1 Nonlinear Equations in One Unknown

1.1 Exploring Equations—A Short Tour

Let us start this course with some standard methods for solving equations in one unknown. Such equations are called *linear* if they can be expressed in the form

$$kx = d, \quad k, d \text{ given}, \quad x \text{ unknown}.$$

If $k \neq 0$, a unique solution exists. Not much else is to say in this simple case, and so, for the time being, we only care about *nonlinear* equations. (Linear equations will concern us more intensively when they appear in masses, as systems with several unknowns).

Analytical versus numerical solutions If algebraic transformations make it possible to write the solution of an equation explicitly in the form $x = \ldots$ (in the above example: $x = d/k$, in general some mathematical expression that uses a finite number of standard operations), one speaks of an *analytical solution.* However, in the vast majority of real-world problems, only *numerical solutions* are possible.

Analytically solvable are, for example, *quadratic* equations, i.e. those that can be expressed as

$$x^2 + px + q = 0 \quad p, q \text{ given}, \quad x \text{ unknown}.$$

You certainly know the quadratic formula

$$x_{1,2} = -\frac{p}{2} \pm \sqrt{\frac{p^2}{4} - q}$$

But it is not enough to write down a solution formula; you must also be sure it will deliver reliable results. In this example, the seemingly trivial solution of a quadratic equation calculated by the above formula can become entirely inaccurate. Let your calculator use it to find the smaller solution of the quadratic equation

$$x^2 - 12345678x + 9 = 0.$$

The exact value, up to sixteen digits, is $x_1 = 7.290\,000\,597\,780\,479 \times 10^{-7}$. Although common calculators work with ten to fourteen digits of precision, they return only the first few digits correctly. The numerically more accurate method first calculates the solution *larger in absolute value*, x_1, using the classical formula and then finds the second solution x_2 with the alternative solution formula

$$x_2 = \frac{q}{x_1}\,.$$

Algebraic versus transcendental equations Linear, quadratic and cubic equations are the simplest examples of *polynomial* equations. A polynomial in a variable x is a sum of powers of x, multiplied by coefficients, i.e. an expression of the form

$$a_n x^n + \cdots + a_2 x^2 + a_1 x + a_0.$$

The highest power occurring is the *order* or *degree* of the polynomial or the equation. Both cubic and fourth-order equations are, in principle, analytically solvable, but the formulas

(Cardano[1]-Tartaglia[2] formula, Ferrari's[3] solution) are so unwieldy that they are hardly used in practice. Numerical methods for such equations are computationally more straightforward and more elegant. They provide approximations that stepwise, with ever-increasing accuracy, approximate the solutions. For equations of order five or higher, no algebraic solution formulas exist anyway.

In 1826, the young Norwegian mathematician Niels Henrik Abel proved the impossibility of solving the general quintic equation by some algebraic formula. Thus, from the fifth degree on, equations cannot (in general) be solved by a finite number of elementary arithmetic operations (i.e., addition, subtraction, multiplication, division, taking roots).

Let us conclude the introduction of the different types: Equations involving also fractions, roots or rational exponents can be transformed (but maybe only by cumbersome manipulations) into systems of polynomial equations. An equation containing functions that cannot be formulated through a finite number of elementary arithmetic operations is something that exceeds the powers of algebra; *"quod vires algebrae transcendit"*, said Leibnitz) and is therefore called *transcendental* . For example, the trigonometric functions, the exponential function and the corresponding inverse functions are transcendental functions. If an equation involves algebraic and transzendental terms, normally only numerical methods can solve it.

Explicit solution formulas exist for low-order polynomial and very simple transzendental equations only. In all other cases, only numerical methods can find solutions.

1.2 Definitions, problems, solutions

Types of problems covered here:

$$\begin{array}{rcll} g(x) & = & h(x), & \text{finding a } \textit{solution} \text{ of an equation} \\ f(x) & = & 0, & \text{finding a } \textit{zero} \text{ of the function } f \\ x & = & \phi(x), & \text{finding a } \textit{fixed point} \text{ of the function } \phi \end{array}$$

A *zero* of the function f is a solution of the equation $f(x) = 0$.
A *fixed point* of the function ϕ is a solution of the equation $x = \phi(x)$.

Of course, any equation in fixed-point form $x = \phi(x)$ can be transormed into $\phi(x) - x = 0$. Thus, any fixed point of ϕ is also a zero of $f(x) = \phi(x) - x$.
Moreover, the names f and ϕ are not reserved for problems involving zeros or fixed points, respectively. These notes, however, usually write $x = \phi(x)$ for an equation resulting from some transformation of $f(x) = 0$.

An *analytical solution* , also called a *closed-form expression* , is an explicit expression involving only well-known quantities and functions. In contrast, a *numerical solution* repeatedly uses a set of calculations to improve a known approximation step by step.

Which functions are assumed to be "well-known" is not exactly defined. Trigonometric functions like sine or cosine definitely count as well-known, but even these can be evaluated by numerical methods only. (You just don't notice, because your calculator does this work for you.)

[1]Girolamo Cardano is also known for the Cardan shaft and the gimbal suspension, which he did not invent either.

[2]Niccolò Fontana Tartaglia, revealed the solution to Cardano under the promise to keep it secret; was extremely upset when Cardano published the formula anyway.

[3]Lodovico Ferrari was mainly responsible for the solution of quartic equations that Cardano published. (The Formula Four, so to say, was won by Ferrari that year.)

Multiple zeros : A function f has at x a root of multiplicit n, if $f(x) = 0, f'(x) = 0, f''(x) = 0, \ldots, f^{(n-1)}(x) = 0$ and $f^{(n)}(x) \neq 0$ (assuming continuous derivatives up to order n exist).

In this lecture, the functions $f, g, \ldots$ and variables $x, y, \ldots$ denote *real* quantities. The *complex* numbers, however, are the natural environment for polynomials and functions (among other things because there polynomials of degree n always have exactly n zeros; multiple zeros are possible, the fundamental theorem of algebra tells how to count correctly). Most definitions and methods can be easily generalized for complex variables and complex-valued functions. Nevertheless, we restrict ourselves (apart from occasional hints) to computational procedures in the real numbers.

Checklist for solving nonlinear equations

Serves also as table of contents and review for the following sections.

- Preliminary work
  - Examine the shape of your functions (table of values, graphical representation).
  - Domain of the functions? Where may a solution lie? How many solutions can exist?
  - Can you find suitable transformations?
- Basic methods using computer or pocket calculator
  - Systematically compute a table of values
  - Plot the function and zoom in
- Standard methods
  - Interval bisection
  - Secant method and Regula Falsi
  - Newton's method (also known as the Newton-Raphson method)
  - Fixed point iteration

1.3 Warm-Up Examples

The exercises and the lecture discuss examples of the following type. Also the next Sections 1.5 and 1.6 present some further explanations.

From financial mathematics

A loan of €100 000 will be repaid in 180 monthly installments of €900 each. What is the interest rate on these terms?

The annuity formula for payment in arrears yields for the (monthly) compounding factor q the equation

$$900 = 100\,000 \frac{q-1}{1-q^{-180}}. \tag{1}$$

Equation of state of a real gas

What is the molar volume of nitrogen at 20 C and 1 bar = 10^5 Pa according to the Van der Waals equation?

The equation of state

$$\left(p + \frac{a}{V_{mol}^2}\right)(V_{mol} - b) = RT$$

describes the relationship between pressure p, molar volume V_{mol} and Temperature T. For nitrogen, the constants a and b are

$$a = 0.129\,\mathrm{Pa\,m^6/mol^2}, \quad b = 38.6 \times 10^{-6}\,\mathrm{m^3/mol}.$$

The molar gas constant is $R = 8.3145$ J/molK. Inserting all numerical values leaves an equation for V_{mol},

$$\left(100\,000 + \frac{0.129}{V_{mol}^2}\right)(V_{mol} - 0.000\,038\,6) = 2437.4 \tag{2}$$

Friction losses in pipe flow

The friction factor f depends on the Reynolds number Re. For laminar flow, the simple rule is $f = 64/\mathrm{Re}$. In the turbulent range, from about Re > 2000, technical manuals list different, partly empirical formulas for f. For a smooth pipe, PRANDTL found the relation

$$f = \frac{1}{(2\log_{10}(\mathrm{Re}\sqrt{f}) - 0.8)^2}\,, \tag{3}$$

which agrees with experiments up to $\mathrm{Re} = 3.4 \times 10^6$. What is the value of f at $\mathrm{Re} = 1 \times 10^6$?

No deeper meaning

It is good if the examples so far have given you the impression of certain practicality. However, the technical background of the equations and the related difficulties in understanding them may obscure the view of the mathematical contents. These notes do not intend to teach physics but numerical methods, which are easier to illustrate with simple examples.

Therefore: Find the solutions to the equation

$$3\cos x = \log x \tag{4}$$

Important note: here log, of course, means the natural logarithm[4]. Arguments in trigonometric functions are always in radians!

[4]There are hardly any arguments in favor of the decadic logarithm, except for the evolutionary coincidence that humans have ten fingers. For people who cannot count to three, the base $e = 2.718\,281\,8\ldots$ is more natural anyway.

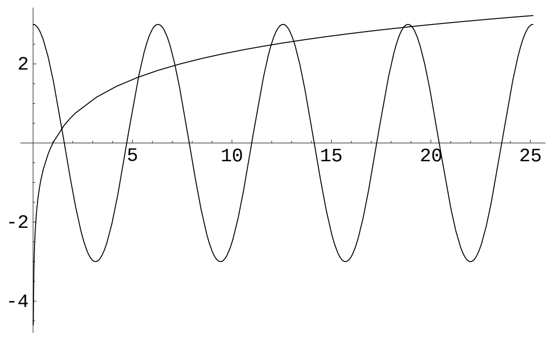


Figure 1: Diagram for the equation $3\cos x = \log x$. The x-values at the intersections of the graphs correspond to the solutions of the equation.

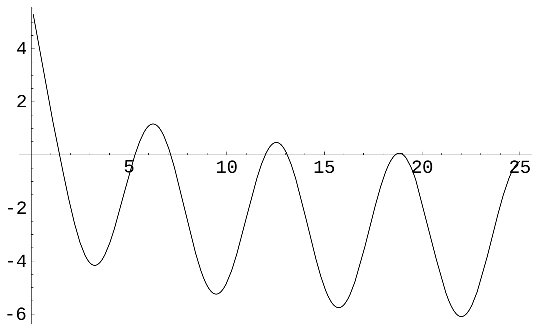


Figure 2: Graph of the function $f(x) = 3\cos x - \log x$. The zeros of f correspond to the x-values at the intersections in Figure 1

1.4 Graphical solution: A picture says more than a thousand formulas

According to the checklist from chapter 1.2, we use the example of Equation 4 to get a first overview. Looking at the equation, it is not immediately evident if, where and how many solutions may exist. Since cosine and logarithm are well-known functions, a graphical representation is helpful. (Figure 1). The graph immediately shows the number and approximate position of the solutions. Computational environments can easily calculate a table of values or zoom into the function graph. This way, they quickly provide suitable values. (the checklist calls these procedures "basic methods using computer or pocket calculator").

1.5 Suitable transformations; zeroes and fixed points

The solutions of the equation $3\cos x = \log x$ are exactly the zeros of the function $f(x) = 3\cos x - \log x$. A comparison of Figure 1 with Figure 2 clarifies this fact and shows, for example, that in the vicinity of $x = 5$, at any rate in the range $4 < x < 6$, one of the zeros of f must lie.

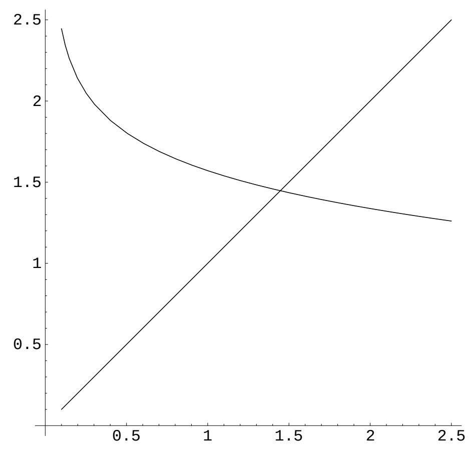


Figure 3: The function $\phi(x) = \arccos((\log x)/3)$ with one fixed point. It corresponds to the zero of f near 1.4. Additional fixed points of ϕ do not exist. Due to the reformulation, all other solutions to the original equation have been lost!

Which form of graphical representation is best chosen depends on the given equation. In this example, the well-known functions cos and log can be quickly sketched. Therefore, the visualization of the solution by the (x-values of) intersections is clear. On the other hand, the representation of $f(x) = 3\cos x - \log x$ lets recognize the immediately. The classical methods for finding zeros from Chapter 1.7 require such a transformation of the equation anyway.

It is, for example, also possible to formulate the equation $3\cos x = \log x$ as

$$x = \arccos \frac{\log x}{3} \quad . \tag{5}$$

In this form, it is a fixed-point problem $x = \phi(x)$, with $\phi(x) = \arccos((\log x)/3)$.

Fixed-point iteration

What happens if you insert a value for x on the right-hand side of Equation 5, evaluate the term, and substitute the result back into the right-hand side? Starting, for example, with $x = 1$, this procedure yields the sequence

$$1; \quad 1.5708; \quad 1.419\,69; \quad 1.453\,72; \quad 1.445\,76; \quad 1.447\,61; \quad 1.447\,18\ldots$$

The sequence converges to $\xi = 1.447\,258\,6$, which is the smallest solution of the given equation and at the same time, the only fixed point of the function

$$\phi(x) = \arccos \frac{\log x}{3}.$$

You can see here an example of a *fixed-point iteration* .

Fixed-point iteration
Given an equation $x = \phi(x)$.

