(x_i) \tag{7.8}$$

for some $f(x)$ of degree $d + 1$.

Equivalent to this is the following formulation: the rule $Q(f; a, b)$ approximating the definite integral $I(f; a, b)$ has DOP d if

$$\int_a^b x^q\,\mathrm{d}x = \sum_{i=0}^{n} w_i\, x_i^q \tag{7.9}$$

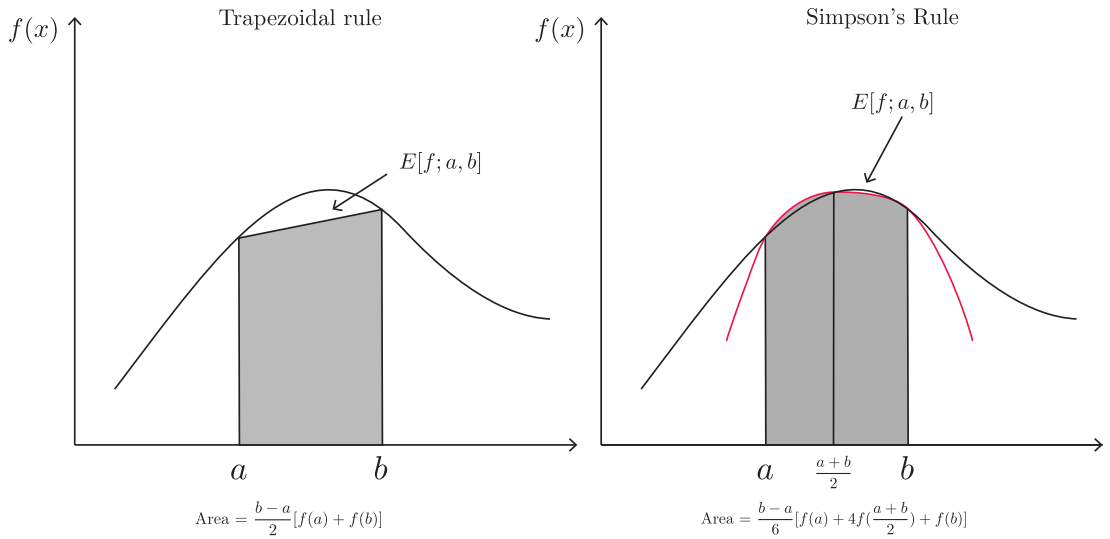


Figure 7.1: Trapezoidal rule (Section 7.4.1) and Simpson's rule (Section 7.4.2) for integral approximation.

for $q = 0, 1, \ldots, d$, but

$$\int_a^b x^q \, \mathrm{d}x \neq \sum_{i=0}^{n} w_i \, x_i^q \tag{7.10}$$

for $q = d+1$. From a practical point of view, if a quadrature formula $Q_1(f;a,b)$ has a higher DOP than another quadrature formula $Q_2(f;a,b)$, then Q_1 is considered more accurate than Q_2. This DOP concept can be used to derive quadrature formulas directly.

Figure 7.1 shows the basic idea: the area under the integrand is approximated by the area under a polynomial interpolant. The error in the approximation $E[f;a,b]$ corresponds to the area "missed" by the fitting polynomial.

7.2 Solving for quadrature weights

Suppose we want to develop a formula of approximate integration

$$\int_a^b f(x) \, \mathrm{d}x \approx \sum_{i=0}^{n} w_i f(x_i)$$

of a given degree of precision at least n. Assume that the nodes $x_0, \ldots, x_n$ are fixed and distinct numbers lying between a and b and prescribed in advance. Then the only degrees of freedom are the $n+1$ weights w_i which can be determined in two ways:

(a) Interpolate the function $f(x)$ at the $n+1$ points $x_0, \ldots, x_n$ by a polynomial of degree $\leq n$. Then integrate this interpolation polynomial exactly. The degree of precision d will be at least n.

(b) Select the weights $w_0, \ldots, w_n$ so that the rule integrates the monomials x^j exactly for $j = 0, \ldots, n$. I.e. we obtain the $n+1$ *integration conditions*:

$$\int_a^b x^j \, \mathrm{d}x = \sum_{i=0}^{n} w_i x_i^j, \quad \forall j = 0, \ldots, n,$$

(compare these conditions with the interpolation conditions). Again the degree of precision will be at least n by construction.

It can be shown that both ways lead to one and the same formula, which is called *"an interpolatory quadrature formula"*. Following (b), the integration conditions yields a linear system for the unknown weights w_i:

$$\begin{pmatrix} 1 & 1 & \cdots & 1 \\ x_0 & x_1 & \cdots & x_n \\ \vdots & \vdots & & \vdots \\ x_0^n & x_1^n & \cdots & x_n^n \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{pmatrix} = \begin{pmatrix} \int_a^b \mathrm{d}x \\ \int_a^b x \mathrm{d}x \\ \vdots \\ \int_a^b x^n \mathrm{d}x \end{pmatrix} \tag{7.11}$$

The matrix is the familiar "*Vandermonde matrix*". It is invertible if and only if the nodes x_i are distinct. Obviously, quadrature formulas of interpolatory type using $n+1$ nodes (i.e. $n+1$ function evaluations) have exact degree of precision $d = n$, i.e. they integrate exactly polynomials of degree at most n.

Exercise 7.2
Construct a rule of the form

$$I(f; -1, 1) = \int_{-1}^{1} f(x) \mathrm{d}x \approx w_0 f(-1/2) + w_1 f(0) + w_2 f(1/2)$$

which is exact for all polynomials of degree ≤ 2.

Rather than fixing x_i and determining w_i, one could prescribe the weights w_i and determining the nodes x_i such that the integration conditions are satisfied for all polynomials of degree $\leq n$. However this yields a *non-linear* system of algebraic equations for the nodes x_i, which is much more difficult to solve then the linear system of equations for w_i.

A more exciting possibility is to determine weights *and* nodes simultaneously such that the degree of precision d is as high as possible. The number of degrees of freedom is $2(n+1)$, so we should be able to satisfy integration conditions of up to degree $2n+1$ – these are the Gauss quadrature rules, and are the best possible integration rules (in the sense that they have highest degree of precision).

7.3 Numerical integration error – Main results

We would like to have an expression for the numerical integration error $E(f;a,b)$ for a quadrature rule with degree of precision d. In this section the result is simply stated, for a derivation see Section 7.8. We define:

$$E(f;a,b) = \int_a^b f(x)\,\mathrm{d}x - \sum_{i=0}^{n} w_i\, f(x_i), \tag{7.12}$$

where $a \le x_0 < x_1 < \ldots < x_n \le b$ are the nodes and w_i are the weights and E is the integration error. By the definition of degree of precision, if the quadrature rule has DOP d then:

$$E(x^j;a,b) = 0, \quad \text{for all } j = 0,\ldots,d, \tag{7.13}$$

and

$$E(x^{d+1};a,b) \neq 0. \tag{7.14}$$

A general expression for E will now be stated. First we must introduce some notation. The truncated power function is defined as

$$(x-s)_+^d := \begin{cases} (x-s)^d & \text{when } x \ge s \\ 0 & \text{when } x < s \end{cases}. \tag{7.15}$$

Theorem 7.3 (Peano kernel theorem)
Let $f \in C^{d+1}([a,b])$ and let $Q(f;a,b) = \sum_{i=0}^{n} w_i f(x_i)$ be a quadrature rule with DOP d. Then,

$$E(f;a,b) = \frac{1}{d!}\int_a^b f^{(d+1)}(s)\,K(s)\,ds, \tag{7.16}$$

where

$$K(s) = E((x-s)_+^d;a,b). \tag{7.17}$$

This result may appear somewhat ponderous, but it captures a beautiful idea: the error in the quadrate formula of f is related to the integral of a function $K(s)$ (called the *Peano kernel* or the *influence function*) that is independent of f.

Under typical circumstances $K(s)$ does not change sign on $[a,b]$. In this circumstance the second law of the mean can be applied to extract f from inside the integral in (7.16). Therefore $E(f;a,b)$ can be described as

$$E(f;a,b) = \frac{f^{(d+1)}(\xi)}{d!}\int_a^b K(s)\,\mathrm{d}s \qquad a < \xi < b, \tag{7.18}$$

$$= \kappa\, f^{(d+1)}(\xi), \qquad a < \xi < b, \tag{7.19}$$

where κ is *independent* of $f(x)$. Since we have this independence, to calculate κ we may make the special choice $f(x) = x^{d+1}$. Then from (7.19) we have

$$E(x^{d+1}; a, b) = \kappa\,(d+1)!, \tag{7.20}$$

which fixes κ:

$$\kappa = \frac{E(x^{d+1}; a, b)}{(d+1)!}.$$

Hence for any $(d+1)$-times continuously differentiable function $f \in C^{d+1}([a,b])$, we have

$$E(f; a, b) = \frac{E(x^{d+1}; a, b)}{(d+1)!} f^{(d+1)}(\xi), \quad a < \xi < b \tag{7.21}$$

so we can test our quadrature rule against x^{d+1}, and if it does well, it will do well for all $f(\cdot)$!

In the case of quadrature rules with equidistant nodes (closed Newton-Cotes rules), we can be even more specific:

Theorem 7.4 (Errors for Newton-Cotes Rules)
Let $Q_n(f; a, b)$ be a quadrature rule on $[a, b]$ with $n+1$ equidistant nodes $x_0, \ldots, x_n$

$$x_i = a + ih, \quad h = \frac{b-a}{n}$$

and degree of precision n. If n is odd (and $f \in C^{n+1}([a,b])$) then the integration error can be written:

$$E_n(f; a, b) = \frac{h^{n+2} f^{(n+1)}(\xi)}{(n+1)!} \int_0^n s(s-1)\cdots(s-n)\,ds. \tag{7.22}$$

If n is even (and $f \in C^{n+2}([a,b])$) it is:

$$E_n(f; a, b) = \frac{h^{n+3} f^{(n+2)}(\xi)}{(n+2)!} \int_0^n (s - \frac{n}{2})\, s(s-1)\cdots(s-n)\,ds, \tag{7.23}$$

where $a < \xi < b$ in both cases.

These expressions are used in the following sections to derive error estimates for closed Newton-Cotes rules of varying degree.

7.4 Newton-Cotes formulas

The Newton-Cotes integration formulas are quadrature rules with *equidistant* nodes. We distinguish two cases:

(a) the end-points of the integration domain are nodes, i.e. $a = x_0$, $b = x_n$. Then, we talk about *closed* Newton-Cotes formulas.

(b) the end-points of the integration domain are not nodes, i.e., $a < x_0$ and $x_n < b$. Then, we talk about *open* Newton-Cotes formulas.

First let us consider closed-type formulas. Thus, the integration domain $[a, b]$ is divided into n subintervals of equal length h at the points

$$x_i = a + ih, \quad i = 0, \ldots, n,$$

where $h = \frac{b-a}{n}$. The number of nodes is $s = n + 1$. For different s we obtain different closed Newton-Cotes formulas:

7.4.1 Closed Newton-Cotes (s=2) – Trapezoidal rule

With $s = 2$ we only have 2 points: $x_0 = a$ and $x_1 = b$. The weights w_0 and w_1 are the solution of the linear system of equations

$$\begin{pmatrix} 1 & 1 \\ a & b \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \end{pmatrix} = \begin{pmatrix} \int_a^b 1 \mathrm{d}x & = & b-a \\ \int_a^b x \mathrm{d}x & = & \frac{1}{2}(b^2 - a^2) \end{pmatrix} \tag{7.24}$$

The solution is $w_0 = w_1 = \frac{b-a}{2}$. Thus, the quadrature formula is

$$\int_a^b f(x)\mathrm{d}x \approx \sum_{i=0}^{1} w_i\, f(x_i) = \frac{b-a}{2}\Big[f(a) + f(b)\Big] \tag{7.25}$$

This quadrature formula is called the *trapezoidal rule*. The same result is obtained if we use the Lagrange representation of the interpolation polynomial through the data points:

$$\begin{aligned} \int_a^b p_1(x)\,\mathrm{d}x = \int_a^b \Big(\frac{x - x_1}{x_0 - x_1}\, f_0 + \frac{x - x_0}{x_1 - x_0}\, f_1\Big)\,\mathrm{d}x &= \frac{h}{2}\Big(f_0 + f_1)\Big) \\ &= \frac{h}{2}\Big(f(a) + f(b)\Big), \quad \text{where } h = b - a. \end{aligned}$$

Since $n = 1$, the integration error is:

$$E(f; a, b) = \frac{h^3\, f''(\xi)}{2} \int_0^1 s(s-1)\, ds = -\frac{1}{12} h^3\, f^{(2)}(\xi), \qquad a < \xi < b. \tag{7.26}$$

We say that $E(f; a, b) = O(h^3)$, which means that $E(f; a, b)$ tends to zero as h^3, $h \to 0$. Note that if f is a linear function, $f'' = 0$, hence $E(f; a, b) = 0$. This was to be expected, because the degree of precision of the trapezoidal rule is, by construction, equal to 1. Remember that the error estimates requires $f \in C^2([a, b])$.

7.4.2 Closed Newton-Cotes (s=3) – Simpson's rule

For $s = 3$ we have 3 points: $x_0 = a$, $x_1 = \frac{a+b}{2}$, $x_2 = b$. The weights are the solution of the linear system

$$\begin{pmatrix} 1 & 1 & 1 \\ a & \frac{a+b}{2} & b \\ a^2 & (\frac{a+b}{2})^2 & b^2 \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} b-a \\ \frac{1}{2}(b^2-a^2) \\ \frac{1}{3}(b^3-a^3) \end{pmatrix} \tag{7.27}$$

The solution is $w_0 = \frac{1}{6}(b-a)$, $w_1 = \frac{2}{3}(b-a)$, $w_3 = \frac{1}{6}(b-a)$, and the quadrature formula is

$$\int_a^b f(x)\mathrm{d}x \approx \sum_{i=0}^{2} w_i\, f(x_i) = \frac{b-a}{6}\Big[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\Big] \tag{7.28}$$

This quadrature formula is called "*Simpson's rule*". Using the Lagrange representation of the degree-2 interpolation polynomial yields

$$\int_a^b p_2(x)\,\mathrm{d}x = \int_a^b \Big[\frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)}\,f_0 + \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)}\,f_1 + \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_0)}\,f_2\Big]\,\mathrm{d}x = \frac{h}{3}\Big(f_0 + 4f_1 + f_2\Big), \quad \text{where } h = \frac{b-a}{2}.$$

It is $n = 2$, and the error of Simpson's rule is

$$E(f;a,b) = \frac{h^5 f^{(4)}(\xi)}{4!}\int_0^2 s(s-1)^2(s-2)\,ds = -\frac{h^5}{90}f^{(4)}(\xi), \quad a < \xi < b. \tag{7.29}$$

Therefore, the error is of the order $O(h^5)$. This is remarkable, because the trapezoidal rule, which uses only one node less, is of the order $O(h^3)$. That is the reason why Simpson's rule is so popular and often used.

7.4.3 Closed Newton-Cotes (s=4) – Simpson's 3/8-rule

For $s = 4$ the points are $x_0 = a$, $x_1 = a + \frac{b-a}{3}$, $x_2 = a + \frac{2}{3}(b-a)$, $x_3 = b$, i.e., $h = \frac{b-a}{3}$. The weights are the solution of the linear system

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ a & \frac{2a+b}{3} & \frac{a+2b}{3} & b \\ a^2 & (\frac{2a+b}{3})^2 & (\frac{a+2b}{3})^2 & b^2 \\ a^3 & (\frac{2a+b}{3})^3 & (\frac{a+2b}{3})^3 & b^3 \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{pmatrix} = \begin{pmatrix} b-a \\ \frac{1}{2}(b^2-a^2) \\ \frac{1}{3}(b^3-a^3) \\ \frac{1}{4}(b^4-a^4) \end{pmatrix} \tag{7.30}$$

The solution is $w_0 = \frac{1}{8}(b-a)$, $w_1 = w_2 = \frac{3}{8}(b-a)$, $w_3 = \frac{1}{8}(b-a)$ and the quadrature formula is

$$\int_a^b f(x)\mathrm{d}x \approx \sum_{i=0}^{3} w_i\, f(x_i) = \frac{b-a}{8}\Big[f(a) + 3f\left(\frac{2a+b}{3}\right) + 3f\left(\frac{a+2b}{3}\right) + f(b)\Big] \tag{7.31}$$

This formula is called "*Simpson's 3/8-rule*". The error is

$$E(f;a,b) = -\frac{3}{80}h^5\, f^{(4)}(\xi) = -\frac{(b-a)^5}{6480}\, f^{(4)}(\xi), \qquad a < \xi < b. \tag{7.32}$$

We observe that the error is of the order $O(h^5)$. Though Simpson's 3/8-rule uses one node more than Simpson's rule, it has the same order $O(h^5)$.

7.4.4 Closed Newton-Cotes (s=5) – Boules's rule

For $s = 5$, $x_i = a + h\,i$, $i = 0, \ldots, 4$, and $h = \frac{b-a}{4}$. The weights are the solution of a linear system which is straightforward to set up, but is not shown here. The result is

$$w_0 = \frac{7}{90}(b-a), \quad w_1 = \frac{32}{90}(b-a), \quad w_2 = \frac{12}{90}(b-a), \quad w_3 = \frac{32}{90}(b-a), \quad w_4 = \frac{7}{90}(b-a)$$

Thus, the quadrature formula is

$$\int_a^b f(x)\mathrm{d}x \approx \frac{b-a}{90}\Big[7f(a) + 32f\left(\frac{3a+b}{4}\right) + 12f\left(\frac{a+b}{2}\right) + 32f\left(\frac{a+3b}{4}\right) + 7f(b)\Big] \tag{7.33}$$

This formula is called "*Boules's rule*" or "*4/90-rule*". The error is of the order $O(h^7)$:

$$E(f;a,b) = -\frac{8}{945}h^7\, f^{(6)}(\xi) = -\frac{(b-a)^7}{1935360}\, f^{(6)}(\xi), \qquad a < \xi < b \tag{7.34}$$

7.4.5 Open Newton-Cotes Rules

The Newton-Cotes formulas discussed so far are called *closed* Newton-Cotes formulas, because the endpoints of he integration interval (i.e., a and b) belong to the nodes. Otherwise, they are called *open* Newton-Cotes formulas. The most simple open Newton-Cotes formula uses $s = 1$ node at $\frac{a+b}{2}$, i.e., the node is at the midpoint of the integration interval. Therefore, this rule is called *midpoint rule*:

$$Q(f;a,b,) = (b-a)\, f(\frac{a+b}{2}), \;\; E(f;a,b) = \frac{(b-a)^3}{24}\, f^{(2)}(\xi), \quad a < \xi < b. \tag{7.35}$$

If the open Newton-Cotes rule has s points, the location of these points may be

$$x_i = a + h(i+1), \;\; i = 0, \ldots, s-1, \;\; h = \frac{b-a}{s+1}. \tag{7.36}$$

Alternatively, open Newton-Cotes formulas with $s = n-1$ points can be obtained by leaving out the nodes $x_0 = a$ and $x_n = b$ of a $(n+1)$-point closed Newton-Cotes formula.

Another well-known open Newton-Cotes formula is *Milne's rule*: $s = 3$, $h = \frac{b-a}{4}$:

$$Q(f;a,b) = \frac{b-a}{3}\Big(2f_0 - f_1 + 2f_2\Big) = \frac{4h}{3}\Big(2f_0 - f_1 + 2f_2\Big), \tag{7.37}$$

$$E(f;a,b) = \frac{7(b-a)^5}{23040} f^{(4)}(\xi) = \frac{14h^5}{45} f^{(4)}(\xi), \quad a < \xi < b. \tag{7.38}$$

Note that Simpson's rule and Milne's rule are even exact with degree of precision 3, though they use only $s = 3$ nodes. This is due to the symmetric distribution of the nodes over the interval $[a, b]$.

Example 7.5
Given the function $f(x) = \frac{1}{1+x^2}$ *for* $x \in [0, 1]$. *We look for an approximation to* $\pi/4$ *by integrating* f *over the interval* $[0, 1]$ *using (a) the trapezoidal rule, (b) Simpson's rule, (c) Simpson's* 3/8*-rule, and (d) the* 4/90*-rule.*

(a) trapezoidal rule: $Q(f;0,1) = \frac{1}{2}(f(0) + f(1)) = \frac{1}{2}(1 + 1/2) = 0.7500$.

(b) Simpson's rule: $Q(f;0,1) = \frac{1}{6}(f(0) + 4f(1/2) + f(1)) = \frac{1}{6}(1 + 3.2 + 0.5) = 0.78333$.

(c) Simpson's 3/8*-rule:* $Q(f;0,1) = \frac{1}{8}(f(0) + 3f(1/3) + 3f(2/3) + f(1)) = \frac{1}{8}(1 + 2.7 + 27/13 + 0.5) = 0.78462$.

(d) 4/90*-rule:* $Q(f;0,1) = \frac{1}{90}(7f(0) + 32f(1/4) + 12f(1/2) + 32f(3/4) + 7f(1)) = \frac{1}{90}(7 + 512/17 + 48/5 + 512/25 + 3.5) = 0.78553$.

The corresponding errors are for (a) $3.5 \cdot 10^{-2}$, *(b)* $2.1 \cdot 10^{-3}$, *(c)* $7.8 \cdot 10^{-4}$, *and (d)* $1.3 \cdot 10^{-4}$.

7.5 Composite Newton-Cotes formulas

The idea of piecewise polynomial interpolation leads to the so-called *composite Newton-Cotes formulas*. The basic idea is to integrate piecewise polynomials, i.e., to subdivide the interval $[a, b]$ into n subintervals $[x_i, x_{i+1}]$, $i = 0, \ldots, n-1$ where $a = x_0 < x_1 < \ldots < x_n = b$ and to use the fact that

$$\int_a^b f(x)\,\mathrm{d}x = \sum_{i=0}^{n-1} \int_{x_i}^{x_{i+1}} f(x)\,\mathrm{d}x. \tag{7.39}$$

We then approximate each of the integrals

$$\int_{x_i}^{x_{i+1}} f(x)\,\mathrm{d}x \tag{7.40}$$

by one of the Newton-Cotes formulas just developed and add the results. Hence, if we use a closed Newton-Cotes formula with s nodes, the composite Newton-Cotes formula has $n(s-1)+1$ nodes. The corresponding rules are known as *composite Newton-Cotes rules*.

7.5.1 Composite mid-point rule

Let $h = \frac{b-a}{n}$, $x_i = a + ih$ for $i = 0, \dots n-1$. The most simple quadrature rule to be used on each subinterval $[x_i, x_{i+1}]$ is the *midpoint rule*:

$$\int_{x_i}^{x_{i+1}} f(x)\,\mathrm{d}x = hf_{i+1/2} + \frac{h^3}{24} f^{(2)}(\xi_i), \quad \xi_i \in [x_i, x_{i+1}], \tag{7.41}$$

where $f_{i+1/2} = f(x_{i+1/2})$ and $x_{i+1/2} = \frac{x_i + x_{i+1}}{2}$. Hence,

$$\begin{aligned}\int_a^b f(x)\,\mathrm{d}x &= h\sum_{i=0}^{n-1} f_{i+1/2} + \frac{h^3}{24}\sum_{i=0}^{n-1} f^{(2)}(\xi_i) \\ &= h\sum_{i=0}^{n-1} f_{i+1/2} + \frac{b-a}{24} h^2 f^{(2)}(\xi), \quad a < \xi < b.\end{aligned}$$

Note that if

$$|f^{(2)}(x)| \le M \text{ for all } x \in [a, b], \tag{7.42}$$

then by choosing h sufficiently small, we can achieve any desired accuracy (neglecting roundoff errors). The rule

$$Q(f; a, b) = h\sum_{i=0}^{n-1} f_{i+1/2}, \quad h = \frac{b-a}{n} \tag{7.43}$$

is called the *composite midpoint rule*.

