sample tests (two random quantities with possibly different distributions are to be compared), *parameter tests*, *distribution tests* or *independence tests*. We make explicit mention here of some examples that are of particular importance: the *approximate binomial tests*, the *Gaussian test*, the *t-test*, the χ^2*-goodness-of-fit test* as well as the *Kolmogorov Smirnov test*.

After this excursion into stochastics and statistics, we now turn toward numerical analysis, which provides suitable simulation tools for numerous classes of models.

2.4 Numerical Aspects

Numerical Analysis, as stochastics, is an area within Applied Mathematics, and is concerned with the design and analysis of computational methods for continuous models, in particular from the area of linear algebra (solving linear systems of equations, computing eigenvalues etc.) and analysis (finding roots or extrema etc.). This is typically associated with *approximations* (solving differential equations, computing integrals) and is therefore rather atypical for traditional mathematics. The analysis of numerical algorithms revolves around the aspects of approximation accuracy, convergence speed, storage requirements and computing time. In particular, the later two topics lie at the center of *numerical programming*, which is an area within computer science that is also particularly concerned with implementation aspects. And everything works toward the goal to perform numerical simulations, in particular on high performance and supercomputers.

Whoever recognizes gaps in their understanding when reading the following sections on numerics and would like to close these before diving into the individual topics of modeling and simulation, is referred to, for example, “An introduction to numerical computations” by Sidney Yakowitz and Ferenc Szidarovszky [60] or to “Numerical Methods in Scientific Computing” by Germund Dahlquist and Åke Björck [14].

2.4.1 Basics

Discretization

In numerical analysis, we are confronted with problem settings that are *continuous*. Computers, however, are initially restricted to deal with *discrete* objects. This concerns numbers, functions, domains as well as operations such as differentiation. The magic word for the necessary transition, “continuous $\rightarrow$ discrete”, is *discretization*. Discretization always sits at the forefront of all numerical activities. One discretizes the real numbers through the introduction of *floating point numbers*, domains (for example, time intervals at the numerical solution of ordinary differential equations or spatial domains at the numerical solution of partial differential equations) through

the introduction of a *grid* consisting of discrete *grid points* and operators such as the derivative $\frac{\partial}{\partial x}$ by forming difference quotients of function values at neighboring grid points.

Floating Point Numbers

The set $\mathbb{R}$ of real numbers is *unbounded* and *continuous*. The set $\mathbb{Z}$ of integers, also unbounded, is discrete with constant unit distance 1 between two neighboring numbers. The set of *exactly representable numbers* on a computer, however, is inevitably finite and therefore discrete and bounded. Probably the easiest realization of such a number set with its corresponding computations is *integer arithmetic*. This, as well as *fixed point arithmetic*, works with fixed number ranges and fixed resolution. *Floating point arithmetic*, in contrast, permits varying the placement of the decimal point and therefore allows for variable magnitude, variable location of the representable number range as well as variable resolution.

The *normalized t-digit floating point numbers w.r.t. the basis B* ($B \in \mathbb{N} \setminus \{1\}$, $t \in \mathbb{N}$) is now defined:

$$\mathbb{F}_{B,t} := \left\{ M \cdot B^E : M = 0 \text{ or } B^{t-1} \leq \mid M \mid < B^t, \;\; M, E \in \mathbb{Z} \right\} .$$

Here, M is called the *mantissa*, and E the *exponent*. The normalization (no leading zero) guarantees the uniqueness of the representation. The introduction of an admissible range for the exponent leads to the *machine numbers*:

$$\mathbb{F}_{B,t,\alpha,\beta} := \{ f \in \mathbb{F}_{B,t} : \alpha \leq E \leq \beta \} .$$

The quadruple (B, t, α, β) completely characterizes the system of machine numbers. Therefore, a concrete number has to store M and E. The terms floating point number and machine number are often used synonymously; in context, B and t are usually obvious so that in the following we only write $\mathbb{F}$.

The *absolute distance* between two neighboring floating point numbers is not constant but depends on the respective magnitude of the numbers. The maximal possible *relative distance* between two neighboring floating point numbers is called the *resolution* ϱ. It holds:

$$\frac{(\mid M \mid +1) \cdot B^E - \mid M \mid \cdot B^E}{\mid M \mid \cdot B^E} = \frac{1 \cdot B^E}{\mid M \mid \cdot B^E} = \frac{1}{\mid M \mid} \leq B^{1-t} =: \varrho .$$

The representable range is furthermore characterized by the *smallest positive machine number* $\sigma := B^{t-1} \cdot B^{\alpha}$ as well as the *largest machine number* $\lambda := (B^t - 1) \cdot B^{\beta}$.

A famous and important example is the number format of the IEEE (Institute of Electrical and Electronics Engineers), which is defined in the US-Norm ANSI/IEEE-Std-754-1985 and which can be traced back to a patent of Konrad

Zuse in the year 1936. It allows for the accuracy levels of *single precision*, *double precision* and *extended precision*. Single precision corresponds to about 6–7 decimal places, whereas double precision ensures about 14 places.

Rounding

Since floating point numbers are discrete, certain real numbers may slip through our fingers. These must be assigned to suitable floating point numbers in a meaningful way—one *rounds*. Important rounding strategies are *rounding down*, *rounding up*, *truncating* as well as *rounding correctly*, which always rounds toward the closest floating point number.

When rounding, one inevitably commits mistakes. These we can distinguish as the *absolute round-off error*, $\mathrm{rd}(x) - x$, and the *relative round-off error*, $\frac{\mathrm{rd}(x)-x}{x}$, for $x \neq 0$. The goal for the construction of floating point numbers is to obtain a high relative accuracy. It is the relative round-off error that plays a central role for all analyses and it must be estimated in order to assess the possible influence of round-off errors in a numerical algorithm. The relative round-off error is directly tied to the resolution.

Floating Point Arithmetic

For the simple rounding of numbers, one knows the exact value. Here, this immediately changes for the simplest computations since, starting with the first arithmetic operation, one exclusively works with approximations. The exact execution of the basic arithmetic operations $* \in \{+, -, \cdot, /\}$ in the system $\mathbb{F}$ of floating point numbers is, in general, impossible—even for elements in $\mathbb{F}$: How shall one write the sum of 1234 and 0.1234 exactly using four places? Therefore, we need a "clean" floating point arithmetic that prevents a build-up of accumulated errors. The ideal case (and required by the IEEE-standard for the basic arithmetic operations and for the square root) is the *ideal arithmetic* in which the computed result corresponds to the rounding of the exact result.

Round-off Error Analysis

A numerical algorithm is a finite sequence of basic arithmetic operations with a uniquely determined succession. Floating point arithmetic provides a significant source for errors in numerical algorithms. For a numerical algorithm, the most important objectives concerning floating point arithmetic are therefore a *small discretization error*, *efficiency* (run-times as small as possible) as well as a *small influence of round-off errors*. The later objective requires an *a-priori round-off error*

analysis: Which bounds can be stated for the total error given that a certain quality is assumed for the elementary operations?

Condition

Conditioning is a central, but usually only qualitatively defined term in numerical analysis: How large is the sensitivity of the solution of a problem toward changes in the input data? In the case of high sensitivity, one speaks of *ill conditioning*, or of an *ill conditioned problem*. In the case of little sensitivity, one correspondingly speaks of *good conditioning* and *well-conditioned problems*. Very important: Conditioning is a property of the considered *problem*, not of the algorithm to be used.

Perturbations δx in the input data must therefore be studied since inputs oftentimes are available only inaccurately (obtained from measurements or from previous calculations) and thus such perturbations are common even in exact arithmetic. Ill-conditioned problems are not only difficult to treat numerically, but could possibly not even be treated at all in the most extreme case.

Among the basic arithmetic operations, only subtraction has the potential to be ill-conditioned. This is due to the so-called *loss of significance* which is the effect that could occur when subtracting two numbers of equal sign in which leading identical digits cancel each other out, i.e., leading non-zeros can disappear. Thus, the number of relevant digits can decrease considerably. Loss of significance is impending in particular when both numbers are of similar magnitude.

Typically, the conditioning of a problem $p(x)$ having input x is not defined as before by computing a simple difference (i.e., the relative error), but via the derivative of the result with respect to the input:

$$\text{cond}(p(x)) \;:=\; \frac{\partial p(x)}{\partial x} \;.$$

When a problem p is decomposed into two or more subproblems, then the chain rule implies

$$\text{cond}(p(x)) \;=\; \text{cond}(r(q(x))) \;=\; \frac{\partial r(z)}{\partial z}\,|_{z=q(x)} \cdot \frac{\partial q(x)}{\partial x} \;.$$

Naturally, the total conditioning of $p(x)$ is independent of the decomposition, but the partial conditionings depend on the decomposition. This can lead to problems: Let p be well-conditioned with an excellently conditioned first part q and a miserably conditioned second part r. If there then occur errors in the first part, these can have catastrophic effects in the second.

A prime example for conditioning is the computation of the intersecting point of two non-parallel lines. If the two lines are almost orthogonal, then the problem of finding the intersection is well-conditioned. However, if they are almost parallel, then we have an ill conditioned problem.

Stability

Using the notion of conditioning allows us to characterize problems. Next, we turn to the characterization of numerical algorithms. As we have readily seen, the input data may be perturbed. Mathematically formulated, this means that they are only determined up to a certain tolerance, e.g., they lie in some neighborhood,

$$U_\varepsilon(x) \;:=\; \{\tilde{x} : \mid \tilde{x} - x \mid < \varepsilon\}$$

with respect to the exact input x. Every such $\tilde{x}$ must therefore be considered as equivalent to x. Thus, the following definition appears obvious: An approximation $\tilde{y}$ for $y = p(x)$ is called *acceptable* if $\tilde{y}$ is the exact solution for one of the above $\tilde{x}$, i.e.,

$$\tilde{y} \;=\; p(\tilde{x}) \; .$$

Here, the error $\tilde{y} - y$ has different sources: Round-off errors as well as method or discretization errors (series and integrals are often approximated by sums, derivatives by difference quotients, iterations are aborted after a few iteration steps).

Stability is another central notion of numerics. Here, a numerical algorithm is called *(numerically) stable*, if it produces acceptable results under the influence of round-off and method errors for all admissible and up to the computational accuracy perturbed input data. A stable algorithm can, by all means, yield large errors—for example, if the problem to be solved is ill-conditioned. The established implementations of the basic arithmetic operations are numerically stable. The composition of stable methods, however, are not necessarily stable—otherwise everything would be numerically stable.

Stability is an important and central topic for methods involving the numerical solution of ordinary and partial differential equations.

2.4.2 Interpolation and Quadrature

Interpolation and *integration* or *quadrature*, are central tasks of numerical analysis. In subsequent chapters on modeling and simulation they are required indirectly at most and as such we will make it brief in the following.