Start with initial value
Insert value on right-hand side and evaluate
Repeat inserting and evaluating until values no longer change

More examples of fixed-point iterations:

- Enter a number into the calculator and press the root key repeatedly. The results converge to 1 (fixed point of $f(x) = \sqrt{x}$).
- Enter a number < 20 into the calculator and press the exp and $1/x$ keys alternately several times. The results (after the $1/x$ step) converge toward 0.567 14 (fixed point of $f(x) = 1/\exp x$).
- Calculation of square roots was an important problem already in Greek antiquity and (for rational numbers) solved. The nonlinear equation defining the square root of a is $x^2 = a$. For $x \neq 0$, this is equivalent to

  $$x = \frac{1}{2}\left(x + \frac{a}{x}\right) .$$

  Already the Babylonians are said to have used the iteration (often called Heron's method)

  $$x^{(0)} = a; \quad x^{(k+1)} = \frac{1}{2}\left(x^{(k)} + \frac{a}{x^{(k)}}\right) \text{ for } k = 0, 1, 2, \ldots$$

- Equation 3 is a fixed-point equation. With the initial value of 0.05 few fixed-point iterations provide an accurate solution.

But it doesn't always work: another possible fixed-point form of Equation 4 is

$$x = \exp(3\cos x) .$$

If you substitute here $x = 1$ on the right, evaluate and iterate, you get the sequence

$$1; \quad 5.057\,68; \quad 2.760\,46; \quad 0.061\,745\,5; \quad 19.971; \quad 3.6805\ldots$$

These values change irregularly and do not converge.

Summary

Not every fixed-point iteration converges. Suitable transformations are not always easy to find. On the other hand, many numerical methods are of the fixed-point iteration type. This justifies a detailed theoretical investigation of such methods in Chapter 1.12.

1.6 Discussion of the Examples: Important and Unimportant Terms

Here we discuss in detail the examples presented in Chapter 1.1

1.6.1 A Nearly Linear Equation

The equation mentioned at the beginning of Chapter 1.1,

$$x^2 - 12345678x + 9 = 0 ,$$

is, when it comes to the smaller of the two solutions, actually not a quadratic equation! Reason: The sought solution is of order 10^{-6} bis 10^{-7}; the term x^2 in the equation is smaller than the linear term $12345678x$ by more than ten orders of magnitude. For all practical purposes, such an equation is just a linear one with a small quadratic correction term. Therefore, solve for the linear term:

$$x = \frac{1}{12345678}(x^2 + 9) \ .$$

The starting value$x^{(0)} = 0$ already gives, even on the cheapest calculators without a root key, a better approximation $x^{(1)} = 7.290\,000\,597\,78 \times 10^{-7}$ than most calculators can achieve by using the standard solution formula.

Loosely speaking, many equations contain terms in which the unknown has little influence compared to other terms. If an equation becomes more manageable this way, you may neglect those terms in a first approximation. In the following steps, you correct the result, using the approximate values for the unimportant terms.

1.6.2 Van der Waals Equation

You may transform the Equation (2) to a cubic equation,

$$-4.9794 \cdot 10^{-6} + 0.129V_{mol} - 2441.3V_{mol}^2 + 100000V_{mol}^3 = 0 \quad , \tag{6}$$

which would be, in principle, analytically solvable. Please don't do it! A little insight into the physical background of this equation suggests a different procedure: At room temperature, nitrogen is almost an ideal gas, which obeys the equation

$$pV_{mol} = RT \ .$$

In the Van der Waals equation

$$\left(p + \frac{a}{V_{mol}^2}\right)(V_{mol} - b) = RT \ , \tag{7}$$

the term a/V_{mol}^2 is a correction of the ideal gas equation and is, compared to p, negligibly small for the given data. Even if you don't see it in the polynomial (6): the original Equation (7) does not stand for a "real" cubic equation but a linear equation in V_{mol} plus a small correction term a/V_{mol}^2. You can solve this equation if you move the "unimportant" terms to the right side. Here we reshape to

$$V_{mol} = \frac{RT}{p + a/V_{mol}^2} + b = \frac{2437.4}{100000 + 0.129/V_{mol}^2} + 0.000\,038\,6 \ .$$

Let's ignore, for the moment, the correction term a/V_{mol}^2. We get a zero approximation for the molar volume,

$$V_0 = \frac{2437.4}{100000} + 0.000\,038\,6 = 0.024\,413 \ .$$

Now, the trick is to insert this approximation for V_{mol} on the right-hand side of the equation. It produces an improved approximation

$$V_1 = \frac{2437.4}{100000 + 0.129/0.024\,413^2} + 0.000\,038\,6 = 0.024\,360 \ .$$

Repeated insertion does not yield any further improvement:

$$V_2 = \frac{2437.4}{100000 + 0.129/0.024\,360^2} + 0.000\,038\,6 = 0.024\,360 \ .$$

Thus we have calculated (at least to five decimal places) the value $V_{mol} = 0.024\,360\,\mathrm{m}^3$.

Penitential exercise for Lent: Look up the Cardan formulae in Wikipedia and solve the problem this way. Compare the time required with the method above.

1.6.3 Financial Mathematics

In equation 1, we expect the compounding factor q to be just above 1. The term q^{-180} in the denominator is likely to be $\ll 1$ and unimportant. Thus, we solve the equation for the q in the numerator.

$$q = 1 + \frac{900}{100000}(1 - q^{-180})$$

If we ignore q^{-180} on the right side, then the zero approximation is

$$q_0 = 1 + \frac{900}{100000} = 1.009$$

Again, the trick works to insert q_0 on the right-hand side and gets us an improved approximation

$$q_1 = 1 + \frac{900}{100000}(1 - 1.009^{-180}) = 1.007\,206\ .$$

Repeated insertion yields

$$q_2 = 1.006\,529 \quad q_3 = 1.006\,210 \quad q_4 = 1.006\,047 \ldots$$

However, it takes a total of 14 iterations here for the values to stabilize at 1.005 851.

Concluding Remarks

If an equation is given in the form $f(x) = g(x)$ (example: Equation 4), it is not immediately recognizable which terms are “important” or “unimportant”. Rule: ignore the unknown on that side of the equation with the *less steep* function graph at the intersection.

Suitable transformations for fixed point iterations often require a deeper understanding of the individual terms in an equation. Fortunately, black-box type solution methods exist. The next chapter presents one of them.

1.7 Bisection

Do you know the story of the two possibilities? It begins with the intermediate value theorem.

Intermediate Value Theorem

A function f that is continuous on a closed interval $[a, b]$ takes on any give value between $f(a)$ and $f(b)$ somewhere inside the interval.

In particular, if f is negative for $x = a$ and positive for $x = b$ (or vice versa), then the intermediate value theorem guarantees that f has at least one zero in this interval.

There are always two possibilities...

Suppose we are looking for a zero of a function continuous in the range $a \leq x \leq b$. We can immediately check whether $f(a)$ and $f(b)$ have different signs. If so, then the intermediate value theorem guarantees the existence of a zero in the domain $a \leq x \leq b$, but we do not know where it lies. Now there are two possibilities: Either $b - a$ is already small, in which case it is good: we can take both a and b as approximations for a zero of f. Otherwise, we calculate the midpoint c of the interval, $c = (a + b)/2$. Now there are again two possibilities. If $f(c) = 0$, it is good: We have found a zero there. Otherwise, f has different signs at the ends of one of the subintervals $a \leq x \leq c$ or $c \leq x \leq b$ (got it? That's the point!). So there must be a zero in one of the two intervals. Let's consider this interval and, for simplicity, call its boundaries a and b again.

Now there are two possibilities: Either $b - a$ is small, in which case it is good: we can take both a and b as approximations for a zero of f. Otherwise we form $c = (a + b)/2$. Now there are again two possibilities...

You can now continue the story yourself. But note that the interval length gets halved each time you take the story one step further. For any arbitrarily small given precision $\epsilon > 0$, you reach an interval with length $b - a < \epsilon$ after a finite number of steps. So, the story ends just as in real life: There may always be two choices, but each decision restricts the freedom for further actions. At some point, the alternatives are exhausted.

Written down in formalized form, this procedure is the

Bisection Method
Given a function f, two values a and b with $f(a)\cdot f(b) < 0$, and an error tolerance $\epsilon > 0$. If f is continuous in the interval $a \leq x \leq b$, then this algorithm finds the approximation c to a zero ξ of f with error $|c\xi| < \epsilon$.

```
Repeat
  set $c \leftarrow (a + b)/2$
  if $f(a) \cdot f(c) < 0$
    set $b \leftarrow c$
  else
    set $a \leftarrow c$
until $|b - a| < \epsilon$ or $f(c) = 0$
```

Linear convergence

The best estimate for the zero is the midpoint of the interval. In this case, the error $\epsilon_0 \leq |b - a|/2$ cannot be larger than half the interval width. Interval bisection reduces this error bound by a factor of $1/2$ per step or, since

$$\left(\frac{1}{2}\right)^{3.3} \approx \frac{1}{10} \quad ,$$

by a factor of $1/10$ per (average) 3.3 steps. One can say: interval bisection produces one correct decimal per 3.3 iterations. The maximum error after the i-th step, ϵ_i, is at most half as large as the previous maximum error ϵ_{i-1} . It is thus holds

$$\epsilon_i \leq C\epsilon_{i-1} \quad \text{with } C = \frac{1}{2} \ .$$

In general: If in a procedure the error bounds of successive iteration steps fulfil

$$\epsilon_i \leq C\epsilon_{i-1} \quad \text{mit } C < 1 \; .$$

this behaviour is called *linear* convergence.

Advantages and Disadvantages

Advantages of interval bisection: easy to understand and simple to program. If the assumptions are met, it converges with certainty. It is an *inclusion method*, which means that it not only provides an approximate value but also bounds the solution from both sides.

Disadvantages: One needs initial values—but that is a problem for any numerical method. Interval bisection is slow, it converges only linearly—but that is for sure.

1.8 *Regula Falsi* (false position method)

Functions running smoothly in the vicinity of a zero can be approximated there by a straight line. Instead of choosing, as with interval bisection, the value c exactly in the middle between a and b, we take c as the zero of the straight line through $(a, f(a))$ and $(b, f(b))$, see Figure 4.

$$c = a - f(a)\frac{a-b}{f(a)-f(b)} = \frac{af(b) - bf(a)}{f(b) - f(a)}$$

Regula Falsi **(false position method)**
Given a function f, two values a and b with $f(a) \cdot f(b) < 0$ and an error tolerance $\epsilon > 0$. If $f(x)$ is continuous in the interval $a \leq x \leq b$, then this algorithm[a] finds an approximation c to a zero ξ of f with error $|c - \xi| < \epsilon$.

```
Repeat
  set $c \leftarrow a - f(a)\frac{a-b}{f(a)-f(b)}$
  if $f(b) \cdot f(c) < 0$
    set $a \leftarrow b$
  else
    (standard version) do nothing
    (Illinois version) reduce $f(a)$ to $\frac{1}{2}f(a)$
    (Pegasus version) reduce $f(a)$ to $\frac{f(a)f(b)}{f(b)+f(c)}$
  set $b \leftarrow c$
until $|b - a| < \epsilon$ or $f(c) = 0$
```

[a] however, for the stopping criterion given here, only the non-standard versions

However, compared to simple bisection, the standard version of *regula falsi* will not significantly improve the convergence behavior. Typically, aafter the first few iterations the interval boundary a will remain fixed. The other bound b will converge to the zero, but the stopping criterion $|b-a| < \epsilon$ will never be reached. Therefore, careful programmers would add an emergency exit in the algorithm above: count the number of iterations and abort if they exceed a maximum number.

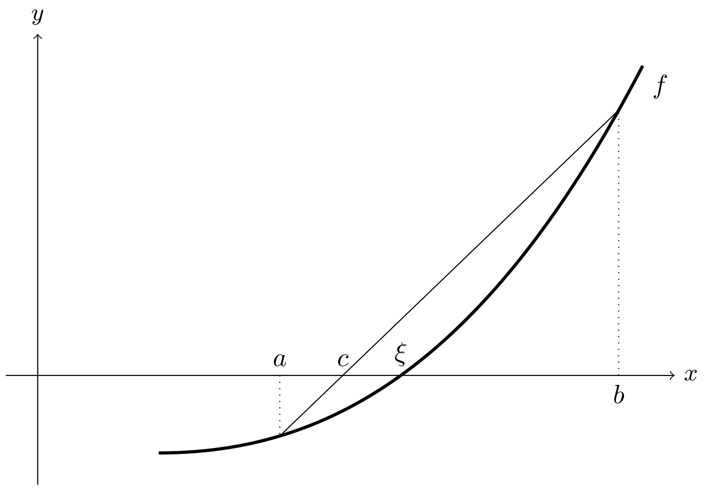


Figure 4: *Regula falsi* computes c, the zero of the connecting staight line, as an approximation to ξ, the zero of f.

The Illinois or Pegasus variants improve the convergence behavior compared to bisection; brave programmers would, in this case, dispense with the query for a maximum number of iterations.

Interval bisection and the various versions of *regula falsi* have in common that they *bracket* the zero from both sides - they are inclusion methods, which is good. The disadvantage is that at the beginning of the method, you need two approximations, one on each side of the zero. Moreover, these methods can only find zeros where the function changes sign. They will not work for multiple zeros of even order.

What is “false” in the *regula falsi*? Not the rule itself, of course, just the assumed starting values a and b. From these two “false solutions,” the rule calculates a better approximate solution.
The method is ancient. Babylonians, Egyptians, Indians, and Chinese used it centuries before Christ to solve linear problems. From Arabic sources, Leonardo of Pisa, known as Fibonacci, brought it to Europe around 1200. He describes among several variants the *regula duarum falsarum positionum*, the “method of the two false positions.” This is what it should be called, but it has been sloppily shortened to *regula falsi*.
Fibonacci solved only linear problems with it; there, the rule calculates the correct solution from two wrong starting values immediately. The application as an iterative method for zeros of non-linear functions is not so old. Small but significant modifications (as in the Illinois or Pegasus variants) have been found around the middle of the last century. Even as recent as 2020, Oliveira und Takahashi proposed a new variant (`https://en.wikipedia.org/wiki/ITP_method`)

1.9 Secant Method

The secant method computes a new approximation by interpolation in the same way as the Regula Falsi, but does not request that the values a and b bracket the zero, see Figure 5.