7.5.2 Composite trapezoidal rule

Following the same idea, we may construct the *composite trapezoidal rule*: let $h = \frac{b-a}{n}$; then,

$$\int_{x_i}^{x_{i+1}} f(x)\,\mathrm{d}x = \frac{h}{2}(f_i + f_{i+1}) - \frac{h^3}{12} f^{(2)}(\xi_i), \quad x_i < \xi < x_{i+1}, \tag{7.44}$$

so that

$$\begin{aligned}\int_a^b f(x)\,\mathrm{d}x &= \frac{h}{2}\sum_{i=0}^{n-1}\Big[f_i + f_{i+1}\Big] - \frac{h^3}{12}\sum_{i=0}^{s-2} f^{(2)}(\xi_i) \\ &= h\Big[\frac{1}{2}f_0 + f_1 + f_2 + \ldots + f_{n-1} + \frac{1}{2}f_n\Big] - \frac{b-a}{12} h^2 f^{(2)}(\xi), \quad a < \xi < b.\end{aligned}$$

Example 7.6 (Error in the composite trapezoidal rule)
We consider the trapezoidal rule

$$Q(f; a, b) = \frac{b-a}{2}\left(f(a) + f(b)\right); \quad E(f; a, b) = -\frac{h^3}{12} f^{(2)}(\xi), \qquad a < \xi < b.$$

We want to apply a composite trapezoidal rule to evaluate approximately

$$\int_a^b f(x)\mathrm{d}x$$

For m subintervals $[a_i, b_i], i = 1, \ldots, m$, we have

$$\int_a^b f(x)\mathrm{d}x = \sum_{i=1}^{m} \int_{a_i}^{b_i} f(x)\mathrm{d}x \approx \sum_{i=1}^{m} \frac{b_i - a_i}{2} \left(f(a_i) + f(b_i)\right),$$

with $a_i = a + (i-1)h$, $b_i = a + ih$, $h = \frac{b-a}{m}$. Since $b_i - a_i = h$, we obtain

$$\begin{aligned} Q_m(f; a, b) &= \sum_{i=1}^{m} \frac{b_i - a_i}{2} \left(f(a_i) + f(b_i)\right) = \\ &= \frac{h}{2}\Big(f(a) + 2f(a+h) + \cdots + 2f(a+(m-1)h) + f(b)\Big). \end{aligned}$$

For the error of the composed rule, we obtain

$$E_m(f; a, b) = -\frac{h^3}{12}\left(f''(\xi_1) + f''(\xi_2) + \cdots + f''(\xi_m)\right)$$

with $\xi_i \in (a_i, b_i), \quad i = 1, \ldots, m$.

Following the mean value theorem that any continuous function g on $[a, b]$ with $c_1 \leq g(x) \leq c_2$ for $x \in [a, b]$ takes on every value between c_1 and c_2 at least once, we find at least one $\xi \in [a, b]$ such that

$$f''(\xi) = \frac{1}{m}\left(f''(\xi_1) + f''(\xi_2) + \cdots + f''(\xi_m)\right).$$

Therefore, we obtain

$$E_m(f; a, b) = -\frac{h^2}{12}\,\frac{b-a}{m}\,m\,f''(\xi) = -\frac{h^2}{12}\,(b-a)\,f''(\xi), \qquad a < \xi < b,$$

which means that the total error is of the order $O(h^2)$, one order less than that of the simple trapezoidal rule, which has $O(h^3)$.

7.6 Interval transformation

Quadrature rules are often developed for special integration intervals, e.g. $[0, 1]$ or $[-1, 1]$. That is, the nodes and weights given in tables or computer software refer to these intervals. How can be use these formulas to approximate

$$\int_a^b f(x)\,\mathrm{d}x? \tag{7.45}$$

Suppose the interval to which the nodes and weights of Q refer is $[-1,1]$. Let $\{\xi_i : i = 1,\dots,s\}$ be the nodes of $Q(f;-1,1)$ and $\{w_i : i = 1,\dots,s\}$ be the weights. Then,

$$\int_a^b f(x)\,\mathrm{d}x = \frac{b-a}{2}\int_{-1}^{1} f(x(\xi))\,d\xi = \frac{b-a}{2}\int_{-1}^{1} g(\xi)\,d\xi = Q(g;-1,1) + E(g;-1,1), \quad (7.46)$$

where

$$x = \frac{b-a}{2}(\xi+1) + a. \quad (7.47)$$

Hence,

$$Q(f;a,b) = \frac{b-a}{2}Q(g;-1,1), \quad g(\xi) := f(x(\xi)). \quad (7.48)$$

That is if $Q(f;-1,1)$ has nodes ξ and weights w_i, then $Q(f;a,b)$ has nodes x_i and weights $\frac{b-a}{2}\,w_i$, where

$$x_i = \frac{b-a}{2}(\xi_i+1) + a. \quad (7.49)$$

Hence,

$$\int_a^b f(x)\,\mathrm{d}x \approx \sum_{i=1}^{s}\Big(\frac{b-a}{2}\,w_i\Big)\,f_i, \quad f_i = f(x_i). \quad (7.50)$$

That means if we want to construct a quadrature rule for an arbitrary integration interval $[a,b]$ from given nodes and weights referring to another interval, say, $[-1,1]$, we only need to find the (linear) mapping that maps $[-1,1]$ onto $[a,b]$. This is exactly Eq. (7.47). The transformation allows to compute the nodes referring to the interval $[a,b]$ and the Jacobian of the mapping multiplied by the weights referring to $[-1,1]$ gives the weights referring to $[a,b]$.

We shall call two rules by the same name, provided that the abscissas and weights are related by a linear transformation as above. For instance, we shall speak of Simpson's rule over the interval $[-1,1]$ and over $[a,b]$.

7.7 Gauss quadrature

Till now we *prescribed* the nodes $\{x_i : i = 1,\dots,s\}$ of the quadrature formula and determined the weights $\{w_i : i = 1,\dots,s\}$ as solution of the linear system

$$\sum_{k=1}^{s} w_k\,x_k^m = \frac{b^{m+1}-a^{m+1}}{m+1}, \qquad m = 0,1,\dots,s-1 \quad (7.51)$$

Then, the resulting quadrature formula has DOP $s-1$, i.e., it is exact for all polynomials of degree at most $s-1$.

If we do not prescribe nodes and weights, the above system of equations contains $2s$ free parameters, namely s nodes and s weights. The system is linear in the weights and non-linear in the nodes. We may use the $2s$ degrees of freedom to demand that the integration

formula is exact for polynomials of degree at most $2s-1$, which are uniquely determined by $2s$ coefficients. That means we require that

$$\int_a^b p(x)\mathrm{d}x = \sum_{i=1}^{s} w_i\, p(x_i), \tag{7.52}$$

for all $p \in \mathbb{P}_{2s-1}$, i.e., for all polynomials of degree at most $2s-1$. This is the idea behind the so-called *Gauss quadrature formulas*. Of course, the question is whether the system of non-linear equations (7.52) is uniquely solvable.

Suppose the integration interval is $[-1,1]$ and suppose there exists nodes and weights so that $E(f;-1,1)=0$ for all $f \in \mathbb{P}_{2s-1}$. Let $q \in \mathbb{P}_{s-1}$ arbitrary and consider the polynomial

$$q(x)\cdot(x-x_0)(x-x_1)\cdots(x-x_{s-1}) \in \mathbb{P}_{2s-1}. \tag{7.53}$$

The polynomial

$$\omega_s(x) := (x-x_0)(x-x_1)\cdots(x-x_{s-1}) \tag{7.54}$$

is the nodal polynomial; it has degree s. Moreover, it holds obviously $Q(q\cdot\omega_s;-1,1)=0$. Hence, it must hold

$$\int_{-1}^{1} q(x)\,\omega_s(x)\,\mathrm{d}x = 0, \quad \text{for all } q \in \mathbb{P}_{s-1}. \tag{7.55}$$

We define in $C([-1,1])$ the scalar product

$$\langle f, g\rangle := \int_{-1}^{1} f(x)g(x)\,\mathrm{d}x. \tag{7.56}$$

Then, condition (7.55) means that $\omega_s(x)$ is orthogonal to all polynomials $q \in \mathbb{P}_{s-1}$ with respect to the scalar product (7.56). We construct a system of orthogonal polynomials L_i, i.e.,

$$\langle L_i, L_j\rangle = \begin{cases} 0 & \text{for } i \neq j \\ \neq 0 & \text{for } i = j \end{cases}. \tag{7.57}$$

This can be done from the monomial basis x^k using the Gram-Schmidt orthogonalization procedure. With $L_0(x)=1$ and $L_1(x)=x$, the solution is

$$L_n(x) = x^n - \sum_{i=0}^{n-1} \frac{\langle x^n, L_i\rangle}{\langle L_i, L_i\rangle}\, L_i(x). \tag{7.58}$$

The polynomials $L_n(x)$ are unique and can be written as

$$L_n(x) = \frac{2^n (n!)^2}{(2n)!}\, P_n(x), \tag{7.59}$$

where $P_n(x)$ is the *Legendre polynomial of degree n*:

$$P_n = \frac{1}{2^n n!}\frac{d^n}{\mathrm{d}x^n}(x^2-1)^n. \tag{7.60}$$

It is $P_0(x) = 1$, $P_1(x) = x$, $P_2(x) = \frac{1}{2}(3x^2 - 1)$, etc. They can be computed recursively:

$$P_{n+1}(x) = \frac{2n+1}{n+1}\, x\, P_n(x) - \frac{n}{n+1}\, P_{n-1}(x), \quad n = 1, 2, \ldots. \tag{7.61}$$

Hence, L_n is up to a scaling factor identical to P_n. The Legendre polynomial $P_n(x)$ has n distinct zeros in the interval $(-1, 1)$ and so has L_n. For given number of nodes s, we choose the zeros of $P_s(x)$ as nodes of the integration formula. For arbitrary $p \in \mathbb{P}_{2s-1}$, we have by polynomial division

$$p(x) = q(x)L_s(x) + r(x), \tag{7.62}$$

where $q \in \mathbb{P}_{s-1}$ and a remainder $r \in \mathbb{P}_{s-1}$. Due to the orthogonality, we have

$$\int_{-1}^{1} p(x)\,\mathrm{d}x = \int_{-1}^{1} q(x)L_s(x)\,\mathrm{d}x + \int_{-1}^{1} r(x)\,\mathrm{d}x. \tag{7.63}$$

The first integral on the right-hand site is zero due to the orthogonality. Hence,

$$\int_{-1}^{1} p(x)\,\mathrm{d}x = \int_{-1}^{1} r(x)\,\mathrm{d}x. \tag{7.64}$$

Moreover, it is for arbitrary weights w_i:

$$Q(p; -1, 1) = Q(qL_s; -1, 1) + Q(r; -1, 1) = \sum_{i=0}^{s-1} w_i q(x_i) L_s(x_i) + Q(r; -1, 1) = Q(r; -1, 1), \tag{7.65}$$

because $L_s(x_i) = 0$. Hence, it holds

$$\int_{-1}^{1} p(x)\,\mathrm{d}x = Q(p; -1, 1), \quad \text{for all } p \in \mathbb{P}_{2s-1}, \tag{7.66}$$

if and only if

$$\int_{-1}^{1} r(x)\,\mathrm{d}x = Q(r; -1, 1), \quad \text{for all } r \in \mathbb{P}_{s-1}. \tag{7.67}$$

That the latter holds can be achieved by a suitable choice of the weights according to the principle of interpolatory quadrature. It is for $r \in \mathbb{P}_{s-1}$:

$$\int_{-1}^{1} r(x)\,\mathrm{d}x = \int_{-1}^{1} \sum_{i=0}^{s-1} r(x_i)\, l_i(x)\,\mathrm{d}x = \sum_{i=0}^{s-1} \Big(\int_{-1}^{1} l_i(x)\,\mathrm{d}x \Big)\, r(x_i). \tag{7.68}$$

Hence, the weights are

$$w_i = \int_{-1}^{1} l_i(x)\,\mathrm{d}x = \int_{-1}^{1} \prod_{j=0, j\neq i} \frac{x - x_j}{x_i - x_j}\,\mathrm{d}x, \quad i = 0, \ldots s-1, \tag{7.69}$$

and can be computed from the nodes $x_0, \ldots x_{s-1}$. The result is the following:

Definition 7.7 (Gauss-Legendre quadrature)
Let $\{x_i : i = 0, \dots s-1\}$ be the s zeros of the Legendre polynomial $P_s(x)$ and let

$$w_i = \int_{-1}^{1} l_i(x)\,\mathrm{d}x, \tag{7.70}$$

where $l_i(x)$ is the i-th Lagrange polynomial associated with the nodes $\{x_i\}$. Then,

$$\int_{-1}^{1} p(x)\,\mathrm{d}x = \sum_{i=0}^{s-1} w_i\, p(x_i) \quad \text{for all } p \in \mathbb{P}_{2s-1}, \tag{7.71}$$

and for $f \in C^{2s}([-1,1])$ the integration error is

$$E(f;-1,1) = \frac{f^{(2s)}(\xi)}{(2s)!}\int_{-1}^{1} L_s(x)^2\,\mathrm{d}x = \frac{2}{(2s+1)(2s)!}\,f^{(2s)}(\xi), \quad -1 < \xi < 1. \tag{7.72}$$

The weights of all Gauss-Legendre quadrature rules are positive. The sum of the weights is always equal to 2, the length of the interval $[-1,1]$. A Gauss-Legendre rule with s nodes has DOP $2s-1$. This is the maximum degree of precision a quadrate rule with s nodes can have. All nodes are in $(-1,1)$. The nodes are placed symmetrically around the origin, and the weights are correspondingly symmetric. For s even, the nodes satisfy $x_0 = -x_{s-1}$, $x_1 = -x_{s-2}$ etc., and the weights satisfy $w_0 = w_{s-1}$, $w_1 = w_{s-2}$ etc. For s odd, the nodes and weights satisfy the same relation as for s even plus we have $x_{\frac{s+1}{2}} = 0$. Notice that we have shown before that the Gauss-Legendre rules are interpolatory rules! If we need to apply a Gauss-Legendre rule to approximate an integral over the interval $[a,b]$ we transform the nodes and weights from the interval $[-1,1]$ to the interval $[a,b]$ as explained in section 7.6.

Example 7.8
When applying a Gauss-Legendre formula with s nodes to a function f on the interval $[-1,1]$, we obtain for

$s = 2$: $\int_{-1}^{1} f(x)\mathrm{d}x \approx f(-\frac{1}{\sqrt{3}}) + f(\frac{1}{\sqrt{3}})$, $E(f;-1,1) = \frac{h^4}{135} f^{(4)}(\xi)$, $-1 < \xi < 1$.

$s = 3$: $\int_{-1}^{1} f(x)\mathrm{d}x \approx \frac{1}{9}\left(5f(-\sqrt{0.6}) + 8f(0) + 5f(\sqrt{0}.6)\right)$, $E(f;-1,1) = \frac{h^6}{15750} f^{(6)}(\xi)$,

$-1 < \xi < 1$.

Obviously, the error estimates are better than for the Newton-Cotes formulas with the same number of nodes. Since no integration rule of type $\sum_{i=1}^{s} w_i\, f(x_i)$ can integrate exactly the function $\prod_{i=1}^{s}(x - x_i)^2 \in \mathbb{P}_{2s}$, we see that Gauss rules are best in the sense that they integrate exactly polynomials of a degree as high as possible with a formula of type $\sum_{i=1}^{s} w_i\, f(x_i)$. Therefore, Gauss formulas possess *maximum degree of precision*. There is no s-point quadrature formula with degree of precision $2s$.

Example 7.9 (Continuation of example 7.5)
We want to apply a Gauss-Legendre formula with $s = 1, 2, 3$, and 4 nodes. The nodes and weights w.r.t. $[-1, 1]$ are given in the following table.

s	nodes $x_i, i = 1, \ldots, 2s-1$	weights $w_i, i = 1, \ldots, 2s-1$
1	$x_0 = 0$	$w_0 = 2$
2	$x_{0,1} = \pm\frac{1}{\sqrt{3}}$	$w_0 = w_1 = 1$
3	$x_{0,2} = \pm\sqrt{0.6}$	$w_0 = w_2 = \frac{5}{9}$
	$x_1 = 0$	$w_1 = \frac{8}{9}$
4	$x_{0,3} = \pm 0.86113631$	$w_0 = w_3 = 0.34785485$
	$x_{1,2} = \pm 0.33998104$	$w_1 = w_2 = 0.65214515$

We obtain, observing the parameter transformation $[-1, 1] \to [0, 1]$:

$s = 1$**:** $Q(f; 0, 1) = f(1/2) = 0.8000$

$s = 2$**:** $Q(f; 0, 1) = \frac{1}{2}(f(0.7887) + f(0.2113)) = 0.7869$

$s = 3$**:** $Q(f; 0, 1) = \frac{5}{18}(f(0.8873) + f(0.1127)) + \frac{4}{9}f(0.5) = 0.7853$

$s = 4$**:** $Q(f; 0, 1) = 0.1739(f(0.9306) + f(0.6943)) + 0.3261(f(0.6700) + f(0.3300)) = 0.7854.$

The corresponding integration errors are $1.5 \cdot 10^{-2}$, $1.5 \cdot 10^{-3}$, $1.3 \cdot 10^{-4}$, and $4.8 \cdot 10^{-6}$, respectively.

Exercise 7.10
Find an approximation to

$$I = \int_1^3 \frac{\sin^2 x}{x} \mathrm{d}x$$

using Gaussian quadrature with $s = 3$ nodes. Evaluate the same integral but now using composed Gaussian quadrature with $m = 2$ subintervals and $s = 3$ nodes each.

7.8 Numerical integration error – Details

We want to derive expressions for the numerical integration error $E(f; a, b)$ for an interpolatory integration formula with $DOP = d$. We consider

$$\int_a^b f(x)\,\mathrm{d}x = \sum_{i=0}^{n} w_i\, f(x_i) + E(f; a, b), \tag{7.73}$$

where $a \le x_0 < x_1 < \ldots < x_n \le b$ are the nodes and $\{w_i\}$ are the weights of the integration formula and E is the integration error. Hence,

$$E(f; a, b) = \int_a^b f(x)\,\mathrm{d}x - \sum_{i=0}^{n} w_i\, f(x_i), \tag{7.74}$$

and

$$E(x^q; a, b) = 0, \quad \text{for all } q \le d, \tag{7.75}$$

and

$$E(x^{q+1}; a, b) \neq 0. \tag{7.76}$$

Suppose $f \in C^{d+1}([a, b])$. Then, for any value of x and $\bar{x} \in [a, b]$, we can write

$$f(x) = f(\bar{x}) + \frac{f'(\bar{x})}{1!}(x - \bar{x}) + \ldots + \frac{f^{(d)}(\bar{x})}{d!}(x - \bar{x})^d + \frac{f^{(d+1)}(\xi)}{(d+1)!}(x - \bar{x})^{d+1}, \tag{7.77}$$

where for any fixed $\bar{x} \in [a, b]$, ξ depends on x, but is in $[a, b]$. We write the last equation as

$$f(x) = p_d(x) + e_d(x), \tag{7.78}$$

where $p_d(x)$ comprises the first $d + 1$ terms, i.e., is a polynomial of degree d. Hence,

$$\begin{aligned} E(f; a, b) &= \int_a^b p_d(x)\,\mathrm{d}x + \int_a^b e_d(x)\,\mathrm{d}x - \sum_{i=0}^{n} w_i \Big(p_d(x_i) - e_d(x_i) \Big) \\ &= \int_a^b p_d(x)\,\mathrm{d}x - \sum_{i=0}^{n} w_i\, p_d(x_i) + \int_a^b e_d(x)\,\mathrm{d}x - \sum_{i=0}^{n} w_i\, e_d(x_i) \\ &= \int_a^b e_d(x) - \sum_{i=0}^{n} w_i\, e_d(x_i), \end{aligned}$$

because $DOP = d$. Inserting the expression for $e_d(x)$ yields

$$E(f; a, b) = \int_a^b \frac{f^{(d+1)}(\xi)}{(d+1)!}\,(x - \bar{x})^{d+1}\,\mathrm{d}x - \sum_{i=0}^{n} w_i \frac{f^{(d+1)}(\xi_i)}{(d+1)!}\,(x_i - \bar{x})^{d+1}, \tag{7.79}$$

where $\xi,\ \xi_0,\ \xi_1,\ \ldots\ \xi_n$ all lie in $[a, b]$. Let

$$M := \max_{x \in [a,b]} |f^{(d+1)}(x)| \tag{7.80}$$

and notice that

$$|x - \bar{x}| \le \frac{b - a}{2} \tag{7.81}$$

in $[a, b]$ when we choose $\bar{x} = \frac{a+b}{2}$. Then,

$$|E(f; a, b)| \le \frac{M(b - a)^{d+1}}{2^{d+1}(d + 1)!} \Big[\int_a^b \mathrm{d}x + \sum_{i=0}^{n} |w_i| \Big] \le \frac{M(b - a)^{d+1}}{2^{d+1}(d + 1)!} \Big[(b - a) + \sum_{i=0}^{n} |w_i| \Big]. \tag{7.82}$$

When all the weights w_i are *positive*, we have

$$\sum_{i=0}^{n} |w_i| = \sum_{i=0}^{n} w_i = b - a, \tag{7.83}$$

because any interpolatory rule integrates a constant exactly, i.e., the sum of all weights must be equal to $b-a$. Hence, for integration rules with positive weights, it holds

$$|E(f;a,b)| \leq \frac{M(b-a)^{d+2}}{2^d(d+1)!}. \tag{7.84}$$

The error bounds (7.82) and (7.84) are often too conservative. A more useful form is obtained when we start with the interpolation error: let

$$f(x) = \sum_{i=0}^{n} l_i(x)\, f(x_i) + \frac{1}{(n+1)!}\omega_{n+1}(x) f^{(n+1)}(\xi), \tag{7.85}$$

where $w_{n+1}(x)$ is the nodal polynomial of degree $n+1$ and

$$\min_{i=0,\ldots,n}(x_i,x) \leq \xi \leq \max_{i=0,\ldots,n}(x_i,x). \tag{7.86}$$

Then,

$$\int_a^b f(x)\,\mathrm{d}x = \sum_{i=0}^{n} w_i\, f(x_i) + \frac{1}{(n+1)!}\int_a^b \omega_{n+1}(x)\, f^{(n+1)}(\xi)\,\mathrm{d}x, \tag{7.87}$$

where

$$w_i = \int_a^b l_i(x)\,\mathrm{d}x. \tag{7.88}$$

Note that $\xi = \xi(x)$, i.e., we can't take the term $f^{(n+1)}(\xi)$ out of the integral. Hence, even if $f^{(n+1)}(x)$ is known analytically, we can't evaluate the second term on the right-hand side, i.e., the integration error, analytically. However, with

$$M := \max_{x\in[a,b]} |f^{(n+1)}(x)|, \tag{7.89}$$

we obtain the estimate

$$|E(f;a,b)| \leq \frac{M}{(n+1)!}\int_a^b |\omega_{n+1}(x)|\,\mathrm{d}x. \tag{7.90}$$

If no one of the abscissas x_i lies in (a,b), for instance if we want to compute the integral

$$\int_{x_i}^{x_{i+1}} f(x)\,\mathrm{d}x, \tag{7.91}$$

the nodal polynomial $\omega_{n+1}(x)$ does not change sign in (a,b) and the *second law of the mean* may be invoked to show that

$$E(f;a,b) = \frac{f^{(n+1)}(\eta)}{(n+1)!}\int_a^b \omega_{n+1}(x)\,\mathrm{d}x. \tag{7.92}$$

Definition 7.11 (Second law of the mean)
If $f \in C([a,b])$ and g does not change sign inside $[a,b]$, then

$$\int_a^b f(x)\, g(x)\, \mathrm{d}x = f(\eta) \int_a^b g(x)\, \mathrm{d}x \tag{7.93}$$

for at least one η such that $a < \eta < b$.