Polynomial Interpolation

For reasons of simplicity we restrict ourselves to the one-dimensional case. In interpolation or intermediate value calculations, an (entirely or partially unknown or just too complicated) function $f(x)$ is to be replaced by a function $p(x)$ that is easy to construct and to work with (evaluate, differentiate, integrate), whereby p

assumes prescribed *node values* $y_i = f(x_i)$ at the given *nodes* $x_i, i = 0, \ldots n$, which define the distance of the *mesh widths* $h_i := x_{i+1} - x_i$. The pairs (x_i, y_i) are called nodal *data points*, while p is called the *interpolant* of f. In view of their simple structure, a class of particularly popular interpolants is given by *polynomials* $p \in \mathbb{P}_n$, the vector space of all polynomials with real coefficients of degree smaller than or equal to n in the single variable x. However, polynomial interpolation is by no means the only one: One can patch together piecewise polynomials from which one obtains the so-called *polynomial splines* which offer several distinct and essential advantages, and one can also interpolate using rational, trigonometric or exponential functions.

If one uses the interpolant p instead of the function f between the nodes, then one commits an *interpolation error* there. The difference $f(x) - p(x)$ is called the *error term* or *remainder*, and it holds that,

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

at some intermediate location ξ,

$$\xi \in [\min(x_0, \ldots, x_n, x), \max(x_0, \ldots, x_n, x)] .$$

In the case of sufficiently smooth functions f, this relationship allows for the estimation of the interpolation error.

There exist different possibilities for the construction and representation of polynomial interpolants: the classical approach of the *point* or *incidence test*, the approach of *Lagrange polynomials*,

$$L_k(x) := \prod_{i:i \neq k} \frac{x - x_i}{x_k - x_i} , \qquad p(x) := \sum_{k=0}^{n} y_k \cdot L_k(x) ,$$

the recursive *scheme by Aitken and Neville* as well as the recursive *Newton interpolation formula*.

For equidistant nodes with mesh width $h := x_{i+1} - x_i$, one can easily estimate the interpolation error:

$$| f(\bar{x}) - p(\bar{x}) | \leq \frac{\max_{[a,b]} | D^{n+1} f(x) |}{n+1} \cdot h^{n+1} = \mathcal{O}(h^{n+1}) .$$

One observes that classical polynomial interpolation with equidistant nodes is very ill-conditioned for larger n (here, the impact of "larger" already begins below 10)—small errors in the central nodal values are, for example, amplified drastically at the boundary of the considered interval by polynomial interpolation. For this reason, one has to attend to methods that are better in this respect.

Polynomial Splines

In order to circumvent the two major disadvantages of the polynomial interpolation (the number of nodes and the degree of polynomial are tightly connected; useless for larger n), one "glues" together polynomial pieces of lower degree in order to also construct a global interpolant and for a larger number of nodes. This leads to *polynomial splines* or *splines* in short. Once again let $a = x_0 < x_1 < \ldots < x_n = b$ and $m \in \mathbb{N}$. Here, the x_i are called *nodes*. We consider here only the special case of *simple nodes*, i.e., $x_i \neq x_j$ for $i \neq j$. A function $s : [a, b] \to \mathbb{R}$ is called a *spline of order m* or *of degree* $m-1$, resp., if $s(x) = p_i(x)$ is $m-2$-times continuously differentiable on $[a, b]$ with $p_i \in \mathbb{P}_{m-1}$, $i = 0, 1, \ldots, n-1$ on $[x_i, x_{i+1}]$. Thus, s is a polynomial of degree $m-1$ between two neighbouring nodes, and globally (i.e., in particular at the nodes themselves!) s is $m-2$-times continuously differentiable.

For $m = 1$, s is a step function (piecewise constant) while for for $m = 2$ it is piecewise linear, etc. The *cubic splines* ($m = 4$) are popular. Splines not only overcome the disadvantages of polynomial interpolation, they can also be constructed efficiently (linear effort in n).

Trigonometric Interpolation

With *trigonometric interpolation*, one considers complex-valued functions that are defined on the unit circle in the complex number plane—and also refers to the *representation in the frequency domain*. Let there be n nodes given on the unit circle of the complex number plane,

$$z_j := \mathrm{e}^{\frac{2\pi i}{n} j}, \qquad j = 0, 1, \ldots, n-1 ,$$

as well as n node values v_j. One desires to find the interpolant

$$p(z) , \quad z = \mathrm{e}^{2\pi i t} , t \in [0, 1] ,$$

with

$$p(z_j) = v_j , \quad j = 0, 1, \ldots, n-1 , \qquad p(z) = \sum_{k=0}^{n-1} c_k z^k = \sum_{k=0}^{n-1} c_k \mathrm{e}^{2\pi i k t} .$$

$p(z)$ is hence set up as a linear combination of exponential functions or—after separation into real and imaginary part—of sine and cosine functions. Finding this p de facto implies the need to compute the coefficients c_k, and these are of course nothing else but the coefficients of the *(discrete) Fourier transformation (DFT)*. A famous and efficient algorithm for this task is the *Fast Fourier Transform (FFT)*.

With the previously introduced p and the abbreviation $\omega := e^{2\pi i/n}$ we obtain the following problem setting: Find n complex numbers $c_0, \ldots, c_{n-1}$ which satisfy

$$v_j = p(\omega^j) = \sum_{k=0}^{n-1} c_k \omega^{jk} \qquad \text{for} \qquad j = 0, 1, \ldots, n-1.$$

With some analysis ($\bar{\omega}$ the complex conjugation of ω) one can show that

$$c_k = \frac{1}{n} \sum_{j=0}^{n-1} v_j \bar{\omega}^{jk} \qquad \text{for} \qquad k = 0, 1, \ldots, n-1 \ .$$

We use c and v to denote the n-dimensional vectors of the discrete Fourier coefficients and the DFT input, resp. Furthermore, we let the matrix M be defined as $M = (\omega^{jk})_{0 \le j,k \le n-1}$. Hence, it holds using matrix-vector notation that

$$v = M \cdot c, \ c = \frac{1}{n} \cdot \overline{M} \cdot v \ .$$

This formula for the computation of the coefficients c_k defines precisely the *discrete Fourier transformation (DFT)* of the input data v_k. The formula for the computation of the values v_j from the Fourier coefficients c_k is called the *inverse discrete Fourier transformation (IDFT)*.

Apparently, the numbers of required arithmetic operations for the DFT and IDFT are of the order $\mathcal{O}(n^2)$.

The FFT algorithm, however, gets by with $\mathcal{O}(n \log n)$ operations for suitable n. The main idea is a recursive reordering of the coefficients into even and odd indices with subsequent exploitation of the recurrence of certain partial sums (*sorting phase* and *combination phase* by means of the so-called *butterfly operation*).

Close relatives, the *discrete cosine transformation* and the *fast cosine transformation*, are used, for example, in JPEG methods for image compression.

Numerical Quadrature

Numerical quadrature is known to be the numerical computation of a certain integral of the type

$$I(f) := \int_{\Omega} f(\mathbf{x}) \mathrm{d}\mathbf{x}$$

for a given function $f : \mathbb{R}^d \supseteq \Omega \to \mathbb{R}$, the *integrand*, and a given *integration domain* Ω. Here, we will exclusively be concerned with *univariate* quadrature, i.e., with the case $d = 1$ over an interval $\Omega = [a, b]$. Obviously, the greater challenges lie in the higher-dimensional case of the *multivariate* quadrature,

occuring for example, in statistics, physics or in mathematical finance and requires sophisticated numerical techniques. Numerical quadrature should only be used if all other solution or simplification techniques such as closed-form formulas or subdivision of the integration domain have failed. Most of the numerical integration techniques require sufficient smoothness (differentiability) of the integrand.

Almost all *quadrature rules*, i.e., rules for numerical quadrature, can be written as *weighted sums of function values (samples)*:

$$I(f) \approx Q(f) := \sum_{i=0}^{n} g_i f(x_i) =: \sum_{i=0}^{n} g_i y_i$$

with *weights* g_i and pairwise distinct *nodes* x_i, where $a \leq x_0 < x_1 < \ldots < x_{n-1} < x_n \leq b$. Since the evaluation of the integrand is often an expensive endeavor, one is interested in rules which allow for high accuracy (a small *quadrature error*) given moderate n.

The standard approach for the derivation of quadrature rules is to replace the integrand f by an easy to construct and easy to integrate approximation function $\tilde{f}$ which is then integrated *exactly*, i.e.,

$$Q(f) := \int_a^b \tilde{f}(x)\mathrm{d}x \ .$$

For simplicity, the approximant $\tilde{f}$ is frequently chosen as a *polynomial interpolant* $p(x)$ of $f(x)$ for the nodes x_i. In this case, the representation of $p(x)$ using the Lagrange polynomial $L_i(x)$ of degree n yields the weights g_i practically without cost:

$$Q(f) := \int_a^b p(x)\mathrm{d}x = \int_a^b \sum_{i=0}^{n} y_i L_i(x)\mathrm{d}x = \sum_{i=0}^{n} \left(y_i \cdot \int_a^b L_i(x)\mathrm{d}x \right) ,$$

where the weights g_i are given explicitly through,

$$g_i := \int_a^b L_i(x)\mathrm{d}x.$$

The integrals of the Lagrange polynomial clearly can be computed a priori—they depend on the chosen grid (the nodes), but not on the integrand f. As a result of the uniqueness of the interpolation problem it holds that $\sum_{i=0}^{n} L_i(x) \equiv 1$ and therefore $\sum_{i=0}^{n} g_i = b - a$. This implies that the sum of the weights always equals $b - a$ if a polynomial interpolant is used for the quadrature. For reasons of conditioning, one in general considers only rules with positive weights.

Simple Quadrature Rules

One distinguishes between *simple* and *composite* quadrature rules. A simple rule treats the entire integration domain $[a, b]$ of length $H := b - a$ in its entirety. A composite rule, however, decomposes the integration domain into subdomains in which simple rules are then applied, their individual sum then forms the overall approximation—a procedure that strongly resembles the spline interpolation.

The *Newton–Cotes-Formula* represent an important class of simple rules:

$$Q_{\mathrm{NC}(n)}(f) := I(p_n) ,$$

where p_n is the polynomial interpolant of f of degree n for the $n + 1$ equidistant nodes $x_i := a + H \cdot i/n$, $i = 0, \ldots, n$. The simplest representative is the *rectangle rule*

$$Q_R(f) := H \cdot f\left(\frac{a+b}{2}\right) = I(p_0) .$$

For its *remainder term* $R_R(f) := Q_R(f) - I(f)$ one can show the relationship,

$$R_R(f) := -H^3 \cdot \frac{f^{(2)}(\xi)}{24}$$

for some intermediate value $\xi \in]a, b[$ if f is twice continuously differentiable on $]a, b[$. Hence, polynomials of degree 0 or 1 is integrated *exactly*.

If one replaces the constant polynomial interpolant p_0 by its linear counterpart p_1, then one obtains the *trapezoidal rule*

$$Q_T(f) := H \cdot \frac{f(a) + f(b)}{2} = I(p_1)$$

with remainder,

$$R_T(f) = H^3 \cdot \frac{f^{(2)}(\xi)}{12} .$$

The polynomial of maximal degree that can be integrated exactly by a quadrature rule is called the *degree of accuracy*, or in short, the *accuracy* of the method. Hence, the rectangle rule as well as the trapezoidal rule have accuracy 1. For $p = 2$, one obtains *Simpson's rule*

$$Q_F(f) := H \cdot \frac{f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)}{6} = I(p_2)$$

with remainder

$$R_F(f) \;=\; H^5 \cdot \frac{f^{(4)}(\xi)}{2880} \;.$$

Assuming sufficiently high differentiability, one therefore obtains in the Newton–Cotes approach methods of higher order and higher accuracy for increasing n. However, for $n = 8$ and $n \geq 10$, there appear negative weights. As previously mentioned, the Newton–Cotes formulas become practically useless in these cases.

