

TMA4320 Introduction to Scientific Computing

Anne Kværnø

3rd June 2024

Contents

1	Preliminaries	5
1.1	Introduction	5
1.2	Norms	6
1.3	Convergence and errors	7
1.3.1	Convergence of an iterative process	7
1.3.2	Convergence of h -dependent approximations	9
1.4	Taylor-expansions	11
1.5	Some other useful results	11
1.6	Floating Point Arithmetic	12
1.7	Error propagation	13
1.8	How to develop a numerical algorithm	15
1.8.1	Problem statement	15
1.8.2	Test problems	15
1.8.3	Algorithm	15
1.8.4	Testing the algorithm	16
1.8.5	Error analysis	16
1.8.6	Numerical verification of theoretical results, and a nice presentation of those	17
1.8.7	Can something go wrong?	17
1.8.8	Numerical differentiation formulas	19
2	Numerical Linear Algebra	21
2.1	Gauss Elimination	21
2.1.1	Notation	21
2.1.2	Naive Gauss elimination	21
2.1.3	Gauss elimination with pivoting	26
2.2	Vector and Matrix Norms	30
2.2.1	Norms	30
2.2.2	Matrix norms	32
2.3	Error analysis of linear system of equations	33
3	Numerical ODEs	35
3.1	Introduction	35
3.1.1	Scalar ODEs.	35
3.1.2	Systems of ODEs.	36
3.1.3	Autonomous ODEs.	37
3.1.4	Higher order ODEs.	37
3.2	Numerical methods for solving ODEs	38

3.2.1	Euler's method	38
3.2.2	Heun's method	39
3.2.3	Runge–Kutta methods	41
3.3	Theory	42
3.3.1	Existence and uniqueness results	42
3.3.2	Error analysis of Euler's method	42
3.3.3	A general convergence result	45
3.3.4	Order conditions for Runge–Kutta methods.	46
3.4	Error estimation and step size selection	49
3.4.1	Error estimation	49
3.4.2	Stepsize control	50
3.5	A summary of some terms and definitions	51
3.6	Stiff ODEs	52
3.6.1	Linear stability analysis	53
3.6.2	Linear systems of ODEs.	56
3.7	A-stable methods.	57
3.7.1	Adaptive methods.	58
3.8	Newton's method for nonlinear equations	59
3.8.1	Newton's method for systems	59

Chapter 1

Preliminaries

1.1 Introduction

In your mathematical courses so far, you have learned how to solve different mathematical problems, like linear and nonlinear equations and differential equations. You have learned how to differentiate and to integrate functions. Unfortunately, only simplified models from "real life" applications can be treated by these techniques, more complex and realistic problems the best we can aim for is some kind of approximate solution, found on a computer by some clever numerical algorithms. In this part of the course, our concern is how to develop, analyse, implement and test a selection of such algorithms.

In this note, we will present some mathematical results that will be used frequently, as well as some definitions and concepts. Most of it should be known from previous courses, in particular from Mathematics 1 and 3.

Whenever some theoretical error analysis has been established, it should be verified numerically. This is done based on a test problem with a known exact solution, so the error in our numerical experiment can be evaluated, and the theory numerically verified. Examples of such verifications will be demonstrated.

Definition 1.1.1. A *real vector space* is a set V together with the operations $+$ (addition) and $\cdot$ (scalar multiplication) which for all $x, y, z \in V$ and $\alpha, \beta \in \mathbb{R}$ satisfy the following conditions:

- 1) $x + y \in V$
- 2) $x + y = y + x$
- 3) $x + (y + z) = (x + y) + z$
- 4) There exist a $0 \in V$ such that $x + 0 = x$
- 5) For all $x \in V$ there is a $-x \in V$ such that $x + (-x) = 0$
- 6) $\alpha x \in V$
- 7) $\alpha(\beta x) = (\alpha\beta)x$
- 8) $1x = x$
- 9) $\alpha(x + y) = \alpha x + \alpha y$
- 10) $(\alpha + \beta)x = \alpha x + \beta x$

Example 1.1.2. The following vector spaces will be used throughout the course:

- $\mathbb{R}^m$ is the set of all real vectors with m components.

- $\mathbb{R}^{m \times n}$ is the set of all $m \times n$ real matrices.
- $C^m[a, b]$ is the set of all functions with continuous first m derivatives on the interval $[a, b]$. It is common to use $C[a, b]$ rather than $C^0[a, b]$ for all continuous functions.
- $\mathbb{P}_n$ is the set of all polynomials of degree at most n .

Notice that $C^n[a, b] \subset C^m[a, b]$ if $n > m$. Further, $\mathbb{P}_n \subset C^\infty[\mathbb{R}]$.

1.2 Norms

The norm $\|\cdot\|$ of an element x in a vector space V is essentially a measure of the size of the element. The norm obeys the following rules:

Definition 1.2.1. Norm $\|\cdot\|$ For all $x, y \in V$ and $\alpha \in \mathbb{R}$ the following holds

$$\begin{aligned} \|x\| &\geq 0 & \text{and} \quad \|x\| = 0 &\Leftrightarrow x \equiv 0 \\ \|\alpha x\| &= |\alpha| \|x\|, \\ \|x + y\| &\leq \|x\| + \|y\| & (\text{Triangle inequality}). \end{aligned}$$

Some examples of norms for $C[a, b]$ and $\mathbb{R}^m$:

- For $f \in C[a, b]$ we have $\|f\|_\infty = \max_{x \in [a, b]} |f(x)|$ and $\|f\|_2 = \sqrt{\int_a^b f(x)^2 dx}$.
- For $\mathbf{x} \in \mathbb{R}^m$ we have $\|\mathbf{x}\|_\infty = \max_{i=1}^m |x_i|$, and $\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^m x_i^2}$.

Note that we will use bold symbols for vectors $\mathbf{x} \in \mathbb{R}^m$.

As is demonstrated in these examples, the norm to which we refer is usually marked with a subscript. We will always use the absolute value as the norm of a real number, thus $\|x\| = |x|$ whenever $x \in \mathbb{R}$.

Example 1.2.2. Let $\mathbf{x} = [1, -6, 3, -1, 5]^T \in \mathbb{R}^5$. Then

$$\begin{aligned} \|\mathbf{x}\|_2 &= \sqrt{1 + 36 + 9 + 1 + 25} = 8.4853, \\ \|\mathbf{x}\|_\infty &= \max\{1, 6, 3, 1, 5\} = 6. \end{aligned}$$

Norms of vectors in $\mathbb{R}^m$ are implemented in Python, in the module ‘numpy.linalg’.

Example 1.2.3. Let $f(x) = \sin(x)$ on $[0, 2\pi]$, so $f \in C^\infty[0, 2\pi]$. In this case

$$\begin{aligned} \|f\|_2 &= \sqrt{\int_0^{2\pi} \sin^2(x) dx} = \sqrt{\pi} = 1.7725 \\ \|f\|_\infty &= \max_{x \in [0, 2\pi]} |\sin(x)| = 1. \end{aligned}$$

Norms on function spaces require some sort of numerical approximations. In this case, the interval $[0, 2\pi]$ has been divided into $N = 1000$ uniform subintervals, so $x_i = (2\pi i)/N$, $i = 0, 1, \dots, N$. The norm $\|f\|_\infty$ is approximated by $\max_i |f(x_i)|$, and the integral required for $\|f\|_2$ is computed by the trapezoidal rule.

See `norm_of_a_function` in `preliminaries.py`.

1.3 Convergence and errors

Definition 1.3.1. Convergence of a sequence Let $\{x_k\}_{k=0}^{\infty}$ be an infinite sequence of real numbers. The sequence converges to x , if, for any $\varepsilon > 0$ there exist a positive integer $N(\varepsilon)$ such that $|x_k - x| < \varepsilon$ whenever $k > N(\varepsilon)$.

Common notations:

$$\lim_{k \rightarrow \infty} x_k = x \quad \text{or} \quad x_k \rightarrow x \text{ as } k \rightarrow \infty.$$

Example 1.3.2. It is known that the sequence $x_k = (1 + 1/k)^k \rightarrow e = 2.7182\dots$ as $k \rightarrow \infty$. The sequence is monotone, so $|x_{k+1} - e| < |x_k - e|$. The following program demonstrates the concept of convergence for this sequence: Given an ε , the positive integer $N(\varepsilon)$ is returned.

See `convergence_sequence` in `preliminaries.py`. Clearly, this code can only be used for monotone series.

Warning: The sequence converges very slowly, so given a very small ‘epsilon’ will result in N too large. This can be compensated by making ‘Nmax’ larger, and increment k in larger steps.

1.3.1 Convergence of an iterative process

Let X be the exact solution of a problem, and X_k a numerical approximation achieved by some iterative process $X_{k+1} = G(X_k)$. In this case the iterations converge towards X if

$$\lim_{k \rightarrow \infty} \|X - X_k\| = 0$$

Let $e_k = \|X - X_k\|$ measure the error. In practice, you have to choose an appropriate norm, which depends on the problem and what you might be interesting in measuring. The order of convergence is p if there exist a positive constant M such that

$$e_{k+1} \leq M e_k^p$$

Notation: The case $p = 1$ is called *linear* convergence, $p = 2$ is called *quadratic* convergence and $p = 3$ *cubic* convergence.

Numerical verification of the order To verify the order, we make the assumptions that $e_{k+1} = C_k e_k^p$, and that the C_k do not change much from one iteration to the next one. These assumptions are usually reasonable when the error becomes small. The order p can then be computed numerically by the following procedure: Take the expressions for the error for two subsequent iterations, assuming that $C_{k+1} \approx C_k \approx C$. Then divide the expression by each other to get rid of the unknown constant C , take the logarithm on both sides and solve for the order p .

$$\begin{aligned}
\frac{e_{k+2}}{e_{k+1}} &\approx \frac{C e_{k+1}^p}{C e_k^p} \Rightarrow \frac{e_{k+2}}{e_{k+1}} \approx \left(\frac{e_{k+1}}{e_k} \right)^p \\
\Rightarrow \log \left(\frac{e_{k+2}}{e_{k+1}} \right) &\approx p \log \left(\frac{e_{k+1}}{e_k} \right) \Rightarrow p \approx \frac{\log(e_{k+2}/e_{k+1})}{\log(e_{k+1}/e_k)}
\end{aligned}$$

We are usually not very interested in the constant C , but given p and the error for two iterations, it can easily be approximated.

Example 1.3.3. Newton's method applied to the equation $f(x) = 0$ is given by

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}.$$

Let r be a solution of the equation. It can be proved that the error $r - x_k$ satisfies

$$r - x_{k+1} = -\frac{f''(\xi_k)}{2f'(x_k)}(r - x_k)^2, \quad \text{where } \xi_k \text{ is a real number between } x_k \text{ and } r.$$

Since the error is a real number, its norm is $e_k = |r - x_k|$ and

$$e_{k+1} = C_k e_k^2 \quad \text{where} \quad C_k = \frac{|f''(\xi_k)|}{2|f'(x_k)|}.$$

Notice that $C_k \rightarrow |f''(r)|/(2|f'(r)|)$ as $x_k \rightarrow r$.

Let M denote an upper bound for C_k . More precisely: let $I_\delta = [x - \delta, x + \delta]$ be some interval around the solution r . Assume there exist constants L and K such that $|f'(x)| \geq L$ and $|f''(x)| \leq K$ for all $x \in I_\delta$. Then $M = K/(2L)$ and

$$e_{k+1} \leq M e_k^2.$$

The convergence is quadratic, and the iterations converge for all starting values x_0 chosen such that

$$M e_0 < 1,$$

often described as "sufficiently close to the solution".

Let us now verify the theoretical result by applying Newton's method to the problem $x^2 - a = 0$ for some $a > 0$. The iterations become

$$x_{k+1} = x_k - \frac{x_k^2 - a}{2x_k} = \frac{x_k^2 + a}{2x_k}, \quad k = 0, 1, 2, \dots$$

with the exact solution $r = \sqrt{a}$. From the discussion above, we expect $C \approx |f''(r)/(2f'(r))|$ with $r = \sqrt{a}$, which in our case becomes $C \approx 1/(2\sqrt{a})$. Use the following code to see if the theoretical considerations hold in practice:

See `ooc_iterations` in `preliminaries.py`.

1.3.2 Convergence of h -dependent approximations

Let X be the exact solution, and $X(h)$ some numerical solution depending on a parameter h , and let $e(h)$ be the norm of the error, so $e(h) = \|X - X(h)\|$. The numerical approximation $X(h)$ converges to X if $e(h) \rightarrow 0$ as $h \rightarrow 0$. The order of the approximation is p if there exists a positive constant M such that

$$e(h) \leq Mh^p$$

The Big $\mathcal{O}$ -notation: A function $f(x) = \mathcal{O}(g(x))$ as $x \rightarrow a$ if and only there exist positive numbers δ and M such that

$$|f(x)| \leq M|g(x)| \quad \text{when} \quad 0 < |x - a| < \delta.$$

Let $a = 0$ and the error of an approximation of order p can be written as

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

This is often used when we are not directly interested in any expression for the constant M , we only need to know it exists.

Numerical verification The following is based on the assumption that $e(h) \approx Ch^p$ for some unknown constant C . This assumption is usually reasonable for sufficiently small h .

Choose a test problem for which the exact solution is known and compute the error for a sequence of smaller h 's, for instance $h_k = H/2^k$, $k = 0, 1, 2, \dots$. The procedure is then quite similar to what was done for iterative processes.

$$\begin{aligned} \frac{e(h_{k+1})}{e(h_k)} &\approx \frac{Ch_{k+1}^p}{Ch_k^p} \Rightarrow \frac{e(h_{k+1})}{e(h_k)} \approx \left(\frac{h_{k+1}}{h_k}\right)^p \Rightarrow p \approx \frac{\log(e(h_{k+1})/e(h_k))}{\log(h_{k+1}/h_k)}. \end{aligned}$$

Since

$$e(h) \approx Ch^p \quad \Rightarrow \quad \log e(h) \approx \log C + p \log h,$$

a plot of $e(h)$ as a function of h using a logarithmic scale on both axes will be a straight line with slope p . Such a plot is referred to as an *error plot* or a *convergence plot*.

Some terminology Let X be the exact solution of some problem, and $\tilde{X}$ the numerical approximation of X . The following concepts are of interest:

- **Error:** $E = X - \tilde{X}$. This is something which obviously is only known if the exact solution is known (which it will be in our test problems, but not in real life problems). Still, most error analysis will start trying to find an expression for this error, but it will typically contain some higher order derivatives evaluated in some unknown point.

- *Error bound* : $\|X - \tilde{X}\| \leq K$. If the approximation depend on some parameter h , we typically search for expressions of the form $K = Mh^p$, if M is known, this can be used to decide how small h has to be to guarantee that the error is below some tolerance.
- *Error estimate*: $\mathcal{E} \approx \|E\|$. This is an approximation to the error (or the norm of the error) and something that can be computed and included in practical codes. How to compute these will be described for each class of problems discussed later in the course.

Example 1.3.4. Consider the trapezoidal rule for numerical integration. It is known that

$$\int_a^b f(x)dx = T(h) + E(h)$$

where $T(h)$ is the numerical approximation given by

$$T(h) = h \left(\frac{1}{2}f(x_0) + \sum_{i=0}^{n-1} f(x_i) + \frac{1}{2}f(x_n) \right), \quad x_n = a + ih, \quad h = \frac{b-a}{n},$$

and the error $E(h)$ is known to be

$$E(h) = -\frac{b-a}{12} f''(\xi) h^2, \quad \xi \in (a, b).$$

Assume there exists an M such that $|f''(x)| \leq M$ for all $x \in (a, b)$. Let $e(h) = |E(h)|$ (notice that $E(h)$ is a scalar) so

$$e(h) \leq Mh^2.$$

So the error of the trapezoidal rule is of order 2, and $e(h) = \mathcal{O}(h^2)$.

Use this to verify the order of the trapezoidal rule, as given above. As test example, choose

$$\int_0^\pi \sin(x)dx = 2.$$

In this case, we expect the order p to be 2. The constant $C = |f''(\xi)|\pi/12$ for some unknown $\xi \in [0, \pi]$. Thus $0 < C < \pi/12 = 0.2617\dots$, but we can not be more precise. An upper bound for the error is thus $e(h) \leq (\pi/12)h^2$.

The following code can be used to confirm the result. It also returns an approximation to C , so we can at least check if it is within the expected bound. See `ooc_h()` in `preliminaries.py`.

1.4 Taylor-expansions

Given a function $f \in C^\infty[a, b]$, choose a point x and an increment h such that $x, x + h \in [a, b]$. The Taylor expansion of f around x is then given by

$$f(x + h) = \sum_{k=0}^{\infty} \frac{h^k}{k!} f^{(k)}(x).$$

The function f is called analytic in x if the series converges for sufficiently small values of $|h|$. In numerics, we will usually work with the truncated series, also known as the Taylor polynomial of degree m :

$$f(x + h) = \sum_{k=0}^m \frac{h^k}{k!} f^{(k)}(x) + R_{m+1}(x).$$

The remainder term $R_{m+1}(x)$ is given by

$$R_{m+1}(x) = \frac{h^{m+1}}{(m+1)!} f^{(m+1)}(\xi),$$

where ξ is some unknown point between x and $x+h$. Unlike the full expansion, constructing the truncated series only requires that $f \in C^{(m+1)}$ near x . The remainder term is often written as $R_{m+1}(x) = \mathcal{O}(h^{m+1})$.

1.5 Some other useful results

Theorem 1.5.1. *Intermediate value theorem* Let $f \in C[a, b]$ and let x be some number between $f(a)$ and $f(b)$, then there exist at least one $\xi \in (a, b)$ such that $f(\xi) = x$.

The generalization of this reads: Given k nodes $x_i \in [a, b]$ and k positive weights $w_i > 0$, $i = 1, \dots, k$. Then there exists at least one $\xi \in (a, b)$ such that

$$\sum_{i=1}^k w_i f(x_i) = f(\xi) \sum_{i=1}^k w_i. \quad (1.1)$$

Theorem 1.5.2. *Mean value theorem* Let $f \in C^1[a, b]$. Then there exists at least one $\xi \in (a, b)$ such that

$$f'(\xi) = \frac{f(b) - f(a)}{b - a} \quad (1.2)$$

Theorem 1.5.3. *Rolle's theorem* Let $f \in C^1[a, b]$ and $f(a) = f(b) = 0$. Then there exists at least one $\xi \in (a, b)$ such that $f'(\xi) = 0$. This is just a special case of the Mean value theorem.