If the nodes are distributed *equidistantly* over $[a,b]$, we can obtain other error estimates. First observe that if $x = a + \frac{b-a}{n}\, s$, $n \in \mathbb{N}_+$, then

$$\int_a^b f(x)\, \mathrm{d}x = \frac{b-a}{n} \int_0^n F(s)\, ds, \tag{7.94}$$

where

$$F(s) = f\left(a + \frac{b-a}{n}\, s\right). \tag{7.95}$$

Suppose $f(x)$ is interpolated by a polynomial of degree n, which agrees with $f(x)$ at the $n+1$ equally spaced nodes x_i in $[a,b]$. Then,

$$\int_a^b F(s)\, ds = \sum_{i=0}^{n} w_i\, F(i) + E_n(F;0,n), \tag{7.96}$$

where

$$w_i = \int_a^b l_i(x)\, \mathrm{d}x = \int_0^n L_i(s)\, ds, \tag{7.97}$$

where

$$L_i(s) = \prod_{j=0, j\neq i} \frac{s-j}{i-j} \tag{7.98}$$

and

$$E_n(F;0,n) = \frac{1}{(n+1)!} \int_0^n s(s-1)\cdots(s-n)\, F^{(n+1)}(\xi_1)\, ds, \quad 0 < \xi_1 < n. \tag{7.99}$$

Since $s(s-1)\cdots(s-n)$ is not of constant sign on $[0,n]$, we can't apply the second law of the mean. However, it can be shown that when n *is odd*, the error can be expressed in the form which would be obtained if the second law of the mean could be applied:

$$E_n(F;0,n) = \frac{F^{(n+1)}(\xi_2)}{(n+1)!} \int_0^n s(s-1)\cdot(s-n)\, ds, \ n \text{ odd}. \tag{7.100}$$

Moreover, it can be shown that when n *is even*, it holds

$$E_n(F;0,n) = \frac{F^{(n+2)}(\xi_2)}{(n+2)!} \int_0^n (s - \frac{n}{2})\, s(s-1)\cdots(s-n)\, ds, \ n \text{ even}, \tag{7.101}$$

where in both cases $0 < \xi_2 < n$. Since we assumed equally spaced nodes, we may write if

$$h = \frac{b-a}{n}, \quad x_i = a + ih,\ i = 0, \ldots, n : \tag{7.102}$$

$$E_n(f; a, b) = \frac{h^{n+2} f^{(n+1)}(\xi)}{(n+1)!} \int_0^n s(s-1)\cdots(s-n)\, ds,\ n \text{ odd} \tag{7.103}$$

and

$$E_n(f; a, b) = \frac{h^{n+3} f^{(n+2)}(\xi)}{(n+2)!} \int_0^n (s - \frac{n}{2})\, s(s-1)\cdots(s-n)\, ds,\ n \text{ even}, \tag{7.104}$$

where $a < \xi < b$ in each case.

We return to the integration error $E(f; a, b)$ for any interpolatory quadrature rule. Instead of

$$e_d(x) = \frac{f^{(d+1)}(\xi)}{(d+1)!}(x - \bar{x})^{d+1}, \tag{7.105}$$

we use the *integral remainder term* for the Taylor series:

$$e_d(x) = \frac{1}{d!} \int_{\bar{x}}^x (x-s)^d f^{(d+1)}(s)\, ds. \tag{7.106}$$

Let $\bar{x} = a$. Then,

$$d!\, E(f; a, b) = \int_a^b \int_a^x (x-s)^d\, f^{(d+1)}(s)\, ds\, \mathrm{d}x - \sum_{i=0}^n w_i \int_a^{x_i} (x_i - s)^d\, f^{(d+1)}(s)\, ds. \tag{7.107}$$

It will be convenient to remove the x from the upper limit of the interior integral above. To this end we introduce the *truncated power function*

$$(x-s)_+^d := \begin{cases} (x-s)^d & \text{when } x \geq s \\ 0 & \text{when } x < s \end{cases}. \tag{7.108}$$

Since $(x-s)_+^d = 0$ for $s > x$, we have

$$e_d(x) = \frac{1}{d!} \int_a^x (x-s)^d f^{(d+1)}(s)\, ds = \frac{1}{d!} \int_a^b (x-s)_+^d f^{(d+1)}(s)\, ds, \tag{7.109}$$

and so

$$\begin{aligned} d!\, E(f; a, b) &= \int_a^b \Big[\int_a^b (x-s)_+^d\, f^{(d+1)}(s)\, ds \Big] \mathrm{d}x - \sum_{i=0}^n w_i \int_a^b (x_i - s)_+^d f^{(d+1)}(s)\, ds \\ &= \int_a^b f^{(d+1)}(s) \Big[\int_a^b (x-s)_+^d\, \mathrm{d}x \Big]\, ds - \int_a^b f^{(d+1)}(s) \sum_{i=0}^n w_i (x_i - s)_+^d\, ds \\ &= \int_a^b f^{(d+1)}(s) \Big[\int_a^b (x-s)_+^d\, \mathrm{d}x - \sum_{i=0}^n w_i (x_i - s)_+^d \Big]\, ds \\ &= \int_a^b f^{(d+1)}(s)\, E((x-s)_+^d; a, b)\, ds. \end{aligned}$$

We have just proved the *Peano kernel theorem* (in fact, the full theorem is a bit more general than what we proved here, though our development is sufficient for analysis of Newton-Cotes rules).

Theorem 7.12 (Peano kernel theorem)
Suppose $f \in C^{d+1}([a,b])$ *and let* $Q(f;a,b) = \sum_{i=0}^{n} w_i f(x_i)$ *be a quadrature rule with DOP* d. *Then,*

$$E(f;a,b) = \frac{1}{d!}\int_a^b f^{(d+1)}(s)\,K(s)\,ds, \tag{7.110}$$

where

$$K(s) = E((x-s)_+^d;a,b). \tag{7.111}$$

This result may appear somewhat ponderous, but it captures a beautiful idea: the error in the quadrate formula of f is related to the integral of a function $K(s)$ (called the *Peano kernel* or the *influence function*) that is independent of f.

In typical circumstances, $K(s)$ does not change sign on $[a,b]$, so the second law of the mean can be applied to extract f from inside the integral. This allows $E(f;a,b)$ to be described as

$$E(f;a,b) = \kappa\, f^{(d+1)}(\xi), \quad a < \xi < b, \tag{7.112}$$

where κ is *independent* of $f(x)$. Hence, to calculate κ, we may make the special choice $f(x) = x^{d+1}$. Then,

$$E(x^{d+1};a,b) = \kappa\,(d+1)!, \tag{7.113}$$

which fixes κ, hence,

$$E(f;a,b) = \frac{E(x^{d+1};a,b)}{(d+1)!} f^{(d+1)}(\xi), \quad a < \xi < b \tag{7.114}$$

if K does not change sign in $[a,b]$. Note that positive weights w_i is not sufficient to guarantee that K does not change sign in $[a,b]$.

7.9 Two-dimensional integration

The problem of two-dimensional integration can be formulated as follows:

Let B define a fixed closed region in $\mathbb{R}^2$ and $\mathrm{d}B = \mathrm{d}x\mathrm{d}y$ the surface element. Find fixed points $(x_1,y_1),(x_2,y_2),\ldots,(x_s,y_s)$ and fixed weights $w_1,w_2,\ldots,w_s$ such that

$$\iint_B w(x,y)f(x,y)\mathrm{d}x\mathrm{d}y \approx \sum_{i=1}^{s} w_i f(x_i,y_i) \tag{7.115}$$

is a useful approximation to the integral on the left for a reasonable large class of functions of two variables defined over B.

In passing from one-dimensional integration to two-dimensional integration a series of new problems arise since the diversity of integrals and the difficulty in handling them greatly increases.

(1) In one dimension only three different types of integration domains are possible: the finite interval (which has been discussed before), the single infinite interval, and the double infinite interval. In two dimensions we have to deal with a variety of domains.

(2) The behaviour of functions of two variables can be considerably more complicated than that of functions of one variable and the analysis of them is limited.

(3) The evaluation of functions of two variables takes much more time.

Only for some *standard regions* the theory is developed much further and the following brief discussion will be restricted to the two most important standard regions, namely

(a) the unit square $U := \{(u, v) : -1 \leq u, v \leq 1\}$

(b) the standard triangle $T := \{(u, v) : u, v \geq 0,\ u + v \leq 1\}$

They allow to transform two-dimensional integrals into two iterated integrals. Arbitrary domains B can sometimes be transformed into a standard region D by affine or other transformations. Then, given cubature formulas for D can be transformed as well providing a cubature formula for B.

To explain this, let B be a region in the x, y-plane and D be our standard region in the u, v-plane. Let the regions B and D be related to each other by means of the transformation

$$\begin{aligned} x &= \phi(u, v), \\ y &= \psi(u, v). \end{aligned}$$

We assume that $\phi, \psi \in C^1(D)$ and that the *Jacobian*

$$J(u, v) = \det \begin{pmatrix} \frac{\partial \phi}{\partial u} & \frac{\partial \phi}{\partial v} \\ \frac{\partial \psi}{\partial u} & \frac{\partial \psi}{\partial v} \end{pmatrix} \tag{7.116}$$

does not vanish in D. Suppose further that a cubature formula for the standard domain D is available, i.e.

$$\iint_D h(u, v) \mathrm{d}u\mathrm{d}v \approx \sum_{i=1}^{s} w_i \, h(u_i, v_i), \qquad (u_i, v_i) \in D, \tag{7.117}$$

with given nodes (u_i, v_i) and given weights w_i. Now we have

$$\begin{aligned} \iint_B f(x,y)\mathrm{d}x\mathrm{d}y &= \iint_D f(\phi(u,v),\psi(u,v))\,|J(u,v)|\mathrm{d}u\mathrm{d}v && (7.118)\\ &\approx \sum_{i=1}^{s} w_i\, f(\phi(u_i,v_i),\psi(u_i,v_i))\,|J(u_i,v_i)| && (7.119)\\ &=: \sum_{i=1}^{s} W_i\, f(x_i,y_i), && (7.120) \end{aligned}$$

where

$$x_i = \phi(u_i,v_i), \quad y_i = \psi(u_i,v_i), \quad W_i = w_i\,|J(u_i,v_i)|.$$

Thus,

$$\iint_B f(x,y)\mathrm{d}x\mathrm{d}y \approx \sum_{i=1}^{s} W_i\, f(x_i,y_i). \tag{7.121}$$

An important special case occurs when B and D are related by a non-singular *affine* transformation:

$$\begin{aligned} x &= a_0 + a_1 u + a_2 v,\\ y &= b_0 + b_1 u + b_2 v. \end{aligned}$$

Then, the Jacobian is *constant*:

$$|J(u,v)| = \left|\det\begin{pmatrix} a_1 & a_2 \\ b_1 & b_2 \end{pmatrix}\right| = |a_1 b_2 - a_2 b_1| \neq 0. \tag{7.122}$$

Examples are the parallelogram which is an affine transformation of the standard square U and any triangle which is an affine transformation of the standard triangle T.

Example 7.13
Given the s nodes (u_i, v_i) and weights w_i of a cubature formula w.r.t. to the standard triangle $T = \{(u,v) : 0 \le u \le 1, 0 \le v \le 1-u\}$. Calculate the nodes (x_i, y_i) and weights W_i of a cubature formula

$$Q(f;B) = \sum_{i=1}^{s} W_i\, f(x_i,y_i) \approx \int_B f(x,y)\mathrm{d}x\,\mathrm{d}y,$$

where $B = \{(x,y) : 0 \le x \le 1, 0 \le y \le x\}$.

First of all we have to find the transformation which maps T onto B. Since both are triangles, the transformation is an affine transformation which can be written as

$$x = a_0 + a_1\, u + a_2\, v; y = b_0 + b_1\, u + b_2\, v.$$

The coefficients are determined using the identical points $(u = 0, v = 0) \to (x = 0, y = 0)$, $(u = 1, v = 0) \to (x = 1, y = 0)$, *and* $(u = 0, v = 1) \to (x = 1, y = 1)$. *This yields* $x = u + v, y = v$. *The Jacobian of the parameter transformation is equal to* 1, *i.e.* $\mathrm{d}x\,\mathrm{d}y = \mathrm{d}u\,\mathrm{d}v$. *Now, we obtain*

$$\begin{aligned}\int_B f(x,y)\mathrm{d}x\,\mathrm{d}y &= \int_T f(x(u,v),y(u,v))\mathrm{d}u\,\mathrm{d}v \\ &\approx \sum_{i=1}^{s} w_i\, f(x(u_i,v_i),y(u_i,v_i)) = \sum_{i=1}^{s} W_i\, f(x_i,y_i) =: Q(f;B).\end{aligned}$$

Therefore the nodes of $Q(f;B)$ *are* $x_i = u_i + v_i, y_i = v_i, i = 1,\ldots,s$ *and the weights are* $W_i = w_i, i = 1,\ldots,s$.

Exercise 7.14
Given a triangle B *with vertices* $(1,1)$, $(2,2)$, *and* $(2,3)$. *Determine the transformation which maps the standard triangle* T *onto* B. *Let* (u_i, v_i) *and* w_i, $i = 1,\ldots,s$ *denote the nodes and weights of a cubature formula* $Q(f;T) = \sum_{i=1}^{s} w_i\, f(u_i,v_i)$ *which approximates the integral* $\int_T f(u,v)\,\mathrm{d}u\,\mathrm{d}v$. *Use these nodes and weights and derive a cubature formula* $Q(g;B)$ *to approximate* $\int_B g(x,y)\mathrm{d}x\,\mathrm{d}y$.

7.9.1 Cartesian products and product rules

Let I_1, I_2 be intervals. The symbol $I_1 \times I_2$ denotes the Cartesian product of I_1 and I_2, and by this we mean the region in $\mathbb{R}^2$ with points (x,y) that satisfy $x \in I_1, y \in I_2$.

Example 7.15
$I_1 : -1 \le x \le 1, I_2 : -1 \le y \le 1$. *Then* $I_1 \times I_2$ *is the unit square* $Q : \{(x,y) : -1 \le x, y \le 1\}$.

Suppose now that Q_1 is an s_1-point rule of integration over I_1 and Q_2 is an s_2-point rule of integration over I_2:

$$Q_1(h,I_1) = \sum_{i=1}^{s_1} w_i\, h(x_i) \approx \int_{I_1} h\,\mathrm{d}x, \tag{7.123}$$

$$Q_2(g,I_2) = \sum_{i=1}^{s_2} v_i\, g(y_i) \approx \int_{I_2} g\,\mathrm{d}y. \tag{7.124}$$

w_i and v_i denote the weights of the quadrature formulas Q_1 and Q_2, respectively. Then, by the product rule of Q_1 and Q_2 we mean the $s_1 \cdot s_2$-point rule applicable to $I_1 \times I_2$ and defined by

$$Q_1 \times Q_2(f; I_1 \times I_2) = \sum_{i=1}^{s_1}\sum_{j=1}^{s_2} w_i\, v_j\, f(x_i,y_j) \tag{7.125}$$

$$\approx \iint\limits_{I_1\times I_2} f(x,y)\,\mathrm{d}x\mathrm{d}y. \tag{7.126}$$

We can show that if Q_1 integrates $h(x)$ exactly over I_1, if Q_2 integrates $g(y)$ exactly over I_2, and if $f(x,y) = h(x)g(y)$, then $Q_1 \times Q_2$ will integrate $f(x,y)$ exactly over $I_1 \times I_2$.

Example 7.16
Let B be the rectangle $a \le x \le b$, $c \le y \le d$. The evaluation of

$$\iint_B f(x,y)\mathrm{d}x\mathrm{d}y$$

by the product of two Simpson's rules yields

$$\begin{aligned}\iint_B f(x,y)\mathrm{d}x\mathrm{d}y \approx \frac{(b-a)(c-d)}{36}\Bigg[&f(a,c)+f(a,d)+f(b,c)+f(b,d)\\ &+4\left(f\left(a,\frac{c+d}{2}\right)+f\left(\frac{a+b}{2},c\right)+f\left(b,\frac{c+d}{2}\right)+f\left(\frac{a+b}{2},d\right)\right)\\ &+16f\left(\frac{a+b}{2},\frac{c+d}{2}\right)\Bigg]. \quad (7.127)\end{aligned}$$

This 9-point rule will integrate exactly all linear combinations of the 16 monomials *$x^i y^j$, $0 \le i,j \le 3$. The corresponding weights are the products of the weights of the two Simpson rules.*

Example 7.17
Given the s nodes and weights of a Gauss-Legendre quadrature formula w.r.t. the standard interval $[-1,1]$. Construct a cubature formula to compute the integral

$$I := \int_D f(x,y)\,\mathrm{d}D,$$

with $D := \{(x,y) : 0 \le x^2 + y^2 \le 1\}$.

D is the unit circle. Introducing polar coordinates $x = r\cos\alpha$, $y = r\sin\alpha$, we can transform D into the rectangle $R := \{(r,\alpha) : 0 \le r \le 1, 0 \le \alpha \le 2\pi\}$. The Jacobian of this transformation is $J_1(r,\alpha) = r$. Therefore, we obtain

$$I = \int_D f\mathrm{d}D = \int_R f|J_1|\,\mathrm{d}R,$$

with $\mathrm{d}R = \mathrm{d}r\,\mathrm{d}\alpha$. The affine transformation $r = \frac{1}{2}(u+1)$, $\alpha = \pi(v+1)$ transforms the unit square $U = [-1,1]\times[-1,1]$ onto R. Its Jacobian is $J_2(u,v) = \frac{\pi}{2}$. Therefore, we obtain

$$I = \int_R f|J_1|\mathrm{d}R = \int_U f|J_1||J_2|\mathrm{d}U,$$

with $dU = \mathrm{d}u\,\mathrm{d}v$. Let $g := f|J_1||J_2|$. Then

$$I = \int_{u=-1}^{u=1}\int_{v=-1}^{v=1} g(u,v)\mathrm{d}u\,\mathrm{d}v.$$

Let $\int_{v=-1}^{v=1} g(u,v)\mathrm{d}v =: h(u)$. *Then, we may write*

$$I = \int_{u=-1}^{u=1} h(u)\mathrm{d}u \approx \sum_{i=1}^{s} w_i\, h(u_i),$$

where u_i *and* w_i *denote the nodes and weights of the Gauss-Legendre quadrature w.r.t. the interval* $[-1,1]$. *Because of the definition of* $h(u)$ *we obtain*

$$I \approx \sum_{i=1}^{s} w_i\, h(u_i) = \sum_{i=1}^{s} w_i \int_{v=-1}^{v=1} g(u_i,v)\,\mathrm{d}v.$$

For fixed i, $g(u_i,v)$ *is a function of* v *only, thus we can again apply the Gauss-Legendre quadrature formula to compute the integral:*

$$\int_{v=-1}^{v=1} g(u_i,v)\,\mathrm{d}v \approx \sum_{j=1}^{s} w_j\, g(u_i,v_j).$$

This yields

$$I \approx \sum_{i=1}^{s}\sum_{j=1}^{s} w_i\, w_j\, g(u_i,v_j),$$

with $g = f\,|J_1|\,|J_2|$, $x = \frac{u+1}{2}\cos(\pi(v+1))$, $y = \frac{u+1}{2}\sin(\pi(v+1))$, $r = \frac{1}{2(u+1)}$, *and* $\alpha = \pi(v+1)$.

We can generalize the product rule approach. If the region B is sufficiently simple, the integral

$$I = \iint_B f(x,y)\,\mathrm{d}x\mathrm{d}y$$

may be expressed as an iterated integral of the form

$$I = \int_a^b \int_{u_1(x)}^{u_2(x)} f(x,y)\,\mathrm{d}x\mathrm{d}y =: \int_a^b g(x)\,\mathrm{d}x, \qquad g(x) := \int_{u_1(x)}^{u_2(x)} f(x,y)\,\mathrm{d}y \tag{7.128}$$

Let us use an s_1-point rule Q_1 for integrating $g(x)$:

$$\int_a^b g(x)\mathrm{d}x \approx \sum_{i=1}^{s_1} w_i\, g(x_i) = \sum_{i=1}^{s_1} w_i \int_{u_1(x_i)}^{u_2(x_i)} f(x_i,y)\mathrm{d}y \tag{7.129}$$

For each of the s_1 integrals we use an s_2-point rule Q_2

$$\int_{u_1(x_i)}^{u_2(x_i)} f(x_i,y)\mathrm{d}y \approx \sum_{j=1}^{s_2} v_{ji}\, f(x_i,y_{ji}) \tag{7.130}$$

The double subscripts on the right reflect the fact that the abscissas and weights of Q_2 must be adjusted for each value of i to the interval $u_1(x_i) \le y \le u_2(x_i)$. Thus

$$I = \sum_{i=1}^{s_1} w_i \sum_{j=1}^{s_2} v_{ji}\, f(x_i,y_{ji}) \tag{7.131}$$

is an $s_1 \cdot s_2$-point rule for I. Note that this is the same formula that results when applying the product rule $Q_1 \times Q_2$ to the transformed integral over the square $[a,b] \times [a,b]$ with Jacobian

$$J(x) = \frac{u_2(x) - u_1(x)}{b-a}.$$

7.9.2 Some remarks on 2D-interpolatory formulas

Univariate interpolatory integration formulas with prescribed nodes $x_1, \ldots, x_s$ can be obtained by requiring $Q(f;a,b)$ to integrate all polynomials in $\mathbb{P}_{s-1}$ exactly. All interpolatory quadrature formulas Q have a degree of precision $d \geq s-1$, where the actual degree of precision d depends on the specific abscissas $x_1, \ldots, x_s$. The largest possible degree of precision, $d = 2s-1$, is attained for Gauss formulas, whereas Newton-Cotes formulas only provide $d = s-1$ and $d = s$, depending on whether s is even or odd. It seems reasonable to construct *multivariate* integration formulas Q by generalizing this univariate approach, i.e. by requiring Q to integrate all polynomials of total degree d exactly. That means that for s given distinct nodes $(x_1,y_1), \ldots, (x_s,y_s)$ the weights $w_1, \ldots, w_s$ of the cubature formula

$$Q(f;I) = \sum_{i=1}^{s} w_i \, f(x_i, y_i)$$

have to be chosen in such a way that

$$E(f;I) = 0 \qquad \text{for all } f \in \mathbb{P}_d. \tag{7.132}$$

The space $\mathbb{P}_d$ has dimension $M := \binom{2+d}{d}$, i.e. all functions from $\mathbb{P}_d$ can be written as a linear combination of $\binom{2+d}{d}$ basis functions ϕ_i. Then (7.132) is equivalent to

$$E(\phi_i;I) = 0, \qquad i = 1, \ldots, M. \tag{7.133}$$

Equations (7.133) are called *moment equations*. For a fixed choice of the basis $\{\phi_1, \ldots, \phi_M\}$ and of the abscissas $(x_1,y_1), \ldots, (x_s,y_s)$ they define a system of linear equations

$$\sum_{i=1}^{s} w_i \, \phi_j(x_i, y_i) = I\phi_j, \qquad j = 1, \ldots, M. \tag{7.134}$$

Then, Q is called an interpolatory cubature formula, provided the system has a unique solution.