As a matter of principle, the problem existing for negative weights g_i for higher polynomial degree n does not imply that polynomial interpolants of higher degree are unsuitable for purposes of numerical quadrature. A possible remedy requires a deviation from the assumption that the nodes are equidistant. This is exactly what is rooted in the principle of the *Clenshaw–Curtis rules* in which the integration interval $[a, b]$ is now replaced by a uniform subdivision of the angles over the semi circle $[0, \pi]$.

As one can easily show, the nodes at the boundaries of the integration interval lie closer to each other compared to the nodes in the middle. The use of such guidelines in the construction of quadrature rules is beneficial since the weights appearing are always positive.

Composite Quadrature Rules

The most important composite rule is the *composite trapezoidal rule*. The integration interval $[a, b]$ is first subdivided into n subintervals of length $h := (b-a)/n$. The equidistant points $x_i := a + ih$, $i = 0, \ldots, n$ serve as nodes. The trapezoidal rule is then applied on each subinterval $[x_i, x_{i+1}]$. Finally, each contribution of the computed integral values are added to produce the integral value over the original interval:

$$Q_{\text{TS}}(f; h) \;:=\; h \cdot \left(\frac{f_0}{2} + f_1 + f_2 + \ldots + f_{n-1} + \frac{f_n}{2} \right) \;,$$

where $f_i := f(x_i)$. It holds that the trapezoidal sum has remainder,

$$R_{\text{TS}}(f; h) \;=\; h^2 \cdot (b-a) \cdot \frac{f^{(2)}(\xi)}{12} \;=\; h^2 \cdot H \cdot \frac{f^{(2)}(\xi)}{12} \;.$$

Compared to the trapezoidal rule, the accuracy remains 1 while the order is reduced to 2 (an order of magnitude is lost by the summation). One gains, however, a rule that is easy to implement for which one can arbitrarily increase n without encountering numerical difficulties.

Just as the composite trapezoidal rule represents a composite quadrature rule based upon the trapezoidal rule, the *Composite Simpson Rule* is the natural

generalization of Simpson's rule. Starting from the same subdivision of the integration interval $[a, b]$ as before, one now applies Simpson's rule over two combined neighboring subintervals:

$$Q_{\mathrm{ss}}(f;h) := \frac{h}{3} \cdot (f_0 + 4f_1 + 2f_2 + 4f_3 + 2f_4 + \ldots + 2f_{n-2} + 4f_{n-1} + f_n) .$$

Further Approaches for Numerical Quadrature

At this junction, let us refer to further important approaches for numerical quadrature. In *extrapolation*, one combines different computed approximations of lower order (e.g., trapezoidal sums for h, $h/2$, $h/4$, etc.) in a clever linear way in order to eliminate certain error terms thus resulting in a significantly better approximation. However, a prerequisite for the extrapolation technique is a high degree of smoothness for the integrand. Additional terms that are important in this context are the *Euler-Maclaurin sum formula* as well as *Romberg quadrature*.

For *Monte-Carlo quadrature*, a stochastic approach is applied. It is, in particular, very popular in the high-dimensional case where classical numerical methods often fail because of the so-called *curse of dimensionality* (a product approach making use of a rule with only 2 points in one dimension already requires 2^d points in d spatial dimensions). Colloquially, one can visualize the process as one in which nodes in the integration domain are selected from some uniform distribution. This defines the nodes where the integrand is evaluated, and simple averaging yields the overall result (obviously under consideration of the size of the domain). The error of the Monte-Carlo quadrature does not depend on the dimensionality; however, the convergence behavior in terms of the number of nodes n is only $\mathcal{O}(1/\sqrt{n})$.

Quasi Monte-Carlo methods or *Methods of minimal discrepancy* move in the same direction, however, they employ suitable sequences of determinitic nodes instead of randomly selected nodes.

The idea behind *Gauß quadrature* consists of placing the nodes so that polynomials of degree as high as possible can still be integrated exactly—i.e., one aims for a degree of accuracy as high as possible:

$$I(p) = \int_{-1}^{1} p(x)\mathrm{d}x \stackrel{!}{=} \sum_{i=1}^{n} g_i\, p(x_i)$$

for all $p \in \mathbb{P}_k$ with k as large as possible. One keeps a series expansion for f at the back of one's mind in which as many leading terms as possible shall be integrated exactly (high error order for decreasing interval width). The Gauß quadrature, which employs the roots of Legendre polynomial as nodes, attains the (hereby largest possible) accuracy degree $2n - 1$.

A very old and yet—especially considered from an algorithmic point of view—elegant approach for numerical quadrature originates from Archimedes. He is one of

the forefathers of the algorithmic paradigm *divide et impera* ubiquitious in computer science. The area under the integrand is thereby exhausted through a sequence of hierarchical (i.e., smaller and smaller) triangles. Similar to the Monte-Carlo quadrature, the main attraction of this approach is rooted in the highdimensional integrals. Here, competitive *sparse grid* algorithms can be formulated.

2.4.3 Direct Solution of Linear Systems of Equations

Linear Systems of Equations

Another important application area for numerical methods is *numerical linear algebra* that is concerned with the numerical solution of problems in linear algebra (matrix-vector product, computation of eigenvalues, solution of linear systems of equations). Of central importance here is the *solution of systems of linear equations*, i.e., for $A = (a_{i,j})_{1 \le i,j \le n} \in \mathbb{R}^{n,n}$ and $b = (b_i)_{1 \le i \le n} \in \mathbb{R}^n$, find $x \in \mathbb{R}^n$ such that $Ax = b$, which, among other applications, are generated in the discretization of models based on differential equations. One distinguishes between *fully populated* matrices (the number of non-zeros in A is of the order of the total number of matrix entries, i.e., $\mathcal{O}(n^2)$) and *sparse* matrices (typically $\mathcal{O}(n)$ or $\mathcal{O}(n \log(n))$ non-zeros). Often, sparse matrices have a *sparsity structure* (diagonal matrices, tridiagonal matrices, general banded structure), which simplifies the solving of the system.

With respect to solution techniques, one distinguishes between *direct* solvers, which yield the exact solution x (modulo round-off error), from *indirect* solvers which, starting from an initial approximation $x^{(0)}$, *iteratively* compute a (hopefully convergent) sequence of approximations $x^{(i)}$ without actually reaching x.

We first address the direct solvers; let the matrix A be non-singular. The explicit computation of the inverse A^{-1} is not performed due to complexity reasons. For the analysis of the conditioning of the problem $Ax = b$, as well as for the analysis of the convergence behavior of iterative methods, one needs the notion of the *condition number* $\kappa(A)$:

$$\|A\| := \max_{\|x\|=1} \|Ax\| , \qquad \kappa(A) := \|A\| \cdot \|A^{-1}\|,$$

where $\|.\|$ denotes a suitable vector or matrix norm, resp. One can show that the following inequality holds,

$$\frac{\|\delta x\|}{\|x\|} \le \frac{2\varepsilon\kappa(A)}{1 - \varepsilon\kappa(A)}$$

where ε denotes an upper bound for the relative input perturbations $\delta A/A$ or $\delta b/b$, resp. The larger the condition number $\kappa(A)$, the larger our upper bound on the right becomes for estimating the impact on the result, and therefore the worse

the conditioning of the problem "solve $Ax = b$". The term "condition number" is therefore a meaningful choice—it represents a measure for the conditioning. Only if $\varepsilon\kappa(A) \ll 1$, which represents a restriction to the order of magnitude of the admissible input perturbation, can the numerical solution of the problem make sense. We then would have the conditioning under control.

An important quantity is the *residual* r. For the approximation $\tilde{x}$ of x, r is defined as

$$r := b - A\tilde{x} = A(x - \tilde{x}) =: -Ae$$

with *error* $e := \tilde{x} - x$. Error and residual can be of very different orders of magnitude. In particular, a small residual does not necessarily imply a small error—in fact, the correlation also involves the condition number $\kappa(A)$. Nevertheless, the residual is helpful: $r = b - A\tilde{x} \Leftrightarrow A\tilde{x} = b - r$ shows that an approximate solution with a small residual represents an acceptable result.

Gaussian Elimination and the *LU* Decomposition

From linear algebra comes the well-known classical method for solving linear systems of equations, *Gaussian elimination*, and it is the natural generalization of solving two equations in two unknowns: Solve one of the n equations (say, the first) for an unknown (say, x_1); substitute the resulting (and dependent on $x_2, \ldots, x_n$) expression for x_1 in the other $n - 1$ equations—x_1 is thus *eliminated* from these; solve the resulting system of $n - 1$ equations in the remaining $n - 1$ unknowns correspondingly and continue until there remains only x_n in a single equation which can therefore be explicitly computed; now plug in x_n in the elimination equation for x_{n-1} so that one obtains x_{n-1} explicitly; continue until finally the elimination equation of x_1 yields the value for x_1 by substituting the (known) values for $x_2, \ldots, x_n$. Visually, the elimination of x_1 means that A and b are modified such that the first column contains only zeros underneath $a_{1,1}$, where the new system (consisting of the first equation and the remaining x_1-free equations) is naturally solved by the same vector x as the old one!

Gaussian elimination is equivalent to the so-called *LU decomposition*, in which A is *factored* into the product $A = LU$ with unit lower triangular matrix L (ones along the diagonal) and upper triangular matrix U. In the special case of a positive definite A, the *Cholesky decomposition* yields a symmetric factorization $A = \tilde{L} \cdot \tilde{L}^T$ which saves about half the cost. The complexity is cubic in the number of unknowns in all three cases.

For Gaussian elimination and LU decomposition, one divides by the diagonal elements (the so-called *pivots*), which may possibly be zero. If a zero occurs, one has to modify the algorithm to enforce an admissible situation, i.e., a non-zero on the diagonal, by row or column exchange (which is of course possible if A is non-singular). Possible exchange partners for a zero on the diagonal can either be found in column i below the diagonal (*column pivoting*) or in the entire remaining part of

the matrix (everything on or after row and column $i+1$, *complete pivoting*). Finally: Even if no zeros occur—for numerical reasons, pivoting is always advisable.

2.4.4 Iteration Methods

Notion of Iteration

Linear systems of equations which need to be solved numerically often originate in the discretization of ordinary (for boundary value problems) or partial differential equations. The direct solution methods previously discussed are generally out of the question. First, n is usually so large (in general, n is directly correlated with the number of grid points and this leads to very large n, in particular, for unsteady partial differential equations (three spatial variables and one temporal variable)) that a cubic effort is unacceptable. Second, such matrices are typically sparse and exhibit a certain structure which naturally corresponds to a decreasing effect on storage and computational time; elimination methods typically destroy this special structure and thus reverse the advantages. Furthermore, the accuracy provided from the exact direct solution often exceeds that required for the simulation.