Theorem 1.5.4. *Mean value theorem for integrals* Let $f \in C[a, b]$ and g an integrable function that does not change sign on $[a, b]$. Then there exists a $\xi \in (a, b)$ such that

$$\int_a^b f(x)g(x)dx = f(\xi) \int_a^b g(x)dx. \quad (1.3)$$

1.6 Floating Point Arithmetic

In these sections, how real numbers are represented in a computer and how this can affect numerical computations, are discussed.

A real number x can be represented as

$$x = \text{sign}(x) \cdot m \cdot b^e,$$

where m is the *mantissa* and $0 \leq m < 1$, b the *basis* (or radix) and e (an integer) the *exponent*. More precisely,

$$\begin{aligned} x &= \text{sign}(x) \left(\sum_{i=1}^{\infty} d_i \cdot b^{-i} \right) b^e \\ &= \underbrace{\text{sign}(x)}_{\pm} \underbrace{0.d_1 d_2 d_3 \dots d_k d_{k+1} \dots}_{\text{mantissa}} \cdot b^e. \end{aligned}$$

where d_i are integers satisfying $0 \leq d_i < b$ and $d_1 \neq 0$.

For the usual decimal system, $b = 10$, most computers works with the binary system, using $b = 2$.

Example 1.6.1.

$$\begin{aligned} \pi &= +0.3141592653589793 \dots \cdot 10^1 && \text{decimal} \\ &= +0.110010010000111110111 \dots \cdot 2^2 && \text{binary} \end{aligned}$$

In a computer, each number can only occupy a certain amount of the memory, that is a certain number of bits. For a standard 64 bits arithmetic, these are typically allocated as follows:

sign of x	1 bit
exponent e	8 bits
mantissa m	52 bits

Only truncated numbers can be represented in the computer, and a real number is rounded to the nearest k -digits number by the following rule:

$$\bar{m} = \begin{cases} 0.d_1 d_2 \dots d_k & \text{if } d_{k+1} < \frac{1}{2}b, \\ 0.d_1 d_2 \dots d_k + b^{-k} & \text{if } d_{k+1} \geq \frac{1}{2}b. \end{cases}$$

The *floating point representation* of x is

$$\text{fl}(x) = \pm \bar{m} \cdot b^e.$$

The relative error admits the following bound:

$$\left| \frac{x - \text{fl}(x)}{x} \right| \leq \frac{1}{2} \cdot b^{-k+1} = \epsilon,$$

where ϵ is referred to as the *machine precision*. So

$$\text{fl}(x) = x(1 + \delta), \quad |\delta| \leq \epsilon.$$

In the double precision binary system, with $b = 2$ and $k = 52$, the machine precision is $\epsilon = 2^{-52} \approx 2.2 \cdot 10^{-16}$.

For a more precise description of how floating point numbers are represented in practice, see e.g. See the IEEE 754 standard.

1.7 Error propagation

In this section, we will see how errors in the data (rounding errors, measure errors,...) will influence the results of some calculations. Let $x \in \mathbb{R}$ be the 'exact' number, and $\bar{x}$ some approximation to x . Then

$$\bar{x} = x + \Delta x = x(1 + \delta x), \quad \delta x = \frac{\Delta x}{x}. \quad (1.4)$$

where Δx is the *error* and δx the *relative error* in x .

Remark 1.7.1. It is usually assumed that the relative error δx is small, so dividing with $\bar{x}$ rather than x for the relative error makes no significant difference.

If $\phi : \mathbb{R}^n \rightarrow \mathbb{R}$, how will errors in the variables influence the value of ϕ ? Let $\mathbf{x} = [x_1, \dots, x_n]^\top$ and $\bar{\mathbf{x}} = [\bar{x}_1, \dots, \bar{x}_n]^\top = \mathbf{x} + \Delta \mathbf{x}$.

By a truncated multivariable Taylor expansion, we get

$$\phi(\mathbf{x}) - \phi(\bar{\mathbf{x}}) = \sum_{i=1}^n \frac{\partial \phi}{\partial x_i} \Delta x_i$$

where the partial derivatives are evaluated somewhere between $\mathbf{x}$ and $\bar{\mathbf{x}}$. The relative error is

$$\frac{\phi(\bar{\mathbf{x}}) - \phi(\mathbf{x})}{\phi(\mathbf{x})} = \sum_{i=1}^n \frac{\partial \phi}{\partial x_i} \frac{x_i}{\phi(\mathbf{x})} \delta x_i.$$

Taking the absolute values of these expressions, and by using the triangle equality, we obtain:

Error propagation law	
Absolute error bound:	$ \phi(\bar{\mathbf{x}}) - \phi(\mathbf{x}) \leq \sum_{i=1}^n \left \frac{\partial \phi}{\partial x_i} \right \Delta x_i $
Relative error bound:	$\left \frac{\phi(\bar{\mathbf{x}}) - \phi(\mathbf{x})}{\phi(\mathbf{x})} \right \leq \sum_{i=1}^n \left \frac{\partial \phi}{\partial x_i} \frac{x_i}{\phi(\mathbf{x})} \right \delta x_i $

The factor $\left| \frac{\partial \phi}{\partial x_i} \frac{x_i}{\phi(\mathbf{x})} \right|$ can be considered as an amplification factor of the relative error in x_i , and is often referred to as the *condition number*. If one of these terms is large, the problem is ill-conditioned, meaning that small errors in the input datas can make large errors in the output. If all the factors are reasonable small, the problem is *well-conditioned*.

Example 1.7.2. Let $z = \phi(x, y) = x + y$ and $\bar{z} = \bar{x} + \bar{y} = x + \Delta x + y + \Delta y$. The absolute and relative errors in z are

$$\begin{aligned} \Delta z &= \Delta x + \Delta y, \\ \delta z &= \frac{x}{x+y} \delta x + \frac{y}{x+y} \delta y. \end{aligned}$$

If the errors here are rounding errors from the floating point arithmetic, then $|\delta x|, |\delta y| \leq \epsilon$.

$$|\delta z| \leq \frac{1}{|x+y|} (|x| + |y|) \epsilon.$$

In this case, we have ignored the additional truncation error coming from z being stored in the computer.

For the usual arithmetic operations, the error propagation will be:

$$\begin{array}{lll}
 z = x \pm y & \Delta z = \Delta x \pm \Delta y, & \delta z = \frac{x}{x \pm y} \delta x \pm \frac{y}{x \pm y} \delta y \\
 z = xy & \Delta z = y \Delta x + x \Delta y, & \delta z = \delta x + \delta y \\
 z = x/y & \Delta z = \frac{1}{y} \Delta x - \frac{x}{y^2} \Delta y & \delta z = \delta x - \delta y.
 \end{array}$$

Remark 1.7.3. When subtracting two almost equal numbers, the relative error will increase a lot. Similar, dividing with a small number may result in a large absolute error.

1.8 How to develop a numerical algorithm

In this section a simple procedure for developing a numerical algorithm is presented. The steps are usually something like this:

1. State the problem, and, if relevant, what are the known theoretical results for the problem, typically existence and uniqueness results.
2. Find some relevant test problems, preferable with a known exact solution. This will be used for testing the algorithm later on.
3. Suggest a method for solving the problem numerically.
4. Implement the method, and apply it to the test problems. Does it behave as expected? How does it behave when changing method parameters? This can often give some clue on what to look for theoretically.
5. Do a theoretical analysis of the method. Exactly what this means depends on the problem/method in question, but some kind of error and convergence analysis is usually very relevant.
6. Perform numerical tests to confirm the theoretical results.
7. Further: Do some stress tests: When will the method break down? Why, and can we do something about it? Is it possible to use the knowledge developed to improve the algorithm, make them more robust/accurate/fast. Et cetera.

In the following, this procedure is demonstrated on an apparently simple problem, that is numerical derivation.

1.8.1 Problem statement

Assume that f is a known and sufficiently smooth function of x (the meaning of sufficiently smooth will be clear later). How can we find a numerical approximation to the derivative $f'(x)$ at a given point x_0 .

1.8.2 Test problems

In the class, three different test problems were suggested:

$$f(x) = x^2, \quad x_0 = 0, \quad f'(x) = 2x \quad (1.5a)$$

$$f(x) = e^x, \quad x_0 = 10, \quad f'(x) = e^x \quad (1.5b)$$

$$f(x) = \sinh x, \quad x_0 = 1, f'(x) = \cosh x \quad (1.5c)$$

In all cases, the exact solution is known, thus they are applicable for testing the algorithm, and the theoretical results which will be derived later.

1.8.3 Algorithm

Let us start with the definition of the derivative:

$$f'(x) = \lim_{h \rightarrow 0^+} \frac{f(x+h) - f(x)}{h} \quad (1.6)$$

which more or less automatically give us the simplest approximation to the derivative: For a given (small) $h > 0$, let

$$f'(x) \approx \hat{f}'(x_0) = \frac{f(x+h) - f(x)}{h} \quad (1.7)$$

where $\hat{f}'$ is the numerical approximation of f' .

1.8.4 Testing the algorithm

The next step is now to see if the algorithm (1.7) behaves reasonable. For each of the test problems an approximation to $f'(x_0)$ were computed, using two different values of h . In each case the absolute value of the error

$$e(h) = \hat{f}'(x_0) - f'(x_0) \quad (1.8)$$

were reported. The results are as follows:

$f(x)$	x_0	h	$\hat{f}'(x_0)$	$ e(h) $
x^2	0	0.01	0.01	0.01
		0.005	0.005	0.005
e^x	10	0.001	22037.48	11.0
			22031.97	5.51
$\sinh x$	1	0.1	1.5490	$5.90 \cdot 10^{-3}$
		0.05	1.5460	$2.94 \cdot 10^{-3}$

Remark 1.8.1.

1. The error $|e(h)|$ reduced by a factor 2 when the stepsize is reduced accordingly. Thus a reasonable hypothesis is:

$$e(h) \approx Ch, \quad \text{for some (unknown) constant } C.$$

This has to be proved, though.

2. In the second example, the error is 11 and 5.5 respectively, although h is smaller than in the other two examples. But the *relative* error $e(h)/\hat{f}'(x_0)$ is $4.99 \cdot 10^{-3}$ and $2.50 \cdot 10^{-3}$, which is more reasonable.

1.8.5 Error analysis

What can we say about the error $e(h)$? In particular, do we have convergence, $e(h) \rightarrow 0$ when $h \rightarrow 0^+$?

The error is the difference between the numerical approximation and the exact solution. Do a Taylor expansion of $f(x_0 + h)$ around x_0 , and then see which terms cancels out. Truncate the expansion at the first non-zero term. More precise:

$$\begin{aligned}
 e(h) &= \frac{f(x_0 + h) - f(x_0)}{h} - f'(x_0) \\
 &= \frac{\overbrace{f(x_0) + hf'(x_0) + \frac{h^2}{2}f''(\xi)}^{\text{Taylor expansion}} - f(x_0)}{h} - f'(x_0) \\
 &= \frac{1}{2}f''(\xi)h, \quad \xi \in (x_0, x_0 + h),
 \end{aligned}$$

Remark 1.8.2.

1. The value of $\xi \in (x_0, x_0 + h)$ is unknown. However, when h is small $f''(\xi) \approx f''(x_0)$.
2. The hypothesis $e(h) \approx Ch$ is correct with $C \approx f''(x_0)$ for sufficiently small values of h .
3. The theory above is only valid if f is twice continuous differentiable around x_0 .

1.8.6 Numerical verification of theoretical results, and a nice presentation of those

To verify the theoretical result, continue to measure the error $|e(h)|$ while different values of h , usually for stepsizes $h_0 \cdot 2^i$, $i = 1, 2, 3, \dots$, and make a log-log plot of the result. Choosing a log-log plot has two advantages: first, we can easily see the errors also for small values of h . But often, we want to verify that a numerical approximation of something has an error of order p , that is

$$|e(h)| \approx Ch^p \quad \Leftrightarrow \quad \log |e(h)| \approx p \log h + \log C,$$

and in a log-log plot the error as a function of h will appear as a straight line with slope p .

In Fig. 1.1 the result of such an experiment is presented. In this case $f(x) = \sinh x$, $x_0 = 1$ and $h_i = 2^i$, $i = 0, 1, \dots, 9$. Comparing with the reference line, we observe that

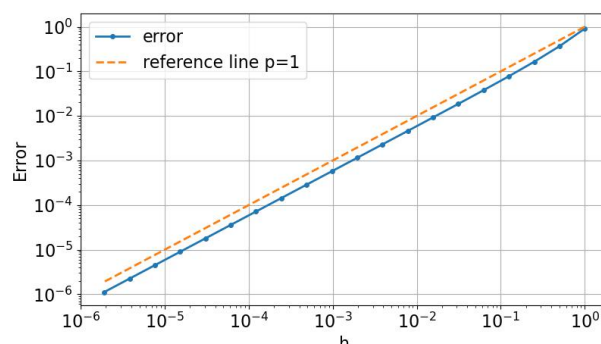


Figure 1.1: Error plot for example 3.

the numerical order for h sufficiently small is 1. For large h , there is some deviation from the straight line. This is quite reasonable (why?).

The Python-code producing this figure is given in `numdiff1.py`. Try the other examples, and/or make your own.

1.8.7 Can something go wrong?

Oh yes. Repeat the experiments for stepsizes 2^{-i} , $i = 0, 1, 2, \dots, 50$. The result is presented in Fig. 1.2. Everything seems fine for $h \leq 10^{-8}$, but for smaller h , the error increases in a rather irregular way. Which can be explained as follows: Let $y_2 = f(x_0 + h)$ and $y_1 = f(x_0)$. Both numbers are represented in the computer with small errors, $\bar{y}_1 = y_1(1 + \delta y_1)$ and similar for y_2 . From Section 1.6 we recall that for $v = y_2 - y_1$

$$\bar{v} = v(1 + \delta v), \quad \text{with} \quad \delta v = \frac{1}{y_2 - y_1} (y_2 \delta y_2 - y_1 \delta y_1).$$

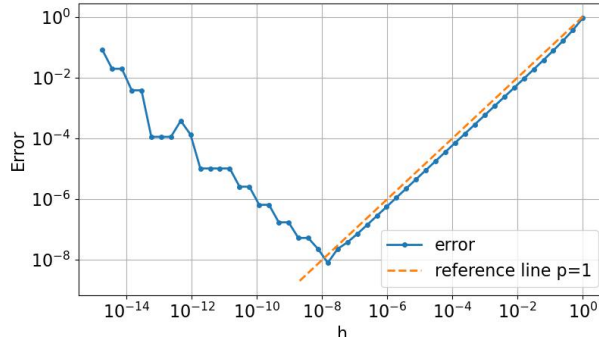


Figure 1.2: Error plot for example 3.

By the mean value theorem, there is an η between x_0 and $x_0 + h$ such that $y_2 - y_1 = f(x_0 + h) - f(x_0) = hf'(\eta)$. Since $h = 2^{-i}$ in our experiments, it is represented exactly. So

$$\bar{z} = \frac{\bar{y}_2 - \bar{y}_1}{h} = z(1 + \delta z)$$

where

$$z = \frac{(f(x_0 + h) - f(x_0))}{h} = f'(x_0) + \frac{h}{2}f''(\xi),$$

and

$$\delta z = \frac{1}{hf''(\eta)}(f(x_0 + h)\delta y_2 - f(x_0)\delta y_1)$$

Combining all this, and ignoring quadratic terms $h\delta y_i$, $i = 1, 2$, we get an expression for the absolute error

$$\bar{z} - f'(x_0) = \frac{h}{2}f''(\xi) + \frac{1}{h} \cdot \frac{f'(x_0)}{f''(\eta)}(f(x_0 + h)\delta y_2 - f(x_0)\delta y_1)$$

Assume $|\delta y_i| \leq \mathbf{u}$, which in the current case is the machine precision ϵ . Further assuming that f, f' and f'' are bounded in a neighbourhood around x_0 , the absolute error satisfies

$$|\bar{z} - f'(x_0)| \leq \frac{1}{2}Ch + \frac{2}{h}D\mathbf{u}.$$

For $h > \bar{h}$, the error is dominated by numerical error, for $h < \bar{h}$ by the rounding errors, and the minimum is approximately where these two intersect:

$$\frac{1}{2}C\bar{h} \approx \frac{2}{\bar{h}}D\mathbf{u} \quad \Rightarrow \quad \bar{h} \approx 2\sqrt{\frac{D}{C}}\sqrt{\mathbf{u}}.$$

The constants mentioned above is not known, so we just claim that the minimum error is obtained for

$$\bar{h} = \mathcal{O}(\sqrt{\epsilon}).$$

If $\epsilon \sim 10^{-16}$, a reasonable choice of h is 10^{-8} .

1.8.8 Numerical differentiation formulas

Below is listed some of the most popular finite difference approximations

$$f'(x) = \begin{cases} \frac{f(x+h) - f(x)}{h} - \frac{h}{2}f''(\xi), & \text{Forward difference} \\ \frac{f(x) - f(x-h)}{h} + \frac{h}{2}f''(\xi), & \text{Backward difference} \\ \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{6}f'''(\xi). & \text{Central difference} \end{cases}$$

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} - \frac{h^2}{12}f^{(4)}(\xi), \quad \text{Central difference}$$

Exercise 1.8.1. Prove the formulas above. Verify the theoretical results numerically, using the test problems above.

Chapter 2

Numerical Linear Algebra

2.1 Gauss Elimination

In this section it will be discussed how to solve a linear system of equations,

$$A\mathbf{x} = \mathbf{b} \tag{2.1}$$

where

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \in \mathbb{R}^{n \times n}, \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \in \mathbb{R}^n, \quad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \in \mathbb{R}^n.$$

Here A and $\mathbf{b}$ are known, $\mathbf{x}$ is the unknown to be found. We will further assume that A is nonsingular, so (2.1) has a unique solution $\mathbf{x}$.