In one dimension, Q is identical to the quadrature formula obtained by interpolation with polynomials from $\mathbb{P}_d$ at the distinct nodes $x_1 \ldots, x_s$. This relationship does not generally hold in two (or more) dimensions!

For arbitrary given distinct points $(x_1,y_1), \ldots, (x_s,y_s)$ the moment equations usually have no unique solution. Thus, when trying to construct s-point interpolatory cubature formulas, the linear system has to be solved not only for the weights $w_1, \ldots, w_s$, but also

for the nodes $(x_1, y_1), \ldots, (x_s, y_s)$. Then the system (7.134) is non-linear in the unknowns $(x_1, y_1), \ldots, (x_s, y_s)$. Each node (x_i, y_i) introduces three unknowns: the weight w_i and the two coordinates of each node (x_i, y_i). Therefore, the cubature formula Q to be constructed has to satisfy a system of $\binom{d+2}{d}$ non-linear equations in $3s$ unknowns. For non-trivial values of s, these non-linear equations are too complex to be solved directly.

However, Chakalov's theorem guarantees at least that such a cubature formula exists:

Theorem 7.18 (Chakalov)
Let B be a closed bounded region in $\mathbb{R}^2$ and let W be a non-negative weight function on B. Then there exist $s = \binom{d+2}{d}$ points $(x_1, y_1), \ldots, (x_s, y_s)$ in B and corresponding positive weights $w_1, \ldots, w_s$ so that the corresponding cubature formula Q has a degree of precision d.

Unfortunately, Chakalov's theorem is not constructive, and therefore useless for the practical construction of cubature formulas. Therefore, the construction of interpolatory cubature formulas is an area of ongoing research. For most of the practical applications, however, product rules as discussed in section 7.9.1 can be derived.

Chapter 8

Numerical Methods for Solving Ordinary Differential Equations

8.1 Introduction

Many problems in technical sciences result in the task to look for a differentiable function $y = y(x)$ of one real variable x, whose derivative $y'(x)$ fulfils an equation of the form

$$y'(x) = f(x, y(x)), \quad x \in I = [x_0, x_n]. \tag{8.1}$$

Equation (8.1) is called *ordinary differential equation*. Since only the *first* derivative of y occurs, the ordinary differential equations (8.1) is of *first order*. In general it has infinitely many solutions $y(x)$. Through additional conditions one can single out a specific one. Two types of conditions are commonly used, the so-called *initial condition*

$$y(x_0) = y_0, \tag{8.2}$$

i.e. the value of y at the initial point x_0 of I is given, and the so-called *boundary condition*

$$g(y(x_0), y(x_n)) = 0, \tag{8.3}$$

where g is a function of two variables. Equations (8.1),(8.2) define an *initial value problem* and equations (8.1),(8.3) define a *boundary value problem*.

More generally, we can also consider *systems* of p ordinary differential equations of first order

$$\begin{array}{rcl} y_1'(x) & = & f_1(x, y_1(x), y_2(x), \ldots, y_p(x)) \\ y_2'(x) & = & f_2(x, y_1(x), y_2(x), \ldots, y_p(x)) \\ & \vdots & \\ y_p'(x) & = & f_p(x, y_1(x), y_2(x), \ldots, y_p(x)) \end{array} \tag{8.4}$$

for p unknown functions $y_i(x)$, $i = 1, \ldots, p$ of a real variable x. Such systems can be written in the form (8.1) when y' and f are interpreted as vector of functions:

$$y'(x) := \begin{bmatrix} y_1'(x) \\ \vdots \\ y_p'(x) \end{bmatrix}, \quad f(x, y(x)) := \begin{bmatrix} f_1(x, y_1(x), \ldots, y_p(x)) \\ \vdots \\ f_p(x, y_1(x), \ldots, y_p(x)) \end{bmatrix}. \tag{8.5}$$

Correspondingly the initial condition (8.2) has to be interpreted as

$$y(x_0) = y_0 = \begin{bmatrix} y_1(x_0) \\ \vdots \\ y_p(x_0) \end{bmatrix} = \begin{bmatrix} y_{1,0} \\ \vdots \\ y_{p,0} \end{bmatrix}. \tag{8.6}$$

In addition to ordinary differential equations of first order there are ordinary differential equations of the m*th order*, which have the form

$$y^{(m)}(x) = f(x, y(x), y'(x), \ldots, y^{(m-1)}(x)). \tag{8.7}$$

The corresponding inital value problem is to determine a function $y(x)$ which is m-times differentiable, satisfies (8.7), and fulfils the initial condition:

$$y^{(i)}(x_0) = y_0^{(i)}, \quad i = 0, 1, \ldots, m-1. \tag{8.8}$$

By introducing *auxiliary functions*

$$\begin{array}{rcl} z_1(x): & = & y(x) \\ z_2(x): & = & y'(x), \\ & \vdots & \\ z_m(x): & = & y^{(m-1)}(x), \end{array} \tag{8.9}$$

the ordinary differential equation of the mth order can always be transformed into an equivalent system of m first order differential equations

$$\begin{bmatrix} z_1' \\ \vdots \\ z_{m-1}' \\ z_m' \end{bmatrix} = \begin{bmatrix} z_2 \\ \vdots \\ z_m \\ f(x, z_1, z_2, \ldots, z_m) \end{bmatrix}. \tag{8.10}$$

In this chapter we will restrict to initial value problems for ordinary differential equations of first order, i.e. to the case of only *one* ordinary differential equation of first order for only *one* unknown function. However, all methods and results holds for *systems* of p ordinary differential equations of first order, as well, provided quantities such as y and $f(x, y)$ are interpreted as vectors, and $|\cdot|$ as norm $\|\cdot\|$. Moreover, we always assume that the initial value problem is uniquely solvable. This is guaranteed by the following theorem:

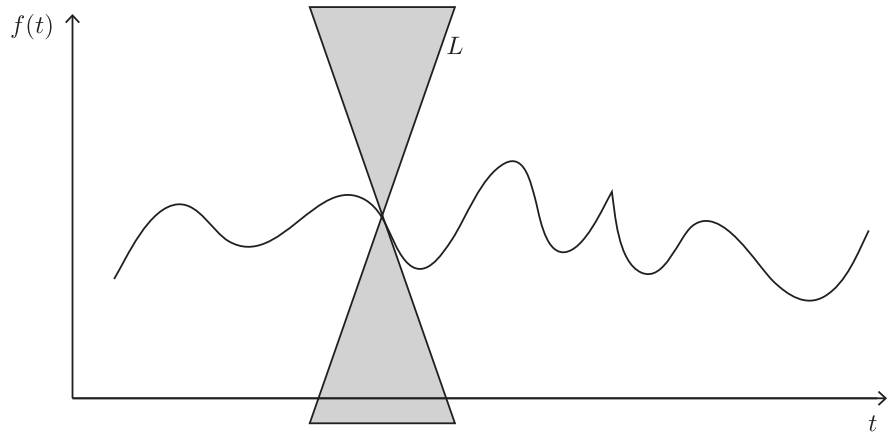


Figure 8.1: Graphical interpretation of Lipschitz continuity: finding a cone at each point on the curve such that the function does not intersect the cone.

Theorem 8.1 (Existence and uniqueness)
Let f be defined and continuous on a strip $G := \{(x, y) : a \leq x \leq b, y \in \mathbb{R}\}$, a, b finite. Further, let there be a constant L such that

$$|f(x, y_1) - f(x, y_2)| \leq L|y_1 - y_2| \tag{8.11}$$

for all $x \in [a, b]$ and all $y_1, y_2 \in \mathbb{R}$ ("Lipschitz condition"). L is called the "Lipschitz constant" of f. Then for every $x_0 \in [a, b]$ and every $y_0 \in \mathbb{R}$ there exists exactly one solution of the initial value problem (8.1),(8.2).

This theorem has a geometric interpretation illustrated in Figure 8.1: if at every point on the curve we can draw a cone (in gray) with some (finite) slope L, such that the curve does not intersect the cone, then the function is Lipschitz continuous. The function sketched in the figure is clearly Lipschitz continuous.

Condition (8.11) is fulfilled if the partial derivative $f_y := \frac{\partial f}{\partial y}$ exists on the strip G and is continuous and bounded there. Then we can choose

$$L = \max_{(x,y)\in G} \big|f_y(x, y)\big|.$$

In most applications f is continuous on G, but the partial derivative f_y is often unbounded on G. Then, the initial value problem (8.1),(8.2) is still solvable, but the solution may only be defined in a certain neighborhood of the initial point x_0 and not on all of $[x_0, x_n]$.

Most initial value problems for ordinary differential equations cannot be solved analytically. Instead, numerical methods have to be used which provide to certain abscissae ("nodes") x_i, $i = 0, \ldots, n$ approximate values $\eta(x_i)$ for the exact values $y(x_i)$. The abscissae

x_i are often equidistant, i.e. $x_i = x_0 + i\,h$, where h is the *step size*. We will discuss various numerical methods and will examine whether and how fast $\eta(x)$ converges to $y(x)$ as $h \to 0$. We will intensively make use of the results of Chapter 1 and Chapter 4.

8.2 Basic concepts and classification

Let us assume that $n+1$ nodes subdivide the integration interval $I = [x_0, x_n]$ into equally large subintervals, i.e.

$$x_i = x_0 + i\,h, \quad i = 0, \ldots, n, \tag{8.12}$$

with the fixed step size $h = \frac{x_n - x_0}{n}$. By formal integration of (8.1) over the interval $[x_i, x_{i+1}]$, $i = 0, \ldots, n-1$, we obtain

$$y(x_{i+1}) = y(x_i) + \int_{x_i}^{x_{i+1}} f(t, y(t))\,dt, \quad i = 0, \ldots, n-1. \tag{8.13}$$

The numerical methods for solving the initial value problem (8.1), (8.2) differ in what quadrature formula $Q(f, y; x_i, x_{i+1})$ is used to compute the integral on the right-hand side of equation (8.13):

$$y(x_{i+1}) = y(x_i) + Q(f, y; x_i, x_{i+1}) + E(f, y; x_i, x_{i+1}), \quad i = 0, \ldots, n-1. \tag{8.14}$$

Approximations $\bar{\eta}(x_{i+1})$ of $y(x_{i+1})$, $i = 0, \ldots, n-1$ can be obtained by neglecting the quadrature error $E(f, y; x_i, x_{i+1})$, yielding an approximation

$$\bar{\eta}(x_{i+1}) = y(x_i) + Q(f, y; x_i, x_{i+1}), \quad i = 0, \ldots, n-1. \tag{8.15}$$

The difference $y(x_{i+1}) - \bar{\eta}(x_{i+1})$ is then equal to the quadrature error $E(f, y; x_i, x_{i+1})$. In the context of ODE it is called *local discretization error* or *local truncation error* of step $i+1$ and denoted by ϵ_{i+1}:

$$\epsilon_{i+1} = E(f, y; x_i, x_{i+1}) = y(x_{i+1}) - \bar{\eta}(x_{i+1}), \quad i = 0, \ldots, n-1. \tag{8.16}$$

For applications it is important how fast the local discretization error decreases as $h \to 0$. This is expressed by the *order* of the local discretization error. It is the largest number p such that

$$\lim_{h \to 0} \frac{\epsilon_{i+1}}{h^p} < \infty. \tag{8.17}$$

We simply write $\epsilon_{i+1} = O(h^p)$ which means nothing else but $\epsilon_{i+1} \approx C\,h^p$ with some $C \in \mathbb{R}$.

Usually $y(x_i)$ is not known, and (8.15) cannot be used directly. Therefore, an additional approximation has to be introduced, for instance, by replacing the terms with y on the right-hand side of (8.15) by the corresponding approximations η, which yields:

$$\eta(x_{i+1}) = \eta(x_i) + Q(f, \eta; x_i, x_{i+1}), \quad i = 0, \ldots, n-1, \tag{8.18}$$

where $y(x_0) = \eta(x_0) = y_0$. To keep notation simple we will often write y_j, η_j instead of $y(x_j)$, $\eta(x_j)$.

Mostly we want to know how large the difference between y_{i+1} and its approximation η_{i+1} is. This question is not answered by the local discretization error because it only tells us how good the numerical integration has been performed in the corresponding step of the algorithm; the numerical integration errors of the previous steps and also the errors introduced by replacing terms with y by the approximations η on the right-hand side of (8.15) are not taken into account. The difference $y_{i+1}-\eta_{i+1}$ is called *global discretization error* or sometimes *global truncation error* at x_{i+1}, denoted by e_{i+1}:

$$e_{i+1} := y_{i+1} - \eta_{i+1}, \quad i = 0, 2, \ldots, n-1. \tag{8.19}$$

The order of the global discretization error is defined in the same way as the order of the local discretization error. It can be shown that the global order is always one less than the local order. It can be estimated using approximations with different step size: If $\eta_{h_j}(x_i)$ is the approximation to $y(x_i)$ obtained with a method of global order $O(h_j^p)$, and step size h_j, $j = 1, 2$, we can proof that

$$e_{i,h_1} = y(x_i) - \eta_{h_1}(x_i) \approx \frac{\eta_{h_1}(x_i) - \eta_{h_2}(x_i)}{1 - \left(\frac{h_1}{h_2}\right)^p} = \bar{e}_{i,h_1}.$$

This estimate of the global discretization error can be used to obtain an improved approximation $\bar{\eta}_{h_1}(x_i)$:

$$\bar{\eta}_{h_1}(x_i) = \eta_{h_1}(x_i) + \bar{e}_{i,h_1} = \frac{\left(\frac{h_2}{h_1}\right)^p \eta_{h_1}(x_i) - \eta_{h_2}(x_i)}{\left(\frac{h_2}{h_1}\right)^p - 1}.$$

The approximation $\bar{\eta}_{h_1}(x_i)$ has a global order of at least $O(h_1^{p+1})$, i.e. the order of the global discretization error is at least one higher than for the original approximation $\eta_{h_1}(x_i)$.

For the special case that $h_2 = 2h_1 = 2h$, we obtain

$$\bar{e}_{i,h}(x_i) = \frac{\eta_h(x_i) - \eta_{2h}(x_i)}{1 - \frac{1}{2}^p},$$

and

$$\bar{\eta}_h(x_i) = \frac{2^p \eta_h(x_i) - \eta_{2h}(x_i)}{2^p - 1}.$$

In addition to discretization errors, there are rounding errors which have to be taken into account. Rounding errors are unavoidable since all numerical computations on a computer are done in floating-point arithmetic with a limited number of decimal digits. They behave like

$$r_i = O(h^{-q}),$$

with some $q > 0$, where the error constant depends on the number of decimal digits in the floating-point arithmetic. Note that the rounding error increases with decreasing step size h, while the discretization error decreases with decreasing h. Therefore, the step size h has to be chosen carefully. Investigations have shown that usually the choice $hL = 0.05, \ldots, 0.20$ yields good results, where L denotes the Lipschitz constant of f (cf. (8.11)). For more information about rounding errors we refere to Stoer and Bulirsch (1993).

In this chapter we will focus on *single-step methods* and *multistep methods*. Single-step methods are of the form

$$\eta_{i+1} = \eta_i + h \cdot \Phi(\eta_{i+1}, \eta_i, x_i, h), \quad i = 0, 1, \ldots, n-1, \tag{8.20}$$

where the function Φ can be linear or non-linear in its arguments. If Φ does not depend on η_{i+1}, the method is called *explicit*, otherwise *implicit*. Characteristic for single-step methods is that they use only *one* previously calculated approximation η_i to get an approximation η_{i+1}. *Multistep methods* (i.e., s-step methods with $s > 1$) make use of $s+1$, $s \geq 1$ previously calculated approximations $\eta_{i-s}, \eta_{i-s+1}, \ldots, \eta_{i-1}, \eta_i$ to calculate η_{i+1}. The linear multistep methods have the form

$$\eta_{i+1} = \sum_{k=1}^{s} a_{s-k}\eta_{i+1-k} + h\sum_{k=0}^{s} b_{s-k}f_{i+1-k}, \quad i = 0, 1, \ldots, n-1, \tag{8.21}$$

with some real coefficients a_j, b_j. If $b_s = 0$ they are called *explicit*, otherwise *implicit*.

A third group of numerical method are so-called *extrapolation methods*, which are similar to Romberg integration methods; they are beyond the scope of this lecture. The reader is referred to Stoer and Bulirsch (1993).

Among the single-step methods we will discuss the methods of *Euler-Cauchy*, the method of *Heun*, and the *classical Runge-Kutta* method. Among the multistep methods we will discuss the method of *Adams-Bashforth* and the method of *Adams-Moulton*. In addition we will discuss a subclass of single-step and multistep methods, the so-called *predictor-corrector methods*. These are methods which first determine an approximation $\eta_{i+1}^{(0)}$ using a single-step or a multistep method; the corresponding formula is called *predictor*. Then, the approximation $\eta_{i+1}^{(0)}$ is improved using another single-step or a multistep method (the so-called "corrector"), yielding approximations $\eta_{i+1}^{(1)}$, $\eta_{i+1}^{(2)}$, ..., $\eta_{i+1}^{(k_0)}$. For instance, the methods of Heun and Adams-Moulton belong to the class of predictor-corrector methods.

What method is used to solve a given initial value problem depends on several factors, among them are the required accuracy, the computer time and memory needed, the flexibility w.r.t. the step size h, and the number of function evaluations. We will give some hints for practical applications.

8.3 Single-step methods

8.3.1 The methods of Euler-Cauchy

We start from equation (8.13) with $i = 0$ and use a closed Newton-Cotes formula with one node, namely at x_0, (see Chapter 4) to calculate the integral on the right-hand side of (8.13). We obtain:

$$y_1 = y_0 + h\, f(x_0, y(x_0)) + E(f, y; x_0, x_1). \tag{8.22}$$

The local discretization error of the first Euler-Cauchy step is simply (see Chapter 4):

$$\epsilon_1^{EC} = \frac{1}{2} h^2\, f'(\xi_0, y(\xi_0)) = \frac{1}{2} h^2\, y''(\xi_0), \quad \xi_0 \in [x_0, x_1].$$

Assuming that $f(x, y(x))$ is differentiable w.r.t. x, we obtain

$$y_1 = y_0 + h\, f(x_0, y_0) + \epsilon_1^{EC} = \eta_1 + \epsilon_1^{EC},$$

with

$$\eta_1 := y_0 + h\, f(x_0, y_0).$$

When integrating over $[x_1, x_2]$ we obtain correspondingly

$$y_2 = y_1 + \int_{x_1}^{x_2} f(t, y(t))\, dt = y_1 + h\, f(x_1, y_1) + \epsilon_2^{EC}.$$

Since y_1 is unknown, we use instead its approximation η_1 to obtain the approximation η_2:

$$\eta_2 = \eta_1 + h\, f(x_1, \eta_1).$$

The local discretization error of the second Euler-Cauchy step is:

$$\epsilon_2^{EC} = \frac{1}{2} h^2\, y''(\xi_1), \quad \xi_1 \in [x_1, x_2].$$

Continuing in the same way, we obtain, when using x_i as node of the one point Newton-Cotes formula over $[x_i, x_{i+1}]$:

$$\begin{aligned} \eta_{i+1} &= \eta_i + h\, f(x_i, \eta_i), \\ \epsilon_{i+1}^{EC} &= \frac{1}{2} h^2\, y''(\xi_i), \quad \xi_i \in [x_i, x_{i+1}], \quad i = 0, \ldots, n-1. \end{aligned} \tag{8.23}$$

The scheme (8.23) is called *forward Euler-Cauchy method*. The local discretization error has order 2, and the global discretization error has order 1. The latter means that if the step size in the Euler-Cauchy method is reduced by a factor of $\frac{1}{2}$, we can expect that the global discretization error will be reduced by a factor of $\frac{1}{2}$. Thus, the forward Euler-Cauchy method converges for $h \to 0$, i.e.,

$$\lim_{h \to 0} (y(x) - \eta(x)) = 0.$$

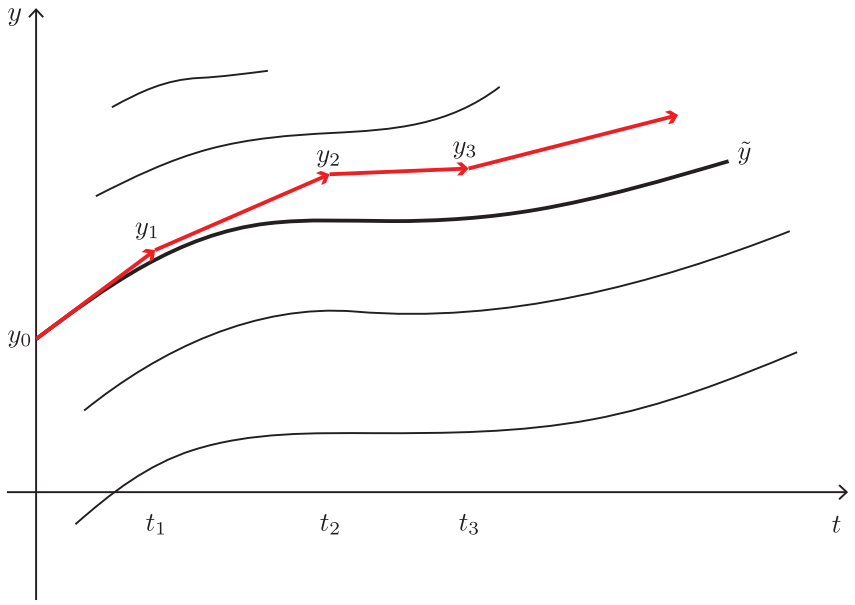


Figure 8.2: Solution space for the ODE $y' = f(t, y)$. Exact solutions with various initial conditions are plotted (black lines). The progress of 4 steps of forward Euler-Cauchy is shown (red arrows).

Graphically we can plot the path of forward Euler-Cauchy in the solution space, see Figure 8.2. The black lines in this figure correspond to exact solutions of the ODE with different initial conditions. The thick black line is the exact solution for the initial condition we are interested in, $\tilde{y}$. Using the ODE $y' = f(x, y)$ the exact derivative of the solution y can be computed at every point in this space. Forward Euler starts from the initial condition, and takes a linear step using the initial derivative. Because the exact solution is curved, forward Euler lands a little to the side of the exact solution. The next step proceeds from this new point. Over a number of steps these errors compound.