Therefore, *iterative* methods are preferred for large and sparse matrices or linear systems of equations. These start with (in general, not just for the situation of linear systems of equations) an *initial approximation* $x^{(0)}$ and generate a sequence of approximation $x^{(i)}$, $i = 1, 2, \ldots$ which, in the case of convergence, converge toward the exact solution x. In this context one also refers to the *method function* Φ of the iteration, $\Phi(x_i) := x_{i+1}$, which determines the new iteration value from the current one. In the case of convergence ($\lim x_i = x$), the iteration has a *fixed point* $\Phi(x) = x$. For sparse matrices, an iteration step typically costs (at least) $\mathcal{O}(n)$ computational operations. Thus, the construction of iterative algorithms is affected by how many iteration steps will be required to reach a certain given accuracy.

Relaxation Methods

Perhaps the oldest iterative methods for the solution of linear systems of equations $Ax = b$ with $A \in \mathbb{R}^{n,n}$ and $x, b \in \mathbb{R}^n$ are the so-called *relaxation methods*: the *Richardson* method, the *Jacobi* method, the *Gauß–Seidel* method as well as the *overrelaxation (SOR)* method. The starting point for each of these methods is the residual $r^{(i)} := b - Ax^{(i)} = -Ae^{(i)}$. Since $e^{(i)}$ is unavailable (the error cannot be computed without knowledge of the exact solution x), it is reasoned by the above relationship to make use of the vector $r^{(i)}$ as the *direction* in which we want to search for an improvement of $x^{(i)}$. The Richardson method takes the residual directly as a correction for $x^{(i)}$. The Jacobi- and Gauß–Seidel methods try a little harder. Their idea for the correction of the k-th component of $x^{(i)}$ is the elimination of $r_k^{(i)}$.

The SOR method and its counterpart, *damped* relaxation, additionally account for the fact that such a correction often overshoots or undershoots its mark, respectively.

In the algorithmic formulation, the four methods are represented as follows:

- *Richardson iteration*:
  `for i = 0,1,...`
  `for k = 1,...,n:` $x_k^{(i+1)} := x_k^{(i)} + r_k^{(i)}$
  Here, the residual $r^{(i)}$ is simply used componentwise as a correction to the current approximation $x^{(i)}$.
- *Jacobi iteration*:
  `for i = 0,1,...`
  `for k = 1,...,n:` $y_k := \frac{1}{a_{kk}} \cdot r_k^{(i)}$
  `for k = 1,...,n:` $x_k^{(i+1)} := x_k^{(i)} + y_k$
  In every partial step k of step i, a correction y_k is computed and stored. If applied immediately, this would lead to the (temporary) disappearance of the k-component of the residual $r^{(i)}$ (easily verified by insertion). Equation k would be solved exactly with this current approximation for x—a progress which would be immediately lost again in the subsequent partial step $k+1$. However, these component corrections are not performed immediately but only at the end of an iteration step (second k-loop).
- *Gauß–Seidel iteration* :
  `for i = 0,1,...`
  `for k = 1,...,n:` $r_k^{(i)} := b_k - \sum_{j=1}^{k-1} a_{kj} x_j^{(i+1)} - \sum_{j=k}^{n} a_{kj} x_j^{(i)}$
  $y_k := \frac{1}{a_{kk}} \cdot r_k^{(i)}, \quad x_k^{(i+1)} := x_k^{(i)} + y_k$
  Here, the same correction as in the Jacobi method is computed, however, the update is now always performed immediately and not only at the end of the iteration step. Thus, at the update of component k, the modified new values for the components 1 to $k-1$ are already available.
- Sometimes, in each of the three sketched methods a *damping* (multiplication of the correction with a factor $0 < \alpha < 1$) or an *overrelaxation* (factor $1 < \alpha < 2$), resp., leads to better convergence behavior:

$$x_k^{(i+1)} := x_k^{(i)} + \alpha y_k \ .$$

  In the Gauß–Seidel case, the version with $\alpha > 1$ is predominantly used which is referred to as the *SOR method*, in the Jacobi case, on the other side, damping is mostly used.

For a brief convergence analysis of the above methods we need an algebraic formulation (instead of the algorithmic). All the approaches presented are based on the simple idea of writing the matrix A as a sum $A = M + (A - M)$, where $Mx = b$ is very easy to solve and the difference $A - M$ should not be too

large wrt. a matrix norm. By means of such a suitable M, the Richardson, Jacobi, Gauß–Seidel- and SOR methods can be written as

$$Mx^{(i+1)} + (A - M)x^{(i)} = b$$

or, solved for $x^{(i+1)}$, as

$$x^{(i+1)} := x^{(i)} + M^{-1}r^{(i)} .$$

Furthermore, we decompose A additively into its diagonal part D_A, its strictly lower triangular part L_A as well as its strictly upper triangular part U_A:

$$A =: L_A + D_A + U_A .$$

Thus we can show the following relationships:

- Richardson: $M := I$,
- Jacobi: $M := D_A$,
- Gauß–Seidel: $M := D_A + L_A$,
- SOR: $M := \frac{1}{\alpha} D_A + L_A$.

Concerning the convergence, there exists a direct consequence from the approach $Mx^{(i+1)} + (A-M)x^{(i)} = b$: If the sequence $\left(x^{(i)}\right)$ converges, then the limit is the exact solution x of our system $Ax = b$. For the analysis, let us further assume that the *iteration matrix* $-M^{-1}(A-M)$ (i.e., the matrix that is applied to $e^{(i)}$ in order to obtain $e^{(i+1)}$) is symmetric. Then the spectral radius ϱ (i.e., the absolute value of the largest eigenvalue) is the critical quantity for the determination of the convergence behavior:

$$\left(\forall x^{(0)} \in \mathbb{R}^n : \lim_{i\to\infty} x^{(i)} = x = A^{-1}b\right) \quad \Leftrightarrow \quad \varrho < 1 .$$

To see this, one subtracts $Mx + (A - M)x = b$ from the above equation of the general approach:

$$Me^{(i+1)} + (A - M)\mathrm{e}^{(i)} = 0 \quad \Leftrightarrow \quad e^{(i+1)} = -M^{-1}(A - M)e^{(i)} .$$

If all eigenvalues are less than 1 in absolute value and hence $\varrho < 1$ holds, then all error components are reduced at each iteration step. For the case that $\varrho > 1$, generally at least one error component will increase. It is then natural, even intuitive, that the objective of the construction of iterative methods is a spectral radius of the iteration matrix as small as possible (as close to zero as possible).

There are a number of results regarding the convergence of the various methods, of which a few of the more significant ones shall be mentioned below:

- Necessary for the convergence of the SOR method is $0 < \alpha < 2$.
- If A is positive definite, then the SOR method (for $0 < \alpha < 2$) as well as the Gauß–Seidel iteration converge.

- If both A and $2D_A - A$ are positive definite, then the Jacobi method converges.
- If A is strictly diagonally dominant (i.e., $a_{ii} > \sum_{j \neq i} \mid a_{ij} \mid$ for all i), then the Jacobi and the Gauß–Seidel methods converge.
- In certain cases the optimal parameter α can be determined (ϱ minimal so that the error reduction per iteration step becomes maximal).

Obviously, ϱ is not only critical for the question whether the iteration directive converges at all, but also for its quality, i.e. its convergence speed: The smaller ϱ, the faster are all the components of the error $e^{(i)}$ reduced in each iteration step. In practice, the above results are unfortunately of rather theoretical value since ϱ is often so close to 1 so that—despite convergence—the number of iteration steps required to reach a sufficient accuracy is far too large. An important example scenario is the discretization of partial differential equations. Here it is typical that ϱ depends on the problem size n and thus on the resolution h of the underlying grid, i.e. for example

$$\varrho \;=\; \mathcal{O}(1 - h_l^2) \;=\; \mathcal{O}\left(1 - \frac{1}{4^l}\right)$$

for a mesh width $h_l = 2^{-l}$. This is an enormous disadvantage: The finer and hence more accurate our grid, the more miserable the resulting convergence behavior becomes for our iterative methods. It is therefore a must to develop faster iterative solvers such as *multigrid methods*, for example.

Minimization Techniques

One of the best-known methods for the solution of linear systems of equations, the method of *conjugate gradients (cg)*, is based on a different principle than that of relaxation. In order to see this, we take an indirect approach for the problem of solving linear systems of equations. In the following, let $A \in \mathbb{R}^{n,n}$ be symmetric and positive definite. In this case, the solution of $Ax = b$ is equivalent to the minimization of the quadratic function

$$f(x) \;:=\; \frac{1}{2} x^T A x - b^T x + c$$

for an arbitrary scalar constant $c \in \mathbb{R}$. Since A is positive definite, the hypersurface that is built from $z := f(x)$ is a paraboloid in $\mathbb{R}^{n+1}$ with n-dimensional ellipsoids as isosurfaces $f(x) = const.$, with f having a global minimum at x. The equivalence of the problems is obvious:

$$f'(x) \;=\; \frac{1}{2} A^T x + \frac{1}{2} A x - b \;=\; Ax - b \;=\; -r(x) \;=\; 0 \quad \Leftrightarrow \quad Ax = b\,.$$

By means of this reformulation, one can now employ *optimization methods* and thus extend the spectrum of potential solution techniques. Let us consider techniques for searching for a minimum. A straightforward possibility is provided by the *method of steepest descent*: For $i = 0, 1, \ldots$, repeat,

$$\begin{aligned} r^{(i)} &:= b - Ax^{(i)} , \\ \alpha_i &:= \frac{{r^{(i)}}^T r^{(i)}}{{r^{(i)}}^T A r^{(i)}} , \\ x^{(i+1)} &:= x^{(i)} + \alpha_i r^{(i)} , \end{aligned}$$

or begin with $r^{(0)} := b - Ax^{(0)}$ and repeat for $i = 0, 1, \ldots$

$$\begin{aligned} \alpha_i &:= \frac{{r^{(i)}}^T r^{(i)}}{{r^{(i)}}^T A r^{(i)}} , \\ x^{(i+1)} &:= x^{(i)} + \alpha_i r^{(i)} , \\ r^{(i+1)} &:= r^{(i)} - \alpha_i A r^{(i)} , \end{aligned}$$

which saves one of the two matrix-vector products (the only really expensive step in the algorithm). The method of steepest descent searches for an improvement in the direction of the negative gradient $-f'(x^{(i)}) = r^{(i)}$ which in fact points to the steepest descent (thus the name). It would obviously be better to search in the direction of the error $e^{(i)} := x^{(i)} - x$, but unfortunately this is not known. But the direction itself is insufficient for one also needs a suitable step length. To this end, we look for the minimum of $f(x^{(i)} + \alpha_i r^{(i)})$ when interpreted as a function of α_i (partial derivative with respect to α_i is set to zero), which after a brief calculation yields the above value for α_i. If, instead of the residuals $r^{(i)}$, one chooses alternatingly the unit vectors in the coordinate directions as search vectors and then once again determines the optimal step widths with respect to these search directions, one actually ends up with the Gauß–Seidel iteration. Thus, despite all differences there are relationships between the relaxation and minimization approach.