The formal description of the algorithm here denotes the initial values a and b by $x^{(0)}$ und $x^{(1)}$ and writes $x^{(k)}, x^{(k+1)}, \ldots$ for the iteratively computet approximations.

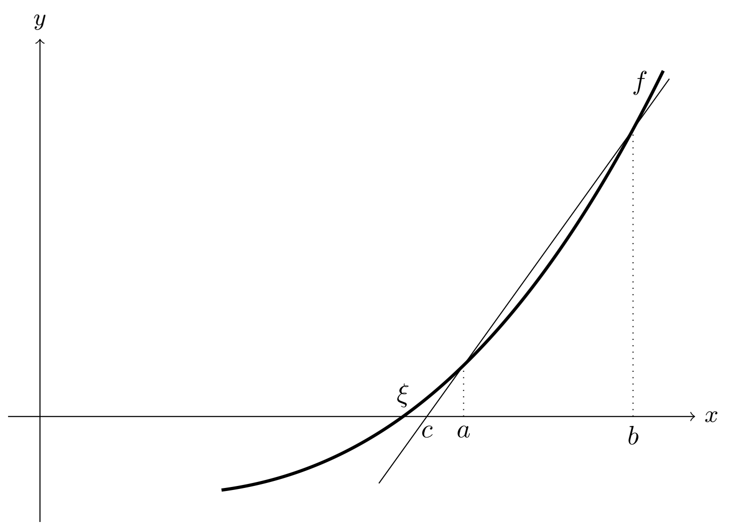


Figure 5: The secant method computes the next approximation c from a straight line (secant) cutting through two points in the graph of f. The values a and b do not necessarily bracket the root ξ.

Secant Method
Given a function f, two values $x^{(0)}$ and $x^{(1)}$, an error tolerance $\epsilon > 0$, and a maximum number of iterations k_{max}. For sufficiently good initial values $x^{(0)}$ and $x^{(1)}$ this algorithm finds the approximation $x^{(k)}$ to a zero ξ of f with accuracy $|x^{(k)} - \xi| \approx \epsilon$ or terminates after a maximum number of k_{max} steps.

```
set $k = 1$
repeat
    set $x^{(k+1)} = x^{(k)} - f(x^{(k)}) \dfrac{x^{(k)} - x^{(k-1)}}{f(x^{(k)}) - f(x^{(k-1)})}$
    increase $k = k + 1$
until $|x^{(k+1)} - x^{(k)}| < \epsilon$ or $k \geq k_{max}$
```

Superlinear Convergence

The secant method shows *superlinear* convergence. (Necessary technical details: f twice continuously differentiable, no multiple zeros). That is, for the error bounds $|x^{(k+1)} - \xi|$ and $|x^{(k)} - \xi|$ of successive steps, provided that $|x^{(k)} - \xi|$ is already sufficiently is sufficiently small:

$$|x^{(k+1)} - \xi| \leq C|x^{(k)} - \xi|^p \quad \text{with } p > 1 \ .$$

Thus, the error is not only reduced by a factor C but additionally by the power p. For the secant method, it can be shown that

$$p = \frac{1 + \sqrt{5}}{2} \approx 1.618 \ .$$

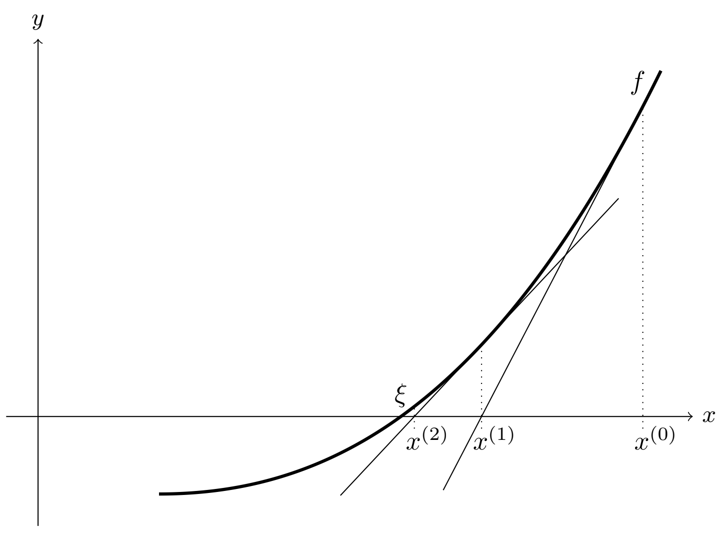


Figure 6: Visualization of Newton's method: The tangent to f in point $(x^{(0)}, f(x^{(0)}))$ intersects the x-axis in $x^{(1)}$. The value $x^{(2)}$ from the next step is alresdy close to the zero ξ.

Assume that $|x^{(k)} - \xi| = 0.01$. Consider which reduces the error more: Multiplying by a factor $C = 1/2$, or exponentiating by $p = 1.6$!

1.10 Newton's method

Also known as the Newton–Raphson method. However, it was Thomas Simpson who, some decades after Isaac Newton and Joseph Raphson, formulated the method as we know it today.

We are looking for a zero of the function f. Let $x^{(0)}$ be a starting value in the vicinity of the zero. Then, the Newton method tries, similar to the secant method, to approximate the function f by a linear function and uses the tangent to f at the point $(x^{(0)}, f(x^{(0)}))$. The point of intersection of the tangent with the x-axis is the next approximation, see Figure 6.

Derivation from the Taylor expansion of f around the point $x^{(0)}$. If f is sufficiently differentiable,

$$f(x) = f(x^{(0)}) + (x - x^{(0)})f'(x^{(0)}) + \frac{(x - x^{(0)})^2}{2!} f''(x^{(0)}) + \dots$$

We want $f(x) = 0$. Neglecting higher-order terms in the expansion results in the equation

$$0 = f(x^{(0)}) + (x - x^{(0)})f'(x^{(0)}) \ ,$$

which we can solve for x,

$$x = x^{(0)} - \frac{f(x^{(0)})}{f'(x^{(0)})} \ .$$

Newton's method
Given a differentiable function f and an initial value $x^{(0)}$.
Wanted: a zero of f.

Iteration

$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})} \qquad \text{for } k = 0, 1, 2 \dots$$

Quadratic convergence

Newton's method exhibits *quadratic* convergence. That means, for the error bounds $\epsilon_{k+1} = |x^{(k+1)} - x|$ and $\epsilon_k = |x^{(k)} - x|$ in successive steps, and if ϵ_k is already small enough,

$$\epsilon_{k+1} \leq C\epsilon_k^2$$

The new error thus is smaller than the *square* of the previous one times factor C.

The specific value of C is not that important. Assume, for example, $\epsilon_k = 10^{-4}$. That would be an error of one unit in the fourth decimal place. Then, in case of quadratic convergence, $\epsilon_{k+1} = C \cdot 10^{-8}$. That would be an error of C units in the eight decimal place. For a value of C around 1, the number of correct decimal digits has doubled.

Quadratic convergence—new error $\sim$ square of old error

Rule of thumb: as soon as a handful of digits are already correct, the next approximation will have about twice as much correct digits.

1.11 Stopping criteria

Computers use only a fixed number of binary digits to store floating point numbers. It is possible that $f(x)$ does not reach the exact value zero for any floating point argument x. If the zero ξ is in the neighborhood of 1, you can easily find an approximation x with absolute error $|x - \xi| < 10^{-6}$. If the zero lies around $\xi \approx 10^{22}$, you will not be able to achieve an absolute error of this quality. A usual choice of the error tolerance ϵ is$\varepsilon_m(|a| + |b|)/2$ if ε_m is the machine precision and a, b are the original interval boundaries. If a, b, and the zero ξ itself are close to zero, you should apply this formula with some caution only. In any case, the termination bound must not be smaller than the smallest positive machine number (typically around 10^{-38} for 4-byte data types, 10^{-308} for 8-byte data types).

Machine precision

The machine precision ε_m is the smallest positive floating point number which, when added to the floating point number 1.0, results in a sum different from 1.0 (typically around $10^{-7}$for 4-byte data types, 10^{-16} for 8-byte data types).

1.12 Fixpunkt-Iteration

In section 1.5, we have already determined fixed points of functions by repeated insertion. Many numerical methods are just special cases of a fixed-point iteration. Therefore statements about the convergence behavior of fixed-point iterations are of general importance.

Fixed-point iteration
Given a function ϕ and an initial value $x^{(0)}$.
Wanted: a fixed point ξ von ϕ.

Iteration
$x^{(k+1)} = \phi(x^{(k)})$ for $k = 0, 1, 2 \dots$

Fixed-point iterations converge for contraction mappings
Let ϕ be a function with fixed point ξ. Let I be an open interval $(\xi - r, \xi + r)$ around the fixed point ξ. If ϕ acts in I as a *contraction mapping*, i. e.,

$$|\phi(x) - \phi(y)| \le C|x - y| \;, \quad C < 1 \text{ for all } x, y \in I \;,$$

then the fixed-point iteration $x^{(k+1)} = \phi(x^{(k)})$ converges for all $x^{(0)} \in I$ at least linearly to ξ.

Proof: First, we show by induction $x^{(k)} \in I$ for all $k = 0, 1, 2 \dots$ The statement is true for $k = 0$.

If, by induction, already $x^{(k)} \in I$, this means $|x^{(k)} - \xi| < r$. We apply the contraction property to $x^{(k)}$ and the fixed point ξ and get

$$|x^{(k+1)} - \xi| = |\phi(x^{(k)}) - \phi(\xi)| \le C|x^{(k)} - \xi| < Cr \;.$$

Since $C < 1$, it also holds that

$$|x^{(k+1)} - \xi| < r \quad \text{and thus,} \quad x^{(k+1)} \in I$$

From this argument, it also follows for the errors $\epsilon^{(k)} = |x^{(k)} - \xi|$ and $\epsilon^{(k+1)} = |x^{(k+1)} - \xi|$:

$$\epsilon^{(k+1)} \le C\epsilon^{(k)} \le C^k \epsilon_0, \text{ and thus, } \epsilon^{(k+1)} \to 0 \text{ for } k \to \infty \;.$$

As formulated here, the theorem already assumes the existence of a fixed point. This condition makes the convergence proof quick and easy. However, a more general formulation and a technically more elaborate argument can prove the existence and uniqueness of a fixed point from the contraction property alone. This version is the famous Banach fixed-point theorem.

Relationship between slope and contraction

The property $|\phi(x) - \phi(y)| \le C|x - y|$ means for $C < 1$: function values differ less than input values. In the limit for small changes, the function's slope determines how much function values change in relation to input values.

If ϕ is continuously differentiable in a neighborhood of ξ and $|\phi'(\xi)| < 1$, the contraction property is satisfied in some neighborhood of ξ: Because of the continuity of ϕ' there is an open interval I around ξ in which $|\phi'| \le C < 1$. For $x, y \in I$, according to the mean value theorem of calculus,

$$\phi(x) - \phi(y) = (x - y)\phi'(\eta) \quad \text{for some } \eta \in I \;.$$

Thus also

$$|\phi(x) - \phi(y)| \leq C|x - y| \;, \quad C < 1$$

A short version of this statement:

The fixed point method converges locally if $|\phi'(\xi)| < 1$.

Figure 7 illustrates the convergence behavior of fixed-point iteration for different functions ϕ.

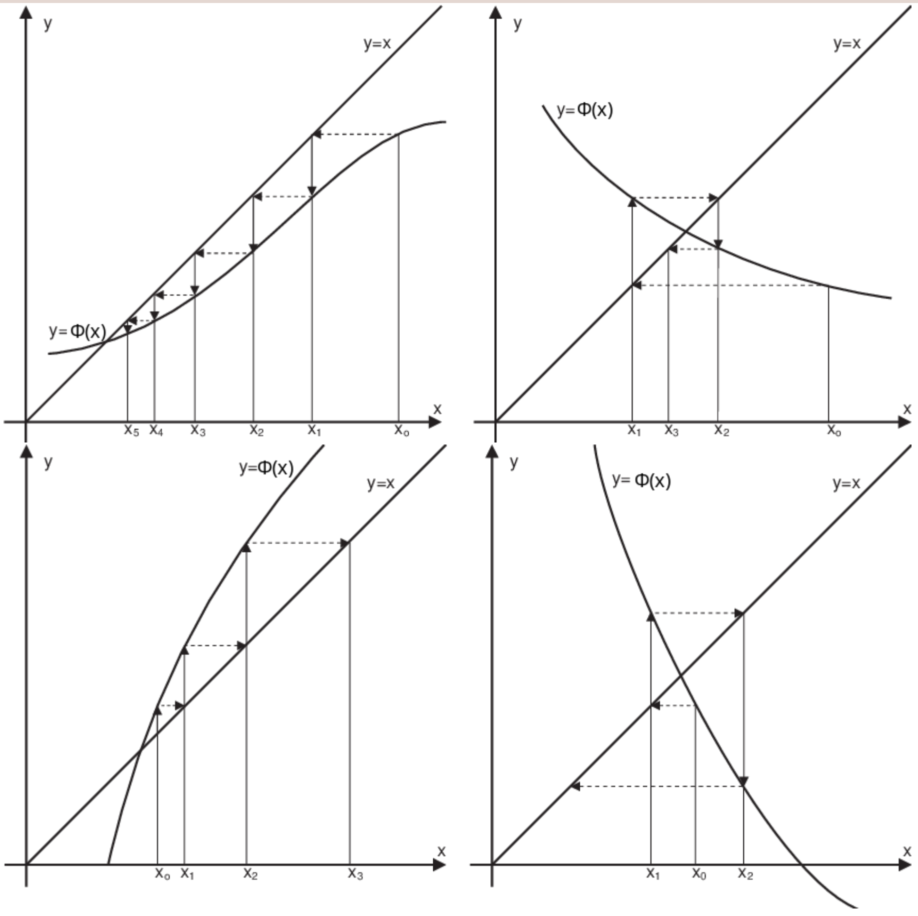


Figure 7: Fixed-point iterations illustrated for different functions ϕ. Possible cases: one-sided approach to the fixed point if $0 < \phi' < 1$ in a neighborhood around the fixed point; alternating convergence if $-1 < \phi' < 0$, divergence if $\phi' > 1$ or $\phi' < -1$.