Example 8.2 (Forward Euler-Cauchy)
Given the initial value problem $y' = y$, $y(0) = 1$. We seek an approximation to $y(0.5)$ using the forward Euler-Cauchy formula with step size $h = 0.1$. Compare the approximations with the true solution at the nodes.

Solution: With $f(x, y(x)) = y$, we obtain the recurrence formula

$$\eta_{i+1} = (1 + h)\eta_i, \quad i = 0, \ldots, 5.$$

The nodes are $x_i = x_0 + ih = 0.1\,i$, $i = 0, \ldots, 5$. With $h = 0.1$ and retaining four decimal places, we obtain

i	0	1	2	3	4	5
x_i	0	0.1	0.2	0.3	0.4	0.5
η_i	1	1.1	1.21	1.331	1.4641	1.61051
y_i	1	1.1052	1.2214	1.3499	1.4918	1.6487
$\lvert y_i - \eta_i \rvert$	0	5.3(-3)	1.1(-2)	1.9 (-2)	2.8(-2)	3.8 (-2)

The exact solution of the initial value problem is $y(x) = e^x$, the correct value at $x = 0.5$ is $y(0.5) = 1.64872$. Thus, the absolute error is $e^{EC}(0.5) = 3.8(-2)$. A smaller step size yields higher accuracies. Taking e.g. $h = 0.005$, we obtain $\eta(0.5) = 1.6467$, which is correct to $2.1(-3)$.

Example 8.3 (Euler-Cauchy: Global discretization error and step size)
Consider the initial value problem $y' = \frac{x-y}{2}$, $y(0) = 1$ over the interval $[0, 3]$ using the forward Euler-Cauchy method with step sizes $h = 1, \frac{1}{2}, \ldots, \frac{1}{64}$ and calculate the global discretization error. The exact solution is $y(x) = 3e^{-\frac{x}{2}} - 2 + x$. The results clearly show that the error in

step size h	number of steps n	$\eta(3)$	$e^{EC}(0.3)$	$O(h) \approx Ch,\ C = 0.256$
1	3	1.375	0.294390	0.256
$\frac{1}{2}$	6	1.533936	0.135454	0.128
$\frac{1}{4}$	12	1.604252	0.065138	0.064
$\frac{1}{8}$	24	1.637429	0.031961	0.032
$\frac{1}{16}$	48	1.653557	0.015833	0.016
$\frac{1}{32}$	96	1.661510	0.007880	0.008
$\frac{1}{64}$	192	1.665459	0.003931	0.004

the approximation of $y(3)$ descreases by about $\frac{1}{2}$ when the step size h is reduced by $\frac{1}{2}$. The error constant can be estimated to $C \approx 0.256$.

Instead of using the node x_i we may use x_{i+1} yielding the so-called *backward Euler-Cauchy formula*:

$$\eta_{i+1} = \eta_i + h\, f(x_{i+1}, \eta_{i+1}), \quad i = 0, 1, \ldots, n-1. \tag{8.24}$$

Obviously it is an *implicit* formula since η_{i+1} also appears on the right-hand side of the equation, in contrast to the explicit forward Euler-Cauchy formula (8.23). For general f we have to solve for the solution η_{i+1} by iteration. Since equation (8.24) is of the form

$$\eta_{i+1} = \varphi(\eta_{i+1}),$$

we can do that with the aid of the fixed-point iteration of Chapter 1. That means we start with an approximation $\eta_{i+1}^{(0)}$, e.g. $\eta_{i+1}^{(0)} = \eta_i + h\, f(x_i, \eta_i)$, and solve the equation iteratively:

$$\eta_{i+1}^{(k+1)} = \varphi\left(\eta_{i+1}^{(k)}\right), \quad k = 0, 1, \ldots, k_0.$$

It can be shown that under the conditions of Theorem 8.1 and for $hL = 0.05, \ldots, 0.20$ the iteration converges. Mostly, one iteration step is already sufficient.

Exercise 8.4 (Backward Euler-Cauchy)
Given the initial value problem $y' = y,\ y(0) = 1$. Compute an approximation $\eta(0.5)$ to $y(0.5)$ using the backward Euler-Cauchy method with step size $h = 0.1$. Compare the results with the forward Euler-Cauchy formula w.r.t. accuracy. Calculate the absolute error for each step.

8.3.2 The method of Heun

The Euler-Cauchy method makes use of the most simple quadrature formula, namely the closed one point Newton-Cotes formula with the node at the lower or upper bound of the integration domain. Higher-order methods can easily be obtained by chosing more accurate quadrature formulas. For instance, when using the trapezoidal rule (see Chapter 4), we obtain

$$y_{i+1} = y_i + \frac{h}{2}\left(f(x_i, y_i) + f(x_{i+1}, y_{i+1})\right) + \epsilon_{i+1}^H,$$

and, correspondingly,

$$\eta_{i+1} = \eta_i + \frac{h}{2}\left(f(x_i, \eta_i) + f(x_{i+1}, \eta_{i+1})\right). \tag{8.25}$$

Equation (8.25) defines an implicit single-step method which is called the *trapezoidal rule*. The local discretization error is

$$\epsilon_{i+1}^H = -\frac{h^3}{12} f''(\xi_i, y(\xi_i)), \quad \xi_i \in [x_i, x_{i+1}], \quad i = 0, \ldots, n-1. \tag{8.26}$$

The right-hand side of (8.25) still contains the unknown value η_{i+1}. We can solve it using the fixed-point iteration:

$$\eta_{i+1}^{(k)} = \eta_i + \frac{h}{2}\left(f(x_i, \eta_i) + f(x_{i+1}, \eta_{i+1}^{(k-1)})\right), \quad k = 1, 2, \ldots, k_0. \tag{8.27}$$

As starting value for the iteration we use the approximation of the forward Euler-Cauchy method:

$$\eta_{i+1}^{(0)} = \eta_i + h\, f(x_i, \eta_i), \quad i = 0, 1, \ldots, n-1, \tag{8.28}$$

Equations (8.27),(8.28) define a predictor-corrector method, which is called *Method of Heun*. The forward Euler-Cauchy formula (8.28) is used as *predictor*; it yields a first approximation $\eta_{i+1}^{(0)}$. The trapezoidal rule is used to *correct* the first approximation and, therefore, is called *corrector* (equation (8.27)). A graphical summary is shown in Figure 8.3.

It can be shown that the corrector converges if $hL < 1$, where L is the Lipschitz constant of the function f (cf. (8.11)). As already mentioned, in practical applications the step size is chosen such that $hL \approx 0.05, \ldots, 0.20$.

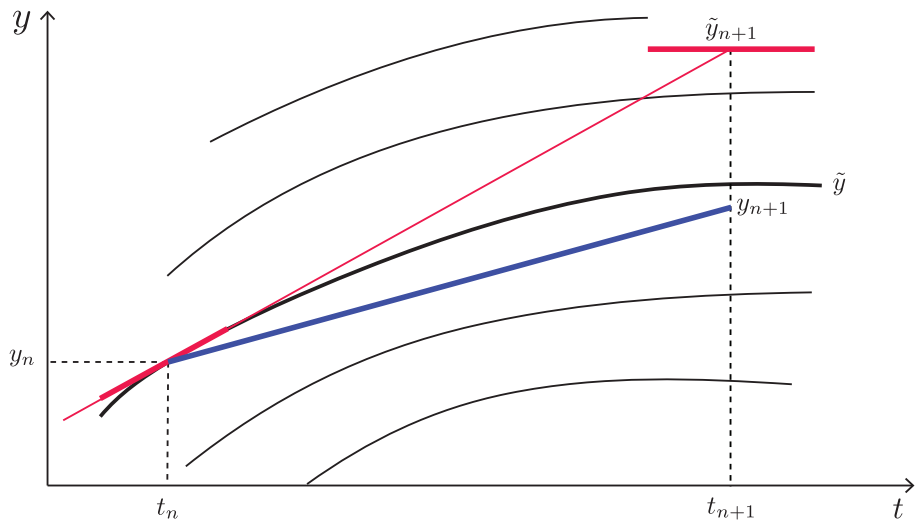


Figure 8.3: Graphical representation of the predictor-corrector in the solution space, with exact solutions plotted (block lines). A prediction is made using the forward Euler-Cauchy formula (thin red line). The solution gradient at the prediction is evaluated (thick red line). This gradient is averaged with the initial solution gradient, and on this basis a new correction step is made (blue line). The final result is much closer to the truth than the predictor.

The local discretization error of the Heun method is identical to the quadrature error of the trapezoidal rule, and given by equation (8.26). Since we do not know exactly the approximation η_{i+1}, an additional iteration error, $\delta_{i+1}^H := \eta_{i+1} - \eta_{i+1}^{(k_0)}$, is introduced, and the total local discretization error becomes $\epsilon_{i+1}^H + \delta_{i+1}^H$. It can be shown that the total local discretization error is still of the order $O(h^3)$, if $hL \approx 0.05, \ldots, 0.20$ is chosen, even with $k_0 = 0$. Thus, the global discretization error is of the order $O(h^2)$ (see e.g. Stoer and Bulirsch (1993)). The latter means that when reducing the step size by a factor $\frac{1}{2}$, the global discretization error will be reduced by a factor of about $\frac{1}{4}$.

Example 8.5 (Method of Heun)
Given the initial value problem $y' = y$, $y(0) = 1$. We seek an approximation to $y(0.5)$ using the method of Heun with step size $h = 0.1$. Use two iterations to get η_{i+1}. Compare the approximations with the true solution at the nodes.

Solution: With $f(x, y(x)) = y$, we obtain the recurrence formula

$$\begin{aligned} \text{predictor: } & \eta_{i+1}^{(0)} = (1+h)\eta_i, \quad i = 0, \ldots, 5, \\ \text{corrector: } & \eta_{i+1}^{(k+1)} = \eta_i + \frac{h}{2}\left(\eta_i + \eta_{i+1}^{(k)}\right), \quad i = 0, \ldots, 5, \quad k = 0, 1. \end{aligned}$$

The nodes are $x_i = x_0 + ih = 0.1\,i$, $i = 0, \ldots, 5$. With $h = 0.1$ and retaining five decimal places, we obtain the results shown in the next table. The absolute error is 5.9×10^{-4},

i	0	1	2	3	4	5
x_i	0	0.1	0.2	0.3	0.4	0.5
η_i	1	1.10525	1.22158	1.35015	1.49225	1.64931
$\eta_{i+1}^{(0)}$	1.1	1.21578	1.34374	1.48517	1.64148	
$\eta_{i+1}^{(1)}$	1.105	1.22130	1.34985	1.49192	1.64894	
$\eta_{i+1}^{(2)}$	1.10525	1.22158	1.35015	1.49225	1.64931	
y_i	1	1.1052	1.2214	1.3499	1.4918	1.6487
$\lvert y_i - \eta_i \rvert$	0	0.8×10^{-5}	1.8×10^{-4}	2.9×10^{-4}	4.3×10^{-4}	5.9×10^{-4}

thus the approximation is about 2 decimals more accuracte than when using the forward Euler-Cauchy method.

Example 8.6 (Heun: Global discretization error and step size)
Consider the initial value problem $y' = \frac{x-y}{2}$, $y(0) = 1$ over the interval $[0, 3]$ using the method of Heun with step sizes $h = 1, \frac{1}{2}, \ldots, \frac{1}{64}$ and calculate the global discretization error. The exact solution is $y(x) = 3e^{-\frac{x}{2}} - 2 + x$.

Exercise 8.7
We seek an error estimate for $\eta(0.2)$ obtained by the method of Heun. We know $\eta_h(0.2) = 1.2217$. Repeat the calculation with step size $2h = \tilde{h} = 0.2$. Estimate the global discretization

step size h	number of steps n	$\eta(3)$	$e^H(0.3)$	$O(h) \approx Ch^2,\ C = -0.0432$
1	3	1.732422	-6.30×10^{-2}	-4.32×10^{-2}
$\frac{1}{2}$	6	1.682121	-1.27×10^{-2}	-1.08×10^{-2}
$\frac{1}{4}$	12	1.672269	-2.88×10^{-3}	-2.70×10^{-3}
$\frac{1}{8}$	24	1.670076	-6.86×10^{-4}	-6.75×10^{-4}
$\frac{1}{16}$	48	1.669558	-1.68×10^{-4}	-1.69×10^{-4}
$\frac{1}{32}$	96	1.669432	-4.20×10^{-5}	-4.20×10^{-4}
$\frac{1}{64}$	192	1.669401	-1.10×10^{-5}	-1.10×10^{-5}

error $e_h^H(0.2)$ and calculate an improved approximation $\bar{\eta}_h(0.2)$. Compare it with the exact solution.

8.3.3 Classical Runge-Kutta method

When we want to improve the accuracy of the methods of Euler-Cauchy and Heun we can take smaller step sizes h and at the same time try to control the rounding errors. However, there are methods which have global discretization errors of orders higher than 1 or 2. Thus, they converge much faster if $h \to 0$ and usually provide higher accuracies for a given h. One class of methods are the so-called *Runge-Kutta methods*. Since the construction of Runge-Kutta formulas is very difficult we restrict to the classical Runge-Kutta method, an explicit method with a local discretization error of order $O(h^5)$ and a global discretization error of order $O(h^4)$. It makes use of the "ansatz"

$$\begin{aligned}\eta_{i+1} &= \eta_i + \sum_{j=1}^{4} w_j \cdot k_{j,i} \\ k_{j,i} &= h \cdot f\left(t_j, \eta_i + \frac{t_j - x_i}{h} \cdot k_{j-1,i}\right), \quad j = 1, \ldots 4, \quad k_{0,i} = 0.\end{aligned}$$

The nodes t_j and weights w_j are determined such that η_{i+1} agrees with the Taylor series expansion of y_{i+1} at x_i up to terms of order $O(h^4)$. This yields a linear system of equations which has two equations less than the number of unknown parameters; so two parameters may be chosen arbitrarily. They are fixed such that the resulting formula has certain symmetry properties w.r.t. the nodes and the weights. The result is

$$\begin{aligned}w_1 &= \frac{1}{6}, \quad w_2 = \frac{1}{3}, \quad w_3 = \frac{1}{3}, \quad w_4 = \frac{1}{6}, \\ t_1 &= x_i, \quad t_2 = x_i + \frac{h}{2}, \quad t_3 = x_i + \frac{h}{2}, \quad t_4 = x_i + h = x_{i+1}.\end{aligned}$$

Therefore, the classical Runge-Kutta formula to perform the step from x_i to x_{i+1} is given by

$$\eta_{i+1} = \eta_i + \frac{1}{6}\left(k_{1,i} + 2k_{2,i} + 2k_{3,i} + k_{4,i}\right), \quad i = 0, \ldots, n-1, \quad \eta(x_0) = y(x_0) = y_0,$$

with

$$
\begin{aligned}
k_{1,i} &= h \cdot f(x_i, \eta_i), \\
k_{2,i} &= h \cdot f(x_i + \frac{h}{2}, \eta_i + \frac{k_{1,i}}{2}), \\
k_{3,i} &= h \cdot f(x_i + \frac{h}{2}, \eta_i + \frac{k_{2,i}}{2}), \\
k_{4,i} &= h \cdot f(x_i + h, \eta_i + k_{3,i}).
\end{aligned}
$$

Obviously, per step four evaluations of the function f are required. That is the price we have to pay for the favorable discretization error. It may be considerable if the function f is complicated.

It is very difficult to specify the optimal choice of the step size h. Mostly, $hL \approx 0.05 \ldots 0.20$ will give good accuracies. After each step it is possible to check whether hL is still in the given range: we use the estimate

$$
hL \approx 2 \left| \frac{k_{2,i} - k_{3,i}}{k_{1,i} - k_{2,i}} \right| .
$$

This enables to vary the step size h during the calculations. It can be shown that, when f does not depend on y, the classical Runge-Kutta formula corresponds to using *Simpson's 3/8-rule* (cf. Chapter 4) for evaluating the integral on the right-hand side of (8.13).

Higher-order Runge-Kutta formulas have also been derived, e.g. the Runge-Kutta-Butcher formula ($m = 6$) and the Runge-Kutta-Shanks formula ($m = 8$). The general structure of all Runge-Kutta formulas is

$$
\begin{aligned}
\eta_{i+1} &= \eta_i + \sum_{j=1}^{m} w_j \cdot k_{j,i} \\
k_{j,i} &= h \cdot f\left(t_j, \eta_i + \sum_{n=1}^{m} a_{j,n} \cdot k_{n,i} \right), \quad j = 1, \ldots, m
\end{aligned}
$$

If $a_{j,n} = 0$ for $n \geq j$, we obtain an *explicit* Runge-Kutta formula, otherwise the formula is *implicit*. An example for an explicit Runge-Kutta formula is the classical Runge-Kutta formula discussed before with $m = 4$ function evaluations per step. With an explicit Runge-Kutta formula we obtain a local discretization error of at most order $O(h^{m+1})$ for $m \leq 4$ whereas for $m > 4$ the best we can get is $O(h^m)$. The optimal local discretization error for implicit Runge-Kutta formulas is $O(h^{2m+1})$ if the m nodes of the corresponding Gauss-Legendre quadrature formula are used. The corresponding formulas are sometimes called *implicit Runge-Kutta formulas of Gauss-type*. They are well-suited if high accuracies are required and if the integration interval is large. For more details see e.g. Engeln-Müllges and Reutter (1985).

Example 8.8 (Classical Runge-Kutta method)
Given the initial value problem $y' = y,\ y(0) = 1$. We seek an approximation to $y(0.2)$ using the classical Runge-Kutta method with step size $h = 0.1$. Compare the approximations with the true solution at the nodes.

Solution: With $f(x, y(x)) = y$, we obtain the recurrence formula

$$\eta_{i+1} = \eta_i + \frac{1}{6}\Big(k_{1,i} + 2k_{2,i} + 2k_{3,i} + k_{4,i}\Big),$$

with

$$k_{1,i} = h\eta_i,\ k_{2,i} = h\left(\eta_i + \frac{k_{1,i}}{2}\right),\ k_{3,i} = h\left(\eta_i + \frac{k_{2,i}}{2}\right),\ k_{4,i} = h\left(\eta_i + k_{3,i}\right), \quad i = 0, 1.$$

The nodes are $x_i = x_0 + ih = 0.1\,i$, $i = 0, \ldots, 5$. With $h = 0.1$ and retaining five decimal places, we obtain the results shown in the next table. It is $\eta(0.2) = 1.22140$, and since

i	t_i	$\bar{y}_i$	$f(t_i, \bar{y}_i)$	$k_{j,i}$
0	0	1.00000	1.00000	0.100000
	0.05	1.05000	1.05000	0.105000
	0.05	1.05250	1.05250	0.110525
	0.1	1.10525	1.10525	0.110525
	0.1	1.10517		
1	0.1	1.10517	1.10517	0.110517
	0.15	1.16043	1.16043	0.116043
	0.15	1.16319	1.16319	0.116319
	0.2	1.22149	1.22149	0.122149
	0.2	1.22140		

$e^{0.2} = 1.22140$, the approximation agrees with the exact solution up to 5 decimal places.

Example 8.9 (Classical Runge-Kutta: Global discretization error and step size)
Consider the initial value problem $y' = \frac{x-y}{2},\ y(0) = 1$ over the interval $[0, 3]$ using the classical Runge-Kutta method with step sizes $h = 1, \frac{1}{2}, \ldots, \frac{1}{8}$ and calculate the global discretization error. The exact solution is $y(x) = 3e^{-\frac{x}{2}} - 2 + x$. The results are shown in the next table.

step size h	number of steps n	$\eta(3)$	$e^H(0.3)$	$O(h) \approx Ch^4,\ C = -6.14 \times 10^{-4}$
1	3	1.670186	-7.96×10^{-4}	-6.14×10^{-4}
$\frac{1}{2}$	6	1.6694308	-4.03×10^{-5}	-3.84×10^{-5}
$\frac{1}{4}$	12	1.6693928	-0.23×10^{-5}	-0.24×10^{-5}
$\frac{1}{8}$	24	1.6693906	-0.01×10^{-5}	-0.01×10^{-5}

Since the global discretization error is of the order $O(h^4)$ we can expect that when reducing the step size by a factor of $\frac{1}{2}$, the error will reduce by about $\frac{1}{16}$.

Exercise 8.10
Solve the equation $y' = -2x - y,\ y(0) = -1$ *with step size* $h = 0.1$ *at* $x = 0.1, \ldots, 0.5$ *using the classical Runge-Kutta method. The analytical solution is* $y(x) = -3e^{-x} - 2x + 2$. *Calculate the global discretization error at each node.*

8.4 Multistep methods

The methods of Euler-Cauchy, Heun, and Runge-Kutta are single-step methods, because they use only the information from the previous node to compute an approximation at the next node. That is, only η_i is used to compute η_{i+1}. Multistep methods for the solution of the initial value problem (8.1),(8.2) use, in order to compute an approximate value η_{i+1}, $s+1$, $s \in \mathbb{N}$, previously computed approximations $\eta_{i-s}, \eta_{i-s+1}, \ldots, \eta_{i-1}, \eta_i$ at equidistant points. To initialize such methods, it is necessary that s starting values $\eta_0, \eta_1, \ldots, \eta_{s-1}$ are at our disposal. s is the step number of the multistep method. Since $\eta_0 = y_0$ is given, the remaining approximations $\eta_1, \ldots, \eta_{s-1}$ must be obtained by other means, e.g. by using single-step methods.

In the following we assume that the starting values $(x_i, f(x_i, \eta_i)),\ i = -s, -s+1, \ldots, 0$ are given. We will often write f_i instead of $f(x_i, \eta_i)$. One class of multistep methods can be derived from the formula

$$y_{i+1} = y_i + \int_{x_i}^{x_{i+1}} f(t, y(t))\, dt, \quad i = 0, \ldots, n-1, \tag{8.29}$$

which is obtained by formally integrating the ODE $y' = f(x, y(x))$. To evaluate the integral on the right-hand side we replace f by the interpolating polynomial P_s of degree s through the past $s+1$ points $(x_{i-s+j}, f_{i-s+j}),\ j = 0, \ldots, s$:

$$P_s(x) = \sum_{j=0}^{s} f_{i-s+j} \cdot l_j(x),$$

with the Lagrange basis functions

$$l_j(x) = \prod_{\substack{k=0 \\ k \neq j}}^{s} \frac{x - x_{i-s+k}}{x_{i-s+j} - x_{i-s+k}}.$$

The approximation η_{i+1} is then obtained from

$$\eta_{i+1} = \eta_i + \int_{x_i}^{x_{i+1}} P_s(t)\, dt = \eta_i + \sum_{j=0}^{s} f_{i-s+j} \int_{x_i}^{x_{i+1}} l_j(x)\, dx,$$

where the integral is calculated exactly. Note that $[x_{i-s}, x_i]$ is the interpolation interval to determine P_s, but P_s is used to integrate over $[x_i, x_{i+1}]$. That means we extrapolate! The

result is an *explicit* formula which yields the approximation η_{i+1}:

$$\eta_{i+1} = \eta_i + \sum_{j=0}^{s} w_j \, f_{i-s+j},$$

with the weights

$$w_j := \int_{x_i}^{x_{i+1}} l_j(x)\, dx.$$

A well-known example of an explicit multistep method is the method of *Adams-Bashforth* with $s = 3$. The corresponding formula is

$$\eta_{i+1} = \eta_i + \frac{h}{24}\left(55f_i - 59f_{i-1} + 37f_{i-2} - 9f_{i-3}\right), \quad i = 0, \ldots, n-1.$$

The local order of the method is $O(h^5)$, the global order is $O(h^4)$. To initialize this formula we need $s+1 = 4$ starting values (x_j, f_j), $j = 0, \ldots, 3$. They have to be computed using a method with the same local order, e.g. the classical Runge-Kutta formula discussed in the previous Section. Compared with the latter, the Adams-Bashforth formula has the advantage that per step only *one* evaluation of f is necessary whereas the classical Runge-Kutta formula requires four function evaluations per step. Thus, the Adams-Bashforth formula is much faster than the classical Runge-Kutta formula although both have the same order.