The convergence behavior of the method of steepest descent is modest. One of the few trivial special cases is the identity matrix: There, the isosurfaces are spheres, the gradient always points to the center (the minimum), and one reaches the goal in one step! In general, one eventually lands arbitrarily close to the minimum, however, this may take arbitrarily long (since we can always destroy part of what we have previously attained). In order to alleviate this disadvantage we stick with our minimization approach but in addition we continue to look for better search directions. If all search directions would be orthogonal, and if the error after i steps would be orthogonal to all previous search directions, then the optimality previously attained would never be lost and one would be at the minimum after at most n steps—just as for the case of a direct solver. For this reason one

calls the cg-method and cg-based methods *semi-iterative methods*. One obtains the cg-method if one pursues the idea of improved search directions, in particular employs *conjugate directions* (two vectors x and y are called *A-orthogonal* or *conjugate* if $x^T Ay = 0$ holds) and combines it all with an inexpensive strategy for computing such conjugate directions. The convergence behavior is clearly better than for the steepest descent, but the "slow-down" effect for increasing problem size is still not overcome.

Nonlinear Equations

This much about linear solvers: Many realistic models, unfortunately, are not linear. Thus, one has to also contemplate the solution of *nonlinear equations*. Typically, this can no longer be done directly, but only iteratively. Here in the following, we restrict ourselves to the easier case in which $n = 1$ (i.e., *one* nonlinear equation). Therefore, consider a continuously differentiable (nonlinear) function $f : \mathbb{R} \to \mathbb{R}$ which has a root $\bar{x} \in]a, b[$ (one may think of a search for roots or extrema). Consider an iteration rule (to be determined) which yields a sequence of approximations, $(x^{(i)})$, $i = 0, 1, \ldots$, that (hopefully) converge toward the simple root $\bar{x}$ of $f(x)$. As a measure of the *convergence speed* one considers the reduction of the error in each step and, in the convergent case

$$| x^{(i+1)} - \bar{x} | \ \leq \ c \cdot | x^{(i)} - \bar{x} |^{\alpha} ,$$

speaks of different types of convergence depending on the maximal possible value of the parameters α: *linear* ($\alpha = 1$ and in addition $0 < c < 1$) or *quadratic* ($\alpha = 2$) convergence etc. There exist *conditionally* or *locally* convergent methods for which convergence is guaranteed only for an already sufficiently good starting value $x^{(0)}$, and *unconditionally* or *globally* convergent methods in which the iteration leads to a root of f independent of the choice of the starting point.

The simple methods are the *bisection method* (start with $[c, d] \subseteq [a, b]$ with $f(c) \cdot f(d) \leq 0$—continuity of f guarantees the existence of (at least) one root in $[c, d]$—and cut each successive search interval in half), the *Regula falsi* (a variant of the bisection method in which the x-axis-intersection of the line through the two points $(c, f(c))$ and $(d, f(d))$ is selected instead of the interval midpoint as an end point of the new and smaller interval), as well as the *secant method* (begin with *two* initial approximations $x^{(0)}$ and $x^{(1)}$; in the following, one determines $x^{(i+1)}$ out of $x^{(i-1)}$ and $x^{(i)}$ by computing the root of the line through $(x^{(i-1)}, f(x^{(i-1)}))$ and $(x^{(i)}, f(x^{(i)}))$ (of the *secant* $s(x)$)).

The most famous method is the *Newton method*. Here one begins with *one* initial approximation $x^{(0)}$ and subsequently determines $x^{(i+1)}$ out of $x^{(i)}$ by searching for the root of the *tangent line* $t(x)$ of $f(x)$ at $x^{(i)}$ (*linearization*: replace f by its tangent or its Taylor polynomial of first degree, resp.):

$$
\begin{aligned}
t(x) &:= f(x^{(i)}) + f'(x^{(i)}) \cdot (x - x^{(i)}) \ , \\
0 \ = \ t(x^{(i+1)}) \ &= \ f(x^{(i)}) + f'(x^{(i)}) \cdot (x^{(i+1)} - x^{(i)}) \ , \\
x^{(i+1)} &:= x^{(i)} - \frac{f(x^{(i)})}{f'(x^{(i)})} \ .
\end{aligned}
$$

The *order of convergence* of the methods introduced is globally linear for bisection and regula falsi, locally quadratic for Newton and locally 1.618 for the secant method. However, since a Newton step requires an evaluation of f and f', resp., it needs to be compared to *two* steps of the other methods. Furthermore, the computation of the derivative often poses a problem.

In most practical applications it holds that $n \gg 1$ (for example, the unknowns are the function values at the grid points!), so that we are confronted with very large nonlinear systems of equations. In the higher dimensional case, the simple derivative $f'(x)$ is replaced by the *Jacobi matrix* $F'(x)$, the matrix of partial derivatives of all vector components of F with respect to all variables. The Newton iteration rule then reads as

$$
x^{(i+1)} \ := \ x^{(i)} - F'(x^{(i)})^{-1} F(x^{(i)}) \ ,
$$

where obviously the matrix $F'(x^{(i)})$ is not inverted but the respective linear system of equations with the right hand side $F(x^{(i)})$ is solved (directly):

compute $F'(x)$;
decompose $F'(x) := LR$;
solve $LRs \ = \ F(x)$;
update $x \ := \ x - s$;
evaluate $F(x)$;

The repeated computation of the Jacobi matrix is highly work-intensive, or is often only possible approximately, since most of the time differentiation must be performed numerically. In addition, the direct solution of a linear system of equations at each Newton step is expensive. Therefore, the Newton method has only been the starting point for a multitude of algorithmic developments. In the *Newton-Chord* or *Shamanskii method*, resp., the Jacobi matrix is not computed and inverted at every Newton step, but $F'(x^{(0)})$ is always used (Chord) or $F'(x^{(i)})$ is always used for several Newton steps (Shamanskii). In the *inexact Newton method*, the linear system of equations is not solved directly (i.e., exactly or via LU decomposition) in each Newton step, but iteratively; one speaks of an *inner* iteration within the (*outer*) Newton iteration. Finally, in the *Quasi-Newton method*, a sequence $B^{(i)}$ of approximations for $F'(\bar{x})$ is generated, and this is not done by expensive new computations but rather by inexpensive *updates*. One exploits the

fact that a *rank-1-update* $(B + uv^T)$ with two arbitrary vectors $u, v \in \mathbb{R}^n$ (invertible if and only if $1 + v^T B^{-1} u \neq 0$) is easy to invert:

$$\left(B + uv^T\right)^{-1} \;=\; \left(I - \frac{(B^{-1}u)v^T}{1 + v^T B^{-1} u}\right) B^{-1} \,.$$

Broyden gave a suitable choice for u and v (s as above in the algorithm):

$$B^{(i+1)} \;:=\; B^{(i)} + \frac{F(x^{(i+1)})s^T}{s^T s} \,.$$

2.4.5 Ordinary Differential Equations

Differential Equations

One of the most important application areas for numerical methods are *differential equations*, see also Sect. 2.2.2 on analysis. In *ordinary differential equations (ODE)*, whose numerical properties we want to discuss in the following, there is only one independent variable appearing. Simple applications include, for example, the oscillating pendulum

$$\ddot{y}(t) \;=\; -\,y(t)$$

having solution $y(t) = c_1 \cdot \sin(t) + c_2 \cdot \cos(t)$, or the exponential growth

$$\dot{y}(t) \;=\; y(t)$$

with the solution $y(t) = c \cdot \mathrm{e}^t$. In *partial differential equations (PDE)*, there appear several independent variables. Here, simple application examples are the *Poisson equation* in 2 D, which, for example, describes the deformation u of a membrane which is fixed at the boundary under an outer load f:

$$\Delta u(x, y) \;:=\; u_{xx}(x, y) + u_{yy}(x, y) \;=\; f(x, y) \quad \text{on } [0, 1]^2$$

(here, the explicit description of the solution is already becoming much harder) or the *heat equation* in 1 D, which, for example, describes the temperature distribution T in a metal wire with given temperature at the endpoints:

$$T_t(x, t) \;=\; T_{xx}(x, t) \quad \text{auf } [0, 1]^2$$

(here one has an *unsteady equation* due to the time dependency).

As it has already been mentioned in Sect. 2.2.2, the differential equation alone, in general, does not uniquely determine the solution, rather it requires additional

conditions. Such conditions appear as *initial conditions* (such as the population size at the beginning of the common era) or as *boundary conditions* (a space shuttle, after all, should start and land at well-defined locations). For such cases, one is looking for a function y that satisfies the differential equation *and* these conditions. Correspondingly, one speaks of *initial value problems (IVP)* or *boundary value problems (BVP)*. In this section, we will only consider IVPs of ODEs, in particular of the type

$$\dot{y}(t) \;=\; f(t, y(t)) \;, \qquad y(t_0) \;=\; y_0 \;.$$

Here we require *one* initial condition since it is an ODE of *first order* (only the first derivative).

As an example we consider the ODE,

$$\dot{y}(t) = -\frac{1}{10} y(t) + \frac{1}{10} \;, \tag{2.1}$$

which, as a linear differential equation with constant coefficients, is one of the simplest conceivable differential equations of the above type. Its solutions can be stated directly as $y(t) = 1 + c \cdot \mathrm{e}^{-t/10}$ with parameter $c \in \mathbb{R}$ as a degree of freedom which allows the flexibility to satisfy an initial condition $y(0) = 1 + c \stackrel{!}{=} y_0$. Figure 2.1 shows solution curves for different initial values y_0.

By means of this simple example we can think about the ways to make at least qualitative statements pertaining to the solutions in complicated cases in which an explicit solution cannot be stated without further ado. Simultaneously, these considerations are the entry point to the numerical solution of ODEs for which reason this discussion is placed in the section on numerical properties instead of the section on analysis.

The idea is simple: We follow a particle in the $t - y$-plane which moves along a curve $\{(t, y(t)), t \in \mathbb{R}_+\}$ where $y(t)$ satisfies the differential equation. Such a curve is called the *trajectory* or *path*. If we assume uniform motion with velocity 1 in t-direction, then the velocity in the y-direction has to be just $\dot{y} = f(t, y(t))$ in order to stay on the trajectory: For each point $(t, y(t))$, the right hand side of the differential equation gives us the direction in which to proceed. If one supplies the $t - y$-plane with a *direction field*—i.e., at every point (t, y) with the vector with component 1 in t-direction and $f(t, y)$ in the y-direction, Fig. 2.1 shows some direction arrows for our example problem—then one can sketch a solution (or approximate it numerically) by exploiting the fact that at every point the direction vector is tangential to the curve.