1.13 Order of Convergence

We have already mentioned linear, superlinear and quadratic convergence. Here, we make the definition more precise.

Order of convergence
Let ξ be a fixed point of ϕ, and for all initial values from an interval around ξ and the corresponding sequence $\{x^{(k)}\}$ generated by the rule $x^{(k+1)} = \phi(x^{(k)}), k = 0, 1, 2 \ldots$ holds

$$|x^{(k+1)} - \xi| \leq C|x^{(k)} - \xi|^p$$

with $p \geq 1$ and $C < 1$ if $p = 1$.
Then the iteration is said to have an order of convergence of (at least) p.

For the local convergence behavior of a fixed-point iteration, the value of the first derivative at the fixed point is decisive. For $|\phi'(\xi)| < 1$, linear convergence is ensured; the smaller the magnitude of the derivative, the faster the method converges. Moreover, $C \approx |\phi'(\xi)|$. However, when $|\phi'(\xi)| = 0$, the convergence behavior becomes superlinear.

Taylor expansion can show : If $\phi(x)$ in a neighborhood of ξ is sufficiently often differentiable and

$$\phi'(\xi) = 0,\ \phi''(\xi) = 0, \ldots, \phi^{(p-1)}(\xi) = 0, \text{ and } \phi^{(p)}(\xi) \neq 0\ ,$$

then for $p = 2, 3, \ldots$ the method is of order p. If $p = 1$, first-order convergence requires the additional condition $|\phi'(\xi)| < 1$.

1.14 Convergence of Newton's Method

Newton's method, applied to the function f, corresponds to a fixed point method for the function ϕ,

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

Now,

$$\phi'(x) = \frac{f''(x)f(x)}{(f'(x))^2}\ ,$$

and since at a simple zero $f(x) = 0, f'(x) \neq 0$, the value $\phi'(x)$ vanishes there. It is easy to check that $\phi''(x) \neq 0$, provided that $f''(x) \neq 0$. From this follows the quadratic convergence of Newton's method for single zeros. For multiple zeros, linear convergence can be proved.

2 Systems of Non-Linear Equations

Section 1.2 defines the terms *solution*, *zero*, and *fixed point* for scalar functions $\mathbb{R} \to \mathbb{R}$. These notations can be easily generalized to vector-valued functions $\mathbb{R}^n \to \mathbb{R}^n$. As in the scalar case, there are different ways to formulate equations.

Notation for vectors and vector-valued functions: Bold

Real-valued functions, scalars:	$f : \mathbb{R} \to \mathbb{R}$,	$y = f(x)$
Vector-valued functions, vectors:	$\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$,	$\mathbf{y} = \mathbf{f}(\mathbf{x})$

Components of a vector $\mathbf{x} \in \mathbb{R}^n$:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad \text{or } \mathbf{x}^T = [x_1, x_2, \ldots, x_n]$$

Normally, $\mathbf{x}$ denotes a column vector while $\mathbf{x}^T$ denotes a row vector.

To count iterations, we set indices (to distinguish them from indices for vector components) as superscripts enclosen in brackets), e. g., $\mathbf{x}^{(k)}, k = 0, 1, 2\ldots$

2.1 Solution, Zero, Fixed Point: the Multi-Dimensional Case

Types of problems for equations in $\mathbb{R}^n$

Let $\mathbf{f}, \mathbf{g}, \mathbf{h}, \mathbf{\Phi}$ be functions $\mathbb{R}^n \to \mathbb{R}^n$ and $\mathbf{x} \in \mathbb{R}^n$
Find an $\mathbf{x}$ that fulfils...

$\mathbf{g}(\mathbf{x}) = \mathbf{h}(\mathbf{x})$,	(Find a *solution* for a system of equations)
$\mathbf{f}(\mathbf{x}) = 0$,	(Find a *zero* of the function $\mathbf{f}$)
$\mathbf{x} = \mathbf{\Phi}(\mathbf{x})$,	(Find a *fixed point* of the function $\mathbf{\Phi}$)

Compared to the definitions of Section 1.2, almost nothing has changed except the typeface.

For example a *nonlinear system of equations* with two unknowns

$$\begin{aligned} 4x_1 - x_2 + x_1x_2 &= 1 \\ -x_1 + 6x_2 &= 2 - \log(x_1x_2) \end{aligned}$$

has the form $\mathbf{g}(\mathbf{x}) = \mathbf{h}(\mathbf{x})$ with

$$\mathbf{g}(\mathbf{x}) = \begin{bmatrix} g_1(x_1, x_2) \\ g_2(x_1, x_2) \end{bmatrix} = \begin{bmatrix} 4x_1 - x_2 + x_1x_2 \\ -x_1 + 6x_2 \end{bmatrix} \quad , \quad \mathbf{h}(\mathbf{x}) = \begin{bmatrix} h_1(x_1, x_2) \\ h_2(x_1, x_2) \end{bmatrix} = \begin{bmatrix} 1 \\ 2 - \log(x_1x_2) \end{bmatrix}$$

This System can be transformed to

$$\begin{aligned} 4x_1 - x_2 + x_1x_2 - 1 &= 0 \\ -x_1 + 6x_2 + \log(x_1x_2) - 2 &= 0 \end{aligned}$$

In this formulation the problem is to find *zeros of the vector-valued function* $\mathbf{f}:\mathbb{R}^2 \to \mathbb{R}^2$, that is, solutions of $\mathbf{f}(\mathbf{x}) = 0$ with

$$\mathbf{f}(\mathbf{x}) = \begin{bmatrix} f_1(x_1,x_2) \\ f_2(x_1,x_2) \end{bmatrix} = \begin{bmatrix} 4x_1 - x_2 + x_1x_2 - 1 \\ -x_1 + 6x_2 + \log(x_1x_2) - 2 \end{bmatrix}$$

Another equivalent transformation would be

$$\begin{aligned} x_1 &= \frac{1}{4}(x_2 - x_1x_2 + 1) \\ x_2 &= \frac{1}{6}(x_1 - \log(x_1x_2) + 2) \end{aligned}$$

Now, *fixed points of the vector-valued function* $\mathbf{\Phi}:\mathbb{R}^2 \to \mathbb{R}^2$ are requested, that is, solutions of $\mathbf{x} = \mathbf{\Phi}(\mathbf{x})$ with

$$\mathbf{\Phi}(\mathbf{x}) = \begin{bmatrix} \phi_1(x_1,x_2) \\ \phi_2(x_1,x_2) \end{bmatrix} = \begin{bmatrix} \frac{1}{4}(x_2 - x_1x_2 + 1) \\ \frac{1}{6}(x_1 - \log(x_1x_2) + 2) \end{bmatrix}$$

One more note on notation: When we have found a particular fixed point, we denote it by $\boldsymbol{\xi}$ in the following to distinguish it from other general $\mathbf{x}$ values.

2.2 Multidimensional Fixed-Point Iterations

Fixed-point iterations are also possible in the multidimensional case. A fixed point of a mapping $\mathbf{\Phi}:\mathbb{R}^n \to \mathbb{R}^n$ is—entirely analogous to the one-dimensional definition— a value $\boldsymbol{\xi} \in \mathbb{R}^n$, for which holds

$$\boldsymbol{\xi} = \mathbf{\Phi}(\boldsymbol{\xi})\ .$$

Just as in the one-dimensional case, fixed-point iteration (if it converges) finds a fixed point. Once again, we write here vectors from the $\mathbb{R}^n$ and vector-valued functions in boldface type $(\mathbf{\Phi}, \boldsymbol{\xi}, \mathbf{x} \ldots)$, to distinguish them from variables and real-valued functions $(\phi, \xi, x, \ldots)$. Otherwise, nothing changes in the scheme of fixed-point iteration.

Multidimensional fixed-point iteration
Given a mapping $\mathbf{\Phi}:\mathbb{R}^n \to \mathbb{R}^n, \quad \mathbf{x} \to \mathbf{\Phi}(\mathbf{x})$.
To find a fixed point $\boldsymbol{\xi}$ of $\mathbf{\Phi}$,

```
    start with initial value $\mathbf{x}^{(0)}$.
    iterate
        $\mathbf{x}^{(k+1)} = \mathbf{\Phi}(\mathbf{x}^{(k)})$ for $k = 0, 1, 2 \ldots$
```

Check the convergence conditions (Section 2.4)!

Example: Fixed-point iteration for a system of two nonlinear equations

Let the following system of nonlinear equations be given (where naturally, log denotes the natural logarithm).

$$\begin{aligned} 4x - y + xy - 1 &= 0 \\ -x + 6y + \log(xy) - 2 &= 0 \end{aligned}$$

Start with the approximative solution $x_0 = y_0 = 1$ and use a suitable fixed-point iteration to determine better approximations.

In the first equation and for the given initial values, the term $4x$ makes the most substantial contribution. The second equation depends most strongly on $6y$. In this situation, you should make variables x and y explicit from these equations where they have the most decisive influence.

$$\begin{aligned} x &= \frac{1}{4}(y - xy + 1) \\ y &= \frac{1}{6}(x - \log(xy) + 2) \end{aligned}$$

Here, the function $\mathbf{\Phi}$ is a vector of two real-valued functions ϕ and ψ, the vector $\mathbf{x}$ has two components x and y.

$$\mathbf{\Phi}(\mathbf{x}) = \begin{bmatrix} \phi(x,y) \\ \psi(x,y) \end{bmatrix} = \begin{bmatrix} \frac{1}{4}(y - xy + 1) \\ \frac{1}{6}(x - \log(xy) + 2) \end{bmatrix}$$

Iteration provides the sequence $(1; 1)$, $(1/4; 1/2)$, $(0.343\,75; 0.721\,574)$, $(0.368\,383; 0.622\,985), \ldots$, which converges to the fixed point $(0.353\,443\,88; 0.639\,968\,47)$.

2.3 Norms

The exact solution, the approximate solution, and the error in systems of equations are all vectors in $\mathbb{R}^n$. We need to measure the magnitude, or length, of error vectors and also the distance of an approximation from the exact solution. In the one-dimensional case, we calculate the "size" of x by the absolute value x, and the distance between two values x and y on the real axis by $|y - x|$.

But while there is only one reasonable definition for the absolute value in $\mathbb{R}$, several possibilities are open in $\mathbb{R}^n$. First, there is the usual definition for the length of a vector, also called *Euclidean* length or *2-Norm* . But often, it is easier to work with other norms. We will use here the *1-Norm* and the *∞-Norm* .

Norms in $\mathbb{R}^n$ for a vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T$

$$\|\mathbf{x}\|_1 = \sum_{i=1}^{n} |x_i| \quad , \quad \text{1-norm, taxicab norm, Manhattan norm}$$

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n} (x_i)^2} \quad , \quad \text{Euklidian norm, 2-norm}$$

$$\|\mathbf{x}\|_\infty = \max_i |x_i| \quad , \quad \text{infinity norm, maximum norm}$$

Do you remember the definition of a norm from Mathematics 2?

A norm in $\mathbb{R}^n$ is a function that assigns to each vector $\mathbf{x} \in \mathbb{R}^n$ a nonnegative real number $\|\mathbf{x}\| \in \mathbb{R}_0^+$, so that $\forall \mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, $\forall \alpha \in \mathbb{R}$ must satisfy three conditions.

- Only the zero vector has norm 0

$$\|\mathbf{x}\| = 0 \quad \Rightarrow \quad \mathbf{x} = 0$$

- Absolute value of scalar α can be factored out

$$\|\alpha \cdot \mathbf{x}\| = |\alpha| \cdot \|\mathbf{x}\|$$

- The triangle inequality holds

$$\|\mathbf{x}+\mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$$

Norm and distance

A norm can also measure the distance between two points $\mathbf{x}$ ans $\mathbf{y}$.

$$\operatorname{dist}(\mathbf{x},\mathbf{y}) = \|\mathbf{x}-\mathbf{y}\|$$

- Cabs in Manhattan measure distances in the 1-norm.

  Therefore the 1-norm is also called taxicab norm or Manhattan norm

- Distance as the crow flies corresponds to the 2-norm.

- Biggest difference in components: ∞-norm.

Matrix norms

- The fundamental destiny of matrices is to multiply vectors.

- The result of a matrix-vector multiplication is also a vector; it is usually rotated and longer or shorter than the original vector.

- A *matrix norm* measures how strongly it can change vector lengths.

- A given matrix cannot extend vectors arbitrarily. For each matrix, there is a “maximum stretch factor.”

The “maximum stretch factor” is a matrix norm.

Some matrix norms

The 1-, 2- and ∞-norms are defined via the corresponding vector norms: They specify how much matrix-vector multiplication maximally can enlarge $\mathbf{y} = A \cdot \mathbf{x}$ compared to $\mathbf{x}$. It is easy to calculate the 1-norm or the ∞-norm of a matrix.

$$\begin{aligned} \|A\|_1 \qquad & \textit{1-norm}: \text{ maximum absolute column sum} \\ \|A\|_\infty \qquad & \infty\textit{-Norm}: \text{ maximum absolute row sum} \end{aligned}$$

Unfortunately, for the frequently used matrix 2-norm no such simple calculation rule exists.

However, MATLAB can easily calculate all norms. $\|A\|_1$ = `norm(A,1)`, $\|A\|_2$ = `norm(A)`, $\|A\|_\infty$ = `norm(A,Inf)`.

Matrix norm, general definition

You can add Matrices and multiply them by scalars. In this sense, they behave precisely like vectors of $\mathbb{R}^n$. We can interpret everything that behaves like a vector as a "vector": The $m \times n$-matrices form a *vector space*. Therefore, term "norm of a matrix" can be defined in the same way as the norm of vectors of $\mathbb{R}^n$. Compare the definition of a norm in $\mathbb{R}^n$ on page 22 and try to find the differences—there are hardly any!

A norm in $\mathbb{R}^m \times \mathbb{R}^n$ is a function that assigns to each $m \times n$ matrix A a nonnegative real number $\|A\| \in \mathbb{R}_0^+$, so that $\forall A, B \in \mathbb{R}^m \times \mathbb{R}^n$, $\forall \alpha \in \mathbb{R}$ must satisfy three conditions.