Example 8.11 (Adams-Bashforth method)
Given the initial value problem $y' = y$, $y(0) = 1$. Choose step size $h = 0.1$ and determine an approximation $\eta(0.4)$ to $y(0.4)$ using the Adams-Bashforth method. Initialize the method using the classical Runge-Kutta formula. Compare the solution with the exact solution $y(0.4)$. Calculate with 9 decimal digits.

Solution: $\eta_0, \ldots, \eta_3$ are calculated using the classical Runge-Kutta formula. Then, the Adams-Bashforth formula has been initialized and η_4 can be calculated.

step i	node x_i	η_i	f_i
0	0	1.00000000	1.00000000
1	0.1	1.10517092	1.10517092
2	0.2	1.22140276	1.22140276
3	0.3	1.34985881	1.34985881
4	0.4	1.49182046	

The exact solution is $y(0.4) = e^{0.4} = 1.49182470$, i.e. the absolute global discretization error at $x = 0.4$ is 4.24×10^{-6}. Computing η_4 with the classical Runge-Kutta formula, we obtain 1.491824586, which has an absolute error of 1.14×10^{-7}. That is, in this case the classical Runge-Kutta method, which has the same local order $O(h^5)$ than the Adams-Bashforth method yields a higher accuracy. The reason is that the Adams-Bashforth method is based on extrapolation which yields larger local errors especially when the step size h is large.

In the same way we can also derive an *implicit* multistep formula if the node x_{i+1} is used to construct the interpolating polynomial P_s instead of the node x_{i-s}. In that case, $f(x, y(x))$ is approximated by the interpolating polynomial P_s of degree s through the $s+1$ points (x_{i-s+j}, f_{i-s+j}), $j = 1, \ldots, s+1$. Then, we avoid to extrapolate but the price we have to pay is that η_{i+1} also appears on the right-hand side, so we have to iterate in order to solve the equation. One example of an implicit multistep formula is the *Adams-Moulton formula* with $s = 4$:

$$\eta_{i+1} = \eta_i + \frac{h}{720}\left(251 f_{i+1} + 646 f_i - 264 f_{i-1} + 106 f_{i-2} - 19 f_{i-3}\right).$$

The fixed-point iteration yields

$$\eta_{i+1}^{(k+1)} = \eta_i + \frac{h}{720}\left(251 f(x_{i+1}, \eta_{i+1}^{(k)}) + 646 f_i - 264 f_{i-1} + 106 f_{i-2} - 19 f_{i-3}\right), \quad k = 0, \ldots, k_0 .$$

The iteration converges if $hL < \frac{360}{251}$. However, when continuing the iteration on the corrector, the result will converge to a fixed point of the Adams-Moulton formula rather than to the ordinary differential equation. Therefore, if a higher accuracy is needed it is more efficient to reduce the step size. In practical applications, mostly $hL \approx 0.05 \ldots 0.20$ is chosen and two iteration steps are sufficient. The Adams-Moulton formula has the local order $O(h^6)$ and the global order $O(h^5)$.

Example 8.12 (Adams-Moulton method)
Given the initial value problem $y' = y$, $y(0) = 1$. Choose step size $h = 0.1$ and determine an approximation $\eta(0.4)$ to $y(0.4)$ using the Adams-Bashforth method. Initialize the method using the classical Runge-Kutta formula. Compare the solution with the exact solution $y(0.4)$. Calculate with 9 decimal digits

step i	node x_i	η_i	f_i
0	0	1.00000000	1.00000000
1	0.1	1.10517092	1.10517092
2	0.2	1.22140276	1.22140276
3	0.3	1.34985881	1.34985881
4 (A-B)	0.4	1.49182046	1.49182046
4 (A-M)	0.4	1.49182458	
		1.49182472	

The iteration even stops after $k_0 = 1$. The exact solution is $y(0.4) = e^{0.4} = 1.49182470$, i.e. the absolute global error at $x = 0.4$ is 0.2×10^{-7}. Thus, Adams-Moulton yields in this case a higher accuracy than the classical Runge-Kutta and the Adams-Bashforth method.

Explicit multistep methods such as the Adams-Bashforth have the disadvantage that due to extrapolation the numerical integration error may become large, especially for large step

size h. Therefore, such formulas should only be used as predictor and should afterwards be corrected by an implicit multistep formula which is used as corrector. For instance, when using Adams-Bashforth as predictor (local order $O(h^5)$) and Adams-Moulton as corrector (local order $O(h^6)$), we obtain to perform the step from x_i to x_{i+1}:

(a) Compute $\eta_{i+1}^{(0)}$ using the Adams-Bashforth formula,

(b) Compute $f(x_{i+1}, \eta_{i+1}^{(0)})$,

(c) Compute $\eta_{i+1}^{(k+1)}$ for $k = 0, \ldots, k_0$ using the Adams-Moulton formula.

Other multistep methods can easily be constructed: in equation (8.29) we replace $f(x, y(x))$ by the interpolation polynomial through the data points (x_{i-s+j}, f_{i-s+j}), $j = 0, \ldots, s$, and integrate over $[x_{i-r}, x_{i+1}]$ with $0 \leq r \leq s$. If $r = 0$ we obtain the Adams-Bashforth formula. More examples and a detailed error analysis of multistep methods is given in Stoer and Bulirsch (1993).

For every predictor-corrector method, where the predictor is of order $O(h^p)$ and the corrector is of order $O(h^c)$, the local discretization error after $k+1$ iteration steps is of order $O(h^{min(c,p+k+1)})$. Therefore, if $p = c - 1$ one iteration is sufficient to obtain the order of the corrector. For arbitrary $p < c$, the order $O(h^c)$ is obtained after $k + 1 = c - p$ iteration steps. However, since the error constant of the predictor are larger than the error constant of the corrector, it may happen that some more iterations are needed to guarantee the order of the corrector. Therefore, in practical applications the order of the corrector is chosen one higher than the order of the predictor, and one iteration step is performed.

Numerous modifications of the discussed algorithms have been developed so far. One example is the *extrapolation method of Bulirsch and Stoer*, which is one of the best algorithms at all w.r.t. accuracy and stability. It is widely used in planetology and satellite geodesy to compute long orbits. In fact it is a predictor-corrector method with a multistep method as predictor. By repeating the predictor for different choices of h a series of approximations is constructed, and the final approximation is obtained by extrapolation. For more information we refer to Engeln-Müllges and Reutter (1985); Stoer and Bulirsch (1993).

8.5 Stability and convergence

The numerical solution of ordinary differential equations provides to a given set of nodes x_i approximations $\eta(x_i)$ to the unknown solutions $y(x_i)$. For all discussed methods we can show that

$$\lim_{h \to 0} |y(x_i) - \eta(x_i)| = 0, \quad x_i \in [x_0, x_n],$$

if certain conditions are fulfiled. That means the approximations $\eta(x_i)$ converge for $h \to 0$ towards the exact values $y(x_i)$, assuming that no rounding errors occur. However, in practical applications it is important to know how discretization errors and rounding errors propagate,

i.e. how stable the algorithm is. An algorithm is called *stable* if an error, which is tolerated in one step, is not amplified when performing the next steps. It is called *instable* if for an arbitrary large number of steps the difference between approximation and unknown solution continuously increases, yielding a totally wrong solution when approaching the upper bound of the integration interval. Responsible for instability can be the ordinary differential equation and/or the used algorithm.

8.5.1 Stability of the ordinary differential equation

Let y denote the solution of the initial value problem

$$y' = f(x, y), \quad y(x_0) = y_0.$$

Let u be a solution of the same ordinary differential equation (ODE), i.e. $u' = f(x, u)$ but let u fulfil a slightly different initial condition $u(x_0) = u_0$. If u differs only slightly from y, we may write

$$u(x) = y(x) + \epsilon\, s(x), \quad u(x_0) = u_0 = y_0 + \epsilon s_0,$$

where s is the so-called *disturbing function* and ϵ a small parameter, i.e. $0 < \epsilon << 1$. Therefore, u fulfils the ODE $u'(x) = y'(x) + \epsilon\, s'(x)$. Taylor series expansion of $f(x, u)$ yields

$$f(x, u) = f(x, y + \epsilon\, s) = f(x, y) + \epsilon\, s\, f_y + O(\epsilon^2),$$

and, when neglecting terms of order $O(\epsilon^2)$, we obtain the so-called *differential variational equation*

$$s' = f_y\, s.$$

Assuming $f_y = c = constant$, this equation has the solution

$$s(x) = s_0\, e^{c(x - x_0)}, \quad s(x_0) = s_0.$$

This equation describes the disturbance at x due to a perturbed initial condition. If $f_y = c < 0$ the disturbance descreases with increasing x, and the ODE is called *stable*, otherwise we call it *unstable*. In case of a stable ODE, the solutions w.r.t. the perturbed initial condition will approach the solutions w.r.t. the unperturbed initial condition. On the other hand, if the initial value problem is unstable, then, as x increases, the solution that starts with the perturbed value $y_0 + \epsilon s_0$ will diverge away from the solution that started at y_0. If the slight difference in initial conditions is due to rounding errors or discretization errors stability (instability) means that this error is damped (amplified) if the integration process continues.

Example 8.13
Let us consider the ODE $y' = f(x, y) = \lambda\, y$, $\lambda \in \mathbb{R}$, and the two initial conditions $y(0) = 1$ and $y(0) = 1 + \epsilon$ with a small parameter $0 < \epsilon << 1$. Obviously f is differentiable w.r.t. y and it is $f_y = \lambda$. The solution of the unperturbed initial value problem is $y(x) = e^{\lambda x}$ and

the solution of the perturbed initial value problem is $u(x) = (1+\epsilon)e^{\lambda x}$*; the difference is* $(u-y)(x) = \epsilon e^{\lambda x}$*. It is clear that when* $\lambda < 0$*, the inital error* ϵ *is strongly damped, while when* $\lambda > 0$ *it is amplified for increasing values* x*.*

If the ODE is stable, a stable numerical algorithm provides a stable solution. However, if the ODE is unstable we can never find a stable numerical algorithm. Therefore, let us assume in the following that the ODE is stable. Then it remains to address the problem of stability of the numerical algorithm.

8.5.2 Stability of the numerical algorithm

Let us consider linear s-step methods, $s \geq 1$, which have the general form

$$\eta_{i+1} = \sum_{k=1}^{s} a_{s-k}\, \eta_{i+1-k} + h \sum_{k=0}^{s} b_{s-k}\, f_{i+1-k}, \quad i = 0, 1, \ldots, n-1. \tag{8.30}$$

We consider the disturbed initial value of the algorithm, $u_0 = \eta_0 + \epsilon H_0$, with a small parameter $0 < \epsilon << 1$. The original solution is denoted by η_i, the disturbed solution by u_i. Let $\delta_i := u_i - \eta_i$. The algorithm (8.30) is called *stable* if

$$\lim_{i\to\infty} |\delta_i| = \lim_{i\to\infty} |u_i - \eta_i| < \infty.$$

With $u_i = \eta_i + \epsilon H_i$, we easily can derive the difference equation for the disturbances δ_i, sometimes called *difference variational equation*. We obtain

$$\delta_{i+1} = \sum_{k=1}^{s} a_{s-k}\, \delta_{i+1-k} + h \sum_{k=0}^{s} b_{s-k} \Big(f(x_{i+1-k}, \eta_{i+1-k} + \delta_{i+1-k}) - f(x_{i+1-k}, \eta_{i+1-k}) \Big). \tag{8.31}$$

It is very difficult to analyse this equation for general functions f. However, a consideration of a certain simplified ODE is sufficient, to give an indication of the stability of an algorithm. This simple ODE is derived from the general equation $y' = f(x, y(x))$ (a) by assuming that f does not depend on x and (b) by restricting to a neighborhood of a value $\bar{y}$. Within this neighborhood we can approximate $y' = f(y)$ by its linear version $y' = f_y(\bar{y})\, y =: c\, y$ with $c \in \mathbb{C}$. The corresponding initial value problem $y' = c\, y$, $y(\bar{x}) = \bar{y}$ has the exact solution $y(x) = \bar{y} \cdot e^{c(x-\bar{x})}$. For our simple ODE, the difference variational equation (8.31) becomes

$$\delta_{i+1} = \sum_{k=1}^{s} a_{s-k}\, \delta_{i+1-k} + hc \sum_{k=0}^{s} b_{s-k}\, \delta_{i+1-k}. \tag{8.32}$$

This is a homogeneous difference equation of order s with constant coefficients. Its solutions can be looked for in the form $\delta_i = \beta^i$ for all i. Substituting into (8.32) yields

$$\beta^{i+1} = \sum_{k=1}^{s} a_{s-k}\, \beta^{i+1-k} + hc \sum_{k=0}^{s} b_{s-k}\, \beta^{i+1-k}. \tag{8.33}$$

Since $\beta \neq 0$, we can divide by β^{i+1-s} and obtain

$$\beta^s = \sum_{k=1}^{s} a_{s-k}\,\beta^{s-k} + hc \sum_{k=0}^{s} b_{s-k}\,\beta^{s-k}, \tag{8.34}$$

or

$$P(\beta) := \beta^s - \sum_{k=1}^{s} a_{s-k}\,\beta^{s-k} - hc \sum_{k=0}^{s} b_{s-k}\,\beta^{s-k} = 0. \tag{8.35}$$

$P(\beta)$ is called *characteristic polynomial* of the difference variational equation (8.31). It is a polynomial of degree s. Assuming that its zeros $\beta_1, \ldots, \beta_s$ are distinct, the general solution is

$$\delta_i = \sum_{j=1}^{s} c_j\,\beta_j^i, \quad \text{for all i,} \tag{8.36}$$

with arbitrary constants c_j. One of the zeros, say β_1^i, will tend to zero as $h \to 0$. All the other zeros are extraneous. If the extraneous zeros satisfy as $h \to 0$ the condition $|\beta_i| < 1,\ i = 2, \ldots, s$, then the algorithm is *absolutely stable*.

Since the zeros are continuous functions of hc, the stability condition $|\beta_j| < 1$ defines a region in the hc-plane, where the algorithm is stable. This region is called *region of stability (S)*, i.e.:

$$S = \{z \in \mathbb{C} : |\beta(z)| < 1\}, \quad z := hc \in \mathbb{C}.$$

That means for any $z = hc \in S$ the algorithm is stable.

Example 8.14 (Stability of the forward Euler-Cauchy formula)
Let us consider the initial value problem $y' = -y^2,\ y(0) = 1$. The forward Euler-Cauchy formula becomes $\eta_{i+1} = \eta_i + h\eta_i$. Comparing with (8.30) it is $s = 1$, $a_0 = 1$, $b_0 = 1$, and $b_1 = 0$. The difference variational equation becomes, observing that

$$f(x_i, \eta_i + \delta_i) - f(x_i, \eta_i) = -2\eta_i\,\delta_i + O(\delta_i^2) :$$

$$\delta_{i+1} = (1 - 2h\eta_i)\,\delta_i.$$

That means, $c = -2\eta_i$. The characteristic polynomial is $\beta = (1 - 2h\eta_i)$ and the stability conditions is $|1 - 2h\eta_i| < 1$. That is the region of stability is $S = \{z \in \mathbb{C} : |1 + z| < 1\}$, where $z := -2h\eta_i$. S is a circle with center at the point -1 on the real line and radius 1. Assuming e.g. $\eta_i > 0$, we have stability of the forward Euler-Cauchy method if $h < \frac{1}{\eta_i}$. Since the stability analysis is based on local linearization, we have to be careful in practise and have to choose h smaller than the criterium says to be on the safe side.

Example 8.15 (Stability of the backward Euler-Cauchy formula)
Let us consider the initial value problem $y' = -y^2,\ y(0) = 1$. The backward Euler-Cauchy formula becomes $\eta_{i+1} = \eta_i + h\,f_{i+1}$. Comparing this equation with (8.30), we observe that $s = 1$, $a_0 = 1$, $b_0 = 0$, and $b_1 = 1$. The difference variational equation is $\delta_{i+1} = \delta_i - 2h\eta_i\delta_{i+1}$,

i.e. $c = -2\eta_i$ *and* $z = hc = -2h\eta_i$. *The characteristic polynomial is* $P(\beta) = 1 + z\,\beta$ *and stability of the backward Euler-Cauchy formula is obtained if* $|1-z|^{-1} < 1$. *Assuming* $\eta_i > 0$, *stability of the backward Euler-Cauchy formula is guaranteed if* $h > 0$, *i.e. for all* h.

In a similar way we can determine the region of stability for other methods. We obtain for instance:

$$\beta(z) = 1 + z \quad \text{(forward Euler-Cauchy)}$$
$$\beta(z) = \frac{1}{1-z} \quad \text{(backward Euler-Cauchy)}$$
$$\beta(z) = \frac{1 + z/2}{1 - z/2} \quad \text{(trapezoidal rule)}$$
$$\beta(z) = 1 + z + \frac{z^2}{2} \quad \text{(Heun)}$$
$$\beta(z) = 1 + z + \frac{z^2}{2} + \frac{z^3}{6} + \frac{z^4}{24} \quad \text{(classical Runge-Kutta)}$$

The stability regions are shown in Figure 8.4 and Figure 8.5.

Example 8.16 (Stability region for the Heun method)
The Heun scheme is $\eta_{i+1} = \eta_i + \frac{h}{2}(f_i + f(x_{i+1}, \eta_i + hf_i))$. *To derive the stability region we neglect the dependency of* f *on* x *and obtain after linearization around* η_i*:*

$$\eta_{i+1} = \eta_i + h\, f_i(1 + \frac{hc}{2})$$

with $c = (f_y)_i$. *Comparing with* (8.30), *it is* $s = 1$, $a_0 = 1$, $b_0 = 1 + \frac{hc}{2}$, *and* $b_1 = 0$. *Thus, the characteristic polynomial is* $P(\beta) = \beta - (1 + z + \frac{z^2}{2})$ *and the stability region is* $S = \{z \in \mathbb{C} : |1 + z + \frac{z^2}{2}| < 1\}$.

8.6 How to choose a suitable method?

Many methods to solve ordinary differential equations have been proposed so far. We could only discuss some of them. None of the methods has such advantages that it can be preferred over all others. On the other hand, the question which method should be used to solve a specific problem still depends on a variety of factors that cannot be discussed in full detail. Examples are the dimension of the system of ODE, the complexity of the function f and the number of function evaluations, the required accuracy, the computer time and memory needed, the flexibility w.r.t. variable step size h, and the stability properties of the initial value problem.

For many applications the computational effort is important. He consists of three parts, (a) the effort for evaluating f, (b) the effort in order to vary the step size, and (c) the effort for all other operations. If f can easily be calculated (a) is not very expensive. Then, for

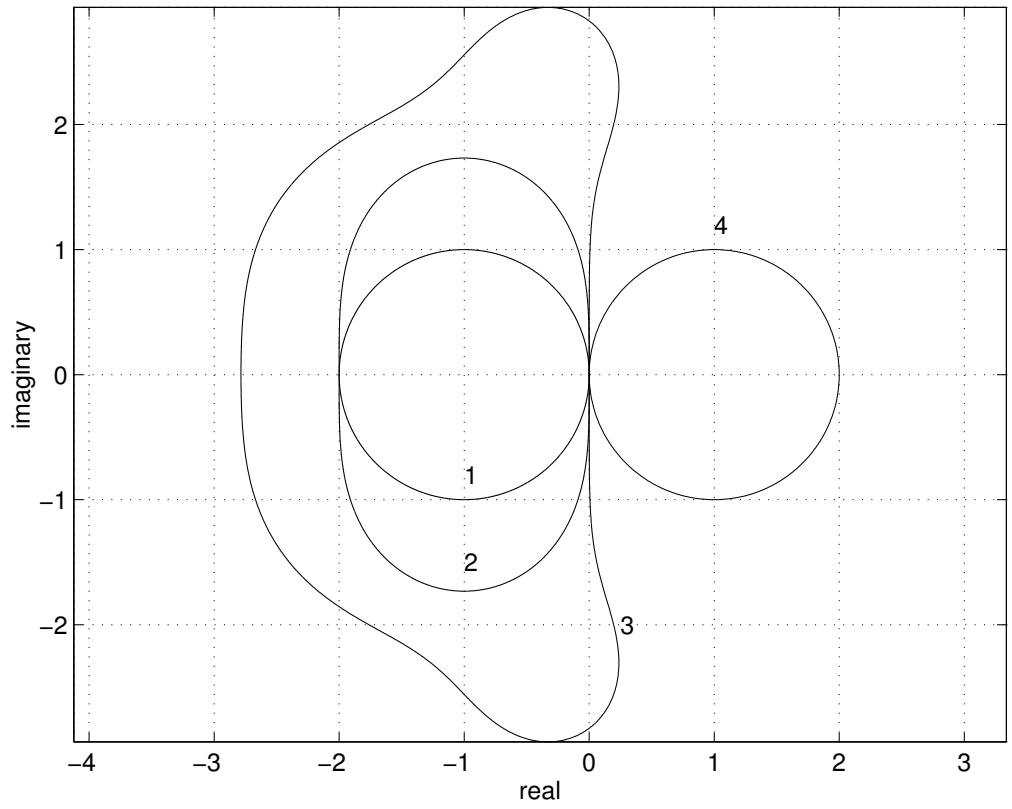


Figure 8.4: The stability region for (1) forward Euler-Cauchy, (2) Heun, (3) classical Runge-Kutta, and (4) backward Euler-Cauchy. Within the curve (1), a circle centered at $(-1, 0)$, we fulfill the condition $|1 + z| < 1$, which provides stability of the forward Euler-Cauchy method. The region within curve (2), defined by $|1 + z + \frac{1}{2}z^2| < 1$, guarantees stability of the Heun method. The classical Runge-Kutta method is stable within region (3), defined by $|1 + z + \frac{1}{2}z^2 + \frac{1}{6}z^3 + \frac{1}{24}z^4| < 1$. The backward Euler-Cauchy method is stable outside the circle (4), i.e. if $|1 - z| > 1$.