An example of a qualitative property of a differential equation, such as all solutions converging to 1 for arbitrary initial values, could have been derived from the direction field even without knowledge of the explicit solution. The fact that the differential equation is *autonomous* due to its constant coefficients (the right hand side is of the form $f(y(t))$ without direct dependence of t), is also not necessary for such qualitative considerations (this will be different for the systems of differential

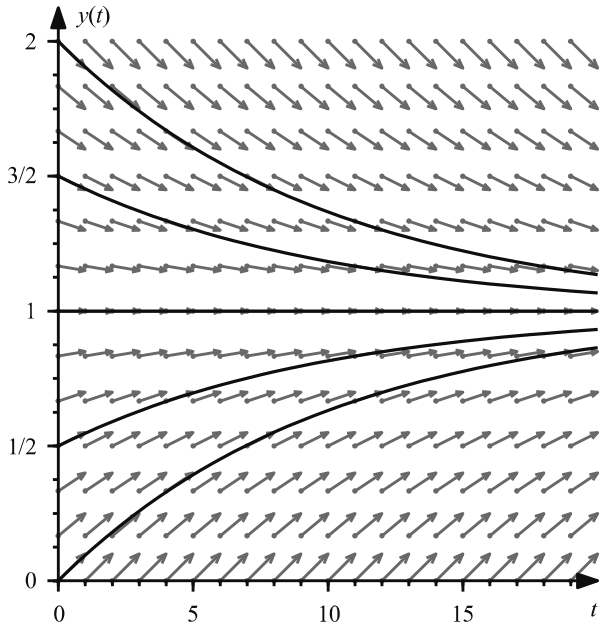

Fig. 2.1 Solutions and direction field of the differential equation (2.1)

equations which will be subsequently discussed). Altogether, direction fields are a useful tool to understand an ODE even before one attempts an explicit or numerical solution.

Finally, we will briefly turn to a very simple class of *systems of ODEs*, the *linear systems with constant coefficients*. As before, we distinguish the *homogeneous* case

$$\dot{x}(t) \;=\; A \cdot x(t)\;, \quad x(t_0) \;=\; x_0\;,$$

as well as the *nonhomogeneous* case

$$\dot{x}(t) \;=\; A \cdot x(t) + c\;, \quad x(t_0) \;=\; x_0\;,$$

where $A \in \mathbb{R}^{n,n}$, $c \in \mathbb{R}^n$ and $x \in \mathbb{R}^n$. For the homogeneous system, the approach $x(t) := \mathrm{e}^{\lambda t} \cdot v$ with initially undetermined parameters $\lambda \in \mathbb{R}$ and $v \in \mathbb{R}^n$ yields after substitution into the ODE that there must hold $Av = \lambda v$. Thus, if one manages successfully to represent the initial vector x_0 as a linear combination of eigenvectors v_i associated with eigenvalues λ_i of A, $x_0 = \sum_{i=1}^{n} v_i$, then

$$x(t) \;:=\; \sum_{i=1}^{n} \mathrm{e}^{\lambda_i t} \cdot v_i$$

satisfies the ODE as well as the initial condition. Somewhat obtained as a freebie, we also learn that the eigenvalues of the system matrix A determine the behavior of the solution. In the case in which all eigenvalues have negative real parts, the solution will converge to zero over time; at least one positive real part leads to $\|x(t)\| \to \infty$ for $t \to \infty$; non-vanishing imaginary parts imply the presence of oscillations.

In the non-homogeneous case one chooses (assuming that the matrix A is regular) $x_\infty := -A^{-1}c$ and with one solution $z(t)$ of the homogeneous equation obtains the solution $x(t) := z(t) + x_\infty$ to the nonhomogeneous system because $\dot{x}(t) = \dot{z}(t) = A \cdot z(t) = A \cdot (x(t) + A^{-1}c) = A \cdot x(t) + c$. Compared to the homogeneous case, the solution is only shifted by x_∞, which has no influence on the convergence, divergence or oscillation behavior, resp.

As an example we consider the following system:

$$\begin{pmatrix} \dot{x}(t) \\ \dot{y}(t) \end{pmatrix} = \begin{pmatrix} -1/10 & 1/20 \\ 1/20 & -1/10 \end{pmatrix} \begin{pmatrix} x(t) \\ y(t) \end{pmatrix} + \begin{pmatrix} 3/40 \\ 0 \end{pmatrix}. \tag{2.2}$$

The system matrix has the eigenvectors

$$v_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

associated with the eigenvalue $\lambda_1 = -0.05$ and

$$v_2 = \begin{pmatrix} -1 \\ 1 \end{pmatrix}$$

associated with the eigenvalue $\lambda_2 = -0.15$. Since both eigenvalues are real and negative, the solutions thus converge to the *equilibrium point*

$$\bar{x} = -A^{-1} \begin{pmatrix} 3/40 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1/2 \end{pmatrix}$$

for arbitrary initial values.Figure 2.2 shows several solution curves in the $x - y$-plane for arbitrary initial values, while Fig. 2.3 shows the components $x(t)$, $y(t)$ of the solution and their dependence on t. The direction field can still be drawn in the case of autonomous systems of differential equations with two components if one imagines the t-axis (which makes no contribution to the direction field) to be perpendicular to the drawing plane and then marks the direction vector $(\dot{x}, \dot{y})$ for a point in the $x - y$-plane.

A completely different behavior is displayed by the solution curves of the system

$$\begin{pmatrix} \dot{x}(t) \\ \dot{y}(t) \end{pmatrix} = \begin{pmatrix} -1/20 & 1/10 \\ 1/10 & -1/20 \end{pmatrix} \begin{pmatrix} x(t) \\ y(t) \end{pmatrix} + \begin{pmatrix} 0 \\ -3/40 \end{pmatrix}, \tag{2.3}$$

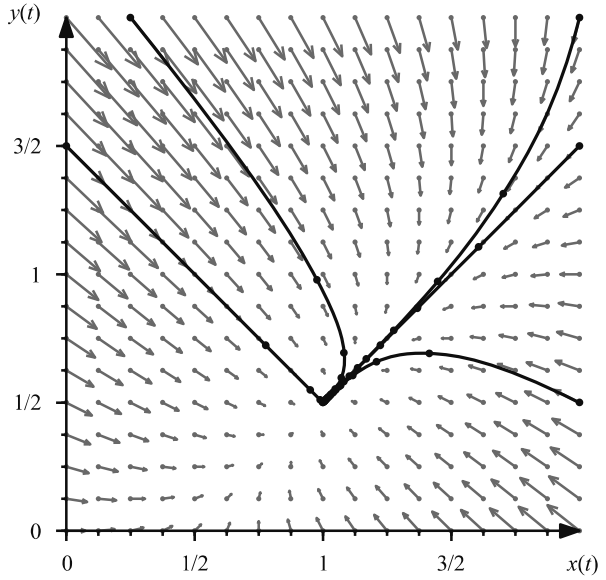

Fig. 2.2 Solutions corresponding to the initial values $(x_0, y_0) = (0, 3/2)$, $(1/4, 2)$, $(2, 2)$, $(2, 3/2)$ and $(2, 1/2)$ and the direction field of the system of differential equations (2.2), *filled circle* symbols represent the step $\delta t = 10$

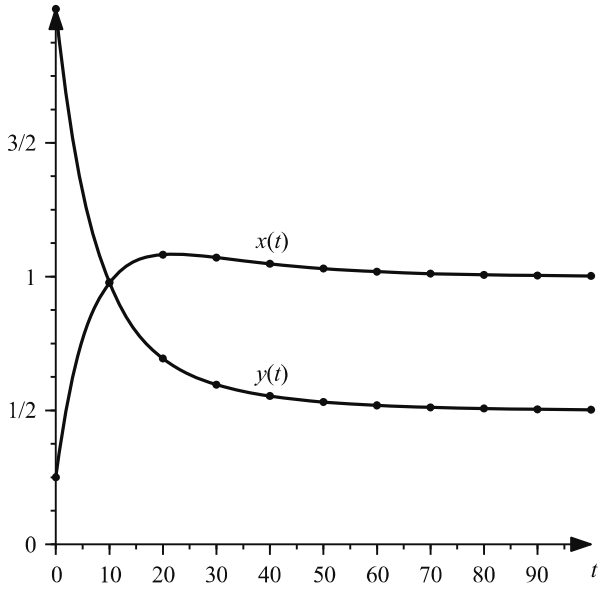

Fig. 2.3 Solution components of the system of differential equations (2.2) with initial values $(x_0, y_0) = (1/4, 2)$

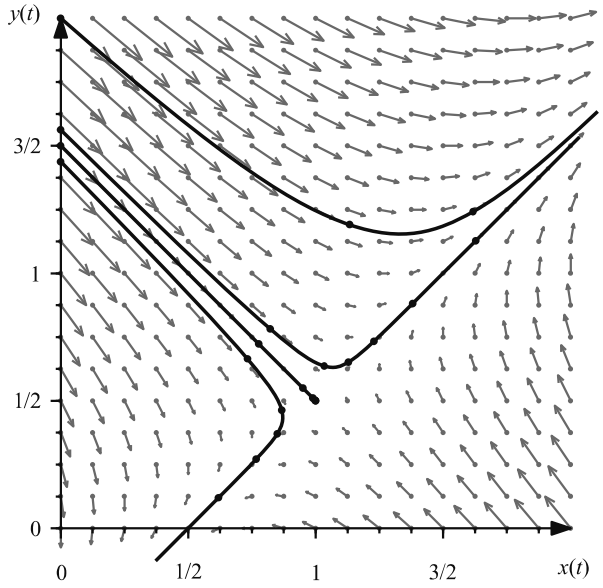

Fig. 2.4 Solutions for the initial values $(x_0, y_0) = (0, 3/2)$, $(0, 3/2 \pm 1/16)$ and $(0, 2)$ and the direction field of the system of differential equations (2.3)

in which the system matrix has the same eigenvectors v_1 and v_2 as in the previous example, but the eigenvalue $\lambda_1 = +0.05$ associated with v_1 is now positive (the other eigenvalue $\lambda_2 = -0.15$ and the equilibrium point x_∞ remain unchanged). Figure 2.4 shows the effects from the change in sign of λ_1: now only those solution curves converge toward x_∞ for which the initial values lie exactly on the line in the direction v_2 from the equilibrium point. In the case of a minor perturbation, the solution curve initially runs almost parallel until the positive eigenvalue λ_1 has a sudden and almost explosive effect—the solution curve is redirected in the direction of v_1 and the velocity by which the solution distances itself from the equilibrium point grows exponentially. Hence, the existence of a positive eigenvalue leads to an unstable equilibrium which will be lost by an arbitrarily small perturbation.

Now let us address the numerical properties of initial value problems of ordinary differential equations. Particularly troublesome, of course, are—as always—the ill-conditioned problems which we will not address in the following. Nevertheless, to serve as a deterrent, we provide an ill-conditioned IVP as an example. Let the equation $\ddot{y}(t) - N\dot{y}(t) - (N+1)y(t) = 0$, and $t \geq 0$ be given with the initial conditions $y(0) = 1$, $\dot{y}(0) = -1$ and the solution $y(t) = \mathrm{e}^{-t}$. If we now perturb just the one initial condition $y_\varepsilon(0) = 1 + \varepsilon$, a new solution $y_\varepsilon(t) = (1 + \frac{N+1}{N+2}\varepsilon)\mathrm{e}^{-t} + \frac{\varepsilon}{N+2}\mathrm{e}^{(N+1)t}$ results. Obviously, $y(t)$ and $y_\varepsilon(t)$ have a completely different character; in particular, $y(t)$ goes to zero as $t \to \infty$, whereas $y_\varepsilon(t)$ grows without bound for $N + 1 > 0$, and this occurs for arbitrary $\varepsilon > 0$ (i.e., in particular arbitrarily small)! This shows that the smallest of perturbations

in the input data (here one of the two initial conditions) can thus have a disastrous effect on the solution of the IVP—a clear case of miserable conditioning!