- Only the zero matrix has norm 0

$$\|A\| = 0 \quad \Rightarrow \quad A = 0$$

- Absolute value of scalar α can be factored out

$$\|\alpha \cdot A\| = |\alpha| \cdot \|A\|$$

- The triangle inequality holds

$$\|A + B\| \leq \|A\| + \|B\|$$

These three basic rules must apply to every norm. However, there are bonus features for the 1-, 2-, or ∞-norm. For these matrix norms, the following additional rules apply.

$$\|A \cdot B\| \leq \|A\| \cdot \|B\| \tag{8}$$

$$\|A \cdot \mathbf{x}\| \leq \|A\| \cdot \|\mathbf{x}\| \tag{9}$$

Compare with the absolute value $|a \cdot b| = |a| \cdot |b|$

Frobenius norm

The Frobenius Norm $\|A\|_F$ is calculated like the 2-norm of a vector: *square root of sum of squares*

Frobenius-Norm: $$\|A\|_F = \sqrt{\sum a_{ij}^2}$$

It is easier to calculate the Frobenius norm instead of the 2-Norm, and you can use it as an upper bound.

$$\|A\|_2 \leq \|A\|_F$$

Moreover, $\|A\|_F$ also provides bonus features similar to those of the 1-, 2-, or ∞-norm,

$$\|A \cdot B\|_F \leq \|A\|_F \cdot \|B\|_F \quad , \quad \|A \cdot \mathbf{x}\|_2 \leq \|A\|_F \|\mathbf{x}\|_2$$

MATLAB: $\|A\|_F =$ `norm(A,'fro')`.

Matrix norms—the small print

Don't think much about it;
incomplete this text would be without it.

The informal explanation *"matrix norm is maximum extension factor"* is mathematically correct for 1-, 2-, and ∞-norms when vector lengths are measured in the respective norms. However, the Frobenius norm usually overestimates the maximum extension factor when vector lengths are measured in the 2-norm. Nevertheless, it provides an upper bound for the lengthening factor.
There is also another rule, $\|A\| = \max_{i,j} |a_{ij}|$, which satisfies the three conditions of a norm, but is not always an upper bound for the lengthening factor.

2.4 Convergence

As in the one-dimensional case, convergence of an n-dimensional fixed-point iteration depends on a contraction property.

Fixed-point iterations in $\mathbb{R}^n$ converge for contraction mappings
Let $\mathbf{\Phi}(x)$ be a function with fixed popint $\boldsymbol{\xi}$: $\mathbf{\Phi}(\boldsymbol{\xi}) = \boldsymbol{\xi}$. Also, let B be an open neighborhood around $\boldsymbol{\xi}$ in the form $B = \{\mathbf{x} : \|\boldsymbol{\xi} - \mathbf{x}\| < r\}, \quad r > 0$. If $\mathbf{\Phi}$ acts in B as a *contraction mapping* in some norm $\|\cdot\|$, i. e

$$\|\mathbf{\Phi}(\mathbf{x}) - \mathbf{\Phi}(\mathbf{y})\| \le C\|\mathbf{x} - \mathbf{y}\| \ , \quad C < 1 \text{ for all } \mathbf{x}, \mathbf{y} \in B \ ,$$

then the fixed-point iteration $\mathbf{x}^{(k+1)} = \mathbf{\Phi}(\mathbf{x}^{(k)})$ converges for all $\mathbf{x}^{(0)} \in B$ at least linearly to $\boldsymbol{\xi}$.

One can prove the convergence of the multidimensional fixed point iteration in the same way as in the one-dimensional case when a contraction property holds. Also, the concept of the order of convergence can be directly applied to the multidimensional case by using norms.

Contraction and Jacobian matrix

The convergence criterion $|\phi'(\xi)| < 1$ from the one-dimensional case (compare Page 17) can be generalized to several dimensions. For this end, one collects the partial derivatives of $\mathbf{\Phi}$ in a matrix D_ϕ, called the *Jacobian matrix*.

Jacobian matrix D_ϕ of a function $\mathbf{\Phi} : \mathbb{R}^n \to \mathbb{R}^n$

$$D_\phi = \begin{bmatrix} \frac{\partial \phi_1}{\partial x_1} & \frac{\partial \phi_1}{\partial x_2} & \cdots & \frac{\partial \phi_1}{\partial x_n} \\ \frac{\partial \phi_2}{\partial x_1} & \frac{\partial \phi_2}{\partial x_2} & \cdots & \frac{\partial \phi_2}{\partial x_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial \phi_n}{\partial x_1} & \frac{\partial \phi_n}{\partial x_2} & \cdots & \frac{\partial \phi_n}{\partial x_n} \end{bmatrix}$$

Then, similar to the one-dimensional case, one can state

Fixed-point iterations converge locally,
if in the 1-,2-, Frobenius- oder ∞-norm holds

$$\|D_\phi\| < 1$$

2.5 Newton's Method for Systems of Equations

Given a vector-valued function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$. Let us find a zero of $\mathbf{f}$. The zero is a vector $\mathbf{x} \in \mathbb{R}^n$ that solves

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

This is the general formulation of a system of n linear or nonlinear equations in n unknowns. And once again, let us note: we put vectors from $\mathbb{R}^n$ and vector-valued functions in bold type $(\mathbf{x}, \mathbf{f}(\mathbf{x}), \ldots)$, as distinguished from variables and real-valued functions $(x, f(x), \ldots)$.

Component-wise written out for

$$\mathbf{f} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix} \quad \text{and} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad \text{, the system is} \quad \begin{array}{c} f_1(x_1, x_2, \ \ldots \ , x_n) = 0 \\ f_2(x_1, x_2, \ \ldots \ , x_n) = 0 \\ \ldots \\ f_n(x_1, x_2, \ \ldots \ , x_n) = 0 \end{array} .$$

Newton's method for systems reduces the solution of a nonlinear system to the solution of a sequence of linear systems. Solving linear equations is comparatively simple compared to nonlinear systems. We will treat systems of linear equations in detail later, but for the time being, we will assume that you are familiar enough with them from school.

Assuming that the corresponding partial derivatives exist, we define the *Jacobian matrix* D_f of $\mathbf{f}$ as

$$D_f = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}$$

Let us assume that an initial value $\mathbf{x}^{(0)}$ is given in the vicinity of a zero. Then Taylor's theorem approximates $\mathbf{f}$ in the neighborhood of $\mathbf{x}^{(0)}$,

$$\mathbf{f}(\mathbf{x}^{(0)} + \Delta\mathbf{x}) = \mathbf{f}(\mathbf{x}^{(0)}) + D_f(\mathbf{x}^{(0)}) \cdot \Delta\mathbf{x} + \mathbf{R}$$

with a remainder term $\mathbf{R}$ that vanishes in the limit $\Delta\mathbf{x} \to 0$ with higher order. We drop this remainder term and require $\mathbf{f}(\mathbf{x}^{(0)} + \Delta\mathbf{x}) = 0$. From the resulting equation

$$0 = \mathbf{f}(\mathbf{x}^{(0)}) + D_f(\mathbf{x}^{(0)}) \cdot \Delta\mathbf{x}$$

it is easy to determine the correction vector $\Delta\mathbf{x}$ and thus an improved approximation $\mathbf{x}^{(1)} = \mathbf{x}^{(0)} + \Delta\mathbf{x}$.

Newton's method for systems

Given a differentiable vector-valued function $\mathbf{f}$ and an initial value $\mathbf{x}^{(0)}$. Wanted a zero of $\mathbf{f}$.

iterate

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \Delta\mathbf{x}^{(k)}$$

with $\Delta\mathbf{x}^{(k)}$ as solution of $D_f(\mathbf{x}^{(k)})\Delta\mathbf{x}^{(k)} = -\mathbf{f}(\mathbf{x}^{(k)})$

Actually, this method is a fixed-point iteration for the function

$$\mathbf{\Phi}(\mathbf{x}) = \mathbf{x} - D_f^{-1}(\mathbf{x})\mathbf{f}(\mathbf{x}).$$

Of course, D_f^{-1} must exist for the method to work.

One can show: If D_f^{-1} exists at the zero then Newton's method converges quadratically for sufficiently accurate initial values.

Since it is often very tedious to calculate all elements of Df for each iteration, one sometimes computes Df just for the initial value $\mathbf{x}^{(0)}$ and keeps this D_f for the next iterations. This procedure is called the simplified Newton method. Here $\mathbf{x}^{(0)}$ should already be a useful approximation. However, this simplified Newton method converges only linearly.

The Newton method for systems requires the solution of a linear system of equations in each step. Therefore, the next chapter brings the systematic treatment of linear systems.

Example: the nonlinear system from Section 2.2

The function $\mathbf{f}$ and its Jacobian D_f are in this example

$$\mathbf{f}(\mathbf{x}) = \begin{bmatrix} 4x - y + xy - 1 \\ -x + 6y + \log(xy) - 2 \end{bmatrix}, \qquad D_f = \begin{bmatrix} 4+y & -1+x \\ -1+\frac{1}{x} & 6+\frac{1}{y} \end{bmatrix}.$$

Inserting the initial value $(1;1)$ gives

$$\mathbf{f}(\mathbf{x}) = \begin{bmatrix} 3 \\ 3 \end{bmatrix}, \qquad D_f = \begin{bmatrix} 5 & 0 \\ 0 & 7 \end{bmatrix}.$$

Now, Newton's method requires the solution of the linear system

$$\begin{bmatrix} 5 & 0 \\ 0 & 7 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = - \begin{bmatrix} 3 \\ 3 \end{bmatrix}$$

Thus, we get the correction vector and the next approximation

$$\Delta\mathbf{x}^{(0)} = \begin{bmatrix} -0,6 \\ -0,428571 \end{bmatrix}, \quad \mathbf{x}^{(1)} = \mathbf{x}^{(0)} + \Delta\mathbf{x}^{(0)} = \begin{bmatrix} 0,4 \\ 0,571429 \end{bmatrix}.$$

The next step evaluates $\mathbf{f}$ and D_f for the new values of $\mathbf{x}$, solves the linear system for the correction term $\Delta\mathbf{x}^{(1)}$, and calculates from this the improved approximation $\mathbf{x}^{(2)} = \mathbf{x}^{(1)} + \Delta\mathbf{x}^{(1)}$. However, the new matrix D_f does not have such nice entries as the initial D_f. Therefore, the system of equations is not as directly solvable as in the first step. The simplified version of Newton's method would re-evaluate $\mathbf{f}$ but keep the simpler diagonal matrix D_f of the first step. The effect would be a more straightforward calculation for the cost of only linear instead of quadratic convergence.

3 Systems of Linear Equations, Direct Methods

We use the standard notation for systems of linear equations,

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

where A denotes the coefficient matrix, $\mathbf{b}$ the right-hand side, and $\mathbf{x}$ the solution vector. If the system consists of n equations and unknowns, then A is an $n \times n$ matrix.

Solution methods for linear systems of equations fall into two main categories: Direct and iterative methods.

- Direct methods compute an exact solution (assuming no rounding error during computation). For example, elimination, substitution, and Cramer's rule fall into this category. If you want to use paper and pencil to solve systems with two or three unknowns, they are the methods of choice. Computers can easily use direct methods for thousands of equations and unknowns.
- Iterative methods start with some initial guess and compute progressively better approximate solutions. They are only suitable for systems of equations with a specific matrix structure. This way, computers can solve huge systems of equations (several million unknowns), such as those arising in numerical flow simulations or structural analysis.

This chapter deals with direct methods and repeats (what you should know from mathematics 1) theoretical statements on the existence and uniqueness of the solution; Chapter 4 will treat iterative methods.

Software of recognized high quality is freely available in the LAPACK program library (`http://www.netlib.org/lapack/`). Even in commercially available software packages you will not find anything better. MATLAB also contains the LAPACK algorithms. (By the way, MATLAB was initially created as a simple user interface to this package).

3.1 Triangular matrices

If A is a lower or upper triangular matrix, one can solve the system $A\mathbf{x} = \mathbf{b}$ directly by forward or backward substitution, respectively.

Otherwise, you could transform the equations to triangular form as described in Section 3.2. Alternatively, you could factorize A into a product of triangular matrices. The corresponding procedure is described in Section 3.5.

We denote triangular matrices by L and U. In the usual notation, in L only the entries in the lower left triangle are different from zero and all entries of the main diagonal are equal to one. In U, only entries in the upper right triangle, including the main diagonal are not equal to zero. Example for $n = 4$:

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \ell_{21} & 1 & 0 & 0 \\ \ell_{31} & \ell_{32} & 1 & 0 \\ \ell_{41} & \ell_{42} & \ell_{43} & 1 \end{bmatrix}, \qquad U = \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix}$$

If the elements of a triangular matrix L are stored on an array `a[i][j]` and the right-hand side $\mathbf{b}$ on a vector `b[i]`, the following Java program segment solves the system $L\mathbf{x} = \mathbf{b}$ stepwise by *forward substitution* .

Note that Java counts array indices from 0 to $n-1$; conventional math notation counts rows and columns from 1 to n.)

```
for (int i=0; i<n; i++) {
   x[i] = b[i];
   for (int j=0; j<i; j++) {
      x[i] -= a[i][j] * x[j];
   }
}
```

A similarly compact formulation is possible for a system of equations with an upper triangular matrix. Differences from before: the backward substitution starts in the last row and proceeds from bottom to top; dividing by the main diagonal element is necessary; the right side is already stored on `x[ ]` at the beginning; the algorithm overwrites `x[ ]` with the solution vector.

```
for (int i=n-1; i>-1; i--) {
   for (int j=i+1; j<n; j++) {
      x[i] -= a[i][j] * x[j];
   }
   x[i] /= a[i][i];
}
```

The computational effort for the forward substitution, measured by the number of multiplications and divisions, is $n^2/2 - n/2$. On the other hand, the backward substitution needs $n^2/2 + n/2$ point operations. For large n, only the quadratic term is essential. Therefore we say:

Computational effort for forward- or backward substitution
Solving an $n \times n$ triangular system requires $O(n^2)$ floating-point operations.