2.1.1 Notation

A matrix $U \in \mathbb{R}^{n \times n}$ is *upper triangular* if $u_{ij} = 0$ whenever $i > j$. Similar, L is *lower triangular* if $l_{ij} = 0$ whenever $i < j$, that is

$$U = \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{pmatrix}, \quad L = \begin{pmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{pmatrix}.$$

2.1.2 Naive Gauss elimination

The aim is here to transform the original system $A\mathbf{x} = \mathbf{b}$ to an upper triangular system $\tilde{A}\mathbf{x} = \tilde{\mathbf{b}}$, which is quite easy to solve. The procedure is that standard Gauss elimination, which for most of you is well known from earlier calculus classes.

Given the system of equations:

$$\begin{array}{lcl} \text{Eq}_1 : & a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n & = b_1 \\ \text{Eq}_2 : & a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n & = b_2 \\ \vdots & & \vdots \\ \text{Eq}_n : & a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n & = b_n \end{array} \quad (2.2)$$

Keep Eq.1, and eliminate the first term of the remaining equations by

$$\text{Eq}_i^{(2)} = \text{Eq}_i - \frac{a_{i1}}{a_{11}} \cdot \text{Eq}_1, \quad i = 2, \dots, n$$

that is,

$$a_{ij}^{(2)} = a_{ij} - m_{i1}a_{1j}, \quad b_i^{(2)} = b_i - m_{i1}b_1, \quad \text{where } m_{i1} = a_{i1}/a_{11}, \quad i, j = 2, \dots, n.$$

The resulting system is of the form

$$\begin{array}{lcl} \text{Eq}_1 : & a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n & = b_1 \\ \text{Eq}_2^{(2)} : & a_{22}^{(2)}x_2 + \cdots + a_{2n}^{(2)}x_n & = b_2^{(2)} \\ \vdots & & \vdots \\ \text{Eq}_n^{(2)} : & a_{n2}^{(2)}x_2 + \cdots + a_{nn}^{(2)}x_n & = b_n^{(2)} \end{array}$$

Continue like this, and we end up with a triangular system of equations:

$$\begin{array}{lcl} \text{Eq}_1 : & a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n & = b_1 \\ \text{Eq}_2^{(2)} : & a_{22}^{(2)}x_2 + \cdots + a_{2n}^{(2)}x_n & = b_2^{(2)} \\ \vdots & & \vdots \\ \text{Eq}_{n-1}^{(n-1)} : & a_{n-1,n-1}^{(n-1)}x_{n-1} + a_{n-1,n}^{(n-1)}x_n & = b_{n-1}^{(n-1)} \\ \text{Eq}_n^{(2)} : & a_{nn}^{(n)}x_n & = b_n^{(n)} \end{array}$$

The transformed system can easily be solved, by starting with the last equation:

$$\begin{aligned} x_n &= b_n^{(n)} / a_{nn}^{(n)} \\ x_{n-1} &= (b_{n-1}^{(n-1)} - a_{n-1,n}^{(n-1)}x_n) / a_{n-1,n-1}^{(n-1)} \\ &\vdots \\ x_1 &= (b_1 - \sum_{j=2}^n a_{1j}x_j) / a_{11}. \end{aligned}$$

The solution of the triangular system is called *the back substitution*. The diagonal elements on the triangular matrix, $a_{ii}^{(i)}$ are called *pivot elements*, and the factors $m_{ik} = a_{ik}^{(k)} / a_{kk}^{(k)}$, for $i > k$ are called multipliers. They play a vital role later on.

Example 2.1.1.

$$\begin{array}{cccc|ccc} x_1 & + & 2x_2 & + & x_3 & = & 3, & a_{11} = 1, \\ 3x_1 & + & 4x_2 & + & & = & 3, & m_{21} = 3, \\ 2x_1 & + & 10x_2 & + & 4x_3 & = & 10, & m_{31} = 2. \end{array}$$

Keep the first equation, and eliminate the first term of the remaining equations.

$$\begin{array}{rrcr|l} x_1 & + & 2x_2 & + & x_3 & = & 3, \\ & - & 2x_2 & - & 3x_3 & = & -6, \\ & & 6x_2 & + & 2x_3 & = & 4, \end{array} \quad \left| \quad \begin{array}{l} a_{22}^{(2)} = -2, \\ m_{32} = -3, \end{array} \right.$$

and the third step is

$$\begin{array}{rrcr|l} x_1 & + & 2x_2 & + & x_3 & = & 3, \\ & - & 2x_2 & - & 3x_3 & = & -6, \\ & & & - & 7x_3 & = & -14, \end{array} \quad \left| \quad \begin{array}{l} a_{33}^{(3)} = -7. \end{array} \right.$$

Back substitution gives

$$x_3 = 2, \quad x_2 = 0, \quad x_1 = 1$$

so the solution vector $\mathbf{x} = (1, 0, 2)^\top$.

Algorithm 1 Naive Gauss elimination with back-solution

Input: $(a_{ij})_{i,j=1}^n, (b_i)_{i=1}^n$.

for $k = 1, 2, \dots, n-1$ **do**

for $i = k+1, \dots, n$ **do**

$m_{ik} \leftarrow a_{ik}/a_{kk}$

for $j = k+1, \dots, n$ **do**

$a_{ij} \leftarrow a_{ij} - m_{ik}a_{kj}$

end for

$b_i \leftarrow b_i - m_{ik}b_k$

end for

end for

$x_n \leftarrow b_n/a_{nn}$

▷ Back substitution

for $i = n-1, n-2, \dots, 1$ **do**

$x_i = (b_i - \sum_{j=i+1}^n a_{ij}x_j) / a_{ii}$

end for

Output $(x_i)_{i=1}^n$.

The algorithm will in principle works successfully as long as all the pivot elements $a_{ii} \neq 0$. This is a topic for further discussion.

The Gauss-elimination process can also be used as factorization method.

Definition 2.1.2. LU-factorization Let $A \in \mathbb{R}^{n \times n}$ be an invertible matrix. An *LU*-factorization of A is given by

$$A = LU$$

where L is a lower triangular matrix with 1's at the diagonal, and U is an upper triangular matrix with nonzero diagonal elements.

Theorem 2.1.3. *If the Gauss-elimination process is successfull, with only nonzero pivot elements, then A has an LU-factorization where U is the upper triangular matrix produced by the process, and*

$$L = \begin{pmatrix} 1 & & & & \\ m_{21} & 1 & & & \\ m_{31} & m_{32} & 1 & & \\ \vdots & & \ddots & \ddots & \\ m_{n1} & m_{n2} & \cdots & m_{n,n-1} & 1 \end{pmatrix}$$

Proof. Step k , $A^{(k+1)} = M_k A^{(k)}$ with $A^{(1)} = A$ in the algorithm is given by

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - m_{ik} a_{kj}^{(k)}.$$

where $m_{ik} = a_{ik}^{(k)} / a_{kk}^{(k)}$ are the multipliers. The Gauss-elimination process can then be considered as a sequence of matrix operations

$$U = A^{(n)} = M_{n-1} M_{n-2} \cdots M_1 A$$

thus

$$A = \overbrace{(M_1^{-1} M_2^{-1} \cdots M_{n-1}^{-1})}^L U,$$

so we need to prove that $L = M_1^{-1} M_2^{-1} \cdots M_{n-1}^{-1}$. Each transformation matrix M_k can then be written as

$$M_k = \begin{pmatrix} 1 & & & & & & \\ 0 & 1 & & & & & \\ \vdots & & \ddots & & & & \\ \vdots & & & 1 & & & \\ \vdots & & & -m_{k+1,k} & 1 & & \\ \vdots & & & \vdots & & \ddots & \\ 0 & \cdots & \cdots & -m_{n,k} & 0 & \cdots & 1 \end{pmatrix} = I_n - \mathbf{m}_k \mathbf{e}_k^\top$$

where I_n is the identity matrix and

$$\mathbf{e}_k = (\overbrace{0, \dots, 0}^{k-1}, 1, \overbrace{0, \dots, 0}^{n-k})^\top, \quad \mathbf{m}_k = (\overbrace{0, \dots, 0}^k, \overbrace{m_{k+1,k}, \dots, m_{n,k}}^{n-k})^\top.$$

Then $M_k^{-1} = I_n + \mathbf{m}_k \mathbf{e}_k^\top$ since

$$M_k^{-1} M_k = (I_n + \mathbf{m}_k \mathbf{e}_k^\top)(I_n - \mathbf{m}_k \mathbf{e}_k^\top) = I_n - \mathbf{m}_k \overbrace{\mathbf{e}_k^\top \mathbf{m}_k}^{=0} \mathbf{e}_k.$$

A similar argument shows that

$$M_k^{-1} M_{k+1}^{-1} = I_n + \mathbf{m}_k \mathbf{e}_k^\top + \mathbf{m}_{k+1} \mathbf{e}_{k+1}^\top$$

thus

$$M_1^{-1} M_2^{-1} \cdots M_{n-1}^{-1} = I_n + \sum_{k=1}^{n-1} \mathbf{m}_k \mathbf{e}_k^\top = L$$

as stated by the theorem. □

Solving the problem $A\mathbf{x} = \mathbf{b}$ is then done in three steps:

$A = LU$	The LU -factorization
Solve $L\mathbf{y} = \mathbf{b}$	Forward substitution
Solve $U\mathbf{x} = \mathbf{y}$	Back substitution

In the remaining part of this chapter, we will therefore use U for $A^{(n)}$, and similar $\mathbf{y}$ for $\mathbf{b}^{(n)}$, but not completely consistent. $\mathbf{b}^{(n)}$

Example 2.1.4. Consider Example 2.1.1 again. The LU -factorization of A is given as

$$\overbrace{\begin{pmatrix} 1 & 2 & 1 \\ 3 & 4 & 0 \\ 2 & 10 & 4 \end{pmatrix}}^A = \overbrace{\begin{pmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 2 & -3 & 1 \end{pmatrix}}^L \cdot \overbrace{\begin{pmatrix} 1 & 2 & 1 \\ 0 & -2 & -3 \\ 0 & 0 & -7 \end{pmatrix}}^U$$

Computational complexity of the Gauss-elimination process

The complexity is measured in flop (floating point operations), that is one addition and one multiplication together, or one division. A matrix-vector multiplication $A\mathbf{x}$ with $A \in \mathbb{R}^{n \times n}$, $\mathbf{x} \in \mathbb{R}^n$ takes n^2 flop. The complexity of the different stages of an LU -factorization and forward and back substitution of a system of n equations are

$$\begin{aligned} A = LU & \quad \overbrace{\sum_{k=1}^n (n-k)^2}^{a_{ij}, (*, +)} + \overbrace{\sum_{k=1}^n (n-k)}^{m_{ij}, (/)} = \frac{1}{3}n^3 - \frac{1}{3}n \approx \frac{1}{3}n^3 \text{ (flop)} \\ L\mathbf{y} = \mathbf{b} & \quad \sum_{k=2}^n (k-1) = \frac{1}{2}n^2 - \frac{1}{2}n \approx \frac{1}{2}n^2 \text{ (flop)} \\ U\mathbf{x} = \mathbf{y} & \quad \sum_{k=2}^n (k-1) + n = \frac{1}{2}n^2 + \frac{1}{2}n \approx \frac{1}{2}n^2 \text{ (flop)} \end{aligned}$$

Remark 2.1.5. In mathematics, it is common to write the solution of a linear system on the form $\mathbf{x} = A^{-1}\mathbf{b}$. If A^{-1} is known (which is usually not), this matrix-vector multiplication requires n^2 flop, exactly the same amount of work as the forward and back substitution if the LU -factorization is done. Thus, the LU factorized matrix works as a computational equivalent to the inverse. This is in particular useful if the same system is solved several times, with the same coefficient matrix but with different right hand sides $\mathbf{b}$.

Let $D = \text{diag}\{u_{11}, u_{22}, \dots, u_{nn}\}$ be the diagonal matrix with the pivot elements u_{11} at the diagonal. Since, by assumption, $u_{ii} \neq 0$, D is invertible, giving rise to alternative LU -factorizations:

$$A = LU = LDD^{-1}U = \hat{L}\hat{U} \quad \hat{L} = LD, \quad \hat{U} = D^{-1}U,$$

where $\hat{U}$ is an upper triangular matrix with 1's at the diagonal. More important, if A is symmetric, that is $A = A^\top$ and $u_{ii} > 0$ for $i = 1, 2, \dots, n$, then

$$A = CC^\top, \quad C = L\sqrt{D},$$

where $\sqrt{D} = \text{diag}\{\sqrt{u_{11}}, \dots, \sqrt{u_{nn}}\}$. This is called a *Cholesky factorization*.

Definition 2.1.6. A matrix $A \in \mathbb{R}^{n \times n}$ is strictly diagonally dominant if

$$|a_{ii}| > \sum_{j=1, j \neq i}^n |a_{ij}|, \quad \text{for } i = 1, 2, \dots, n.$$

Theorem 2.1.7. *If $A \in \mathbb{R}^{n \times n}$ is strictly diagonally dominant then it is nonsingular and it has an LU-factorization.*

Proof. We want to prove that all the pivot elements $a_{ii}^{(i)}$ created from the Gauss elimination process are nonzero. Assume A is diagonal dominant. Clearly, the first pivot element $a_{11} \neq 0$. The first step in the Gauss elimination process transforms

$$A \rightarrow \begin{pmatrix} a_{11} & \cdots \\ \mathbf{0} & \tilde{A}^{(2)} \end{pmatrix}$$

and it is sufficient to prove that also $\tilde{A}^{(2)}$ is strictly diagonal dominant, that is

$$|a_{ii}^{(2)}| > \sum_{j=2, j \neq i} |a_{ij}^{(2)}|, \quad \text{for } i = 2, 3, \dots, n, \quad (2.3)$$

where

$$a_{ij}^{(2)} = a_{ij} - \frac{a_{i1}}{a_{11}} a_{1j}. \quad (2.4)$$

In the proof, we will also use the two triangle inequalities $|x + y| \leq |x| + |y|$ and $|x - y| \geq |x| - |y|$. We then have:

$$\begin{aligned} \sum_{j=2, j \neq i} |a_{ij}^{(2)}| &= \sum_{j=2, j \neq i} \left| a_{ij} - \frac{a_{i1}}{a_{11}} a_{1j} \right| && \text{from (2.4)} \\ &\leq \sum_{j=2, j \neq i} |a_{ij}| + \frac{|a_{i1}|}{|a_{11}|} |a_{1j}| && |x + y| \leq |x| + |y| \\ &= \sum_{j=1, j \neq i}^n |a_{ij}| - |a_{i1}| + \frac{|a_{i1}|}{|a_{11}|} \left(\sum_{j=2}^n |a_{1j}| - |a_{1i}| \right) \\ &\leq |a_{ii}| - |a_{i1}| + \frac{|a_{i1}|}{|a_{11}|} (|a_{11}| - |a_{1i}|) && \text{since } A \text{ is sdd} \\ &= |a_{ii}| - \frac{|a_{i1}|}{|a_{11}|} |a_{1i}| \\ &\leq \left| a_{ii} - \frac{a_{i1}}{a_{11}} a_{1i} \right| = |a_{ii}^{(2)}|, && |x| - |y| \leq |x - y| \text{ and (2.4)} \end{aligned}$$

proving (2.3). □

2.1.3 Gauss elimination with pivoting

Obviously, even for nonsingular A -matrices, the Gauss elimination process may fail, if one of the pivot elements $a_{kk}^{(k)}$ becomes 0. This is not the only problem that may occur, which the following (quite standard) problem demonstrates:

Example 2.1.8.

$$\begin{array}{rcrcrcrcl} \varepsilon x_1 & + & x_2 & = & 1, \\ x_1 & + & x_2 & = & 2 \end{array}$$

where $|\varepsilon| \ll 1$. Obviously, the solution to this problem is $x_1 \approx 1$ and $x_2 \approx 1$. By the naive Gauss elimination process, the multiplier $m_{21} = 1/\varepsilon$, and the second equation is transferred into

$$(1 - 1/\varepsilon)x_2 = (2 - 1/\varepsilon) \quad \Rightarrow \quad x_2 = \frac{2 - 1/\varepsilon}{1 - 1/\varepsilon} \approx 1$$

as expected. The first unknown x_1 can be found from the first equation

$$x_1 = (1 - x_2)/\varepsilon \approx 0,$$

if the approximation to x_2 is used. Which is obviously wrong. The problem here is that we two almost equal numbers, 1 and x_2 , are subtracted, leading to large relative errors (see Section 1.6. Then this number is divided by a small number, ε , leading to large absolute errors.

Switching the two equations will solve the problem (try it yourself).

The problematic part is visible in the back substitution part of Algorithm 1,

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

If x_i is of a reasonable size, the pivot element $a_{ii}^{(i)}$ very small, then also $b_i^{(i)} \approx \sum_{j=i+1}^n a_{ij}^{(i)} x_j$, which are operations subjected to large errors in the computations. The remedy is to make the pivot elements small, which can be done by changing the order of the equations, a process called *pivoting*.

- **Partial row pivoting:** For each k in the algorithm, find the smallest q such that $|a_{qk}^{(k)}| = \max_{k \leq i \leq n} |a_{ik}^{(k)}|$. Switch row q and k .
- **Scaled partial row pivoting:**
 - Before the elimination process starts, find the scale $\mathbf{s} \in \mathbb{R}^n$ with the elements $s_i = \max_j |a_{ij}|$. This vector is not recomputed during the elimination process.
 - For each elimination step k , find the smallest q such that

$$\frac{a_{qk}^{(k)}}{s_q} = \max_{k \leq i \leq n} \frac{|a_{ik}^{(k)}|}{s_i}.$$

Switch row k and q (including s_k and s_q).