Finite Differences

We now turn our attention to specific solution algorithms. In the following we consider the general IVP of first order $\dot{y}(t) = f(t, y(t))$, $y(a) = y_a$, $t \in [a, b]$ just mentioned and assume it has a unique solution. If f does not depend on its second argument y, then this is a simple integration problem! The starting point is, as always, the *discretization*, i.e., here: Replace derivatives or *differential quotients* with *difference quotients* or *finite differences*, resp. For example, the *forward*, *backward* or *central difference*

$$\frac{y(t+\delta t) - y(t)}{\delta t} \quad \text{or} \quad \frac{y(t) - y(t-\delta t)}{\delta t} \quad \text{or} \quad \frac{y(t+\delta t) - y(t-\delta t)}{2 \cdot \delta t}$$

for $\dot{y}(t)$ respectively or, for IVPs of second order,

$$\frac{\frac{y(t+\delta t)-y(t)}{\delta t} - \frac{y(t)-y(t-\delta t)}{\delta t}}{\delta t} = \frac{y(t+\delta t) - 2 \cdot y(t) + y(t-\delta t)}{(\delta t)^2}$$

for $\ddot{y}(t)$, etc. From above, the first of the approximations for $\dot{y}(t)$ leads to

$$\begin{aligned} y(a+\delta t) &\approx y(a) + \delta t \cdot f(t, y(a)) \ , \qquad \text{also} \\ y_{k+1} &:= y_k + \delta t \cdot f(t_k, y_k) \ , \\ t_k &= a + k\delta t \ , \ k = 0, 1, \ldots, N \ , \ a + N \cdot \delta t = b \end{aligned}$$

and is the simplest rule to produce discrete approximations y_k for $y(t_k)$. Hence, at the time step t_k one takes the approximation y_k that has been already computed and then, together with f, calculates an approximation for the slope (derivative) of y and uses this for an estimate of y in the next time step t_{k+1}. This method is called *Euler's method*. One can also interpret or derive Euler's method via Taylor approximations of the solution $y(t)$. To this end, consider the Taylor expansion

$$y(t_{k+1}) = y(t_k) + \delta t \cdot \dot{y}(t_k) + R \approx y(t_k) + \delta t \cdot f(t_k, y_k)$$

in which all terms of higher order (i.e., with $(\delta t)^2$ etc.) are neglected.

In addition to this, there exist a number of related methods for IVPs of ODEs. The method by *Heun* is one such example:

$$y_{k+1} := y_k + \frac{\delta t}{2} \left(f(t_k, y_k) + f(t_{k+1}, y_k + \delta t f(t_k, y_k)) \right) \ .$$

The basic pattern, however, remains the same: One takes the approximation y_k at t_k that has already been computed and uses this to determine an approximation of the slope $\dot{y}$. This is then used to compute an approximation for the value of the solution y at the next time step t_{k+1} through multiplication with the step size δt. What is new is how to estimate the slope. In Euler's method, one simply takes $f(t_k, y_k)$. Heun's method attempts to better approximate the slope over the entire interval $[t_k, t_{k+1}]$ by computing the average of two estimates for $\dot{y}$ taken at t_k and t_{k+1}. The difficulty arising from y_{k+1} not being determined yet can be avoided by using an Euler estimate as the second argument of f. It is easily seen that the individual time step has become more work-intensive (two function evaluations of f and more elementary computational operations when compared to the simple Euler's method).

The method by *Runge* and *Kutta* goes one additional step further along these lines:

$$y_{k+1} := y_k + \frac{\delta t}{6}(T_1 + 2T_2 + 2T_3 + T_4)$$

with

$$\begin{aligned} T_1 &:= f(t_k, y_k) \,, \\ T_2 &:= f\left(t_k + \frac{\delta t}{2}, y_k + \frac{\delta t}{2} T_1\right) \,, \\ T_3 &:= f\left(t_k + \frac{\delta t}{2}, y_k + \frac{\delta t}{2} T_2\right) \,, \\ T_4 &:= f(t_{k+1}, y_k + \delta t\, T_3) \,. \end{aligned}$$

This rule also follows the basic principle

$$y_{k+1} := y_k + \delta_t \cdot (\mathit{Approximation\_of\_the\_slope}) \,,$$

however, the computation for the approximation of $\dot{y}$ has now become even more complicated. Starting with the simple Euler approximation $f(t_k, y_k)$, a skillfully nested procedure generates four suitable approximations which when—appropriately weighted—will then be used for the approximation. The appeal of these complicated rules is naturally rooted in the higher accuracy of the produced discrete approximations for $y(t)$. To quantify this, we must get a better handle on the notion of the *accuracy* of a method for the discretization of ordinary differential equations. Two issues have to be carefully separated: first, the error that accumulates *locally* at every point t_k and is independent from the use of any approximate solutions but is present due to the use of the difference quotients (based on the exact $y(t)$) used in the algorithm rather than the derivatives $\dot{y}(t)$ of the exact solution $y(t)$; second, the error that accumulates *globally* over the course of the computations from a to b, i.e., over the entire time interval being considered. In like manner, we distinguish two types of discretization errors, the *local discretization*

error $l(\delta t)$ (i.e., the part that is produced as a new error at each time regardless whether the difference quotient is determined using the exact $y(t)$) as well as the *global discretization error*

$$e(\delta t) := \max_{k=0,\ldots,N} \{ \mid y_k - y(t_k) \mid \}$$

(i.e., the greatest possible amount by which one's calculations performed over the entire time interval may be off the mark).

If $l(\delta t) \to 0$ for $\delta t \to 0$, then the discretization scheme is called *consistent*. Consistency is obviously the minimum that needs to be required. An inconsistent discretization is utterly useless: If the approximations per time step are not even locally reasonable, meaning that more and more computational effort does not lead to better results, one cannot expect our given IVP to be reasonably solved. If $e(\delta t) \to 0$ for $\delta t \to 0$, then the discretization scheme is called *convergent*. The investment of more and more computational effort (smaller and smaller time steps δt) then leads to better and better approximations of the exact solution (vanishing error). Consistency is the weaker of the two concepts, rather of a technical nature and oftentimes this is relatively easy to prove. Convergence, in contrast, is the stronger concept (Convergence implies consistency, but not vice versa!), it is of fundamental and practical significance and oftentimes not quite trivial to show.

All of the three methods that have been introduced so far are consistent and convergent; Euler's method is of first order, Heun's method of second order and the Runge–Kutta method of fourth order, i.e.,

$$l(\delta t) = \mathcal{O}((\delta t)^4) , \qquad e(\delta t) = \mathcal{O}((\delta t)^4) .$$

Here, their difference in quality is made apparent: The higher the order of the method, the more is gained from an increase in computational effort. When dividing the step size δt by two, for example, the error in Euler's, or Heun's, or the Runge–Kutta method is asymptotically reduced by a factor of 2, 4 and 16, respectively. The more expensive methods are thus the more effective (at least asymptotically). But we are still not satisfied. The number of *evaluations* of the function f for different arguments has strongly increased (compare the Runge–Kutta formulas: T_2, T_3 and T_4 in which each require an additional evaluation of f). In numerical practice, f is typically very complicated (oftentimes a single evaluation of f requires another differential equation to be solved), so that one evaluation of f already involves a high computational effort.

The previous methods are examples of the so-called *one-step methods*: For the computation of y_{k+1}, no time points prior to t_k are exploited (but rather—as mentioned—new evaluation points). This is different for the *multistep methods*: Here, no new evaluation points of f are produced, but rather older

(and already computed) function values will be reused, for example in t_{k-1} for the *Adams–Bashforth method of second order*:

$$y_{k+1} := y_k + \frac{\delta t}{2}\,(3f(t_k, y_k) - f(t_{k-1}, y_{k-1}))$$

(second order consistency can easily be shown). Methods of even higher order can be constructed analogously by utilizing time steps t_{k-i}, $i = 1, 2, \ldots$ that lie even further in the past. The principle here is a familiar relative to quadrature: Replace f by a polynomial p of suitable degree which interpolates f at the points (t_i, y_i) being considered and then use this p in accordance to

$$y_{k+1} := y_k + \int_{t_k}^{t_{k+1}} f(t, y(t))\mathrm{d}t \approx y_k + \int_{t_k}^{t_{k+1}} p(t)\mathrm{d}t \ ,$$

in order to compute y_{k+1} (the polynomial p is easy to integrate). At the beginning, i.e., as long as there are not sufficiently many "old" values, one typically uses a suitable one-step method. The multistep methods that are based on this interpolation principle are called *Adams–Bashforth methods*.

Stability

The methods of Euler, Heun and Runge–Kutta are consistent and convergent. This is also true for the class of multistep methods of the Adams–Bashforth type just introduced. However, it turns out that for multistep methods, consistence and convergence do not always hold simultaneously. In order to be convergent, a consistent method must additionally be *stable* (in the sense of our definition of a numerically stable algorithm). The proof of stability is thus of eminent importance. The basic rule holds which says,

$$\text{consistence} \quad + \quad \text{stability} \quad \Rightarrow \quad \text{convergence} \ .$$

An example for an unstable discretization rule is found in the *midpoint rule* $y_{k+1} := y_{k-1} + 2\delta t\, f_k$ which is a consistent 2-step method (easy to show via Taylor expansion). Applying the midpoint rule, for example, to the IVP

$$\dot{y}(t) = -2y(t) + 1 \ , \qquad y(0) = 1 \ , \qquad t \geq 0 \ ,$$

having solution,

$$y(t) = \frac{1}{2}(\mathrm{e}^{-2t} + 1),$$

leads to *oscillations* and *divergence*—no matter how small the chosen δt.

Stiffness, Implicit Methods

As a final phenomenon we consider *stiff* differential equations. These are understood to be scenarios in which an insignificant local property of the solution imposes an extremely high resolution upon the entire domain. An example is provided by the (well-conditioned) IVP $\dot{y} = -1000y + 1000, t \geq 0,\ y(0) = y_0 = 2$ having the solution $y(t) = \mathrm{e}^{-1000t} + 1$. If one applies the (convergent!) Euler's method, then $\delta t \geq 0.002$ leads to no convergence. This is a contradiction only at first glance. The concepts of consistency, convergence and stability are of *asymptotic* nature: $\delta t \to 0$ and $\mathcal{O}(\delta t)$ always implies "for sufficiently small δt", and in the current case, this "sufficiently small" turns out to be "unacceptably small". A cure for a problem's stiffness is given through the use of *implicit methods* in which the unknown y_{k+1} also appears on the right hand side of the method so that it has to be first solved for y_{k+1}. The simplest example of an implicit method is the *implicit Euler's* or *backward Euler's method*:

$$y_{k+1} = y_k + \delta t f(t_{k+1}, y_{k+1}) \ .$$

One notes the appearance of the (still to be determined) y_{k+1} on the right hand side of the rule. For the stiff problem considered above, an application of the implicit Euler's method leads to a situation where $\delta t > 0$ can now be chosen arbitrarily.