Computational effort grows quadratically with the number of equations.

Question: What do you do when a system of equations is not triangular? Answer: You transform it into a triangular system. Of course, you must do this so that the original and the transformed system are equivalent (which means both systems have exactly the same solutions).

The following section shows how to do so.

3.2 Gaussian Elimination

The following box describes a basic version of Gaussian elimination.

Gaussian elimination, basic form
Given an $n \times n$ matrix A and right-hand side $\mathbf{b}$. Provided it will not divide by a zero a_{kk}, this algorithm transforms the system $A\mathbf{x} = \mathbf{b}$ to an equivalent upper triangular system $U\mathbf{x} = \mathbf{c}$.

```
for all columns $k = 1, \dots n - 1$
    in column $k$ eliminate all entries below the main diagonal
```

Elimination in column k proceeds in the following manner

```
for all rows $i = k + 1, \dots, n$ below the main diagonal
    set $p = a_{ik}/a_{kk}$
    subtract the $p$-th multiple of row $k$ from row $i$
```

This subtraction is done via

```
for all columns $j = k, \dots n$
    $a_{ij} = a_{ij} - pa_{kj}$
for right-hand side: $b_i = b_i - pb_k$
```

This algorithm *overwrites* entries in A and $\mathbf{b}$ repeatedly by intermediate results and, finally, by the enties of U und $\mathbf{c}$. Entries below the main diagonal, which should be zero in U, will not be erased, however. (It will turn out later, in Section 3.5, that these entries are quite important.)

This algorithm performs $n^3/3 - n/3$ operations (counting multiplications and divisions only) to transform the matrix, and $n^2/2 - n/2$ operations to transform the right-hand side.

In JAVA source code, Gaussian elimination looks surprisingly simple. Assuming that `x[ ]` initially contains the right-hand side, the algorithm performs three nested loops so that `x[ ]` finally stores the solution vevtor.

```
for (int k=0; k<n; k++) {
   for (int i=k+1; i<n; i++) {
      double p = a[i][k] / a[k][k];
      for (int j=k+1; j<n; j++) {
         a[i][j] -= p * a[k][j];
      }
      x[i] -= p * x[k];
   }
}
```

The program does not calculate results that will be zero anyway. Thus, in step k, it will not erase the elements below the main diagonal in column k. (Actually, these entries will be necessary for the LU matrix decomposition discussed in Section 3.5.)

The stepwise backward insertion can be done with $n^2/2 + O(n)$ operations according to the program segment from the previous section. Note that te double loop in that code uses only the the upper triangle of A. Therefoer, any values $\neq 0$ below the main diagonal do not affect the computation.

These two code segments combined provide a simple equation solver.

Computational effort for basic Gaussian elimination grows cubically with the number of equations
To solve an $n \times n$ system $A\mathbf{x} = \mathbf{b}$, Gaussian elimination needs $O(n^3)$ operations.

(A precise count of multiplications and divisions results in $n^3/3 + n^2 - n/3 = n^3/3 + O(n^2)$ operations.)

3.3 Pivoting

Our basic implementation of Gaussian elimination has a catch: At the step $p = a_{ik}/a_{kk}$, a division by zero may occur. This is quite unlikely to happen for a matrix of randomly chosen real numbers, but Murphy's law says: *If anything can go wrong, it will.* Actually, the method fails already for systems as simple as

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix},$$

because the very first operation would be a division by zero. This failure is not the fault of the equations, which have the unique solution $x_1 = 1, x_2 = 1$.

If you switch the first and second equations, the procedure runs without problems. Also, for reasons of numerical accuracy, it may be beneficial to swap equations or unknowns systematically. This procedure is called pivoting.

Gaussian elimination with complete pivoting
Given an $n \times n$-Matrix A and righ-hand side $\mathbf{b}$. This algorithm transforms the system $A\mathbf{x} = \mathbf{b}$ to upper triangular form $U\mathbf{x} = \mathbf{c}$. It overwrites the matrix A with U and the vector $\mathbf{b}$ with $\mathbf{c}$.

```
for all columns $k = 1, \dots, n-1$
    find the element largest in absolute value in
    the square submatrix ranging from rows and
    columns $k$ to $n$.

    switch equations and unknowns so that this
    element ends up in position $a_{kk}$.

    if $a_{kk} = 0$
        stop.
    else
        in column $k$ eliminate all entries
        below the main diagonal in the same way as
        in the basic version ofGaussian Elimination
```

The pivotal step in Gaussian elimination is selecting which equation to use to eliminate the corresponding unknown in the remaining equations. Therefore, the element a_{kk} used in the calculation $p = a_{ik}/a_{kk}$ is called the *pivot element.* To find a favorable pivot element by suitably swapping or reordering equations and unknowns is called *pivot search* or *pivot selection* .

Row and column interchanges complicate the algorithm considerably compared to the triple loop of Gaussian elimination in its basic form. We will not list code for a complete pivot search because any possible gain of additional insight is not reasonably related to the cumbersome technical details.

Usually, equation solvers do not perform complete but only row pivot search. That means they only swap rows (=equations) for simplicity[5] . As a result, this procedure is somewhat more sensitive to numerical errors than a full pivot search.

[5] For example, the entry *Gaussian elimination* in Wikipedia lists pseudocode.

3.4 Existence of Solutions

The rule of thumb, “If there are as many equations as unknowns, there is always a solution,” *is wrong*. I repeat (because I hear it again and again during exams): *IS WRONG!*

For the three linear systems

$$\begin{aligned} x + y &= 2 \\ 2x + 2y &= 4 \end{aligned} \qquad \begin{aligned} x + y &= 2 \\ x + 2y &= 3 \end{aligned} \qquad \begin{aligned} x + y &= 2 \\ 2x + 2y &= 3 \end{aligned}$$

you hopefully see with the naked eye: $x = 1, y = 1$ solves the first and the second ones. The third system is unsolvable. However, for the first system, there are infinitely many more solutions. These examples illustrate the general case.

Linearer systems—three cases
There are three possibilities for a linear system of equations. It may have

- infinitely many solutions;
- a single unique solution;
- no solution at all.

(You should remember this from your Mathematics introductory lecture.)

This section deals only with systems with the same number of equations and unknowns, but the above statement also applies to linear systems with more equations than unknowns. Only two cases are possible for fewer equations than unknowns: no solution or infinitely many solutions.

An essential property of the Gaussian elimination method is that it can distinguish the three cases

3.4.1 Elimination

Gaussian elimination with complete or row pivot search transforms the original matrix A and the right-hand side $\mathbf{b}$ into a system in *row echelon form:* From each row to the next one, the number of leading zeroes (seen from the left) increases by at least one.

Possible outcomes of the elimination procedure
The system is in row echelon form.

- Zero rows occur in A, and all corresponding entries in $\mathbf{b}$ are zero as well: *infinitely many solutions.*
- Zero rows occur in A, but at least one corresponding entry in $\mathbf{b}$ is not zero: *no solution.*
- No zero rows occur in A: *a unique solution.*

When computers perform the elimination in floating-point arithmetic, it is not so easy to check whether entries are precisely equal to zero. Due to roundoff errors in the input data and during the calculation, matrix elements that should be zero might become tiny floating point numbers. It is a delicate numerical question to decide how small an entry can be considered zero. Those dubious cases when matrix rows are nearly zero so that the algorithm just barely can find the solution are called *numerically singular.*

3.4.2 Rank of matrix and augmented matrix

The *rank of an $m \times n$ matrix* is the number of its linearly independent rows or, equivalently, columns.

Even if the matrix has different numbers of rows and columns, there are always exactly as many linearly independent rows as columns; in short: row rank = column rank.The numbers of linearly independent rows or columns are always the same; in short: row rank = column rank.

The MATLAB command `rank(A)` determines the rank of the $n \times n$ matrix A. or columns) of the n×n matrix A, and `rank([A,b])` determines the rank of the augmented coefficient matrix (this is the linear system matrix combined with the right-hand side as its last column).

Rank determines solution set

- `rank(A)` $=$ `rank([A,b])` $< n$: *infinitely many solutions.*
- `rank(A)` $< n$ und `rank(A)` $\neq$ `rank([A,b])`: *no solution*
- `rank(A)` $= n$: *a unique solution*

There are several methods to calculate the rank of a matrix, for example, transformation to step form by Gaussian elimination: The rank is the number of non-zero rows in the matrix. However, there is no simple decision whether a value is non-zero due to rounding errors or truly non-zero. MATLAB uses a sophisticated procedure (singular value decomposition), which provides a reliable rank estimate. If a system of equations has infinitely many solutions, you may read off the general solution from the rref([A,b]) result or use the following commands:

`pinv(A)*b` returns a particular solution
`null(A)` returns the *null space* of A: a list of linearly independent solutions of the *homogeneous system* $A\mathbf{x} = 0$.

The general solution is the sum of a particular solution and an arbitrary linear combination from the null space.

`null(A,'r')` You may use `null(A,'r')` if A is a small matrix with small integer elements. This variant also returns a list of linearly independent solutions, but with "nicer" numbers, i.e., ratios of small integers. However, this method is numerically less accurate than `null(A)` . (Never let yourself be blinded by external beauty if falseness lurks behind it).

For example, consider the system $A\mathbf{x} = \mathbf{b}$ with

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 5 & 6 \\ -1 & -2 & -2 & -2 \\ 3 & 6 & 8 & 10 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 2 \end{bmatrix}, \quad \texttt{rref([A,b])} = \begin{bmatrix} 1 & 2 & 0 & -2 & -2 \\ 0 & 0 & 1 & 2 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The result of the `rref([A,b])` command means: you may choose $x_2 = \lambda$ and $x_4 = \mu$ as free parameters; $x_3 = 1 - 2\mu, x_1 = -2 - 2\lambda + 2\mu$. In vector notation,

$$\mathbf{x} = \begin{bmatrix} -2 \\ 0 \\ 1 \\ 0 \end{bmatrix} + \lambda \begin{bmatrix} -2 \\ 1 \\ 0 \\ 0 \end{bmatrix} + \mu \begin{bmatrix} 2 \\ 0 \\ -2 \\ 1 \end{bmatrix}$$

MATLAB can also compute a partial solution in the form `x=A\b` and the null space via `null(A,'r')` for a matrix as simple as this one. However, these commands ares not recommended for real-world problems. Nevertheless, here they yield (apart from a warning message) the "more beautiful" result

$$\mathbf{x} = \begin{bmatrix} 0 \\ -1 \\ 1 \\ 0 \end{bmatrix} + \lambda \begin{bmatrix} -2 \\ 1 \\ 0 \\ 0 \end{bmatrix} + \mu \begin{bmatrix} 2 \\ 0 \\ -2 \\ 1 \end{bmatrix}$$

`pinv(A)*b` and `null(A)` return the numerically preferred representation,

$$\mathbf{x} = \begin{bmatrix} -0.2069 \\ -0.41379 \\ 0.034483 \\ 0.48276 \end{bmatrix} + \lambda \begin{bmatrix} -0.77069 \\ 0.10421 \\ 0.56227 \\ -0.28113 \end{bmatrix} + \mu \begin{bmatrix} -0.48335 \\ 0.54725 \\ -0.61115 \\ 0.30558 \end{bmatrix}$$

MATLAB calculates this result using sophisticated and reliable numerical procedures. However, in contrast to the previous representations of the solution, substitution into the expression $A\mathbf{x}-\mathbf{b}$ does not yield precisely zero, but due to rounding errors, values in the range of 1×10^{-15}. (Sometimes, beauty signals some sort of truth indeed.)

3.4.3 Determinant

The determinant determines wether a linear system has a unique solution.

Linear systems $A\mathbf{x} = \mathbf{b}$ with $\det A \neq 0$ have a unique solution.

However, this rule is useless for numerical computation.

For example, the MATLAB command `A=rosser` creates the 8×8 matrix

$$A = \begin{bmatrix} 611 & 196 & -192 & 407 & -8 & -52 & -49 & 29 \\ 196 & 899 & 113 & -192 & -71 & -43 & -8 & -44 \\ -192 & 113 & 899 & 196 & 61 & 49 & 8 & 52 \\ 407 & -192 & 196 & 611 & 8 & 44 & 59 & -23 \\ -8 & -71 & 61 & 8 & 411 & -599 & 208 & 208 \\ -52 & -43 & 49 & 44 & -599 & 411 & 208 & 208 \\ -49 & -8 & 8 & 59 & 208 & 208 & 99 & -911 \\ 29 & -44 & 52 & -23 & 208 & 208 & -911 & 99 \end{bmatrix}$$

For this matrix, MATLAB currently[6] computes $\det A = -9480.580$, so you definitely would think $\det A \neq 0$. Thus, a linear system with this matrix A should have a unique solution. However, MATLAB correctly computes the rank of A as `rank(A)`=7. Because $7 < 8$, a unique solution cannot exist. MATLAB's value for the determinant is plainly wrong.

For the 6×6matrix H, a so-called Hilbert matrix,

$$H = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} \\ \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} \end{bmatrix}$$

[6] with version 2022b. The value for version 2021b was $\det A = -10\,611$. Previous versions around 2018 gave $\det A = -9478.9$; the version from 2015 gives -9448.8; prior to 2013 the value was $\det A = -13\,017$. It should worry you deeply that a numerical result varies that much depending on the program version!

MATLAB computes $\det A = 5.3673\times10^{-18}$, and this you could quite reasonably round off and interpret as $\det A = 0$. However, for the rank MATLAB (correctly) calculates `rank(H)`=6. Thus, linear systems with H do have a unique solution (although, in this case, the solution is highly sensitive to roundoff errors).

These examples illustrate:

> The numerically calculated value of a determinant says nothing about the solvability of a linear system.

3.5 *LU* decomposition

The simple Gauss elimination yields (if it does not break down) more than the transformation to a triangular shape. It can, at the same time, give the decomposition

$$A = LU$$

where L is a lower triangular matrix with ones in the main diagonal and U is an upper triangular matrix.