Example 2.1.9. Let us apply these procedures on the problem $\mathbf{Ax} = \mathbf{b}$, where

$$A = \begin{pmatrix} 1 & 2 & 1 \\ 3 & 4 & 0 \\ 2 & 10 & 4 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 3 \\ 3 \\ 10 \end{pmatrix}.$$

Partial row pivoting:

$$\left[\begin{array}{ccc|c} 1 & 2 & 1 & 3 \\ \color{red}{3} & 4 & 0 & 3 \\ 2 & 10 & 4 & 10 \end{array} \right]$$

Choose the pivot row $q = 2$ $\Downarrow$

Switch row 1 and 2

$$\left[\begin{array}{ccc|c} \color{red}{3} & 4 & 0 & 3 \\ 1 & 2 & 1 & 3 \\ 2 & 10 & 4 & 10 \end{array} \right]$$

 $m_{21} = 1/3, m_{31} = 2/3$ $\Downarrow$

Elimination step

$$\left[\begin{array}{ccc|c} \color{red}{3} & 4 & 0 & 3 \\ 0 & 2/3 & 1 & 2 \\ 0 & \color{red}{22/3} & 4 & 8 \end{array} \right]$$

Choose the pivot row $q = 3$ $\Downarrow$

Switch row 2 and 3

$$\left[\begin{array}{ccc|c} \color{red}{3} & 4 & 0 & 3 \\ 0 & \color{red}{22/3} & 4 & 8 \\ 0 & 2/3 & 1 & 2 \end{array} \right]$$

 $m_{32} = 1/11$ $\Downarrow$

Elimination step

$$\left[\begin{array}{ccc|c} \color{red}{3} & 4 & 0 & 3 \\ 0 & \color{red}{22/3} & 4 & 8 \\ 0 & 0 & \color{red}{7/11} & 14/11 \end{array} \right]$$

The back substitution gives $\mathbf{x} = (1, 0, 2)^\top$.

Scaled partial row pivoting The scale vector is $\mathbf{s} = (2, 4, 10)^\top$. Then the first pivot element is chosen from $\max_i \{a_{i1}/s_i\} = \max\{1/2, 3/4, 2/10\}$, thus $q = 2$. Switch row 1 and 2 as above, including the relevant elements in the scale vectors. The next pivot element is chosen from $\max\{(2/3)/2, (23/3)/10\} = \max\{1/3, 23/30\}$, so $q = 3$. Switch row 2 and 3. In this particular case, the two pivoting strategies produces the same result, this is in general not the case.

In an implementation of these algorithm, the order of the rows are not changed, but the sequence in which the elimination process is done is kept track of by a pivot vector $\mathbf{p}$.

- Start with $\mathbf{p} = (1, 2, 3, \dots, n)^\top$.
- Replace all row indices with p_i , $(a_{p_i,j}, b_{p_i}, s_{p_i})$.
- Before each elimination step, find the pivot row q and switch p_q and p_i .

A *permutation matrix* $P \in \mathbb{R}^{n \times n}$ is a permutation of the identity matrix I_n , thus at each row and column, there is one and only one element equal to 1, all other elements are 0.

- PA changes the order of the *rows* of A .
- AP changes the order of the *columns* of A .

Example 2.1.10.

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 2 & 1 \\ 3 & 4 & 0 \\ 2 & 10 & 4 \end{pmatrix} = \begin{pmatrix} 3 & 4 & 0 \\ 2 & 10 & 4 \\ 1 & 2 & 1 \end{pmatrix}$$

We conclude this section with the following important result, here stated without proof:

Theorem 2.1.11. *If $A \in \mathbb{R}^{n \times n}$ is nonsingular, then there exist a permutation matrix P , a lower triangular matrix L with 1s at the diagonal, and an upper triangular matrix U such that*

$$PA = LU.$$

2.2 Vector and Matrix Norms

Definition 2.2.1. A *real vector space* is a set V together with the operations $+$ (addition) and $\cdot$ (scalar multiplication) which for all $x, y, z \in V$ and $\alpha, \beta \in \mathbb{R}$ satisfy the following conditions:

- 1) $x + y \in V$
- 2) $x + y = y + x$
- 3) $x + (y + z) = (x + y) + z$
- 4) There exist a $0 \in V$ such that $x + 0 = x$
- 5) For all $x \in V$ there is a $-x \in V$ such that $x + (-x) = 0$
- 6) $\alpha x \in V$
- 7) $\alpha(\beta x) = (\alpha\beta)x$
- 8) $1x = x$
- 9) $\alpha(x + y) = \alpha x + \alpha y$
- 10) $(\alpha + \beta)x = \alpha x + \beta x$

Example 2.2.2. • $\mathbb{R}^n$ is the set of all real vectors with n components.

- $\mathbb{R}^{m \times n}$ is the set of all $m \times n$ real matrices.
- $C^m[a, b]$ is the set of all functions with continuous first m derivatives on the interval $[a, b]$. It is common to use $C[a, b]$ rather than $C^0[a, b]$ for all continuous functions.
- $\mathbb{P}_n$ is the set of all polynomials of degree at most n .

A set of vectors in $\{v_i\}_{i=1}^q \subset V$ are *linear independent* if

$$\sum_{i=1}^q a_i v_i = 0 \quad \Leftrightarrow \quad a_i = 0, \quad i = 1, \dots, q. \quad (a_i \in \mathbb{R}).$$

If there is a set of linear independent vectors $\{v_i\}_{i=1}^n$ such that all $w \in V$ can be expressed as

$$w = \sum_{i=1}^n a_i v_i, \quad a_i \in \mathbb{R}$$

then $\{v_i\}_{i=1}^n$ is a *basis* of V and we write $V = \text{span}\{v_i\}_{i=1}^n$. In this case, the space V is *n-dimensional*.

In this section, we focus on the vector spaces $\mathbb{R}^n$ and $\mathbb{R}^{n \times n}$.

2.2.1 Norms

Definition 2.2.3. A *norm* on a vector space V is a function $\|\cdot\| \rightarrow \mathbb{R}$ with the following properties:

- 1) $\|v\| \geq 0$ and $\|v\| = 0 \Leftrightarrow v \equiv 0$ for all $v \in V$,
- 2) $\|\alpha v\| = |\alpha| \|v\|$, for all $v \in V$ and $\alpha \in \mathbb{R}$,
- 3) $\|v + w\| \leq \|v\| + \|w\|$ for all $v, w \in V$ (Triangle inequality).

A *normed vector space* is a vector space with a norm, denoted by $(V, \|\cdot\|)$.

Proposition 2.2.4. Let $\mathbf{x} = (x_1, \dots, x_n)^\top \in \mathbb{R}^n$. The following are all norms:

- *Summation norm* $\|\mathbf{x}\|_1 = \sum_{i=1}^n |x_i|$,

- *Euclidian norm (2-norm)*: $\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^n x_i^2} = \sqrt{\mathbf{x}^\top \mathbf{x}}$,
- *Max-norm*: $\|\mathbf{x}\|_\infty = \max_{i=1}^n |x_i|$.

NB!: These are only the most commonly used norms. There are infinitely many other choices!

For the proof, we will need the following famous result:

Lemma 2.2.5 (Cauchy-Schwarz's inequality).

$$|\mathbf{x}^\top \mathbf{y}| \leq \|\mathbf{x}\|_2 \|\mathbf{y}\|_2, \quad \forall \mathbf{x}, \mathbf{y} \in \mathbb{R}^n.$$

Proof. The result is trivial if either $\mathbf{x} = \mathbf{0}$ or $\mathbf{y} = \mathbf{0}$. So let us now assume that both are non-zero. Define the polynomial

$$p(t) = \|\mathbf{x} + t\mathbf{y}\|^2, \quad t \in \mathbb{R}^n.$$

This is a second order polynomial of t , and obviously, $p(t) \geq 0$ for all t . We will now find the minimum value of $p(t)$ and use this to prove the Lemma 2.2.5. We have

$$p(t) = \|\mathbf{x}\|^2 + 2t\mathbf{x}^\top \mathbf{y} + t^2 \|\mathbf{y}\|^2$$

The polynomial reaches it's minimim at $t_{\min} = -\mathbf{x}^\top \mathbf{y} / \|\mathbf{y}\|^2$, the minimum value of the polynomial is

$$p(t_{\min}) = \|\mathbf{x}\|^2 - \frac{(\mathbf{x}^\top \mathbf{y})^2}{\|\mathbf{y}\|^2} \geq 0.$$

By some reorganization this is

$$(\mathbf{x}^\top \mathbf{y})^2 \leq \|\mathbf{x}\|^2 \|\mathbf{y}\|^2$$

and taking the square root on both sides provides the desired result. $\square$

Proof of Proposition 2.2.4. The proofs for the summation and the max-norms are trivial, and are ommitted. For the euclidian norm, the following holds:

- 1) Trivial.
- 2) $\|\alpha \mathbf{x}\|_2^2 = \sum_{i=1}^n \alpha^2 x_i^2 = \alpha^2 \sum_{i=1}^n x_i^2 = |\alpha|^2 \|\mathbf{x}\|_2^2$
- 3) $\|\mathbf{x} + \mathbf{y}\|_2^2 = \mathbf{x}^\top \mathbf{x} + 2\mathbf{x}^\top \mathbf{y} + \mathbf{y}^\top \mathbf{y} \leq \|\mathbf{x}\|_2^2 + 2\|\mathbf{x}\| \|\mathbf{y}\| + \|\mathbf{y}\|_2^2 = (\|\mathbf{x}\|_2 + \|\mathbf{y}\|_2)^2$.

Then, for point 1) and 2): Take the square root on both side to complete the argument. $\square$

To prove that the summation and the max-norms actually are norms, is left to the reader.

As is demonstrated in these examples, the norm to which we refer is usually marked with a subscript. We will always use the absolute value as the norm of a real number, thus $\|x\|$ whenever $x \in \mathbb{R}$.

Example 2.2.6. Let $\mathbf{x} = (1, -6, 3, -1, 5)^\top \in \mathbb{R}^5$. Then

$$\begin{aligned} \|\mathbf{x}\|_2 &= \sqrt{1 + 36 + 9 + 1 + 25} = 8.4853, \\ \|\mathbf{x}\|_\infty &= \max\{1, 6, 3, 1, 5\} = 6. \end{aligned}$$

Norms of vectors in $\mathbb{R}^n$ are implemented in Python, in the module `numpy.linalg`.

2.2.2 Matrix norms

It is of course possible to define norms on $\mathbb{R}^{n \times n}$ just satisfying the conditions in Definition 2.2.3. But it can make even more sense to think about a matrix A as a linear operator from $A : \mathbb{R}^n \rightarrow \mathbb{R}^n$.

Definition 2.2.7. A *matrix norm* obeys the following:

- i) $\|A\| \geq 0$ and $\|A\| = 0 \Leftrightarrow A = 0$.
- ii) $\|\alpha A\| = |\alpha| \|A\|$,
- iii) $\|A + B\| \leq \|A\| + \|B\|$,
- iv) $\|AB\| \leq \|A\| \|B\|$

for all $A, B \in \mathbb{R}^{n \times n}$ and $\alpha \in \mathbb{R}$

Theorem 2.2.8 (Natural norms). *For all norms $\|\cdot\|$ on $\mathbb{R}^n$ there is a corresponding matrix norm*

$$\|A\| = \max_{\substack{\mathbf{x} \in \mathbb{R}^n \\ \mathbf{x} \neq 0}} \frac{\|A\mathbf{x}\|}{\|\mathbf{x}\|} = \max_{\substack{\mathbf{x} \in \mathbb{R}^n \\ \|\mathbf{x}\|=1}} \|A\mathbf{x}\|.$$

Such norms are called natural, subordinate and induced.

The proof is left as an exercise.

Proposition 2.2.9. *The natural matrix norm given from the vector norms defined above is:*

If $A \in \mathbb{R}^{n \times n}$ then

$$\|A\|_1 = \max_j \sum_{i=1}^n |a_{ij}|, \quad \|A\|_\infty = \max_i \sum_{j=1}^n |a_{ij}| \quad \text{and} \quad \|A\|_2 = \sqrt{\max_i \lambda_i}$$

where $\{\lambda_i\}_{i=1}^n$ are the eigenvalues of $A^\top A$.

Partial proof. Only the result for $\|\cdot\|_\infty$ is given here.

For $\|\cdot\|$ to be a natural norm, we need to prove the following: For all $A \in \mathbb{R}^{n \times n}$ and for all $\mathbf{x} \in \mathbb{R}^n$, $\|\mathbf{x}\| = 1$ $\|A\mathbf{x}\| \leq \|A\|$, and that there exist an $\mathbf{x}$ with norm 1 such that $\|A\mathbf{x}\| = \|A\|$.

So let us see what this means for the max-norm:

Start with the first condition. Since $\|\mathbf{x}\|_\infty = 1$ we know that $|x_i| \leq 1$ for $i = 1, 2, \dots, n$, and there is at least one q for which $|x_q| = 1$. Thus

$$\|A\mathbf{x}\|_\infty = \max_i \left| \sum_{j=1}^n a_{ij} x_j \right| \leq \max_i \sum_{j=1}^n |a_{ij} x_j| \leq \max_i \sum_{j=1}^n |a_{ij}| = \|A\|_\infty$$

proving the first part. For the second part, choose p such that

$$\sum_{j=1}^n |a_{pj}| = \max_i \sum_{j=1}^n |a_{ij}|.$$

Then, let $\mathbf{x}$ be the vector given by

$$x_j = \begin{cases} 1 & \text{if } a_{pj} \geq 0, \\ -1 & \text{if } a_{pj} < 0, \end{cases}$$

such that $a_{pj}x_j \geq 0$ and $|a_{pj}x_j| = |a_{pj}|$. For this choice of $\mathbf{x}$ the following holds:

$$\|A\|_\infty = \max_i \sum_{j=1}^n |a_{ij}| = \sum_{j=1}^n |a_{pj}| = \sum_{j=1}^n |a_{pj}x_j| = \max_i \left| \sum_{j=1}^n a_{ij}x_j \right| = \|A\mathbf{x}\|_\infty.$$

□

2.3 Error analysis of linear system of equations

Definition 2.3.1. The *condition number* of a matrix $A \in \mathbb{R}^{n \times n}$ is defined as

$$\kappa(A) = \|A\| \|A^{-1}\|.$$

Proposition 2.3.2. Let $A \in \mathbb{R}^{n \times n}$ be invertible, and $\mathbf{x}$ is the solution of $A\mathbf{x} = \mathbf{b}$. Further, let $\tilde{\mathbf{x}}$ be the solution of the perturbed system $A\tilde{\mathbf{x}} = \tilde{\mathbf{b}}$. Then

$$\frac{\|\mathbf{x} - \tilde{\mathbf{x}}\|}{\|\mathbf{x}\|} \leq \kappa(A) \frac{\|\mathbf{b} - \tilde{\mathbf{b}}\|}{\|\mathbf{b}\|}.$$

Proof. Let $\delta\mathbf{x} = \mathbf{x} - \tilde{\mathbf{x}}$ and $\delta\mathbf{b} = \mathbf{b} - \tilde{\mathbf{b}}$. Then the norm of the absolute value is

$$\|\delta\mathbf{x}\| = \|\mathbf{x} - \tilde{\mathbf{x}}\| = \|A^{-1}(\mathbf{b} - \tilde{\mathbf{b}})\| = \|A^{-1}\delta\mathbf{b}\| \leq \|A^{-1}\| \|\delta\mathbf{b}\|.$$

The relative error satisfies

$$\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \frac{\|A^{-1}\| \|\delta\mathbf{b}\|}{\|\mathbf{x}\|} = \|A^{-1}\| \frac{\|\mathbf{b}\|}{\|\mathbf{x}\|} \frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|} \stackrel{\mathbf{b}=A\mathbf{x}}{=} \|A^{-1}\| \frac{\|A\mathbf{x}\|}{\|\mathbf{x}\|} \frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|} \stackrel{\|A\mathbf{x}\| \leq \|A\| \|\mathbf{x}\|}{\leq} \|A\| \|A^{-1}\| \frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|}$$

□

If $\kappa(A) \gtrsim 1$ then A is *well conditioned*, small errors in $\mathbf{b}$ will only result in small errors in the solution $\mathbf{x}$. If $\kappa(A) \gg 1$, the matrix is *ill-conditioned*. Small errors in $\mathbf{b}$ may result in huge errors in $\mathbf{x}$.

Chapter 3

Numerical Solution of Ordinary Differential Equations

3.1 Introduction

We will here discuss the numerical solution of systems of ordinary differential equations (ODEs). There is a simple code enclosed, which can be used for some experiments.

3.1.1 Scalar ODEs.

A scalar ODE is an equation of the form

$$y'(t) = f(t, y(t)), \quad y(t_0) = y_0,$$

where $y'(t) = \frac{dy}{dt}$. The *initial condition* $y(t_0) = y_0$ is required for a unique solution.

NB! It is common to use the term *initial value problem (IVP)* for an ODE for which the initial value $y(t_0) = y_0$ is given, and we only are interested in the solution for $t > t_0$. In this note, only initial value problems are considered.