Why is that? Overly simplified, explicit methods approximate the solution of an IVP using polynomials whereas implicit methods use rational functions. Polynomials, however, cannot approximate e^{-t} for $t \to \infty$, for example, while rational functions can do so very well. Therefore, implicit methods are indispensible for stiff differential equations. However, the equation cannot always be solved easily for y_{k+1}. In the general case this may require a (nonlinear) iteration technique. Oftentimes, the *predictor-corrector approach* is of help: First employing the use of an implicit method, one determines an initial approximation for y_{k+1}, substituting this approximation into the right hand side of the implicit rule—the rule of thumb, twice explicitly thus turns into once implicitly, holds under certain prerequisites. Basically it holds that an implicit time step is more expensive than an explicit one. But in view of non-existent or at least less restrictive bounds for the step widths, implicit methods often require significantly fewer time steps.

Additional topics (not mentioned here) involving numerical properties of ODEs include *systems of ODEs*, *ODEs of higher order* as well as *boundary value problems*.

2.4.6 Partial Differential Equations

Classification

Partial differential equations (PDE) are likely to be even less analytically approachable than ODEs—not only because solutions in closed-form can hardly ever be

stated for practical and relevant cases, but also because the question regarding existence and uniqueness of solutions is often an open one. This turns the numerical treatment of PDEs into a must—despite the high computational effort that includes, in particular, the case of full spatial resolution. Concrete models are formulated as *boundary value problems* (e.g., in the steady case) or as *boundary-initial value problems*. Important boundary conditions are the *Dirichlet boundary conditions* (here, the function values are prescribed on the boundary) as well as the *Neumann boundary conditions*, in which the normal derivative is prescribed on the boundary.

An important class of PDEs is the *linear PDE of second order* in d dimensions:

$$\sum_{i,j=1}^{n} a_{i,j}(x) \cdot \partial_{i,j}^2 u(x) \; + \; \sum_{i=1}^{n} a_i(x) \cdot \partial_i u(x) \; + \; a(x) \cdot u(x) \; = \; f(x) \; .$$

Within this class, one distinguishes between *elliptic* PDEs (the matrix A of coefficient functions $a_{i,j}(x)$ is positive or negative definite), *hyperbolic* PDEs (the matrix A has a positive and $n-1$ negative eigenvalues or vice versa) as well as *parabolic* PDEs (an eigenvalue of A is zero, all the others have the same sign, and the rank of A together with the vector of coefficient functions $a_i(x)$ is maximal or full—if there is no second derivative with respect to a variable, then there is at least a first derivative). Simple yet well-known examples include the *Laplace-* or *potential equation* $\Delta u = 0$ for the elliptic case, the *heat equation* $\Delta T = T_t$ for the parabolic case and the *wave equation* $\Delta u = u_{tt}$ for the hyperbolic case. This distinction is not reserved for the accounting department—these three types display different analytic properties (e.g., shock phenomena such as the Mach cone exist only for hyperbolic PDEs) and require different numerical approaches for their solutions.

Discretization Approaches

Particularly true for the higher dimensional case, the discretization of the underlying domain is already work intensive. One distinguishes between *structured grids* (e.g., cartesian grids), in which the coordinates of grid points or cells, resp., do not have to be stored explicitly but can be determined via index computations, and *unstructured grids*, in which the entire geometry (coordinates) and topology (neighborhoods, etc.) have to be administered and stored. Examples for unstructured grids include irregular triangulations or tetrahedral grids.

Discretization techniques for PDEs have a variety of approaches that have been established over the course of years. *Finite difference methods* discretize the PDE directly in the sense that all derivatives are replaced by approximating difference quotients—a straightforward and easy to implement approach which, however, has its weaknesses with respect to its theoretical background and is furthermore basically restricted to structured or orthogonal grids. *Finite volume methods* are popular, in particular, in the context of fluid flow simulations. They discretize continuum mechanical conservation laws on small control volumes.

In contrast, *Finite-element methods* apply a variational approach—they look for a minimal-energy solution which is in direct relation to the underlying physics. Here the implementation is often more work intensive, but the pay-off is found from a high flexibility with respect to the employed grids (from structured to unstructured) as well as a very nice and rich mathematical theory. In addition there exist further concepts such as *spectral methods* or *meshfree methods* in which the perspective of moving particles is assumed instead of a fixed reference grid.

Finite Differences

Let a domain $\Omega \subseteq \mathbb{R}^d$, $d \in \{1, 2, 3\}$ be given. On it, we introduce a (structured) grid Ω_h with *grid width* $h = (h_x, h_y, h_z)$ (in 3 D). The definition of *finite differences* or difference quotients, follows analogously with the previous section on ODEs. For the first derivative, as in the case for the ODEs, *forward*, *backward* or *central differences* are customary,

$$\frac{\partial u}{\partial x}(\xi) \doteq \frac{u(\xi + h_x) - u(\xi)}{h_x} \,, \quad \frac{u(\xi) - u(\xi - h_x)}{h_x} \,, \quad \frac{u(\xi + h_x) - u(\xi - h_x)}{2h_x} \,,$$

and for the second derivative, for example, there is the *3-point stencil*

$$\frac{\partial^2 u}{\partial x} \doteq \frac{u(\xi - h_x) - 2u(\xi) + u(\xi + h_x)}{h_x^2} \,.$$

The Laplace operator Δu correspondingly leads to the *5-point stencil* in 2 D and to the *7-point stencil* in 3 D. The notation "stencil" indicates which and how many neighboring points are used in the discretization. Wider stencils involve more points and therefore result in a higher approximation quality (elimination of further terms in the Taylor expansion).

For each interior gridpoint the PDE has now been replaced by a *difference equation*. The unknowns (*degrees of freedom*) are thereby the approximate values of the function values $u(\xi)$ at the discrete grid points ξ. Thus, one obtains one unknown per each grid point and per each unknown scalar quantity. Points on or near the boundary require special treatment. In the case of *Dirichlet boundary conditions*, one does not set up a difference equation in the boundary points (there, the function values are given and not unknown). In points next to the boundary, one obtains a "degraded" discrete equation since the stencils in parts involve known values which are moved to the right hand side. In the case of *Neumann boundary conditions*, one also prescribes difference equations at the boundary points. However, these equations have a modified form due to the condition for the normal derivative.

Overall, one obtains a large (one row for every scalar quantity in each (inner) grid point), sparse (only a few non-zeros resulting from the stencil) system of linear equations which subsequently needs to be solved efficiently.

Resulting is a certain *order of consistency* for the discretization that depends on the selected discrete operators; In addition, stability must also be proven. The areas to first consider for improvements could be the choice of stencils of higher order or local grid refinement (*adaptivity*). For the case of higher dimensions, as they appear in quantum mechanics or in mathematical finance, for example, the *curse of dimensionality* comes to the forefront: In d spatial dimensions, $\mathcal{O}(h^{-d})$ grid points and unknowns are required, which is naturally no longer manageable for $d = 10$ or $d = 100$.

Finite Elements

With *finite element methods (FEM)*, the derivatives are not discretized directly. Rather, the PDE is transformed into its so-called *weak form* (integral form). We summarize below the five essential steps:

1. *Substructuring* and *grid generation*: decompose the domain into individual parcels of a given pattern and of finite extension (*finite elements*);
2. *weak form*: no longer satisfy the PDE pointwise everywhere but only weakened (in form of an inner product) or averaged (as an integral), resp.;
3. *finite dimensional trial space*: replace the continuous solution in the weak form by a suitable finite dimensional approximation;
4. *system of linear equations*: use test functions to generate an equation for each degree of freedom;
5. *Solution of the linear system*: employ a suitable iteration method for the solution of the linear system.

The generation of finite elements can be interpreted as a top-down or a bottom-up process. Top-down decomposes the domain into standard components whose behavior is easy to describe; bottom-up originates from particular elements and uses these to approximate the given domain. In 3 D, one thus obtains a *finite element net* of *elements* (3 D atoms, e.g., cubes or tetrahedra), *surfaces* (2 D surface structures, e.g., triangles or squares), *edges* (1 D boundary structures of the element) as well as *grid points* or *nodes* in which the unknowns typically (but not necessarily) live. Each node is associated with a *trial function* φ_k, a basis function with finite support (non-zero only in neighbouring elements). Together, all trial functions span the linear and finite dimensional *trial space* V_n and form a basis. Within this trial space one seaks the approximation to the solution of the PDE.

We now address the weak form of the PDE whose representation we give through the general form $Lu = f$ with differential operator L (e.g., Δ), right hand side f and continuous solution u. Instead of $Lu = f$ on Ω, one now considers

$$\int_{\Omega} Lu \cdot \psi_l \mathrm{d}\Omega \;=\; \int_{\Omega} f \cdot \psi_l \mathrm{d}\Omega$$

for a basis of *test functions* ψ_l in a finite dimensional *test space* W_n. This procedure is also called the *method of weighted residuals* or, equivalently, the *Galerkin approach*. If the test and trial space are chosen to be the same ($V_n = W_n$), one then speaks of a *Ritz–Galerkin approach*, otherwise of a *Petrov–Galerkin approach*. One now writes the above equation in a slightly different way using both a *bilinear form* $a(.,.)$ and a *linear form* $b(.)$,

$$a(u, \psi_l) = b(\psi_l) \qquad \forall \psi_l \in W_n \ ,$$

hence obtaining n discrete linear equations.

Lastly, one replaces the continuous solution u by its discrete approximation

$$u_n := \sum_{k=1}^{n} \alpha_k \varphi_k \qquad \in V_n$$

thus obtaining,

$$a(u_n, \psi_l) = \sum_{k=1}^{n} \alpha_k \cdot a(\varphi_k, \psi_l) = b(\psi_l) \qquad \forall \psi_l \in W_n \ .$$

This yields a linear system of equations in the n unknowns α_k: The matrix consists of the entries $a(\varphi_k, \psi_l)$, the right hand side of the $b(\psi_l)$; all quantities do not depend on the solution but only on the problem.

As for finite differences, the FEM discetization has led to a discrete problem $Ax = b$. Here, one recognizes a tight coupling between the discretization and the solution: Some properties of the matrix are desirable (positive definite, sparse, as "close" as possible to diagonality) to ensure the efficient solution of the linear system. This is attempted through a "good" choice for the test and trial space.

Given the numerous application fields involving FEM, a multitude of different elements has been introduced over the years, each differing with respect to the form as well as with respect to the location and number of unknowns. For an increase in efficiency, *adaptive refinement* is very popular in which *local error estimation* (its importance is rooted with the decision where to refine) as well as *global error estimation* (whose importance is rooted to the decision when one can stop) are of great significance. One also refers to the *h-version* of FEM. Conversely, one can successively increase the approximation quality of the elements (*p-version* of FEM) or combine these two (*hp version*). With regard to the geometries that are to be described, the complexity increases if they are very *complicated* or if they are *changing* with respect to time, for example, if the problem involves free surfaces. To close, the remark pertaining to finite differences and the curse of dimensionality naturally applies here as well.