> ***LU* decomposition**
> Gaussian elimination without pivot search factorizes (if it does not break down) a matrix A into a product $A = LU$ of a lower triangular matrix L and an upper triangular matrix U.

In case of Gaussian elimination with pivoting, the product of the lower and upper triangular parts does not give the original matrix, but a matrix with permuted rows and columns.

The elements of L are 1 along the main diagonal, and below that equal to the multipliers $p = a_{ik}/a_{kk}$ at the corresponding positions (i, k). The elements of U are exactly those that the elimination procedure writes into the upper right triangle.

The only change in the algorithm on page 29 is remembering the intermediate results `p`. Conveniently, one can store each `p` at the position of the corresponding field element `a[i][k]`; the procedure eliminates just this entry, thus generating a zero. Instead of this zero, the algorithm stores the intermediate result `p` at this position.

Computer programs usually formulate the procedure in such a way that the original matrix A is overwritten by R and U. The upper triangle of A contains, after successful completion, the non-zero entries of U. The all-ones main diagonal of L is self-evident, no values need to be stored. Below the main diagonal of A the remaining non-zero elements of L are stored. The elegance of this storage method is that it arises in the course of the procedure quasi by itself.

See the lab notes for more information!

For the LU decomposition, you don't need a right-hand side. It comes into play later. The way to solve a system $A\mathbf{x} = \mathbf{b}$ when $A = LU$ is already available is a sequence of transformations.

$$\begin{aligned}
A\mathbf{x} &= \mathbf{b} \\
(LU)\mathbf{x} &= \mathbf{b} \\
L(U\mathbf{x}) &= \mathbf{b} && \text{set } \mathbf{y} = U\mathbf{x} \\
L\mathbf{y} &= \mathbf{b} && \text{forward substitution solves for } \mathbf{y} \\
U\mathbf{x} &= \mathbf{y} && \text{backward substitution solves for} \mathbf{x}
\end{aligned}$$

The computational process and effort are completely equivalent to standard Gaussian elimination. However, the advantage of the LU decomposition becomes apparent when solving several systems with the same matrix A and different right-hand sides $\mathbf{b}_1, \mathbf{b}_2, \ldots$ The LU decomposition is the labor-intensive part ($\sim n^3/3$ multiplications) and has to be performed only once. The individual solutions then cost only $\sim n^2$ multiplications per right-hand side.

Special variants of Gaussian elimination exist for symmetric matrices. Exploiting the symmetry saves arithmetic operations and storage space. A possible decomposition is

$$A = LDL^T \ ,$$

with a diagonal matrix D. The *Cholesky decomposition*

$$A = LL^T$$

is the method of choice for symmetric positive definite matrices.

3.6 A Numerical Example for Gaussian Elimination

Gaussian elimination is an algorithm for systematically eliminating unknowns. For small linear systems with “nice” numbers, one often deviates from the systematic way and tries to shorten the calculation (e. g., by exploiting already existing zeros). On the other hand, there is always the danger of calculating “around in circles,” not using equations, or using them twice. Therefore, while taking shortcuts for uncomplicated systems is perfectly OK, having a well-defined algorithm for the general case is essential.

Let us work out the calculations for the system $A \cdot \mathbf{x} = \mathbf{b}$ with

$$A = \begin{bmatrix} 5 & 6 & 7 \\ 10 & 20 & 23 \\ 15 & 50 & 67 \end{bmatrix} \quad , \quad \mathbf{b} = \begin{bmatrix} 6 \\ 6 \\ 14 \end{bmatrix}$$

We work with the *augmented coefficient matrix*

$$[A\,\mathbf{b}] = \begin{bmatrix} 5 & 6 & 7 & 6 \\ 10 & 20 & 23 & 6 \\ 15 & 50 & 67 & 14 \end{bmatrix}$$

Compare the box “Gaussian elimination, basic form” and the JAVA code on Page 29—our calculations follow these instructions.

Eliminating the first column

$n = 3$; in column $k = 1$ all entries below the main diagonal will be eliminated, i.e., the entries in rows $i = 2, 3$

$k = 1, i = 2$: Elimination in first column, second row

Eliminate $a_{ik} = a_{21} = 10$ using the diagonal element $a_{kk} = 5$ from row $k = 1$. Multiply first row by $a_{ik}/a_{kk} = 2$ and subtract.

$$\begin{array}{cccccc} 10 & 20 & 23 & 6 & \\ 10 & 12 & 14 & 12 & \vert - \\ \hline 0 & 8 & 9 & -6 & \end{array}$$

$k = 1, i = 3$: Elimination in first column, third row

Eliminate $a_{ik} = a_{31} = 15$ using the diagonal element $a_{kk} = 5$ from row $k = 1$. Multiply first row by $a_{ik}/a_{kk} = 3$ and subtract.

$$\begin{array}{rrrrl} 15 & 50 & 67 & 14 & \\ 15 & 18 & 21 & 18 & |- \\ \hline 0 & 32 & 46 & -4 & \end{array}$$

Transformed augmented matrix

after processing first column:

$$[A\,\mathbf{b}]^{(1)} = \begin{bmatrix} 5 & 6 & 7 & 6 \\ 0 & 8 & 9 & -6 \\ 0 & 32 & 46 & -4 \end{bmatrix}$$

Eliminating the second column

In column $k = 2$ all entries below the main diagonal will be eliminated. Here, this is the element in row $i = 3$ only.

$k = 2, i = 3$: Elimination in second column, third row

Eliminate $a_{ik} = a_{32} = 32$ using the diagonal element $a_{kk} = 8$ from row $k = 2$. Multiply second row by $a_{ik}/a_{kk} = 4$ and subtract.

$$\begin{array}{rrrrl} 0 & 32 & 46 & -4 & \\ 0 & 32 & 36 & -24 & |- \\ \hline 0 & 0 & 10 & 20 & \end{array}$$

Transformed augmented matrix

Having processed the second column, Gaussian elimination is complete.

$$[A\,\mathbf{b}]^{(2)} = \begin{bmatrix} 5 & 6 & 7 & 6 \\ 0 & 8 & 9 & -6 \\ 0 & 0 & 10 & 20 \end{bmatrix}$$

Backsubstitution

From the third row,

$$\begin{aligned} 10x_3 &= 20 \\ x_3 &= 2 \end{aligned}$$

Substitute for x_3 in second row

$$\begin{aligned} 8x_2 + 9x_3 &= -6 \\ 8x_2 + 18 &= -6 \\ 8x_2 &= -24 \\ x_2 &= -3 \end{aligned}$$

Substitute for x_2 and x_3 in first row,

$$\begin{aligned} 5x_1 + 6x_2 + 7x_3 &= 6 \\ 5x_1 - 18 + 14 &= 6 \\ 5x_1 &= 10 \\ x_1 &= 2 \end{aligned}$$

MATLAB's command `x = A\b` basically works this way, but in addition may interchange rows when selecting the pivot element.

multiple right-hand sides

To solve for several right-hand sides with the same matrix A, you augmente the matrix by all right-hand sides as additional columns and proceed as above.

Pivoting

In this example, no reordering of equations is necessary to avoid division by zero. However, a column pivot search would exchange first and third equation before the first step. Thus, the element largest in absolute value would be in position $(1, 1)$.

LU decomposition

The transformed matrix and the corresponding pivot factors also provide the LU decomposition. A right-hand side is not necessary for the LU decomposition.

$$\begin{bmatrix} 5 & 6 & 7 \\ 10 & 20 & 23 \\ 15 & 50 & 67 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 4 & 1 \end{bmatrix} \cdot \begin{bmatrix} 5 & 6 & 7 \\ 0 & 8 & 9 \\ 0 & 0 & 10 \end{bmatrix}$$

The MATLAB command `[L, U]=lu(A)` uses the Gaussian elimination method in principle but delivers a different LU decomposition for this numerical example. The matrix U is a genuinely upper triangular matrix, but L is a *permuted* lower triangular matrix. Reason: Column pivot search switches rows in the matrix during the elimination process. (Pivoting reduces the rounding errors.)

Decomposition obtained by `[L, U]=lu(A)`

$$\begin{bmatrix} 5 & 6 & 7 \\ 10 & 20 & 23 \\ 15 & 50 & 67 \end{bmatrix} = \begin{bmatrix} 1/3 & 4/5 & 1 \\ 2/3 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 15 & 50 & 67 \\ 0 & -40/3 & -65/3 \\ 0 & 0 & 2 \end{bmatrix}$$

3.7 More Applications of LU decomposition

3.7.1 Determinant

Determinant
Given a decomposition $A = LU$, the determinant of A is the product of all entries along the main diagonal of U.

Proof: The determinant of a triangular matrix is always the product of all entries along its main diagonal. Since L has an all-ones main diagonal, $\det L = 1$. Now, from the properties of the determinant,

$$\det A = \det(LR) = (\det L)(\det R) = \det R\ .$$

Counting multiplications and divisions only, computing an $n \times n$ matrix determinant this way requires $n^3/3 + 2n/3 - 1$ operations.

Compare this number of operations to the computational costs of the classical method Laplace expansion along rows or columns. For an $n \times n$ matrix, let $w(n)$ be the computational cost. For this matrix, Laplace expansion computes n subdeterminants of $(n-1) \times (n-1)$ matrices and multiplies them with corresponding matrix entries. For the computational effort thus applies the recursive relationship

$$w(n) = nw(n-1) + n = n(w(n-1) + 1)\ .$$

The function $w(n)$ grows rapidly, even stronger than the factorial $n!$. A moderately fast PC (performing 10^7 multiplications per second) would compute $\det A$ for a 10×10 matrix in less than one second. However, already for a 13×13 matrix, a quarter-hour coffee break is in order. For the result in case of a 15×15 matrix you will wait two and a half days, thirteen millennia for a 20×20 matrix, and a 25×25 matrix would not be ready after 80 billion years. The following table illustrates the rapid growth of w(n) and the comparatively small effort of the LU decomposition. It intends to point out the importance of computationally efficient algorithms and the difference between polynomial and exponential runtime.

n	$w(n)$	$n^3/3 + 2n/3 - 1$
2	2	3
3	9	10
4	40	23
5	205	44
6	1 236	75
7	8 659	118
8	69 280	175
9	623 529	248
10	6 235 300	339
15	2 246 953 104 075	1 134
20	4 180 411 311 071 440 000	2 679
25	26 652 630 354 867 072 870 693 625	5 224

3.7.2 Inverse

You will rarely need the inverse matrix A^{-1} of a given (nonsingular) matrix A explicitly. example, if some algorithm requires the vector $\mathbf{x} = A^{-1}\mathbf{y}$, you can as well solve the linear system $A\mathbf{x} = \mathbf{y}$ with less computational effort and better numerical accuracy.

Warning 1: Before calculating an inverse, ask yourself thrice whether you need the inverse explicitly.

Warning 2: If you still remember the formula known from linear algebra (the one with the determinants of the cofactors): forget it. It is of theoretical importance because it proves the existence of the inverse of a non-singular matrix. You should never (except in trivial examples) compute the inverse in this way. Consider: computational effort $O(n^5)$, if you calculate the individual subdeterminants employing LU decomposition; exponential computational effort, if you calculate determinants by Laplacian expansion.

If you can't avoid it, proceed this way. Call the first column of the inverse $\mathbf{x}_1$. The first column of the identity matrix I is the unit vector $\mathbf{e}_1 = (1, 0, \ldots, 0)^T$. By definition,

$$AA^{-1} = I\ .$$

The first column in this matrix equality is

$$A\mathbf{x}_1 = \mathbf{e}_1 \ .$$

Thus, you get the first column of the inverse by solving a linear system with the unit vector $\mathbf{e}_1$ on its right-hand side.

A straightforward generalization of this argument:

Inverse
The i-th column vector of A^{-1} solves the linear system

$$A\mathbf{x}_i = \mathbf{e}_i \ .$$

So you have to solve a linear system with multiple right-hand sides. Recipe:

```
decompose $A = LU$;        (needs $(n^3 - n)/3$ operations).
for $i = 1, \dots, n$
    solve $LU\mathbf{x}_i = \mathbf{e}_i$;        (needs $n^2$ operations per step).
```

Computational costs (counting multiplications and divisions) $(4n^3 - n)/3$.

The *Gauss–Jordan elimination* is an exceptionally well-organized variant of Gaussian elimination. It is well suited for pen-and-paper calculations. (You should know this algorithm from the introductory mathematics lectures.)

3.7.3 symmetric positive-definite matrices

Elementary arguments from linear algebra show: for symmetric positive-definite matrices, the basic variant of Gaussian elimination will never break down because of some $a_{kk} = 0$. Therefore, no pivoting is necessary. Conversely, symmetric positive-definite matrices are characterized by the property $a_{kk} > 0$ during all steps of an LU decomposition.

However, as mentioned on Page 35, in the case of symmetric positive-definite matrices, the decomposition is usually done in the form $A = LL^T$ (Cholesky decomposition) or$A = LDL^T$. Benefits: efficient implementations use less storage space and arithmetic operations as compared to a general LU decomposition.

3.7.4 Incomplete LU factorization

Even if most entries are zero in some matrix A, the factors L and U can have a much larger number of nonzero entries. Gaussian elimination introduces additional nonzeros during computation. These entries $\neq 0$ at positions where the initial matrix had elements $= 0$ are called *fill-in* . Simply ignoring all fill-in (or all fill-in entries smaller than some threshold) will significantly reduce the computational effort and memory size. Of course, then no longer $LU = A$, but $LU = \tilde{A}$ for an approximation $\tilde{A}$ to A. In this case, $LU = \tilde{A}$ is called an *incomplete factorization* (or *incomplete decomposition*). Many iterative solvers for linear systems work with incomplete factorizations.

A tiny change in the example program for the LU decomposition illustrates the basic idea (see the lab matrial for further information): Replace in the innermost loop

```
For j = k + 1 To n
   a(i, j) = a(i, j) - p * a(k, j)
Next
```

by

```
For j = k + 1 To n
   if a(i, j) <> 0 then
      a(i, j) = a(i, j) - p * a(k, j)
Next
```

Thus, the LR decomposition has become an incomplete factorization. However, the program does not save any memory space or computing time in this form. Particular data structures that store only the non-zero elements would be necessary for an efficient implementation (sparse data structures).