Example 3.1.1. Consider the solution of the ODE

$$y'(t) = -2ty(t),$$

which can be written in the form $y'(t) = f(t, y(t))$ with

$$f(t, y) = -2ty.$$

The general solution is the function

$$y(t) = Ce^{-t^2},$$

where C is a constant that depends on the initial condition $y(t_0)$, which we can find by using the techniques from Mathematics 1: We write first $y' = dy/dt$ and then multiply formally with dt to obtain

$$dy = -2ty \, dt.$$

If y is equal to 0, we see that the solution of the equation is the constant zero function. Else we can divide by y and obtain

$$\frac{1}{y} dy = -2t \, dt.$$

Now we compute indefinite integrals of both sides, which yields

$$\log|y| = -t^2 + K.$$

Taking the exponential on both sides, we get

$$|y| = e^K e^{-t^2},$$

or

$$y = \pm e^K e^{-t^2}.$$

Setting $C = \pm e^K$, we finally obtain the solution

$$y(t) = C e^{-t^2}$$

for some $C \in \mathbb{R}$ (the case $C = 0$ corresponds to the constant zero solution we have discussed previously). In order to find the constant C , we have to use the initial condition for $y(t_0)$. For instance, we obtain for $t_0 = 0$ and $y(0) = 1$ the solution

$$y(t) = e^{-t^2}.$$

3.1.2 Systems of ODEs.

A system of m ODEs is given by

$$\begin{aligned} y_1' &= f_1(t, y_1, y_2, \dots, y_m), & y_1(t_0) &= y_{1,0} \\ y_2' &= f_2(t, y_1, y_2, \dots, y_m), & y_2(t_0) &= y_{2,0} \\ &\vdots & &\vdots \\ y_m' &= f_m(t, y_1, y_2, \dots, y_m), & y_m(t_0) &= y_{m,0} \end{aligned}$$

or, more compactly, by

$$\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t)), \quad \mathbf{y}(t_0) = \mathbf{y}_0$$

where we use boldface to denote vectors in $\mathbb{R}^m$.

$$\mathbf{y}(t) = \begin{pmatrix} y_1(t) \\ y_2(t) \\ \vdots \\ y_m(t) \end{pmatrix}, \quad \mathbf{f}(t, \mathbf{y}) = \begin{pmatrix} f_1(t, y_1, y_2, \dots, y_m) \\ f_2(t, y_1, y_2, \dots, y_m) \\ \vdots \\ f_m(t, y_1, y_2, \dots, y_m) \end{pmatrix}, \quad \mathbf{y}_0 = \begin{pmatrix} y_{1,0} \\ y_{2,0} \\ \vdots \\ y_{m,0} \end{pmatrix},$$

Example 3.1.2. The Lotka-Volterra equation is a system of two ODEs describing the interaction between predators and prey over time. The system is given as

$$\begin{aligned} y_1'(t) &= \alpha y_1(t) - \beta y_1(t) y_2(t), \\ y_2'(t) &= \delta y_1(t) y_2(t) - \gamma y_2(t). \end{aligned}$$

Here t denotes time, $y_1(t)$ describes the population of the prey species, and $y_2(t)$ the population of predators. The positive parameters α , β , δ , and γ depend on the populations to be modelled. We can write this system in the form $\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t))$ with

$$\mathbf{f}(t, \mathbf{y}) = \begin{pmatrix} \alpha y_1 - \beta y_1 y_2 \\ \delta y_1 y_2 - \gamma y_2 \end{pmatrix}.$$

Note that the parameter t here actually does not appear on the right hand side.

3.1.3 Autonomous ODEs.

An ODE is called *autonomous* if $\mathbf{f}$ is not a function of t , but only of $\mathbf{y}$. The Lotka-Volterra equation is an example of an autonomous ODE. A nonautonomous system can be made autonomous by a simple trick: Just add the equation

$$y'_{m+1} = 1, \quad y_{m+1}(t_0) = t_0,$$

and replace t with y_{m+1} . Thus, the ODE from Example 1, with initial values $y(0) = 1$ can be written as

$$\begin{aligned} y'_1 &= -2y_2y_1, & y_1(0) &= 1, \\ y'_2 &= 1, & y_2(0) &= 0. \end{aligned}$$

3.1.4 Higher order ODEs.

An initial value problem for an ODE of order m is given by

$$u^{(m)} = f(t, u, u', \dots, u^{(m-1)}), \quad u(t_0) = u_0, \quad u'(t_0) = u'_0, \quad \dots, \quad u^{(m-1)}(t_0) = u_0^{(m-1)}.$$

Here $u^{(1)} = u'$ and $u^{(m+1)} = \frac{du^{(m)}}{dt}$ for $m > 0$.

Such equations can be written as a system of first order ODEs by the following trick:

Let

$$y_1(t) = u(t), \quad y_2(t) = u'(t), \quad y_3(t) = u^{(2)}(t), \quad \dots, \quad y_m(t) = u^{(m-1)}(t)$$

such that

$$\begin{aligned} y'_1 &= y_2, & y_1(t_0) &= u_0, \\ y'_2 &= y_3, & y_2(t_0) &= u'_0, \\ &\vdots & &\vdots \\ y'_{m-1} &= y_m, & y_{m-1}(t_0) &= u_0^{(m-2)}, \\ y'_m &= f(t, y_1, y_2, \dots, y_{m-1}, y_m), & y_m(t_0) &= u_0^{(m-1)}. \end{aligned}$$

This is a normal system of first order ODEs. The solution u of the original m -th order ODE is just the first component y_1 of the solution of this system.

Example 3.1.3. Van der Pol's equation is a second order differential equation, given by

$$u'' = \mu(1 - u^2)u' - u, \quad u(0) = u_0, \quad u'(0) = u'_0,$$

where $\mu > 0$ is some constant. Common choices for initial values are $u_0 = 2$ and $u'_0 = 0$.

It can be rewritten as a system of first order ODEs:

$$\begin{aligned} y'_1 &= y_2, & y_1(0) &= u_0, \\ y'_2 &= \mu(1 - y_1^2)y_2 - y_1, & y_2(0) &= u'_0. \end{aligned}$$

3.2 Numerical methods for solving ODEs

In this note, we will discuss some techniques for the numerical solution of ordinary differential equations. For simplicity or presentation, we will develop and discuss these methods mostly based on scalar ODEs. The same methods, however, are equally applicable for systems of equations.

All the methods that we will discuss are so-called *one-step methods*. Given the ODE and the initial values (t_0, y_0) , we choose some step size h and let $t_1 = t_0 + h$. Based on this information, we calculate an approximation y_1 to $y(t_1)$. Then, we repeat this process starting from (t_1, y_1) in order to calculate an approximation y_2 of $y(t_2)$, where $t_2 = t_1 + h$. This process is repeated until some final point, here called t_{end} is reached.

In one-step methods, the approximation y_{k+1} of $y(t_{k+1})$ does not depend on the values of $y_{k-1}, y_{k-2}, \dots, y_0$. The main alternative to this type of methods are *multi-step methods*, where the approximation y_{k+1} of $y(t_{k+1})$ takes into account those values as well.

It should be emphasized that this strategy only will find approximations to the exact solution in some discrete points $t_n, n = 0, 1, \dots$

3.2.1 Euler's method

Let us start with the simplest example, Euler's method, known from Mathematics 1.

Given a (scalar) IVP

$$y'(t) = f(t, y(t)), \quad y(t_0) = y_0,$$

choose some step size h . The trick is as follows:

Perform a Taylor expansion (*Preliminaries*, section 4) of the exact (but unknown) solution $y(t_0 + h)$ around t_0 :

$$y(t_0 + h) = y(t_0) + hy'(t_0) + \frac{1}{2}h^2y''(t_0) + \dots$$

Assume the step size h to be small, such that the solution is dominated by the first two terms. In that case, these can be used as the numerical approximation in the next step, as the ODE implies that

$$y'(t_0) = f(t_0, y(t_0)) = f(t_0, y_0).$$

From this we obtain the approximation

$$y(t_0 + h) \approx y(t_0) + hy'(t_0) = y_0 + hf(t_0, y_0),$$

giving

$$y_1 = y_0 + hf(t_0, y_0).$$

This extends trivially to a system of equations, and to a general n . In practice, we would like to compute the solution over a finite interval t_0, t_{end} , using a finite number of steps, say N , in which case the stepsize is $h = (t_{\text{end}} - t_0)/N$. The complete algorithm (for a system of ODEs) becomes:

Algorithm 2 Euler's method

Input: Right hand side $\mathbf{f}$, interval (t_0, t_{end}) , initial value $\mathbf{y}_0$ and number of steps N .
 $h = (t_{\text{end}} - t_0)/N$
for $n = 0, 1, 2, \dots, N - 1$ **do**
 $\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{f}(t_n, \mathbf{y}_n)$
 $t_{n+1} = t_n + h$
end for
Output: $(t_n, \mathbf{y}_n)_{n=0}^N$

Exercise 3.2.1. Test the implementation of Euler's method on the problem

$$y'(t) = -2ty(t), \quad y(0) = 1, \quad 0 \leq t \leq 1,$$

which has the analytic solution

$$y(t) = e^{-t^2}.$$

Try with different step sizes, for instance $h = 0.1$, $h = 0.05$ and $h = 0.01$. In each case, compare the numerical solution with the exact one.

Exercise 3.2.2. Solve the Lotka-Volterra equation

$$\begin{aligned} y_1'(t) &= \alpha y_1(t) - \beta y_1(t)y_2(t), & y_1(0) &= y_{1,0}, \\ y_2'(t) &= \delta y_1(t)y_2(t) - \gamma y_2(t), & y_2(0) &= y_{2,0}. \end{aligned}$$

In this exercise, use the parameters and initial values

$$\alpha = 2, \quad \beta = 1, \quad \delta = 0.5, \quad \gamma = 1, \quad y_{1,0} = 2, \quad y_{2,0} = 0.5.$$

Solve the equation over the interval $[0, 20]$, and use $h = 0.02$. Try also other step sizes, e.g. $h = 0.1$ and $h = 0.002$. **NB!** In this case, the exact solution is not known. What is known is that the solutions are periodic and positive. Is this the case for the numerical solutions as well? Check for different values of h .

3.2.2 Heun's method

We will now discuss a first, improved, alternative to Euler's method:

Assume that we want to solve an

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

Its exact solution can be written in integral form as

$$\mathbf{y}(t_n + h) = \mathbf{y}(t_n) + \int_{t_n}^{t_n+h} \mathbf{y}'(t) dt = \mathbf{y}(t_n) + \int_{t_n}^{t_n+h} \mathbf{f}(t, \mathbf{y}(t)) dt.$$

We now replace the integral on the right hand side by a numerical approximation using the trapezoidal rule:

$$\mathbf{y}(t_n + h) \approx \mathbf{y}(t_n) + \frac{h}{2} (\mathbf{f}(t_n, \mathbf{y}(t_n)) + \mathbf{f}(t_{n+1}, \mathbf{y}(t_{n+1}))).$$

Then we replace $\mathbf{y}(t_n)$ and $\mathbf{y}(t_{n+1})$ by the approximations $\mathbf{y}_n$ and $\mathbf{y}_{n+1}$. The resulting method is the *trapezoidal rule for ODEs*, given by

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{h}{2}(\mathbf{f}(t_n, \mathbf{y}_n) + \mathbf{f}(t_{n+1}, \mathbf{y}_{n+1})).$$

This is an example of an *implicit* method. The numerical approximation $\mathbf{y}_{n+1}$ appears on both sides of this equation and is therefore only implicitly given: If $t_n, \mathbf{y}_n$ is known, a nonlinear equation has to be solved in order to find $\mathbf{y}_{n+1}$, and this has to be done for each step. There are important applications where this actually makes sense, which we will discuss in a later lecture in the context of *stiff ODEs*.

However, for the moment we want to avoid this additional complication and thus replace the $\mathbf{y}_{n+1}$ on the right hand side by some approximation. One natural possibility here is to apply one step of Euler's method. This results in Heun's method:

$$\begin{aligned} \mathbf{u}_{n+1} &= \mathbf{y}_n + h\mathbf{f}(t_n, \mathbf{y}_n), \\ \mathbf{y}_{n+1} &= \mathbf{y}_n + \frac{h}{2}(\mathbf{f}(t_n, \mathbf{y}_n) + \mathbf{f}(t_{n+1}, \mathbf{u}_{n+1})). \end{aligned}$$

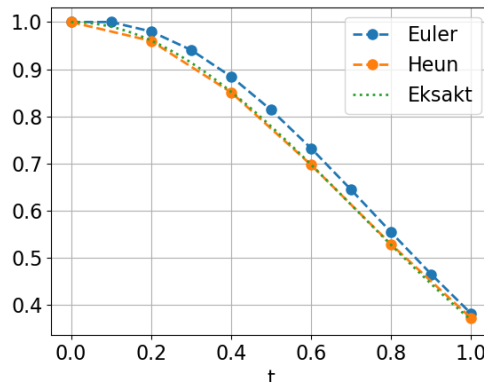
The method is commonly written in the form

$$\begin{aligned} \mathbf{k}_1 &= \mathbf{f}(t_n, \mathbf{y}_n), \\ \mathbf{k}_2 &= \mathbf{f}(t_n + h, \mathbf{y}_n + h\mathbf{k}_1), \\ \mathbf{y}_{n+1} &= \mathbf{y}_n + \frac{h}{2}(\mathbf{k}_1 + \mathbf{k}_2). \end{aligned}$$

Example 3.2.1. Let us compare the numerical solution from Euler's and Heun's methods on the scalar test problem

$$y' = -2ty, \quad y(0) = 1$$

with the exact solution $y(t) = e^{-t^2}$ on the interval $[0, 1]$. Use $h = 0.1$ for Euler's method and $h = 0.2$ for Heun's method. The result is:



and it is clear that Heun's method is significantly more accurate than Euler's method.

The main computational work in these methods lies in the evaluations of the function $\mathbf{f}$, especially in the case of large systems of ODEs. Euler's method requires one function evaluation in each iteration, whereas Heun's method requires two function evaluations: one for the computation of $\mathbf{k}_1$ and a second one for the computation of $\mathbf{k}_2$. This means

that a single iteration of Heun's method is roughly double as computationally expensive as one iteration of Euler's method. If we want to have a fair comparison between the two methods, we should thus use half as many iterations for Heun's method as for Euler's method, which in turn means that the step length h for Heun's method should be double that of Euler's method. In this example, using Euler's method with $h = 0.1$ and Heun's method with $h = 0.2$ both leads to 10 function evaluations for the solution on the interval $[0, 1]$, and thus to a comparable total amount of computational work.

Doing so, we observe that Heun's method is significantly more accurate than Euler's method, even when the number of function evaluations are the same. Further, we notice that the error in Heun's method is reduced by a factor of approximately $1/4$ whenever the step size is reduced by a factor $1/2$, indicating that the error satisfies $|y(t_{\text{end}}) - y_N| \approx Ch^2$, and the method is of order 2. In the theory part below, we will prove that this certainly is the case.

Exercise 3.2.3. Solve the Lotka-Volterra equation by Euler's and Heun's methods, again using twice as many steps for Euler's method as for Heun's method.

- 1) Use $h = 0.01$ for Euler's method and $h = 0.02$ for Heun's method.
- 2) Use $h = 0.1$ for Euler's method and $h = 0.2$ for Heun's method.

Exercise 3.2.4. Solve Van der Pol's equation by use of Heun's method and with Euler's method, and compare. Experiment with different choices of the step size h . Use $\mu = 2$, and solve the equation on the interval $[0, 20]$. Experiment with different values of μ .

Exercise 3.2.5. Implement the classical Runge-Kutta method (also known as RK4) and verify numerically that the order of the method is 4. The method is given by

$$\begin{aligned} \mathbf{k}_1 &= \mathbf{f}(t_n, \mathbf{y}_n) \\ \mathbf{k}_2 &= \mathbf{f}\left(t_n + \frac{h}{2}, \mathbf{y}_n + \frac{h}{2}\mathbf{k}_1\right) \\ \mathbf{k}_3 &= \mathbf{f}\left(t_n + \frac{h}{2}, \mathbf{y}_n + \frac{h}{2}\mathbf{k}_2\right) \\ \mathbf{k}_4 &= \mathbf{f}(t_n + h, \mathbf{y}_n + h\mathbf{k}_3) \\ \mathbf{y}_{n+1} &= \mathbf{y}_n + \frac{h}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4). \end{aligned}$$

3.2.3 Runge-Kutta methods

An s -stage Runge-Kutta method is given by

$$\begin{aligned} \mathbf{k}_i &= \mathbf{f}\left(t_n + c_i h, \mathbf{y}_n + h \sum_{j=1}^s a_{ij} \mathbf{k}_j\right), \quad i = 1, 2, \dots, s, \\ \mathbf{y}_{n+1} &= \mathbf{y}_n + h \sum_{i=1}^s b_i \mathbf{k}_i. \end{aligned}$$

The method is defined by its coefficients, which are given in a *Butcher tableau*

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \cdots & a_{1s} \\ c_2 & a_{21} & a_{22} & \cdots & a_{2s} \\ \vdots & \vdots & & & \vdots \\ c_s & a_{s1} & a_{s2} & \cdots & a_{ss} \\ \hline & b_1 & b_2 & \cdots & b_s \end{array}$$

with

$$c_i = \sum_{j=1}^s a_{ij}, \quad i = 1, \dots, s.$$

The method is *explicit* if $a_{ij} = 0$ whenever $j \geq i$; otherwise it is *implicit*.

The Butcher-tableaux for some well known methods are:

$\begin{array}{c c} 0 & 0 \\ \hline & 1 \end{array}$	$\begin{array}{c cc} 0 & 0 & 0 \\ \hline 1 & 1 & 0 \\ & \frac{1}{2} & \frac{1}{2} \end{array}$	$\begin{array}{c cc} 0 & 0 & 0 \\ \hline 1 & \frac{1}{2} & \frac{1}{2} \\ & \frac{1}{2} & \frac{1}{2} \end{array}$	$\begin{array}{c cccc} 0 & 0 & 0 & 0 & 0 \\ \hline \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}$
Euler	Heun	trapezoidal rule	RK4

See the list of Runge–Kutta methods on Wikipedia for more examples.

3.3 Theory

3.3.1 Existence and uniqueness results

Let us first state the conditions for which the ODE has a unique solution. We will need the following definition (which will also be used later in this note):

Definition 3.3.1. Lipschitz condition A function $\mathbf{f}: \mathbb{R} \times \mathbb{R}^m \rightarrow \mathbb{R}^m$ satisfies a *Lipschitz condition* with respect to y on a domain $(a, b) \times D$ where $D \subset \mathbb{R}^m$ if there exist a constant L so that

$$\|\mathbf{f}(t, \mathbf{y}) - \mathbf{f}(t, \mathbf{z})\| \leq L \|\mathbf{y} - \mathbf{z}\|, \quad \text{for all } t \in (a, b) \text{ and } \mathbf{y}, \mathbf{z} \in D.$$

The constant L is called *the Lipschitz constant*.

It is not hard to show that the function $\mathbf{f}$ satisfies the Lipschitz condition if the functions $\partial f_i / \partial y_j$, $i, j = 1, \dots, m$ are continuous and bounded on the domain and D is open and convex.

Theorem 3.3.2. *Existence and uniqueness of a solution* Consider the initial value problem

$$\mathbf{y}' = \mathbf{f}(t, \mathbf{y}), \quad \mathbf{y}(t_0) = \mathbf{y}_0.$$

If

- $\mathbf{f}(t, \mathbf{y})$ is continuous in $(a, b) \times D$,
- $\mathbf{f}(t, \mathbf{y})$ satisfies the Lipschitz condition with respect to $\mathbf{y}$ in $(a, b) \times D$,

with given initial values $t_0 \in (a, b)$ and $\mathbf{y}_0 \in D$, then the ODE has one and only one solution in $(a, b) \times D$.