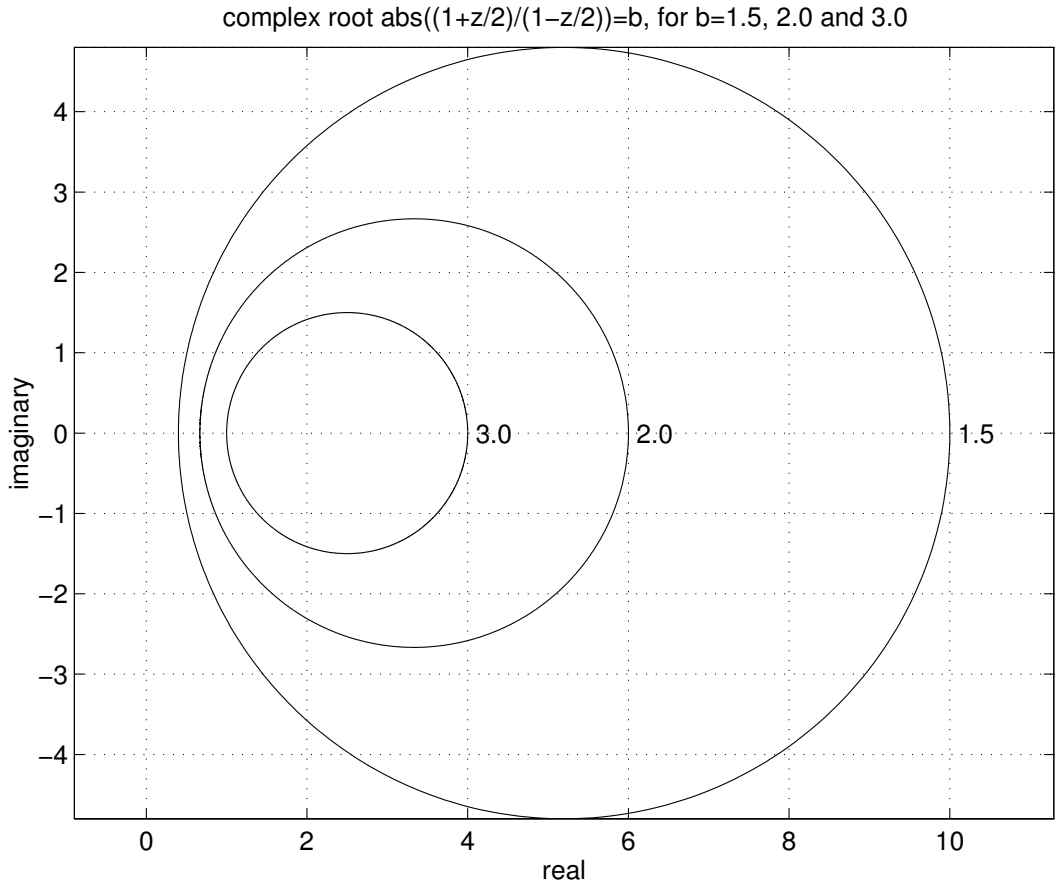


Figure 8.5: Regions $\left|\frac{1+z/2}{1-z/2}\right| = b$ for various values b. The stability region of the trapezoidal rule is obtained if $b = 1$. Obviously the trapezoidal rule is stable for $Re(z) < 0$, i.e. for the complete left half-plane.

moderate accuracies $< 10^{-4}$ the classical Runge-Kutta formula is a good choice. For lower accuracies $> 10^{-4}$ single-step methods with local order < 4 can be used. If the computation of f is costly, multistep methods are superior to single-step methods, although they are more costly when the step size has to be changed during the calculations. Very efficient are implicit Adams methods of variable order. Implicit Runge-Kutta methods, which have not been discussed, are suitable if very high accuracies ($10^{-10} \dots 10^{-20}$) are needed.

Independent on the initial value problem we can summarize the following:

(a) Runge-Kutta methods have the advantage that they have not to be initialized, they have a high local order, they are easy to handle, and an automatic step size control is easy to implement. A drawback is that each step requires several evaluations of the function f.

(b) Multistep methods have the advantage that in general only 2-3 function evaluations per step are needed (including iterations). Moreover, formulas of arbitrarily high order can easily be constructed. A drawback is that they have to be initialized, especially if the step size has to be changed since after any change of the step size, a new initialization is necessary.

Chapter 9

Numerical Optimization

9.1 Statement of the problem

The basic continuous optimization problem is to find $\mathbf{x} \in \Omega$ such that some function $f(\mathbf{x})$ is minimized (or maximized) on the domain $\Omega \subset \mathbb{R}^N$, the design space. We say: Find $\bar{\mathbf{x}} \in \Omega$ such that

$$f(\bar{\mathbf{x}}) \leq f(\mathbf{x}), \qquad \forall \mathbf{x} \in \Omega. \tag{9.1}$$

Equivalently: Find

$$\bar{f} := \min_{\mathbf{x} \in \Omega} f(\mathbf{x}).$$

And equivalently again: Find

$$\bar{\mathbf{x}} := \arg\min_{\mathbf{x} \in \Omega} f(\mathbf{x}),$$

where $\bar{f} = f(\bar{\mathbf{x}})$. In the terminology of optimization

- $f(\cdot)$ is the *objective function* or *cost function*.
- $\mathbf{x} = (x_0, \ldots, x_{N-1})$ is the vector of *design variables*.
- Ω is the *design space*.
- N is the *dimension* of the problem.
- We say the solution $\mathbf{x}$ is *constrained* to lie in Ω.

In 1-dimension $\Omega = [a, b]$ is typically an interval of the real line. As for all other numerical methods in these notes, we are mainly interested in the richer and more challenging multi-dimensional case.

More generally we can consider optimization problems where the design space is partly or wholly discrete, e.g. find $\Omega = \mathbb{R}^N \times \mathbb{N}$. Further we may have problems with an additional

constraint on $\mathbf{x}$, such as: Find $\mathbf{x} \in \Omega$ such that

$$\bar{f} := \min_{\mathbf{x}\in\Omega} f(\mathbf{x}), \quad \text{subject to} \tag{9.2}$$

$$\mathbf{g}(\mathbf{x}) = 0, \quad \text{and} \tag{9.3}$$

$$\mathbf{h}(\mathbf{x}) \geq 0, \tag{9.4}$$

where $\mathbf{g}(\cdot) = 0$ is called an *equality constraint* and $\mathbf{h}(\cdot) \geq 0$ an *inequality constraint*. The problem is now known as a *constrained optimization problem*, an important example of which is *PDE-constrained optimization*, where Ω is a function space and $\mathbf{g}(\mathbf{x}) = 0$ is a partial differential equation. Of course in all cases the minimization can be replaced with a maximization.

Example 9.1 (Drag minimization for aircraft)
When designing aircraft wings one goal is to minimize the drag coefficient c_D, which we can reduce by modifying the shape of the wing. First we choose some parameters $\mathbf{x}$ that specify the shape, e.g. $x_0 \in [0, C_{\max}]$ could be camber, $x_1 \in [0, \infty)$ leading-edge radius, x_2 mid-chord thickness, etc. where the intervals exclude unrealistic shapes. By choosing sufficiently many parameters we can allow for a wide variety of wing shapes - this process is known as parameterization. In practice to specify the shape of a 3d wing fully we might need $\mathcal{O}(100)$ parameters, giving a design space and optimization problem of the same dimension. Let $\mathbf{x}^{(0)}$ be our initial design.

Given a parameterization $\mathbf{x} \in \Omega$, our basic optimization problem for drag is: Find $\mathbf{x} \in \Omega$ such that

$$\bar{c}_D := \min_{\mathbf{x}\in\Omega} c_D(\mathbf{x}).$$

But the new shape might result in an undesirable reduction in lift c_L compared to the initial design, in which case we should introduce a constraint that guarantees the lift is not reduced

$$c_L(\mathbf{x}) - c_L(\mathbf{x}^{(0)}) \geq 0,$$

and we now have a constrained optimization problem. Furthermore imagine that we are at a very early stage in the design of the aircraft, and have a choice a regarding the number of engines we should install on the wing. The number of engines is a new discrete design variable, $N_{\text{eng}} \in \{1, 2, 4\}$, and the optimization problem becomes continuous-discrete inequality constrained.

In any case, to solve the problem we need to evaluate $c_D(\cdot)$ and $c_L(\cdot)$ for different $\mathbf{x}$ representing different wing geometries. This is the job of Computational Fluid Dynamics (CFD), and may be a computationally expensive operation. We therefore need efficient numerical methods that find $\bar{\mathbf{x}}$ with as few evaluations of $c_D(\cdot)$ as possible.

The numerical methods required for solving each type of optimization problem given above are quite different. In the following we consider methods for the most common unconstrained continuous optimization problem only.

9.2 Global and local minima

As for previous problem statements we would first like to establish the existence and uniqueness of solutions to (9.1). Unfortunately in optimization the situation is hopeless. By considering the optimization problems

$$\min_{x\in[0,2\pi M]} \sin x$$

where $M > 0$ and

$$\min_{x\in[0,1]} 1$$

it's easy to convince ourselves respectively that optimization problems may have any number of solutions, and even infinitely many solutions on a finite interval - so solutions can definitely not be regarded as unique in general. As for existence, consider the problem

$$\min_{x\in(0,1)} x,$$

where $(0, 1)$ indicates the *open* interval (not including the end points). This problem has no solution by the following proof by contradiction: assume there exists a solution $\bar{x} = \epsilon \in (0, 1)$; now consider $\epsilon/2$ which is certainly in $(0, 1)$ and $\epsilon/2 < \epsilon$, so ϵ is not the solution. Contradiction. QED.[1] So we can not establish existence of solutions in general either. However in engineering practice (e.g. Example 9.1) we rarely deal with open intervals and therefore don't encounter issues of existence. And if there are multiple solutions any may be a satisfactory design, or we may choose one based on other considerations.

In fact we often reduce the scope of the problem by accepting *local* solutions. In particular a *global optimum* is a solution of (9.1), while a *local optimum* is a solution of the related problem: Find $\mathbf{x} \in E(\epsilon) \subset \Omega$ such that

$$\tilde{\mathbf{x}} := \arg\min_{x\in E} f(\mathbf{x}), \tag{9.5}$$

where $\epsilon > 0$, and

$$E(\epsilon) := \{\mathbf{x} \,|\, \|\mathbf{x} - \bar{\mathbf{x}}\| < \epsilon\} \cap \Omega,$$

i.e. the ball of radius ϵ surrounding the local solution $\tilde{\mathbf{x}}$. In other words a local minima represents the optimal choice in a local sense – the value of the objective function can not be reduced without performing a step greater than some small but finite $\epsilon > 0$. Note that a global optimum is necessarily also a local optimum. Needless to say, we may have multiple local and global optima in a single problem.

The reason we often compromise and only ask for local optima, is that it is easy to find a local optima, but very difficult to determine if a given optima is also global — one has to

[1] This is the reason we need the concept of the *infimum* in real analysis.

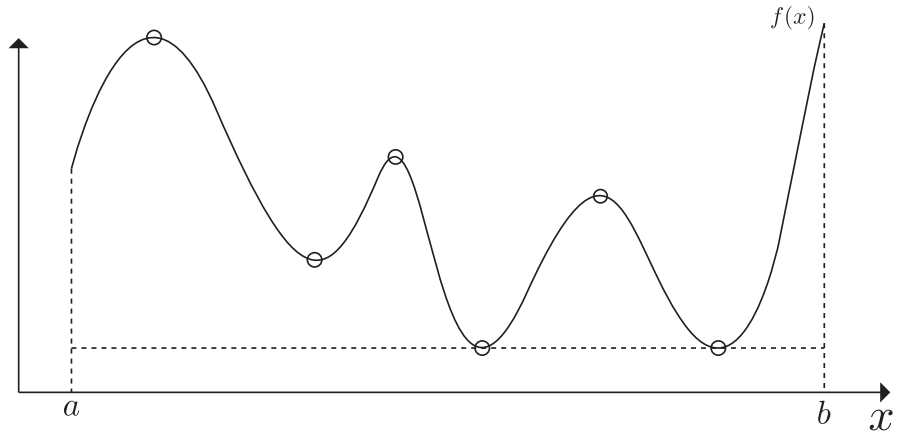


Figure 9.1: One-d objective function $f(x)$ containing multiple local and global optima. Extrema are marked with circles.

search all Ω and make sure there are no other (potentially very limited) regions where f is smaller. With the algorithms presented here we can find local optima quickly and cheaply close to a starting point $\mathbf{x}^{(0)}$. This is often sufficient for engineering purposes, where an acceptable design is known *a priori* — e.g. we know roughly what a wing looks like and want to find another very similar shape with lower drag.

Several algorithms for finding optima will be discussed below. They all find optima close to the starting point $\mathbf{x}^{(0)}$, and are therefore all local. Optimization algorithms are divided into gradient-based methods that use f', and gradient-free methods that require only the ability to evaluate f. In general the former are effective for any N, the latter only for $N \sim \mathcal{O}(10)$.

9.3 Golden-section search

Golden-section search is a gradient-free method that does not rely on the differentiability of $f'(\mathbf{x})$, and is guaranteed to converge to some minimum. However it only applies to problems of dimension 1, which limits its usefulness. It may be considered the contemporary of the recursive bisection for root-finding. Whereas in recursive bisection we needed 2 points to establish the presence of a root in an interval, in Golden-section search we need 3 points to establish the presence of an optimum in an interval.

Consider the case of minimization, and that we wish to find a minimum of $f(x)$ on the interval $[a, b]$. We first choose two nodes $x_L, x_R \in [a, b]$ with $x_L < x_R$, and evaluate the objective function at these points. If $f(x_L) < f(x_R)$ we can conclude that f is decreasing as we head from x_R towards x_L, and that there is guaranteed to be a minimum in the the interval $[a, x_R]$ (which is not at x_R). Similarly if $f(x_L) > f(x_R)$ we can conclude that there is

certainly a minimum in the interval $[x_L, b]$ (that is not at x_L. See Figure 9.2 for a graphical interpretation.

This suggests a recursive algorithm where we progressively reduce the size of the interval on which we know an minimum to exist.

Algorithm 9.2 (Recursive interval minimization)
Assume an initial interval $[a, b] \subset \mathbb{R}$ and a continuous function $f(x)$, and choose $\alpha, \beta \in (0, 1)$ with $\alpha < \beta$. Then pseudocode for $N_{\max}$ iterations of an interval minimization search is:

$a_0 \leftarrow a$; $b_0 \leftarrow b$
for $i = [0 : N_{\max}]$ ***do***
  $x_L \leftarrow a_i + \alpha(b_i - a_i)$
  $x_R \leftarrow a_i + \beta(b_i - a_i)$
  if $f(x_L) < f(x_R)$ ***then***
    $a_{i+1} \leftarrow a_i$
    $b_{i+1} \leftarrow x_R$
  else
    $a_{i+1} \leftarrow x_L$
    $b_{i+1} \leftarrow b_i$
  end if
end for
return $[a_{i+1}, b_{i+1}]$

We might also augment this algorithm with a convergence criteria, stopping the algorithm after the size of the interval is less than some tolerance.

The question remains: how should we choose x_L and x_R, or α and β in the above algorithm? We specify a desirable property: that we can reuse one of $f(x_L)$ or $f(x_R)$ from the previous step — that is, if we choose the sub-interval $[a_n, x_{R,n}]$ on step n, then $x_{L,n}$ should be in the position of $x_{R,n+1}$ on step $n + 1$. Similarly if we choose the sub-interval $[x_{L,n}, b_n]$ then $x_{R,n}$ should be in the position of $x_{L,n+1}$ on step $n + 1$. Thus we reduce the number of evaluations of $f(\cdot)$ to 1 per step, rather than 2 per step.

Satisfying this conditions is possible using the *Golden section* (or *Gulden snede* in Dutch). The conditions require that

$$\frac{x_R - x_L}{x_L - a} = \frac{x_L - a}{b - x_L},$$

and

$$\frac{x_R - x_L}{b - (x_R - x_L)} = \frac{x_L - a}{b - x_L}.$$

Eliminating $x_R - x_L$ from these equations we get

$$\varphi^2 = \varphi + 1 \tag{9.6}$$

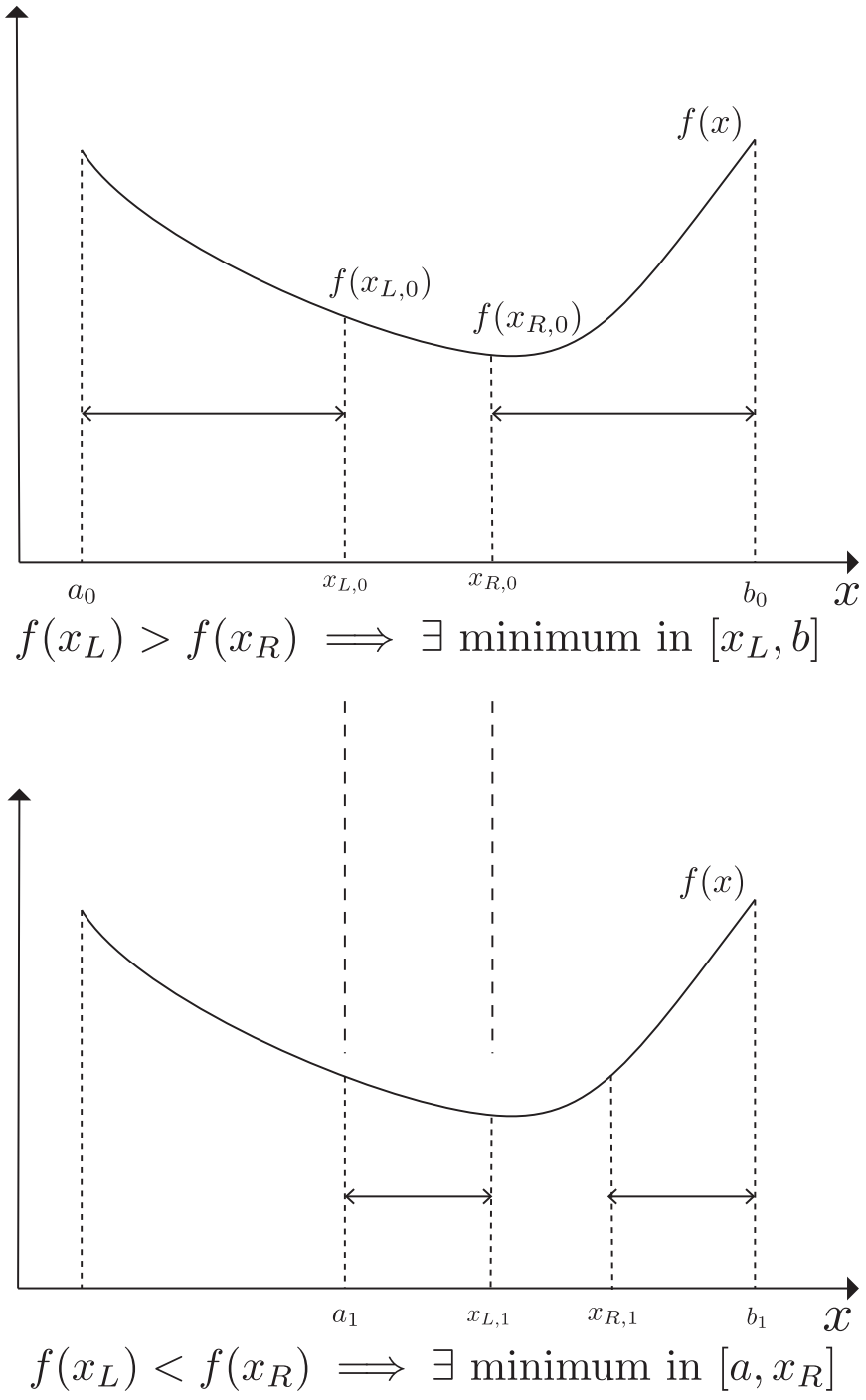


Figure 9.2: One iteration of the golden-section search.

where

$$\varphi = \frac{b - x_L}{x_L - a}. \tag{9.7}$$

The quadratic equation (9.6) has one positive and one negative root, the positive one is

$$\varphi = \frac{1 + \sqrt{5}}{2} = 1.618033988...$$

the Golden ratio. Solving (9.7) for x_L gives

$$x_L = a + (1 - \varphi^{-1})(b - a)$$

given which

$$x_R = a + \varphi^{-1}(b - a)$$

and therefore

$$\alpha = 1 - \varphi^{-1} \approx 0.3820, \qquad \beta = \varphi^{-1} \approx 0.6180.$$

With these values of α and β the above algorithm is known as the *golden-section search* method. The interval width decreases by a constant factor φ^{-1} on each iteration, no matter which side of the interval is choosen. Therefore if we take the midpoint of the interval as our approxiation of the minimum, we make an error of at most

$$\epsilon_0 = \frac{b - a}{2}$$

on the first step, and

$$\epsilon_n = (\varphi^{-1})^n \frac{b - a}{2}$$

on the nth step. As for recursive bisection we have therefore linear convergence, in this case at a rate of φ^{-1}. Applying the method to various hand-sketched functions is a good way of getting an intuition for the behaviour of the method.

9.4 Newton's method

One strategy for solving (9.1) is the following: assume that the objective function $f(\mathbf{x})$ is twice continuously differentiable, i.e. $f \in \mathcal{C}^2(\mathbb{R}^N)$, that the derivative

$$f'(\mathbf{x}) = \left(\frac{\partial f}{\partial x_0} \quad \frac{\partial f}{\partial x_1} \quad \cdots \quad \frac{\partial f}{\partial x_{N-1}} \right) \Big|_{\mathbf{x}}^T$$

is available, and further that $\Omega = \mathbb{R}^N$. In this case we can rewrite (9.1) as: Find one $\bar{\mathbf{x}} \in \Omega$ such that

$$f'(\bar{\mathbf{x}}) = 0, \tag{9.8}$$

which is an N-dimensional root-finding problem. This gives us a location where f is stationary, and then it only remains to establish whether $f(\bar{\mathbf{x}})$ is a local maxima, local minima, or saddle point. This can be done by examining the $N \times N$ Hessian matrix

$$f''(\bar{\mathbf{x}}) = \left(\begin{array}{cccc} \frac{\partial^2 f}{\partial x_0^2} & \frac{\partial^2 f}{\partial x_0 \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_0 \partial x_{N-1}} \\ \frac{\partial^2 f}{\partial x_1 \partial x_0} & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_{N-1} \partial x_0} & \cdots & \cdots & \frac{\partial^2 f}{\partial x_{N-1}^2} \end{array} \right) \Bigg|_{\bar{\mathbf{x}}}.$$

The Hessian is symmetric so all eigenvalues λ are real. If all eigenvalues are positive we have a local minimum, all negative a local maximum, and mixed implies a saddle point. See Example 9.3.[2]

In order to solve (9.8) we can use any of the methods of Chapter 2, for example Newton's method:

$$\mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} - \left[f''(\mathbf{x}^{(n)}) \right]^{-1} f'(\mathbf{x}^{(n)}), \tag{9.9}$$

with a suitable choice of starting point. The properties of Newton's method for root-finding are transfered to this algorithm. In particular we know that Newton's method converges quadratically, so we expect only a few iterations are required for an accurate solution. Furthermore Newton tends to converge to a solution close to the initial guess $\mathbf{x}^{(0)}$, so this algorithm will have the property of finding a local optimum, and different starting points may give different optima.