In the following, we will assume that these conditions are satisfied.

3.3.2 Error analysis of Euler's method

When an ODE is solved by Euler's method over some interval $[t_0, t_{\text{end}}]$, how will the error at t_{end} (or some arbitrary point) depend on the number of steps? Or more specifically,

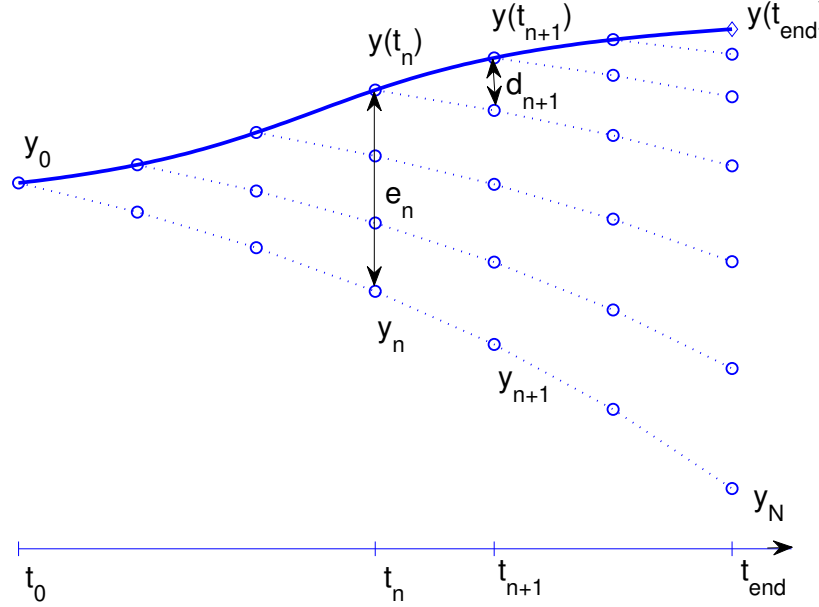


Figure 3.1: Lady Windermere's fan, named after a comedy play by Oscar Wilde. The figure describes the transport and the accumulation of the local truncation errors d_{n+1} into the global error at the end point, $e_N = y(t_{\text{end}}) - y_N$.

choose the number of steps N , let the step size be $h = (t_{\text{end}} - t_0)/N$, such that $t_{\text{end}} = t_N$. What can we say about the error $e_N = y(t_{\text{end}}) - y_N$?

The error analysis will be done on a scalar equation, but it also holds for systems of equations, as described in the end of the section.

Local and global errors. In this discussion we have to consider two kinds of errors:

- *Local truncation error d_{n+1} :* This is the error made in one single step, starting from $(t_n, y(t_n))$.
- *Global error e_n :* This is the difference between the exact and the numerical solution after n steps, $e_n = y(t_n) - y_n$.

Theorem 3.3.3. *Consider the scalar ODE*

$$y' = f(t, y), \quad y(t_0) = y_0$$

to be solved over the interval $[t_0, t_{\text{end}}]$ by Euler's method, using stepsizes $h = (t_{\text{end}} - t_0)/N$. Assume that f , f_y and f_t are bounded in the neighbourhood of the solution. Then there is a constant $C > 0$, independent of h such that the global error is

$$|e_N| \leq Ch,$$

thus the method is convergent of order 1.

Proof. The proof is divided in two parts. First, find an expression for the local truncation error. Then, prove how these errors propagate and add up to find a bound for the global error.

The Taylor expansion of the exact solution of the ODE, starting from $(t_n, y(t_n))$ is given by

$$y(t_n + h) = y(t_n) + hy'(t_n) + \frac{1}{2}h^2y''(\xi), \quad t_n < \xi < t_n + h.$$

and one step with Euler's method, starting from $(t_n, y(t_n))$ is

$$y(t_n + h) = y(t_n) + hf(t_n, y(t_n)).$$

where $y'(t_n) = f(t_n, y(t_n))$. Thus the local truncation error is

$$d_{n+1} = y(t_n + h) - (y(t_n) + hy'(t_n)) = \frac{1}{2}h^2y''(\xi), \quad \xi \in (t_n, t_n + h). \quad (3.1)$$

The next step is to see how the global error e_n propagates over one step. We have

$$y(t_n + h) = y(t_n) + hf(t_n, y(t_n)) + d_{n+1}, \quad \text{Equation (3.1)}$$

$$y_{n+1} = y_n + hf(t_n, y_n), \quad \text{Euler's method.}$$

Subtract the second equation from the first, use that $e_n = y(t_n) - y_n$ and the mean value theorem for functions¹. This yields the expression

$$e_{n+1} = e_n + h(f(t_n, y(t_n)) - f(t_n, y_n)) + d_{n+1} = e_n + hf_y(t_n, \eta)e_n + d_{n+1},$$

where $f_y = \frac{\partial f}{\partial y}$, and η is some value between y_n and $y(t_n)$. We now take the absolute value on each side, and apply the triangle inequality:

$$|e_{n+1}| = |e_n + hf_y(t_n, \eta)e_n + d_{n+1}| \leq |e_n| + h|f_y(t_n, \eta)||e_n| + |d_{n+1}|.$$

From the assumptions we know that f_y is bounded, e.g. there is a constant $L > 0$ such that $|f_y| < L$. Similar $y'' = (f(t, y))' = f_t + f_y f$ is also bounded, there is a $D > 0$ such that $|y''(t)| < 2D$.

From this the following inequality is obtained:

$$|e_{n+1}| \leq (1 + hL)|e_n| + Dh^2.$$

Since $y_0 = y(t_0)$ the initial error $e_0 = 0$. The inequality above therefore results in the following bounds for the global errors:

$$\begin{aligned} |e_1| &\leq Dh^2 \\ |e_2| &\leq (1 + hL)|e_1| + Dh^2 \leq ((1 + hL) + 1)Dh^2 \\ |e_3| &\leq (1 + hL)|e_2| + Dh^2 \leq ((1 + hL)^2 + (1 + hL) + 1)Dh^2 \\ &\vdots \\ |e_N| &\leq (1 + hL)|e_{N+1}| + Dh^2 \leq \sum_{n=0}^{N-1} (1 + hL)^n Dh^2 \end{aligned}$$

Using the formula for the sum of the first terms of a geometric series² and the series of

¹The mean value theorem: There is a ν between y and z such that $g(y) - g(z) = g'(\nu)(y - z)$. In our case, use $g(y) = f(t, y)$ for a fixed t .

²Summation formula: $\sum_{n=0}^{N-1} r^n = \frac{r^N - 1}{r - 1}$ for $r \in \mathbb{R}$.

the exponential function³ we obtain the following:

$$\sum_{n=0}^{N-1} (1 + hL)^n = \frac{(1 + hL)^N - 1}{(1 + hL) - 1} < \frac{(e^{hL})^N - 1}{hL} = \frac{e^{hLN} - 1}{hL} = \frac{e^{L(t_{\text{end}} - t_0)} - 1}{hL},$$

where the last equality holds because $(t_{\text{end}} - t_0) = hN$. Finally, we plug this into the inequality for $|e_N|$ above, and we see that we have proved the the following upper bound for the global:

$$|y(t_{\text{end}}) - y_N| = |e_N| \leq \frac{e^{L(t_{\text{end}} - t_0)} - 1}{L} Dh = Ch,$$

where the constant $C = \frac{e^{L(t_{\text{end}} - t_0)} - 1}{L} D$ depends on the length of the integration interval $t_{\text{end}} - t_0$, of certain properties of the equation (L and D), but *not* on the step size h . $\square$

The numerical solution converges to the exact solution since

$$\lim_{N \rightarrow \infty} |e_N| = 0.$$

If the step size is reduced by a factor of 0.5, so will the error. This is in agreement with the previous numerical result.

Remark 3.3.4. Following the proof of the error estimate for Euler's method, we see that a local truncation error of size h^2 leads to a final, global error of size h . Intuitively, this is because we need to take roughly $1/h$ steps in order to reach the point t_{end} , although a precise proof is quite a bit more complicated than that, as we have seen. However, what this also should mean is that a method with a truncation order of size h^{p+1} should lead to a global error of h^p . The results of the next section show that, under certain conditions, this is indeed the case.

3.3.3 A general convergence result

A one-step method applied to a system of ODEs $\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t))$ can be written in the generic form

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\Phi(t_n, \mathbf{y}_n; h),$$

where the *increment function* Φ typically depends on the function $\mathbf{f}$ and some parameters defining the method.

For example, for Euler's method we have

$$\Phi(t_n, \mathbf{y}_n; h) = \mathbf{f}(t_n, \mathbf{y}_n),$$

whereas for Heun's method

$$\Phi(t_n, \mathbf{y}_n; h) = \frac{1}{2} \left(\mathbf{f}(t_n, \mathbf{y}_n) + \mathbf{f}(t_n + h, \mathbf{y}_n + h\mathbf{f}(t_n, \mathbf{y}_n)) \right).$$

All Runge–Kutta methods can in principle be written in this form as well (though in general for a rather complicated increment function Φ).

³The exponential: $e^x = 1 + x + \frac{1}{2}x^2 + \cdots = 1 + x + \sum_{n=2}^{\infty} \frac{x^n}{n!}$. Thus $1 + x < e^x$ whenever $x > 0$.

Definition 3.3.5. Order of a method. A method is of order $p > 0$ if there is a constant $C > 0$ such that

$$\|\mathbf{e}_N\| = \|\mathbf{y}(t_{\text{end}}) - \mathbf{y}_N\| \leq Ch^p,$$

where N is the number of steps taken to reach t_{end} using the step size $h = (t_{\text{end}} - t_0)/N$.

The local truncation error $\mathbf{d}_{n+1}$ of such a method is

$$\mathbf{d}_{n+1} = \mathbf{y}(t_{n+1}) - (\mathbf{y}(t_n) + h\Phi(t_n, \mathbf{y}(t_n); h))$$

Replace the absolute values in the above proof with norms (*Preliminaries*, section 1), and the argument above can be used to prove the following:

Theorem 3.3.6. *Convergence of one-step methods. Assume that there exist positive constants M and D such that the increment function satisfies*

$$\|\Phi(t, \mathbf{y}; h) - \Phi(t, \mathbf{z}; h)\| \leq M\|\mathbf{y} - \mathbf{z}\| \quad (3.2)$$

and the local truncation error satisfies

$$\|\mathbf{y}(t+h) - (\mathbf{y}(t) + h\Phi(t, \mathbf{y}(t), h))\| \leq Dh^{p+1}$$

for all t , $\mathbf{y}$ and $\mathbf{z}$ in a neighbourhood of the solution. In that case, the global error satisfies

$$\|\mathbf{e}_N\| = \|\mathbf{y}(t_{\text{end}}) - \mathbf{y}_N\| \leq Ch^p, \quad \text{with } C = \frac{e^{M(t_{\text{end}}-t_0)} - 1}{M}D.$$

In the case where Φ is a Runge–Kutta method, one can show that (3.2) is always satisfied for some constant M provided that the function $\mathbf{f}$ satisfies a Lipschitz condition.

Example 3.3.7. Let us consider Eulers method, $\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{f}(t_n, \mathbf{y}_n)$. So in this case $\Phi(t, \mathbf{y}) = \mathbf{f}(t, \mathbf{y})$ and (3.2) is exactly the same as the Lipschitz condition for $\mathbf{f}$.

3.3.4 Order conditions for Runge–Kutta methods.

It can be proved that a Runge–Kutta method is of order p with $p \leq 4$ if all the conditions up to and including p in the table below are satisfied:

p	conditions
1	$\sum_i b_i = 1$
2	$\sum_i b_i c_i = 1/2$
3	$\sum_i b_i c_i^2 = 1/3$ $\sum_{i,j} b_i a_{ij} c_j = 1/6$
4	$\sum_i b_i c_i^3 = 1/4$ $\sum_{i,j} b_i c_i a_{ij} c_j = 1/8$ $\sum_{i,j} b_i a_{ij} c_j^2 = 1/12$ $\sum_{i,j,k} b_i a_{ij} a_{jk} c_k = 1/24$

Here sums are taken over all the indices from 1 to s , with s being the number of stages of the method. It is possible to derive similar conditions also for higher orders p , but the conditions rapidly become more and more complex.

Recall also, that we have always assumed that Runge–Kutta methods satisfy the conditions $c_i = \sum_j a_{ij}$ for all $i = 1, \dots, s$.

Example 3.3.8. Apply the conditions to Heun's method, for which $s = 2$ and the Butcher tableau is

$$\begin{array}{c|cc} c_1 & a_{11} & a_{12} \\ c_2 & a_{21} & a_{22} \\ \hline & b_1 & b_2 \end{array} = \begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \end{array}.$$

The order conditions are:

$$p = 1 \qquad b_1 + b_2 = \frac{1}{2} + \frac{1}{2} = 1 \qquad \text{OK}$$

$$p = 2 \qquad b_1 c_1 + b_2 c_2 = \frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 1 = \frac{1}{2} \qquad \text{OK}$$

$$\begin{aligned} p = 3 \qquad b_1 c_1^2 + b_2 c_2^2 &= \frac{1}{2} \cdot 0^2 + \frac{1}{2} \cdot 1^2 = \frac{1}{2} \neq \frac{1}{3} && \text{Not satisfied} \\ b_1(a_{11}c_1 + a_{12}c_2) + b_2(a_{21}c_1 + a_{22}c_2) &= \frac{1}{2}(0 \cdot 0 + 0 \cdot 1) + \frac{1}{2}(1 \cdot 0 + 0 \cdot 1) \\ &= 0 \neq \frac{1}{6} && \text{Not satisfied} \end{aligned}$$

The method is of order 2.

Convergence properties of Heun's method (optional). This is included mostly to demonstrate that convergence analysis is not a trivial task.

To prove convergence and to find the order of a method two things are needed:

- the local truncation error, expressed as a power series in the step size h
- the condition $\|\Phi(t, \mathbf{y}; h) - \Phi(t, \mathbf{z}; h)\| \leq M\|\mathbf{y} - \mathbf{z}\|$

The local truncation error is found by comparing Taylor series expansions of the exact and the numerical solutions starting from the same point. In practice, this is not trivial. For simplicity, we will here do this only for a scalar equation $y'(t) = f(t, y(t))$. The result is valid for systems as well.

In the following, we will use the notation

$$f_t = \frac{\partial f}{\partial t}, \quad f_y = \frac{\partial f}{\partial y}, \quad f_{tt} = \frac{\partial^2 f}{\partial t^2}, \quad f_{ty} = \frac{\partial^2 f}{\partial t \partial y} \quad \text{etc.}$$

Further, we will suppress the arguments of the function f and its derivatives. So f is to be understood as $f(t, y(t))$ although it is not explicitly written.

The Taylor expansion of the exact solution $y(t + h)$ is given by

$$y(t + h) = y(t) + hy'(t) + \frac{h^2}{2}y''(t) + \frac{h^3}{6}y'''(t) + \dots$$

Higher derivatives of $y(t)$ can be expressed in terms of the function f by using the chain rule and the product rule for differentiation:

$$\begin{aligned} y'(t) &= f, \\ y''(t) &= f_t + f_y y' = f_t + f_y f, \\ y'''(t) &= f_{tt} + f_{ty} y' + f_{yt} f + f_{yy} y' f + f_y f_t + f_y f_y y' = f_{tt} + 2f_{ty} f + f_{yy} f^2 + f_y f_t + (f_y)^2 f. \end{aligned}$$

We then find the series of the exact and the numerical solution around x_0, y_0 (any other point will do equally well). From the discussion above, the series for the exact solution becomes

$$y(t_0 + h) = y_0 + hf + \frac{h^2}{2}(f_t + f_y f) + \frac{h^3}{6}(f_{tt} + 2f_{ty} f + f_{yy} f^2 + f_y f_t + (f_y)^2 f) + \dots,$$

where f and all its derivatives are evaluated in (t_0, y_0) . For the numerical solution we get

$$\begin{aligned} k_1 &= f(t_0, y_0) = f, \\ k_2 &= f(t_0 + h, y_0 + hk_1) \\ &= f + hf_t + f_y hk_1 + \frac{1}{2}f_{tt}h^2 + f_{ty}h^2k_1 + \frac{1}{2}f_{yy}h^2k_1^2 + \dots \\ &= f + h(f_t + f_y f) + \frac{h^2}{2}(f_{tt} + 2f_{ty} f + f_{yy} f^2) + \dots, \\ y_1 &= y_0 + \frac{h}{2}(k_1 + k_2) = y_0 + \frac{h}{2}\left(f + f + h(f_t + f_y f) + \frac{h^2}{2}(f_{tt} + 2f_{ty} f + f_{yy} f^2) + \dots\right) \\ &= y_0 + hf + \frac{h^2}{2}(f_t + f_y f) + \frac{h^3}{4}(f_{tt} + 2f_{ty} f + f_{yy} f^2) + \dots, \end{aligned}$$

and the local truncation error will be

$$d_1 = y(t_0 + h) - y_1 = \frac{h^3}{12}(-f_{tt} - 2f_{ty} f - f_{yy} f^2 + 2f_y f_t + 2(f_y)^2 f) + \dots$$

The first nonzero term in the local truncation error series is called *the principal error term*. For h sufficiently small this is the term dominating the error, and this fact will be used later.

Although the series has been developed around the initial point, series around $t_n, y(t_n)$ will give similar results, and it is possible to conclude that, given sufficient differentiability of f , there is a constant D such that

$$|d_n| \leq Dh^3.$$

Further, we have to prove the condition on the increment function $\Phi(t, y)$. For f differentiable, there is for all y, z some ξ between t and y such that $f(t, y) - f(t, z) = f_y(t, \xi)(y - z)$. Let L be a constant such that $|f_y| \leq L$, and for all t, y, z of interest we get

$$|f(t, y) - f(t, z)| \leq L|y - z|.$$

The increment function for Heun's method is given by

$$\Phi(t, y) = \frac{1}{2}(f(t, y) + f(t + h, y + hf(t, y))).$$

By repeated use of the condition above and the triangle inequality for absolute values we get

$$\begin{aligned}
 |\Phi(t, y) - \Phi(t, z)| &= \frac{1}{2} |f(t, y) + f(t + h, y + hf(t, y)) - f(t, z) - hf(t + h, z + hf(t, z))| \\
 &\leq \frac{1}{2} (|f(t, y) - f(t, z)| + |f(t + h, y + hf(t, y)) - f(t + h, z + hf(t, z))|) \\
 &\leq \frac{1}{2} (L|y - z| + L|y + hf(t, y) - z - hf(t, z)|) \\
 &\leq \frac{1}{2} (2L|y - z| + hL^2|y - z|) \\
 &= \left(L + \frac{h}{2}L^2\right)|y - z|.
 \end{aligned}$$

Assuming that the step size h is bounded above by some H , we can conclude that

$$|\Phi(t, y) - \Phi(t, z)| \leq M|y - z|, \quad M = L + \frac{H}{2}L^2.$$

In conclusion: Heun's method is convergent of order 2.

3.4 Error estimation and step size selection

To control the global error $y(t_n) - y_n$ is notoriously difficult, and far beyond what will be discussed in this course. To control the local error in each step and adjust the step size accordingly is rather straightforward, as we will see.

3.4.1 Error estimation

Given two methods, one of order p and the other of order $p + 1$ or higher. Assume we have reached a point $(t_n, \mathbf{y}_n)$. One step forward with each of these methods can be written as

$$\begin{aligned}
 \mathbf{y}_{n+1} &= \mathbf{y}_n + h\Phi(t_n, \mathbf{y}_n; h), & \text{order } p, \\
 \hat{\mathbf{y}}_{n+1} &= \mathbf{y}_n + h\hat{\Phi}(t_n, \mathbf{y}_n; h), & \text{order } p + 1 \text{ or more.}
 \end{aligned}$$

Let $\mathbf{y}(t_{n+1}; t_n, \mathbf{y}_n)$ be the exact solution of the ODE through $(t_n, \mathbf{y}_n)$. We would like to find an estimate for *the local error* $\mathbf{l}_{n+1}$, that is, the error in one step starting from $(t_n, \mathbf{y}_n)$,

$$\mathbf{l}_{n+1} = \mathbf{y}(t_{n+1}; t_n, \mathbf{y}_n) - \mathbf{y}_{n+1}.$$

As we already have seen, the local error is found by finding the power series in h of the exact and the numerical solution. The local error is of order p if the lowest order terms in the series where the exact and the numerical solution differ is of order $p + 1$. So the local errors of the two methods are

$$\begin{aligned}
 \mathbf{y}(t_{n+1}; t_n, \mathbf{y}_n) - \mathbf{y}_{n+1} &= \Psi(t_n, \mathbf{y}_n)h^{p+1} + \dots, \\
 \mathbf{y}(t_{n+1}; t_n, \mathbf{y}_n) - \hat{\mathbf{y}}_{n+1} &= \dots,
 \end{aligned}$$

where $\Psi(t_n, \mathbf{y}_n)$ is a term consisting of method parameters and differentials of $\mathbf{f}$ and ... contains all the terms of the series of order $p + 2$ or higher. Taking the difference gives

$$\hat{\mathbf{y}}_{n+1} - \mathbf{y}_{n+1} = \Psi(t_n, \mathbf{y}_n)h^{p+1} + \dots$$

Assume now that h is small, such that the *principal error term* $\Psi(t_n, \mathbf{y}_n)h^{p+1}$ dominates the error series. Then a reasonable approximation to the unknown local error $\mathbf{l}_{n+1}$ is the *local error estimate* $\mathbf{le}_{n+1}$:

$$\mathbf{le}_{n+1} = \hat{\mathbf{y}}_{n+1} - \mathbf{y}_{n+1} \approx \mathbf{y}(t_{n+1}; t_n, \mathbf{y}_n) - \mathbf{y}_{n+1}.$$

Example 3.4.1. Apply Euler's method of order 1 and Heun's method of order 2 with $h = 0.1$ to the equation

$$y' = -2ty, \quad y(0) = 1.$$

Use this to find an approximation to the error after one step.

Euler's method:

$$y_1 = 1.0 - 0.1 \cdot 2 \cdot 0 \cdot 1.0 = 1.0.$$

Heun's method

$$\begin{aligned} k_1 &= -2 \cdot 0.0 \cdot 1.0 = 0.0, \\ k_2 &= -2 \cdot 0.1 \cdot (1 + 0.0) = -0.2, \\ \hat{y}_1 &= 1.0 + \frac{0.1}{2} \cdot (0.0 - 0.2) = 0.99. \end{aligned}$$

The error estimate and the local error are respectively

$$le_1 = \hat{y}_1 - y_1 = -10^{-2}, \quad l_1 = y(0.1) - y_1 = e^{-0.1^2} - 1.0 = -0.995 \cdot 10^{-2}.$$

So in this case the error estimate is a quite decent approximation to the actual local error.

3.4.2 Stepsize control

The next step is to control the local error, that is, choose the step size so that $\|\mathbf{le}_{n+1}\| \leq \text{Tol}$ for some given tolerance Tol, and for some chosen norm $\|\cdot\|$.

Essentially:

Given $t_n, \mathbf{y}_n$ and a step size h_n .

- Perform one step with the method of choice, and find the error estimate $\mathbf{le}_{n+1}$.
- if $\|\mathbf{le}_{n+1}\| < \text{Tol}$
 - Accept the solution $t_{n+1}, \mathbf{y}_{n+1}$.
 - If possible, increase the step size for the next step.
- else
 - Repeat the step from $(t_n, \mathbf{y}_n)$ with a reduced step size h_n .

In both cases, the step size will change. But how?

From the discussion above, we have that

$$\|\mathbf{le}_{n+1}\| \approx Dh_n^{p+1}.$$

where $\mathbf{le}_{n+1}$ is the error estimate we can compute, D is some unknown quantity, which we assume almost constant from one step to the next. We aim for a step size h_{new} such that

$$\text{Tol} \approx Dh_{\text{new}}^{p+1}.$$

From these two approximations we get:

$$\frac{\text{Tol}}{\|\mathbf{le}_{n+1}\|} \approx \left(\frac{h_{\text{new}}}{h_n} \right)^{p+1} \Rightarrow h_{\text{new}} \approx \left(\frac{\text{Tol}}{\|\mathbf{le}_{n+1}\|} \right)^{\frac{1}{p+1}} h_n.$$

To avoid too many rejected steps, it is wise to be a bit conservative when choosing the new step size, so the following is used in practice:

$$h_{\text{new}} = P \cdot \left(\frac{\text{Tol}}{\|\mathbf{le}_{n+1}\|} \right)^{\frac{1}{p+1}} h_n.$$

where the *pessimist factor* $P < 1$ is some constant, normally chosen between 0.5 and 0.95.

We have all the bits and pieces for constructing an adaptive ODE solver based on Euler's and Heun's methods. There are still some practical aspects to consider:

- The combination of the two methods can be written as

$$\begin{aligned} \mathbf{k}_1 &= \mathbf{f}(t_n, \mathbf{y}_n), \\ \mathbf{k}_2 &= \mathbf{f}(t_n + h, \mathbf{y}_n + h\mathbf{k}_1), \\ \mathbf{y}_{n+1} &= \mathbf{y}_n + h\mathbf{k}_1, & \text{Euler} \\ \hat{\mathbf{y}}_{n+1} &= \mathbf{y}_n + \frac{h}{2}(\mathbf{k}_1 + \mathbf{k}_2), & \text{Heun} \\ \mathbf{le}_{n+1} &= \|\hat{\mathbf{y}}_{n+1} - \mathbf{y}_{n+1}\| = \frac{h}{2}\|\mathbf{k}_2 - \mathbf{k}_1\|. \end{aligned}$$

- Even if the error estimate is derived for the lower order method, in this case Euler's method, it is common to advance the solution with the higher order method, since the additional accuracy is for free. This is usually referred to as *local extrapolation*.
- Adjust the last step to be able to terminate the solutions exactly in t_{end} .
- To avoid infinite loops, add some stopping criteria. In the code below, there is a maximum number of allowed steps (rejected or accepted).

3.5 A summary of some terms and definitions

There have been quite a few definitions and different error terms in this note. So let us list some of them (not exclusive):

Definitions:

$\mathbf{y}' = \mathbf{f}(t, \mathbf{y})$	the ODE
$\mathbf{y}(t; t^*, \mathbf{y}^*)$,	the exact solution of the ODE through $(t^*, \mathbf{y}^*)$
$\mathbf{y}(t) = \mathbf{y}(t; t_0, \mathbf{y}_0)$,	the exact solution of $\mathbf{y}' = \mathbf{f}(t, \mathbf{y})$, $\mathbf{y}(t_0) = \mathbf{y}_0$
$\mathbf{y}_{n+1} = \mathbf{y}_n + h\Phi(t_n, \mathbf{y}_n; h)$,	one step of the method

Let Φ represent a method of order p and $\hat{\Phi}$ a method of order $p + 1$.

Error concepts:

$$\mathbf{d}_{n+1} = \mathbf{y}(t_n + h; t_n, \mathbf{y}(t_n)) - \left(\mathbf{y}(t_n) + h\Phi(t_n, \mathbf{y}(t_n); h) \right), \quad \text{the local truncation error}$$

$$\mathbf{l}_{n+1} = \mathbf{y}(t_n + h; t_n, \mathbf{y}_n) - \left(\mathbf{y}_n + h\Phi(t_n, \mathbf{y}_n; h) \right), \quad \text{the local error}$$

$$\mathbf{le}_{n+1} = h \left(\hat{\Phi}(t_n, \mathbf{y}_n; h) - \Phi(t_n, \mathbf{y}_n; h) \right), \quad \text{the local error estimate, } \mathbf{le}_{n+1} \approx \mathbf{l}_{n+1}$$

$$\mathbf{e}_n = \mathbf{y}(t_n) - \mathbf{y}_n \quad \text{the global error}$$

3.6 Stiff ODEs

Let us start this section by an example:

Example 3.6.1. Given the following system of two ODEs

$$\begin{aligned} y_1' &= -2y_1 + y_2 + 2\sin(t), & y_1(0) &= 2, \\ y_2' &= (a-1)y_1 - ay_2 + a(\cos(t) - \sin(t)), & y_2(0) &= 3, \end{aligned}$$

where a is some positive parameter. The exact solution, which is independent of the parameter, is

$$y_1(t) = 2e^{-t} + \sin(t), \quad y_2(t) = 2e^{-t} + \cos(t).$$

Solve this problem now with some adaptive ODE solver, for instance the Heun-Euler scheme.

Now try the tolerances $\text{Tol} = 10^{-2}, 10^{-4}, 10^{-6}$, and perform the experiment with two different values of the parameters, $a = 2$ and $a = 999$. For $a = 2$ the expected behaviour is observed, for higher accuracy, more steps are required. But the example $a = 999$ requires much more steps, and the step size seems almost independent of the tolerance, at least for $\text{Tol} = 10^{-2}, 10^{-4}$.

The example above with $a = 999$ is a typically example of a *stiff ODE*. What defines these types of ODEs is that there are (at least) two different time scales at play at the same time: a slow time scale that dominates the time evolution of the solution of the ODE, and a fast time scale at which small perturbations of the solution may occur. In physical systems, this might be due to very strong damping of selected components of the system.

Let us study example 3.6.1 a bit closer: The general solution of the ODE can be shown to be

$$\mathbf{y}(t) = c_1 \begin{pmatrix} 1 \\ 1 \end{pmatrix} e^{-t} + c_2 \begin{pmatrix} -1 \\ a-1 \end{pmatrix} e^{-(a+1)t} + \begin{pmatrix} \sin(t) \\ \cos(t) \end{pmatrix}$$

for some constants c_1, c_2 . The terms e^{-t} , $\sin(t)$, and $\cos(t)$ evolve at a time scale of order 1. In contrast, the term $e^{-(a+1)t}$ reverts back to being essentially equal to zero at a time scale of order $1/(a+1)$.

When a stiff ODE is solved by some explicit adaptive method like the Heun-Euler scheme, an unreasonably large number of steps is required, and this number seems independent of the tolerance. The problem is that, for explicit methods, the local error is dominated by

what is happening at the fast time scale, and the step length will be adapted to that time scale as well. Even worse, any larger step size will lead to instabilities and exponentially increasing oscillations in the numerical solution.

In the remaining part of this note we will explain why this happens, and how we can overcome the problem. For simplicity, the discussion is restricted to linear problems, but also nonlinear ODEs can be stiff, and often will be.

Exercise 3.6.1. Repeat the experiment on the Van der Pol equation

$$\begin{aligned} y_1' &= y_2, & y_1(0) &= 2, \\ y_2' &= \mu(1 - y_1^2)y_2 - y_1, & y_2(0) &= 0. \end{aligned}$$

Use $\mu = 2$, $\mu = 5$ and $\mu = 50$.

3.6.1 Linear stability analysis

The phenomenon observed above can be analyzed by studying the very simple linear test equation

$$y' = \lambda y, \quad y(0) = y_0,$$

where the parameter $\lambda \in \mathbb{C}$ satisfies

$$\Re \lambda < 0.$$

and represent the fastest decaying component in a system. A more extensive explanation will be given below. Here, and in the following, $\Re \lambda$ will denote the real part of λ , and $\Im \lambda$ will denote the imaginary part of λ .

The analytic solution of this problem is

$$y(x) = y_0 e^{\lambda x} = y_0 e^{\Re \lambda x} (\cos(\Im \lambda x) + i \sin(\Im \lambda x)).$$

Since $\Re \lambda < 0$, the solution $y(x)$ tends to zero as $x \rightarrow \infty$. We want a similar behaviour for the numerical solution, that is $|y_n| \rightarrow 0$ when $n \rightarrow \infty$.

One step of some Runge–Kutta method applied to the linear test equation can always be written as

$$y_{n+1} = R(z)y_n, \quad z = \lambda h.$$

The function $R(z): \mathbb{C} \rightarrow \mathbb{C}$ is called the *stability function* of the method. $R(z)$ is either a polynomial or a rational function of z .

Example 3.6.2. The application of Euler's method for the linear test equation results in the sequence

$$y_{n+1} = y_n + h\lambda y_n = (1 + h\lambda)y_n = (1 + z)y_n \quad \text{with } z = h\lambda.$$

The stability function of Euler's method is therefore the function

$$R(z) = 1 + z.$$

For a comparison, Heun's method for this test equation is

$$\begin{aligned} k_1 &= \lambda y_n, \\ k_2 &= \lambda(y_n + hk_1), \\ y_{n+1} &= y_n + \frac{h}{2}(k_1 + k_2), \end{aligned}$$

which can be rewritten as

$$y_{n+1} = y_n + \frac{h}{2}(\lambda y_n + \lambda(y_n + h\lambda y_n)) = y_n + h\lambda y_n + \frac{(h\lambda)^2}{2}y_n.$$

As a consequence, the stability function for Heun's method is

$$R(z) = 1 + z + \frac{z^2}{2}.$$

One step with the Trapezoidal rule is

$$y_{n+1} = y_n + h\frac{1}{2}(\lambda y_n + \lambda y_{n+1})$$

and the corresponding stability function is

$$R(z) = \frac{1 + \frac{1}{2}z}{1 - \frac{1}{2}z}$$

We now return back to the analysis of the behaviour of an arbitrary Runge-Kutta method with stability function R . Taking the absolute value on each side of the expression

$$y_{n+1} = R(z)y_n,$$

we see that there are three possible outcomes:

$$\begin{array}{llll} |R(z)| < 1 & \Rightarrow & |y_{n+1}| < |y_n| & \Rightarrow & y_n \rightarrow 0 & \text{(stable)} \\ |R(z)| = 1 & \Rightarrow & |y_{n+1}| = |y_n| & & & \\ |R(z)| > 1 & \Rightarrow & |y_{n+1}| > |y_n| & \Rightarrow & |y_n| \rightarrow \infty & \text{(unstable)} \end{array}$$

The *stability region* of a method is defined by

$$\mathcal{S} = \{z \in \mathbb{C} : |R(z)| \leq 1\}.$$

To get a stable numerical solution, we have to choose the step size h such that $z = \lambda h \in \mathcal{S}$.

Example 3.6.3. In the case of Euler's method, we have obtained the stability function

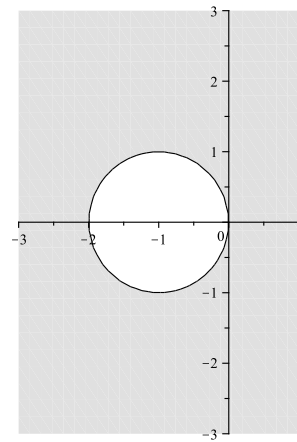
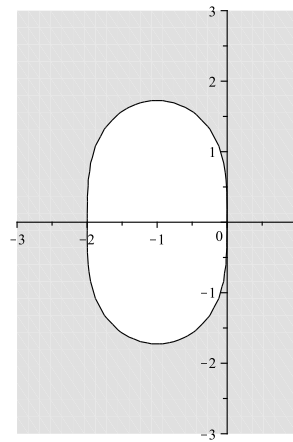
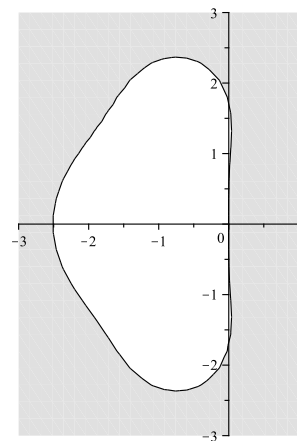
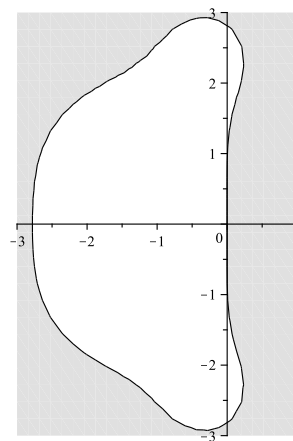
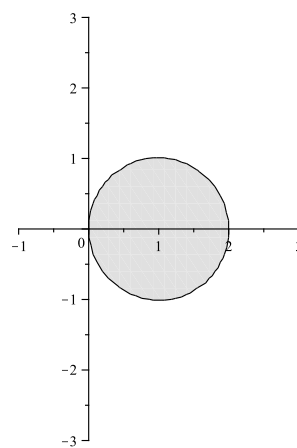
$$R(z) = 1 + z.$$

As a consequence, the stability region for Euler's method is

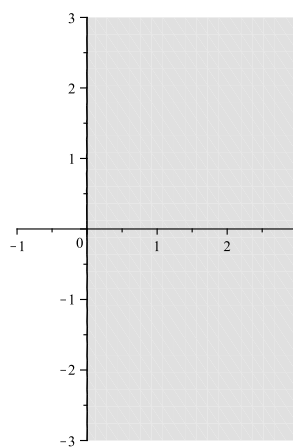
$$\mathcal{S} = \{z \in \mathbb{C} : |1 + z| \leq 1\}.$$

This is a ball in the complex plane, which is centered at -1 and has a radius of 1.