Example 9.3 (Quadratic forms)
Consider the special objective function

$$f(\mathbf{x}) = c + \mathbf{b}^T \mathbf{x} + \frac{1}{2} \mathbf{x}^T A \mathbf{x}, \tag{9.10}$$

known as a quadratic form, where $A = A^T$ is a symmetric matrix, b a vector and c a constant. This is a useful objective function to consider as it is the first 3 terms of a Taylor expansion of a general function f:

$$f(\mathbf{x}_0 + \mathbf{h}) = f(\mathbf{x}_0) + f'(\mathbf{x}_0)^T \mathbf{h} + \frac{1}{2!} \mathbf{h}^T f''(\mathbf{x}_0) \mathbf{h} + \mathcal{O}(\|\mathbf{h}\|^3),$$

where we can immediately identify

$$\begin{aligned} c &= f(\mathbf{x}_0) \\ \mathbf{b} &= f'(\mathbf{x}_0) \\ A &= f''(\mathbf{x}_0). \end{aligned}$$

[2] This is a generalization of the 1d principle that if the 1st-derivative of f is zero and the 2nd-derivative is positive we have a local minimum (consider what this means for the Taylor expansion of f about $\bar{\mathbf{x}}$).

Therefore for small $\|\mathbf{h}\|$, f is approximated well by a quadratic form. In particular if f has a stationary point at $\bar{\mathbf{x}}$ then $f' = 0$ and

$$f(\bar{\mathbf{x}} + \mathbf{h}) \approx f(\bar{\mathbf{x}}) + \frac{1}{2}\mathbf{h}^T f''(\bar{\mathbf{x}})\mathbf{h}. \tag{9.11}$$

Now we would like to understand under what conditions on $A = f''(\bar{\mathbf{x}})$ the function f has a minimum at $\bar{\mathbf{x}}$ (rather than a maximum or something else). Since A is symmetric (i) all eigenvalues are real, and (ii) we can find an orthonormal basis of eigenvectors $\mathbf{v}_i$ for $\mathbb{R}^{N-1}$, satisfying $A\mathbf{v}_i = \lambda_i \mathbf{v}_i$ and $\mathbf{v}_i^T \mathbf{v}_j = \delta_{ij}$. Now consider (9.11)*, and write $\mathbf{h}$ as a sum of eigenvectors of A:*

$$\mathbf{h} = \sum_{i=0}^{N-1} a_i \mathbf{v}_i,$$

substituting into the second term in (9.11) *we have*

$$\frac{1}{2}\mathbf{h}^T A\mathbf{h} = \frac{1}{2}\sum_{i=0}^{N-1}\sum_{j=0}^{N-1} a_i \mathbf{v}_i A \mathbf{v}_j a_j = \frac{1}{2}\sum_{i=0}^{N-1}\sum_{j=0}^{N-1} a_i \mathbf{v}_i \lambda_j \mathbf{v}_j a_j = \frac{1}{2}\sum_{j=0}^{N-1} \lambda_j a_j^2.$$

Now if $\lambda_j > 0$, $\forall j$, then this term will be positive no matter what direction $\mathbf{h}$ we choose, and therefore the function increases in every direction, and $\bar{\mathbf{x}}$ is a local minimum. Similarly if $\lambda_j < 0$, $\forall j$ then this term will be negative for all directions $\mathbf{h}$, and we have a local maximum. If λ_j are mixed positive and negative, then in some directions f increases, and in others it decreases — these are known as saddle-points.[3] *A quadratic form with $\lambda_j > 0$ in 2d is sketched in Figure 9.3.*

Example 9.4 (Newton's method applied to a quadratic form)
To apply Newton to the quadratic form

$$f(\mathbf{x}) = c + \mathbf{b}^T\mathbf{x} + \frac{1}{2}\mathbf{x}^T A\mathbf{x},$$

where we assume A is positive definite, we first differentiate once:

$$f'(\mathbf{x}) = \mathbf{b} + \frac{1}{2}\left[\mathbf{x}^T A + A\mathbf{x}\right] = \mathbf{b} + A\mathbf{x}$$

and then again:

$$f''(\mathbf{x}) = A,$$

and apply the Newton iteration (9.9)

$$\mathbf{x}^{(1)} := \mathbf{x}^{(0)} - A^{-1}(\mathbf{b} + A\mathbf{x}^{(0)}) = -A^{-1}\mathbf{b}.$$

But then $\mathbf{x}^{(1)}$ satisfies immediately $f'(\mathbf{x}^{(1)}) = 0$. So if f is a quadratic form Newton converges to the exact optimum in 1 iteration from any starting point $x^{(0)}$! Compare this result to the result from root-finding that if f is linear Newton finds the root $f(\bar{\mathbf{x}}) = 0$ in one iteration.

[3] A symmetric matrix A is called *strictly positive definite* if $\lambda_j > 0$, $\forall j$, which is equivalent to $\mathbf{x}^T A\mathbf{x} > 0$, $\forall \mathbf{x} \in \mathbb{R}^{N-1}$. Therefore we are mainly interested in quadratic forms with strictly positive definite A.

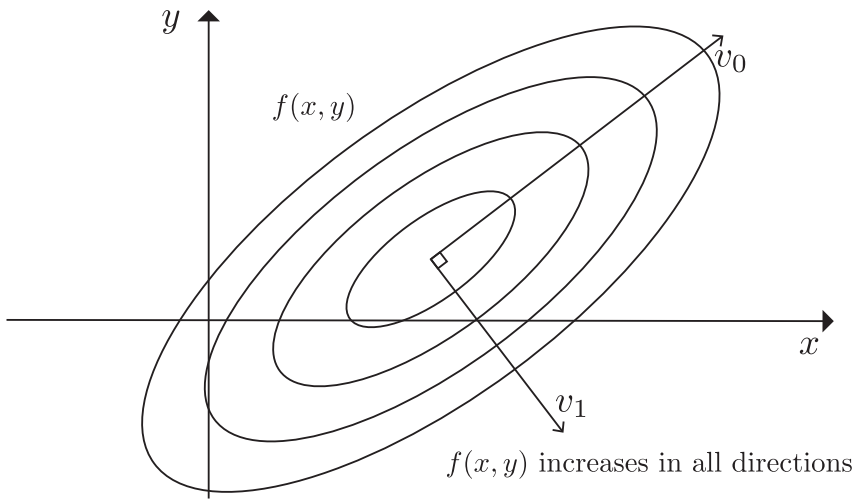


Figure 9.3: Contours of a quadratic form in 2-d with positive definite A.

9.5 Steepest descent method

In practice we may wish to optimize a function where we have access to f' but not f''. We cannot apply Newton but would still like to use the information provided by f'. The essential idea of the steepest descent method is that, given a starting guess $\mathbf{x}^{(n)}$, we search for a minimum of f in the direction in which f decreases fastest. This direction is simply $r_n = -f'(\mathbf{x}^{(n)})$. We then solve the 1-d optimization problem: Find $\alpha^{(n)} \in [0, \infty)$ such that

$$\alpha^{(n)} := \underset{\alpha \in [0,\infty)}{\arg\min}\, g(\alpha),$$
$$g(\alpha) = f\left(\mathbf{x}^{(n)} - \alpha f'(\mathbf{x}^{(n)})\right),$$

using Golden-section search (for example). Then the new guess is

$$\mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} - \alpha^{(n)} f'(\mathbf{x}^{(n)}),$$

and the algorithm repeats with a new search direction. See Figure 9.4 for a sketch of the progress of this method in 2d. At each step we are guaranteed to reduce the value of the objective function:

$$f(\mathbf{x}^{(n+1)}) \leq f(\mathbf{x}^{(n)}),$$

and if $\mathbf{x}^{(n)} = \bar{\mathbf{x}}$ then $\mathbf{x}^{(n+1)} = \bar{\mathbf{x}}$ and the exact solution is preserved, but it unclear how fast this method converges. For this analysis we return to the example of a quadratic form, which will describe the behaviour of the algorithm close to the optimum of any function f.

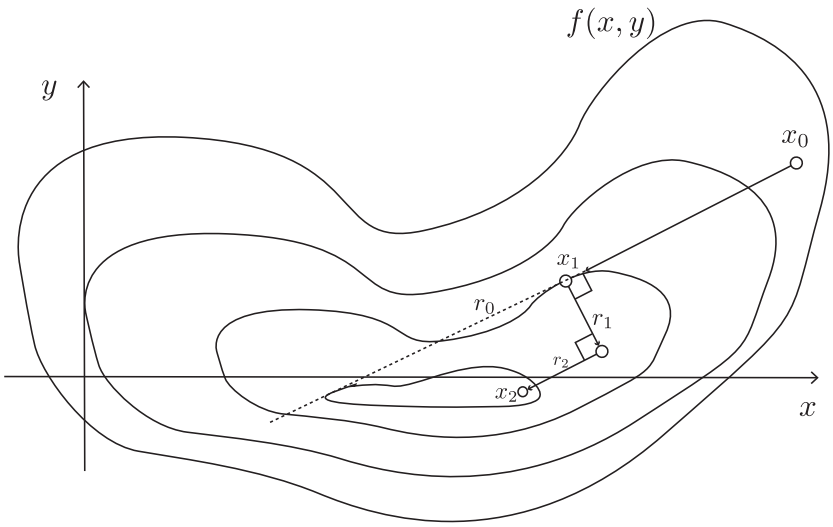


Figure 9.4: Three iterations of the steepest descent method.

Example 9.5 (The steepest descent method applied to a quadratic form)
In the case of a quadratic form we do not need to apply an algorithm to search for the minimum in the search direction, we can get an explicit expression for the size of step to take. Consider the quadratic form:

$$f(\mathbf{x}) = c + \mathbf{b}^T\mathbf{x} + \frac{1}{2}\mathbf{x}^T A\mathbf{x}.$$

The search direction on step n of the method is

$$\mathbf{r}_n = -f'(\mathbf{x}^{(n)}) = -\mathbf{b} - A\mathbf{x}^{(n)}.$$

Now the function $f(\cdot)$ will have a minimum in this direction when the gradient of $f(\cdot)$ is orthogonal to the search direction, i.e. when

$$f'(\mathbf{x}^{(n+1)})^T \cdot f'(\mathbf{x}^{(n)}) = 0.$$

Starting from this point we derive an expression for the step-size $\alpha^{(n)}$:

$$\begin{aligned}
r_{n+1}^T \cdot r_n &= 0 && \\
(-\mathbf{b} - A\mathbf{x}^{(n+1)})^T \cdot r_n &= 0 && \text{Defn. of } r_{n+1} \\
(-\mathbf{b} - A(\mathbf{x}^{(n)} + \alpha^{(n)} r_n))^T \cdot r_n &= 0 && \text{Defn. of } \mathbf{x}^{(n+1)} \\
(-\mathbf{b} - A\mathbf{x}^{(n)})^T \cdot r_n - \alpha^{(n)}(Ar_n)^T \cdot r_n &= 0 && \text{Linearity of } A \\
r_n^T \cdot r_n - \alpha^{(n)} r_n^T A r_n &= 0 && \text{Symmetry of } A
\end{aligned}$$

so that

$$\alpha^{(n)} = \frac{r_n^T r_n}{r_n^T A r_n}. \tag{9.12}$$

So for a step of steepest descent for a quadratic form we know exactly what distance to step in the direction r_n. We could also take this step if we know the Taylor expansion of f including the quadratic term; though in general this will be a different step than that performed using a numerical 1d search method (because of the additional higher-order terms neglected).

Example 9.6 (Convergence of steepest descent for a stiff problem)
Consider the simple quadratic form

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \begin{pmatrix} 1 & 0 \\ 0 & 1000 \end{pmatrix} \mathbf{x},$$

with exact solution $\bar{\mathbf{x}} = 0$, $\bar{f} = 0$.[4] Apply steepest decent with an initial guess $x^{(0)} = (-1, 0.001)$. The derivative is

$$f'(\mathbf{x}) = \begin{pmatrix} 1 & 0 \\ 0 & 1000 \end{pmatrix} \mathbf{x} = \begin{pmatrix} x_0 \\ 1000 x_1 \end{pmatrix}.$$

and therefore the first search direction is

$$r_0 = -f'(\mathbf{x}^{(0)}) = (1, -1),$$

and by (9.12):

$$\alpha^{(0)} = \frac{r_0^T r_0}{r_0^T A r_0} = \frac{2}{1001}.$$

Note that this is a tiny step in comparision to the distance to the true minimum at 0, which is ≈ 1. We have

$$x^{(1)} = \frac{1}{1001}\begin{pmatrix} -999 & -\frac{999}{1000} \end{pmatrix}^T$$

which is extremely close to the starting point. The next search direction is

$$r_1 = -f'(\mathbf{x}^{(1)}) = \frac{999}{1001}(1, 1),$$

which is just 90° to the previous direction but slightly shorter. The next step is

$$\alpha^{(1)} = \frac{r_1^T r_1}{r_1^T A r_1} = \frac{2}{1001},$$

i.e. the same as before, but since r_1 is slightly shorter the actual step is slightly shorter. The iteration proceeds in this way, with the search direction alternating between $(1, 1)$ and

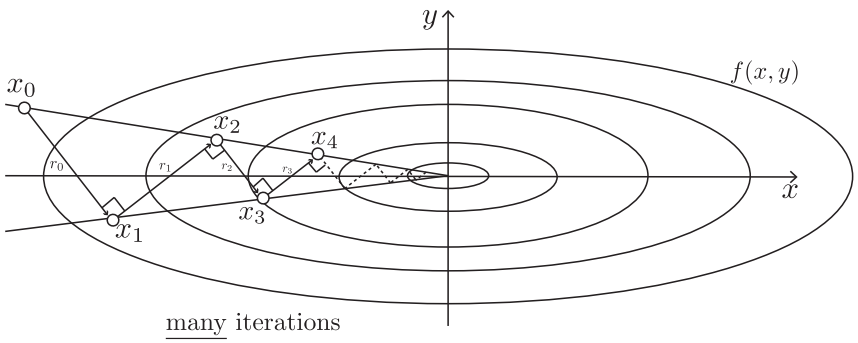


Figure 9.5: Slow progress of the steepest descent method for the stiff problem of Example 9.6.

$(1, -1)$, $\alpha = 2/1001$, *and the total step reducing a factor of* $999/1001$ *at each step. See Figure 9.5*

Therefore the error approximate minimum at step n *can be approximately written:*

$$\epsilon_n \approx \left(\frac{999}{1001}\right)^n.$$

This is familiar linear convergence, but extremely slow. After 100 *steps the error is* $\epsilon_{100} \approx 0.82$, *after* 1000 *steps still* $\epsilon_{1000} \approx 0.14$. *To solve this issue with stiffness causing slow convergence there is a related algorithm called the conjugate gradient method, which make a cleverer choice of search direction, but this is outside the scope of this course.*

9.6 Nelder-Mead simplex method

The final algorithm we consider in this course takes the most space to describe, but as we have seen again and again the definition of a numerical method is only the tip of the iceberg; understanding convergence, accuracy and other important properties requires investigating deep below the water-line. We do not analyse the Nelder-Mead simplex method in detail, but note that it has many nice properties: it is gradient-free, converges rapidly for moderate N and stretched objective functions, is stable, and robust to noise in the objective function.[5]

In the following the method is defined for 2 dimensions in which the simplex is a triangle. In 3 dimensions the simplex is a tetrahedron, and in higher dimensions a hyper-tetrahedron or "N-simplex", the generalizations of the operations to these cases is clear.

[4] In the case of ODEs we would call this a stiff problem, where there are 2 or more disparate scales in the same system— a similar concept applies here.

[5] Probably because of these favorable properties the Nelder-Mead simplex method happens to be the default method in the `fmin` optimization routine in MATLAB.

Algorithm 9.7 (Nelder-Mead Simplex)
Start with an initial triangle with three nodes $(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)$ *not lying on a line. Perform the following steps:*

1. *ORDER the nodes according to the values of* f *at the vertices:*

$$f(\mathbf{x}_1) \leq f(\mathbf{x}_2) \leq f(\mathbf{x}_3)$$

   i.e. $\mathbf{x}_1$ *corresponds to the best value of the objective function, and* $\mathbf{x}_3$ *the worst.*

2. *Compute* $\mathbf{x}_0 = \frac{1}{2}(\mathbf{x}_1 + \mathbf{x}_2)$*, the midpoint of all nodes except the worst* $\mathbf{x}_3$*.*

3. *REFLECT: Compute* $\mathbf{x}_R = \mathbf{x}_0 + (\mathbf{x}_0 - \mathbf{x}_3)$*, the reflection of* $\mathbf{x}_3$ *in the line containing the two other points. Evaluate* $f(\mathbf{x}_R)$*, and:*

   - *If* $f(\mathbf{x}_R) < f(\mathbf{x}_1)$ *then EXPAND.*
   - *If* $f(\mathbf{x}_R) > f(\mathbf{x}_2)$ *then CONTRACT.*
   - *Otherwise replace* $\mathbf{x}_3$ *with* $\mathbf{x}_R$ *and goto 1.*

4. *EXPAND: Compute* $\mathbf{x}_E = \mathbf{x}_0 + 2(\mathbf{x}_0 - \mathbf{x}_3)$*, the*

   - *If* $f(\mathbf{x}_E) < f(\mathbf{x}_R)$ *then replace* $\mathbf{x}_3$ *with* $\mathbf{x}_E$ *and goto 1.*
   - *Otherwise replace* $\mathbf{x}_3$ *with* $\mathbf{x}_R$ *and goto 1.*

5. *CONTRACT: Compute* $\mathbf{x}_C = \mathbf{x}_0 + \frac{1}{2}(\mathbf{x}_0 - \mathbf{x}_3)$*,*

   - *If* $f(\mathbf{x}_C) < f(\mathbf{x}_3)$ *then replace* $\mathbf{x}_3$ *with* $\mathbf{x}_C$ *and goto 1.*
   - *Otherwise REDUCE.*

6. *REDUCE: Nothing is producing an improvement, so replace all the nodes but the best node with the midpoints of the triangle edges, to create a triangle in the same location but half the size:*

   - $\mathbf{x}_2 = \mathbf{x}_1 + \frac{1}{2}(\mathbf{x}_2 - \mathbf{x}_1)$
   - $\mathbf{x}_3 = \mathbf{x}_1 + \frac{1}{2}(\mathbf{x}_3 - \mathbf{x}_1)$ *and goto 1.*

The basic idea is that the values of f at the 3 nodes of the triangle give a clue about the best direction in which to search for a new point without needing any gradient information. At each step the algorithm tries to move away from the worst point in the triangle (REFLECT). If this strategy is successful then it goes even further and stretches the triangle in that direction (EXPAND); if not successful it is more conservative (CONTRACT); and if none of this seems to work then it makes the entire triangle smaller (REDUCE), before trying again. See Figure 9.6

After a number of iterations the stretching of the simplex corresponds roughly to the stretching of the objective function in the design space. If the EXPAND operation is performed repeatedly the result will be an increasingly stretched triangle, but this will only occur if this strategy is producing a consistent reduction in that direction. This flexibility allows the method to take large steps when necessary, and thereby gain some of the efficiency advantages of gradient-based methods.

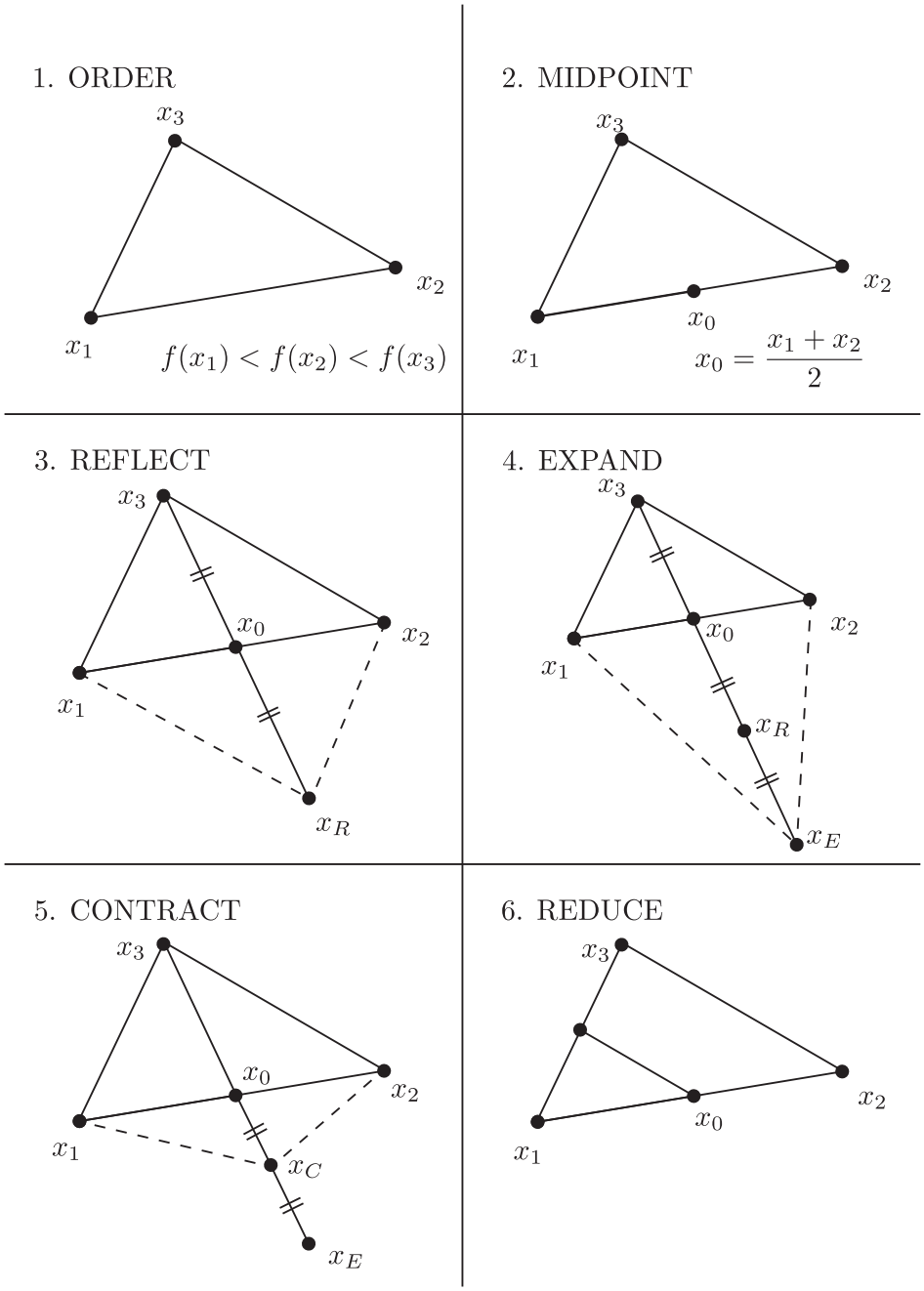


Figure 9.6: The component operations of Nelder-Mead simplex.

Bibliography

Buchanan, F. L. and Turner, P. R. (1992). *Numerical Methods and Analysis.* McGraw-Hill.

Engeln-Müllges, G. and Reutter, F. (1985). *Numerische Mathemathik für Ingenieure.* B.I.-Wissenschaftsverlag, Mannheim.

Gerald, C. F. and Wheatly, P. O. (1994). *Applied numerical Analysis.* Addison-Wesley, 5th edition.

Stoer, J. and Bulirsch, R. (1993). *Introduction to numerical Analysis.* Springer, New York, 2nd edition.

Vetterling, W. T., Teukolsky, S. A., and Press, W. H. (1992). *Numerical Recipes in FORTRAN; the Art of scientific Computing; Numerical Recipes Example Book.* Cambridge University Press, New York.