In Fig. 3.2 the stability regions for some common methods are presented.

 $p = s = 1$  $p = s = 2$  $p = s = 3$  $p = s = 4$ 

Backward Euler



Trapezoidal rule

Figure 3.2: Stability regions in $\mathbb{C}^-$: The first four are the stability regions for explicit RK methods of order $p = s$. The white regions are stable, the grey unstable.

3.6.2 Linear systems of ODEs.

We are given a system of m differential equation of the form

$$\mathbf{y}' = A\mathbf{y} + \mathbf{g}(x). \quad (*)$$

Such systems have been discussed in Mathematics 3, and the technique for finding the exact solution will shortly be repeated here:

Solve the homogenous system $\mathbf{y}' = A\mathbf{y}$, that is, find the eigenvalues λ_i and the corresponding eigenvectors $\mathbf{v}_i$ satisfying

$$A\mathbf{v}_i = \lambda_i\mathbf{v}_i, \quad i = 1, 2, \dots, m. \quad (**)$$

Assume that A has a full set of linearly independent (complex) eigenvectors $\mathbf{v}_i$ with corresponding (complex) eigenvalues λ_i . Let $V = [\mathbf{v}_1, \dots, \mathbf{v}_m]$, and $\Lambda = \text{diag}\{\lambda_1, \dots, \lambda_m\}$. Then V is invertible and

$$AV = V\Lambda \quad \text{and therefore} \quad V^{-1}AV = \Lambda.$$

The ODE $(*)$ can thus be rewritten as

$$V^{-1}\mathbf{y}' = V^{-1}AVV^{-1}\mathbf{y} + V^{-1}\mathbf{g}(t).$$

Let $\mathbf{z} = V^{-1}\mathbf{y}$ and $\mathbf{q}(t) = V^{-1}\mathbf{g}(t)$ such that the equation can be decoupled into a set of independent scalar differential equations

$$\mathbf{z}' = \Lambda\mathbf{z} + \mathbf{q}(t) \quad \text{or, equivalently} \quad z'_i = \lambda_i z_i + q_i(t), \quad i = 1, \dots, m.$$

The solution of such equations has been discussed in Mathematics 1. When these solutions are found, the exact solution is given by

$$\mathbf{y}(t) = V\mathbf{z}(t),$$

and possible integration constants are given by the initial values.

As it turns out, the eigenvalues $\lambda_i \in \mathbb{C}$ are the key to understanding the behaviour of the ODE integrators, and it motivates the study of the stability properties of the very simplified, though complex, linear test equation

$$y' = \lambda y, \quad \lambda \in \mathbb{C}^-.$$

The discussion below is also relevant for nonlinear ODEs $\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t))$, in which case we have to consider the eigenvalues of the Jacobian $\mathbf{f}_{\mathbf{y}}$ of $\mathbf{f}$ with respect to $\mathbf{y}$.

We now return to the introductory example. There, the ODE can be written as

$$\mathbf{y}' = A\mathbf{y} + \mathbf{g}(t),$$

with

$$A = \begin{pmatrix} -2 & 1 \\ a-1 & -a \end{pmatrix}, \quad \mathbf{g}(t) = \begin{pmatrix} \sin(t) \\ a(\cos(t) - \sin(t)) \end{pmatrix}.$$

The eigenvalues of the matrix A are $\lambda_1 = -1$ and $\lambda_2 = -(a+1)$. The general solution is given by

$$\mathbf{y}(t) = c_1 \begin{pmatrix} 1 \\ 1 \end{pmatrix} e^{-t} + c_2 \begin{pmatrix} -1 \\ a-1 \end{pmatrix} e^{-(a+1)t} + \begin{pmatrix} \sin(t) \\ \cos(t) \end{pmatrix}.$$

In the introductory example, the initial values were chosen such that $c_1 = 2$ and $c_2 = 0$. However, for large values of a , the term $e^{-(a+1)t}$ will still go to 0 almost immediately, even if $c_2 \neq 0$. It is this term that creates problems for the numerical solution.

Example 3.6.4. We have already discussed the stability function and stability region for Euler's method in the example above. We now solve the introductory problem

$$\mathbf{y}' = \begin{pmatrix} -2 & 1 \\ a-1 & -a \end{pmatrix} \mathbf{y} + \begin{pmatrix} \sin(t) \\ a(\cos(t) - \sin(t)) \end{pmatrix}, \quad \mathbf{y}(0) = \begin{pmatrix} 2 \\ 3 \end{pmatrix}, \quad a > 0.$$

by Euler's method. We know that the eigenvalues of the matrix A are $\lambda_1 = -1$ and $\lambda_2 = -(1+a)$.

For the numerical solution to be stable for both eigenvalues, we have to require that the step length h satisfies

$$|1 + \lambda_1 h| \leq 1 \quad \text{and} \quad |1 + \lambda_2 h| \leq 1.$$

Since both eigenvalues in this case are real and negative, we see after a short computation that this results in the requirement that

$$h \leq \frac{2}{1+a}.$$

Try $a = 9$ and $a = 999$. Choose step sizes a little bit over and under the stability boundary, and you can experience that the result is sharp. If h is just a tiny bit above, you may have to increase the interval of integration to see the unstable solution.

3.7 A-stable methods.

In an ideal world, we would prefer the stability region to satisfy

$$\mathcal{S} \supset \mathbb{C}^- := \{z \in \mathbb{C} : \Re z \leq 0\},$$

such that the method is stable for all $\lambda \in \mathbb{C}$ with $\Re \lambda \leq 0$ and for all h . Such methods are called *A-stable*. For all explicit methods, like Euler's and Heun's, the stability function will be a polynomial, and $|R(z)| \rightarrow \infty$ as $\Re z \rightarrow -\infty$. Explicit methods can never be *A-stable*, and we therefore have to search among implicit methods. The simplest of those is the implicit, or backward, Euler's method, given by

$$y_{n+1} = y_n + hf(t_{n+1}, y_{n+1}).$$

Applied to the linear test equation $y' = \lambda y$, this results in the update

$$y_{n+1} = y_n + h\lambda y_{n+1} \quad \text{or} \quad y_{n+1} = \frac{1}{1 - h\lambda} y_n.$$

We therefore see that we have the stability function

$$R(z) = \frac{1}{1 - z}.$$

The stability region for the implicit Euler function is thus

$$\mathcal{S} = \left\{ z \in \mathbb{C} : \left| \frac{1}{1-z} \right| \leq 1 \right\} = \{ z \in \mathbb{C} : |1-z| \geq 1 \}.$$

This is the whole complex plane apart from an open ball with center $+1$ and radius 1. Thus the method is A -stable, as every complex number z with $\Re z \leq 0$ is contained in $\mathcal{S}$.

3.7.1 Adaptive methods.

Implicit Euler is a method of order 1, and the trapezoidal rule of order 2. Thus, these can be used for error estimation: Perform one step with each of the methods, use the difference between the solutions as an error estimate, and use the solution from the trapezoidal rule to advance the solution.

Example 3.7.1. Repeat the experiment from the introduction, using a combination of backward Euler and the trapezoidal rule.

We observe that there are no longer any step size restriction because of stability. The algorithm behaves as expected.

Comment: Implicit methods can of course also be applied for nonlinear ODEs. Implicit Euler's method will be

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{f}(x, \mathbf{y}_{n+1}),$$

which is a nonlinear system which has to be solved for each step. Similar for the trapezoidal rule. Usually these equations are solved by Newton's method or some simplification of it.

For an implementation example, see the programming exercise of the exam May 2023.

3.8 Newton's method for nonlinear equations

We start with the scalar equation

$$f(x) = 0.$$

(**NB!** Everything that follows is formulated for an equation of the form $f(x) = 0$; if the equation you have is given in any other way, you first have to rewrite it in the form $f(x) = 0$.)

Our idea is to set up an iterative method for the solution of this equation. That is, we assume that we have already found some x_k that is reasonably close to the solution. We want to find an update Δ_k such that $x_{k+1} := x_k + \Delta_k$ satisfies $f(x_{k+1}) \approx 0$. Using a first order Taylor series expansion, we now obtain that

$$0 \approx f(x_k + \Delta_k) = f(x_k) + \Delta_k f'(x_k) + O(\Delta_k^2) \approx f(x_k) + \Delta_k f'(x_k).$$

Solving this (approximate) equation for the update Δ_k , we see that we should choose it as

$$\Delta_k = -\frac{f(x_k)}{f'(x_k)}.$$

Or, using that $x_{k+1} = x_k + \Delta_k$, we get the well known Newton's method

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}, \quad k = 0, 1, \dots$$

As a stopping criterion, use either $|\Delta_k| < \text{tol}$ or $|f(x_k)| \leq \text{tol}$ or a combination of those.

Example 3.8.1. Solve $f(x) = x^3 + x^2 - 3x - 3 = 0$ by Newton's method. Choose $x_0 = 1.5$ and stop the iterations when $|\Delta_k| < 1.e - 10$.

3.8.1 Newton's method for systems

In this section we will discuss how to solve systems of non-linear equations, given by

$$\begin{aligned} f_1(x_1, x_2, \dots, x_n) &= 0 \\ f_2(x_1, x_2, \dots, x_n) &= 0 \\ &\vdots \\ f_n(x_1, x_2, \dots, x_n) &= 0 \end{aligned}$$

or short by

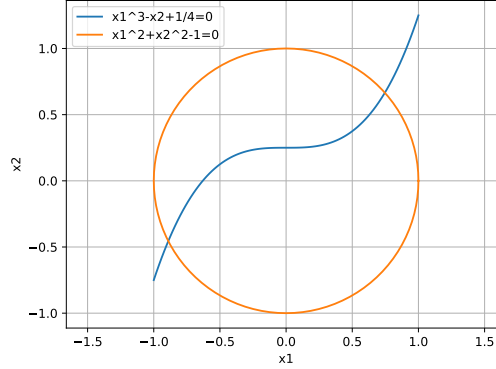
$$\mathbf{f}(\mathbf{x}) = \mathbf{0}.$$

where $\mathbf{f} : \mathbb{R}^m \rightarrow \mathbb{R}^m$.

Example 3.8.2. Given the two equations

$$x_1^3 - x_2 + \frac{1}{4} = 0$$

$$x_1^2 + x_2^2 - 1 = 0$$



The solutions are in the intersections of the two graphs. Thus, there are two solutions, one in the first and one in the third quadrant.

So how to construct an iteration scheme similar to Newton's method for a scalar equation to a system? To avoid completely getting lost in indices, only discuss systems of two equations and two unknowns are discussed here, that is:

$$f(x, y) = 0$$

$$g(x, y) = 0$$

Let $\mathbf{x}^* = [x^*, y^*]^\top$ be an exact solution of these equations and some $\mathbf{x}_k = [x_k, y_k]^\top$ a known approximation to $\mathbf{x}^*$, starting with $\mathbf{x}_0$. We search for a better approximation $\mathbf{x}_{k+1}$.

Let $\delta\mathbf{x}_k = \mathbf{x}^* - \mathbf{x}_k$ be the displacement error of iteration k . Then, by a multidimensional Taylor expansion around the known value $\mathbf{x}_k$ we get

$$0 = f(x^*, y^*) = f(x_k + \delta x_k, y_k + \delta y_k) = f(x_k, y_k) + \frac{\partial f}{\partial x}(x_k, y_k)\delta x_k + \frac{\partial f}{\partial y}(x_k, y_k)\delta y_k + \mathcal{O}(\delta\mathbf{x}_k^2),$$

$$0 = g(x^*, y^*) = g(x_k + \delta x_k, y_k + \delta y_k) = g(x_k, y_k) + \frac{\partial g}{\partial x}(x_k, y_k)\delta x_k + \frac{\partial g}{\partial y}(x_k, y_k)\delta y_k + \mathcal{O}(\delta\mathbf{x}_k^2).$$

where $\mathcal{O}(\delta\mathbf{x}_k^2)$ represents higher order terms, which are assumed to be small compared to the linear terms. By ignoring these terms we get a *linear approximation* to $\mathbf{f}(\mathbf{x})$, and by replacing the exact error $\delta\mathbf{x}_k = \mathbf{x}^* - \mathbf{x}_k$ with its approximation $\Delta\mathbf{x}$, the whole thing can be written as a linear system of equation

$$0 = f(x_k, y_k) + \frac{\partial f}{\partial x}(x_k, y_k)\Delta x_k + \frac{\partial f}{\partial y}(x_k, y_k)\Delta y_k$$

$$0 = g(x_k, y_k) + \frac{\partial g}{\partial x}(x_k, y_k)\Delta x_k + \frac{\partial g}{\partial y}(x_k, y_k)\Delta y_k.$$

or more compact

$$\mathbf{f}(\mathbf{x}_k) + J(\mathbf{x}_k)\Delta\mathbf{x}_k = 0,$$

where the Jacobian $J(\mathbf{x})$ is given by

$$J(\mathbf{x}) = \begin{pmatrix} \frac{\partial f}{\partial x}(x, y) & \frac{\partial f}{\partial y}(x, y) \\ \frac{\partial g}{\partial x}(x, y) & \frac{\partial g}{\partial y}(x, y) \end{pmatrix}$$

It is reasonable to assume that $\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta\mathbf{x}_k$ is a better approximation to $\mathbf{x}^*$ than $\mathbf{x}_k$. Newton's method for systems of equations can then be written as

$$\mathbf{x}_{k+1} = \mathbf{x}_k + J^{-1}(\mathbf{x}_k)\mathbf{f}(\mathbf{x}_k), \quad k = 0, 1, \dots$$

and the implementable algorithm,

- Given $\mathbf{f}(\mathbf{x})$, the Jacobi matrix $J(\mathbf{x})$ and a starting value $\mathbf{x}_0$.
- For $k = 0, 1, 2, 3, \dots$
 - Compute $\mathbf{f}(\mathbf{x}_k)$ og $J(\mathbf{x}_k)$.
 - Solve the linear system $J(\mathbf{x}_k)\Delta\mathbf{x}_k = -\mathbf{f}(\mathbf{x}_k)$ with respect to $\Delta\mathbf{x}_k$.
 - Update: $\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta\mathbf{x}_k$.

Stop the iterations when $\|\mathbf{f}(\mathbf{x}_k)\| \leq \text{Tol}$ or $\|\Delta\mathbf{x}_k\| \leq \text{Tol}$, or a combination of this.

The same algorithm is used for solving a system of m equations, in which case the Jacobian matrix becomes

$$J(\mathbf{x}) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1}(\mathbf{x}) & \frac{\partial f_1}{\partial x_2}(\mathbf{x}) & \dots & \frac{\partial f_1}{\partial x_n}(\mathbf{x}) \\ \frac{\partial f_2}{\partial x_1}(\mathbf{x}) & \frac{\partial f_2}{\partial x_2}(\mathbf{x}) & \dots & \frac{\partial f_2}{\partial x_n}(\mathbf{x}) \\ \vdots & \vdots & & \vdots \\ \frac{\partial f_m}{\partial x_1}(\mathbf{x}) & \frac{\partial f_m}{\partial x_2}(\mathbf{x}) & \dots & \frac{\partial f_m}{\partial x_m}(\mathbf{x}) \end{pmatrix}.$$

A complete convergence analysis is rather hard, and will not be given here. But under certain assumptions it is possible to state

- If the problem has a solution, then Newton's method converges given that the starting values are close enough to the solution.
- But it is often hard to find sufficiently good starting values.

Example 3.8.3. Apply the Newton's method to the problem from Example 3.8.2

$$\begin{aligned} x_1^3 - x_2 + \frac{1}{4} &= 0 \\ x_1^2 + x_2^2 - 1 &= 0 \end{aligned}$$

In this case, the Jacobian is

$$J(\mathbf{x}) = \begin{bmatrix} 3x_1^2 & -1 \\ 2x_1 & 2x_2 \end{bmatrix}$$

Starting with e.g. $\mathbf{x}_0 = [0.6, 0.5]^\top$, the first iteration takes the values

$$\mathbf{f}(\mathbf{x}_0) = \begin{bmatrix} -0.034 \\ -0.390 \end{bmatrix}, \quad J(\mathbf{x}_0) = \begin{bmatrix} 1.08 & -1.0 \\ 1.2 & 1.0 \end{bmatrix}, \quad \Delta\mathbf{x}_0 = \begin{bmatrix} 0.18596491 \\ 0.16684211 \end{bmatrix}, \quad \mathbf{x}_1 = \begin{bmatrix} 0.78596491 \\ 0.66684211 \end{bmatrix}$$

The first iterations are

k	x_1	x_2	$\ \Delta\mathbf{x}_k\ $
0	0.60000000	0.50000000	$0.25 \cdot 10^{-1}$
1	0.78596491	0.66684211	$3.81 \cdot 10^{-2}$
2	0.74787521	0.66493393	$1.74 \cdot 10^{-3}$
3	0.74628412	0.66562980	$2.99 \cdot 10^{-6}$
4	0.74628128	0.66563072	$9.47 \cdot 10^{-12}$
5	0.74628128	0.66563072	$1.28 \cdot 10^{-16}$