3.8 Sensitivity to Small Perturbations

Roundoff errors and noisy input data may change a matrix from A to $A + \delta A$ and the right-hand side from $\mathbf{b}$ to $\mathbf{b} + \delta\mathbf{b}$. The solution of this *perturbed* system will deviate by a (hopefully, small) $\delta\mathbf{x}$ from the true solution of the original system.

$$(A + \delta A)(\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b} \quad .$$

How does $\delta\mathbf{x}$ depend on δA and $\delta\mathbf{b}$?

Condition number
The *condition number* $\kappa(A)$ measures for the system $A\mathbf{x} = \mathbf{b}$, how the relative error of $\mathbf{x}$ depends on small relative changes in A and $\mathbf{b}$.

$$\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|} \le \kappa(A) \left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|} \right)$$

In the inequality above, $\|\cdot\|$ denotes both a vector norm (for example, the 1-, 2-, or ∞-norm) and the corresponding matrix norm. A short calculation using the properties of the norm (see Section 2.3) shows

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

Sketch of the proof: Start with the perturbed system

$$(A+\delta A)(\mathbf{x}+\delta\mathbf{x}) = \mathbf{b}+\delta\mathbf{b}\,,$$

expand the brackets,

$$A\mathbf{x} + A\cdot\delta\mathbf{x} + \delta A\cdot\mathbf{x} + \delta A\cdot\delta\mathbf{x} = \mathbf{b}+\delta\mathbf{b}\,;$$

since $A\mathbf{x}=\mathbf{b}$, we can cancel $A\mathbf{x}$ on the left and $\mathbf{b}$ on the righ-hand side. For small $\delta\mathbf{b}$ and δA, the product $\delta A\cdot\delta\mathbf{x}$ is small of higher order; we neglect this term. Thus,

$$A\cdot\delta\mathbf{x} + \delta A\cdot\mathbf{x} = \delta\mathbf{b}\ .$$

Make $\delta\mathbf{x}$ explicit,

$$\delta\mathbf{x} = A^{-1}\left(\delta\mathbf{b} - \delta A\cdot\mathbf{x}\right)\ ,$$

apply a vector norm on both sides,

$$\|\delta\mathbf{x}\| = \|A^{-1}\left(\delta\mathbf{b} - \delta A\cdot\mathbf{x}\right)\|\,,$$

use a property of matrix norms, inequality (9)

$$\|\delta\mathbf{x}\| \le \|A^{-1}\|\cdot\|\left(\delta\mathbf{b} - \delta A\cdot\mathbf{x}\right)\|\,,$$

employ the triangle inequality

$$\|\delta\mathbf{x}\| \le \|A^{-1}\|\left(\|\delta\mathbf{b}\| + \|\delta A\cdot\mathbf{x}\|\right)\ ,$$

expand the terms in the bracket

$$\|\delta\mathbf{x}\| \le \|A^{-1}\|\left(\frac{\|A\mathbf{x}\|}{\|\mathbf{b}\|}\|\delta\mathbf{b}\| + \frac{\|A\|}{\|A\|}\|\delta A\cdot\mathbf{x}\|\right)\ ,$$

use again an inequality for matrix norms,

$$\|\delta\mathbf{x}\| \le \|A^{-1}\|\cdot\|A\|\left(\frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|}\|\mathbf{x}\| + \frac{\|\delta A\|}{\|A\|}\|\mathbf{x}\|\right)\ ,$$

and finally, divide by $\|\mathbf{x}\|$,

$$\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|} \le \|A^{-1}\|\cdot\|A\|\left(\frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|} + \frac{\|\delta A\|}{\|A\|}\right)\ .$$

Thus, the relative error in $\mathbf{x}$ can be $\kappa(A)$ times larger than the relative error in A and $\mathbf{b}$. Errors in the input data will strongly affect a system of equations whose matrix has a large condition number. Such a system is called *ill-conditioned*. Geometric illustration: two straight lines intersecting at a small angle.

The calculation of the condition number directly according to the definition would require the computation of the inverse and would be nonsensically expensive. Many equation solvers provide estimates of $\kappa(A)$ as a byproduct. For example, it holds

$$\kappa(A) \ge \frac{\max|\lambda|}{\min|\lambda|}$$

(Ratio of largest to smallest magnitude of eigenvalues; Section 9 treats eigenvalues.)

4 Iterative Solvers for Linear Systems

Let a linear system in n equations and unknowns be given.

$$\begin{array}{llcl} a_{11}x_1 + a_{12}x_2 + & \dots & +a_{1n}x_n & = & b_1 \\ a_{21}x_1 + a_{22}x_2 + & \dots & +a_{2n}x_n & = & b_2 \\ \vdots & & \vdots & & \vdots \\ a_{n1}x_1 + a_{n2}x_2 + & \dots & +a_{nn}x_n & = & b_n \end{array} \tag{10}$$

In matrix notation,

$$A\mathbf{x} = \mathbf{b}\ . \tag{11}$$

Gaussian elimination is the classical solution method, at least for systems with up to several thousand equations. Chapter 3 treats it in detail. However, many applications (flow simulation, seismics, tomography, structural analysis…) produce systems with hundred thousands or millions of unknowns. Iterative solvers work well for large systems of this kind. Here you will get to know some basic methods only. They are fundamental in building more powerful iterative solvers.

4.1 Basic iterative solvers: Jacobi, Gauss-Seidel, SOR

Suppose the diagonal elements a_{ii} of an $n \times n$ matrix A are all nonzero. Then the following recipe for solving $A\mathbf{x} = \mathbf{b}$ (a fixed point method) would be possible:

Jacobi mehtod for $A\mathrm{x} = \mathrm{b}$, loosely formulated
Solve each equation for its diagonal element, set initial values, and iterate.

In more detail, using the component-wise notation (10): In row i, bring all terms except for the i-th to the right-hand side, and solve for x_i.

A correspondingly transformed 3×3 system then looks like this:

$$\begin{array}{lcl} x_1 & = & (b_1 - a_{12}x_2 - a_{13}x_3)/a_{11} \\ x_2 & = & (b_2 - a_{21}x_1 - a_{23}x_3)/a_{22} \\ x_3 & = & (b_3 - a_{31}x_1 - a_{32}x_2)/a_{33} \end{array}$$

Suppose $\mathbf{x}^{(k)}$ is some approximate solution. The Jacobi method computes a new approximation by

$$\begin{array}{lcl} x_1^{(k+1)} & = & (b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)})/a_{11} \\ x_2^{(k+1)} & = & (b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)})/a_{22} \\ x_3^{(k+1)} & = & (b_3 - a_{31}x_1^{(k)} - a_{32}x_2^{(k)})/a_{33} \end{array}$$

You may find matrix notation clearer. For this, we define a matrix $D = [d_{ij}]$ with the same diagonal elements as A and zero in all off-diagonal elements. The remaining elements of A we write into a matrix E.

$$A = D + E \text{ with } D = [d_{ij}], \quad d_{ij} = \begin{cases} a_{ii} & \text{if } i = j, \\ 0 & \text{else.} \end{cases} \qquad E = A - D\ . \tag{12}$$

The linear system (11) then may be transformed,

$$\begin{aligned} A\mathbf{x} &= \mathbf{b} \\ (D+E)\mathbf{x} &= \mathbf{b} \\ D\mathbf{x} &= \mathbf{b} - E\mathbf{x} \\ \mathbf{x} &= D^{-1}(\mathbf{b} - E\mathbf{x})\ . \end{aligned}$$

The last equation is in fixed-point form. The corresponding fixed-point iteration

$$\mathbf{x}^{(k+1)} = D^{-1}(\mathbf{b} - E\mathbf{x}^{(k)}) \tag{13}$$

is called *Jacobi method* .

Iteration step of the Jacobi method
In matrix notation for the splitting $A = D + E$:

$$\mathbf{x}^{(k+1)} = D^{-1}(\mathbf{b} - E\mathbf{x}^{(k)})$$

Component-wise notation for $i = 1, \ldots, n$

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

The Jacobi method does not take advantage of the most recent information to calculate $x_i^{(k+1)}$. For example, it uses $x_1^{(k)}$ when calculating $x_2^{(k+1)}$, even though the more recent approximation $x_1^{(k+1)}$ is already available. If we formulate the method in such a way that it always uses the most current values of the x_i, we get the *Gauss-Seidel method* .

For the matrix notation of the Gauss-Seidel method, we define a matrix $C = [c_{ij}]$ with the same elements as A in and below the main diagonal, and zero above the main diagonal. We write the remaining elements of A into a matrix E:

$$A = C + E \text{ with } C = [c_{ij}], \quad c_{ij} = \begin{cases} a_{ij} & \text{if } i \geq j, \\ 0 & \text{else.} \end{cases} \qquad E = A - C\ . \tag{14}$$

The same steps that led to the fixed point equation 13 for the Jacobi method, we can repeat with the matrix C instead of D and obtain the iteration rule for the Gauss-Seidel method:

$$\mathbf{x}^{(k+1)} = C^{-1}(\mathbf{b} - E\mathbf{x}^{(k)}) \tag{15}$$

Iteration step of the Gauss-Seidel method
loosely formulated
Solve each equation for its diagonal element, set initial values, iterate using the most recently calculated approximations.

Matrix notation for splitting $A = C + E$

$$\mathbf{x}^{(k+1)} = C^{-1}(\mathbf{b} - E\mathbf{x}^{(k)})$$

Component-wise notation
for $i = 1, \dots, n$

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

It is possible to accelerate the Gauss-Seidel method considerably if the new value $x_i^{(k+1)}$ is not used directly, but in combinatio with the old value in the form $\omega x_i^{(k+1)} + (1-\omega) x_i^{(k)}$ with some extrapolation factor $\omega > 1$. This iterative procedure is called the *SOR method* (SOR stands for successive overrelaxation). However, it is difficult to give a suitable value for ω. The theory says $1 \leq \omega < 2$ with values relatively close to 2. For $\omega = 1$, SOR reduces to Gauss-Seidel.

Iteration step of the SOR method
loosely formulated
For each i, calculate first an intermediate result $y_i^{(k+1)}$ via a Gauss-Seidel step; get the new value by extrapolation (overrelaxation)from old value and intermediate result: $x_i^{(k+1)} = \omega y_i^{(k+1)} + (1-\omega) x_i^{(k)}$

The component-wise notation already looks a bit confusing here.

for $i = 1, \dots, n$

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

This method can also be written with a decomposition $A = B + E$ similar to equations 13 and 15,

$$\mathbf{x}^{(k+1)} = B^{-1}(\mathbf{b} - E\mathbf{x}^{(k)}) \tag{16}$$

For SOR, B is a combination of the matrices C and D from before (12, 14) in the form

$$B = C + \left(\frac{1}{\omega} - 1 \right) D$$

4.2 Convergence Criteria for the Jacobi and Gauss-Seidel Methods

The three methods presented above will not necessarily converge for an arbitrary Matrix A. However, it is possible to prove the convergence of the Jacobi method by showing that the fixed point iteration is a contraction mapping. For this purpose, we define

An $n \times n$matrix $A = [a_{ij}]$ is called *strictly diagonally dominant* , if

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

Thus, in each row the sum of the absolute values of the non-diagonal elements must be smaller than the absolute value of the diagonal element.

Convergence of Jacobi Method
For linear systems with strictly diagonally dominant matrices the Jacobi method converges to the unique solution.

Proof: We show that the function $\mathbf{\Phi}(\mathbf{x}) = D^{-1}(\mathbf{b} - E\mathbf{x})$, which defines the iteration (13), is a contration mapping in the maximum norm for all $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$. According to section 2.4, this ensures convergence.

$$\mathbf{\Phi}(\mathbf{x}) - \mathbf{\Phi}(\mathbf{y}) = D^{-1}(\mathbf{b} - E\mathbf{x}) - D^{-1}(\mathbf{b} - E\mathbf{y}) = D^{-1}E(\mathbf{y} - \mathbf{x})$$

Row i of the matrix $D^{-1}E$ is

$$\frac{a_{i1}}{a_{ii}} \quad \frac{a_{i2}}{a_{ii}} \quad \dots \quad \frac{a_{i,i-1}}{a_{ii}} \quad 0 \quad \frac{a_{i,i+1}}{a_{ii}} \quad \dots \quad \frac{a_{in}}{a_{ii}}$$

The sum of the absolute values in this row is < 1 (diagonal dominance ensures that the nominator is smaller than the denominator). Since this holds for all rows of $D^{-1}E$, the maximum norm (infiniyty norm) fulfills

$$\|D^{-1}E\|_\infty < 1 \ .$$

A property of the norm, inequality (9), immediately gives the contraction property.

$$\|\mathbf{\Phi}(\mathbf{x}) - \mathbf{\Phi}(\mathbf{y})\|_\infty = \|D^{-1}E(\mathbf{y} - \mathbf{x})\|_\infty \leq \|D^{-1}E\|_\infty \cdot \|\mathbf{y} - \mathbf{x}\|_\infty \leq C\|\mathbf{y} - \mathbf{x}\|_\infty$$

mit $C = \|D^{-1}E\|_\infty < 1$.

With considerably more effort convergence of the Jacobi method can be shown also for a larger class of matrices (weakly diagonally dominant, irreducible matrices). This statement is important, because many real-world problems yield exactly such matrices. For the sake of completeness here are the definitions:
An $n \times n$ matrix $A = [a_{ij}]$ is *weakly diagonally dominant* , if

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

and at least for one i strict inequality holds. To check for irreducibility, you draw a point for each i. For each matrix entry $a_{ij} \neq 0$ in A you join points i and j by an arrow $i \to j$. This drawing represents a *directed graph* If you can reach, following the arrows, any point starting from any other point, then this graph is *connected*, and the corresponding matrix A is *irreducible*.

As a rule, the Gauss-Seidel method converges more rapidly than the Jacobi method. It typically needs only half as many iterations for the same accuracy. The SOR method with optimally chosen relaxation parameter ω needs $O(\sqrt{N})$ iterations, where the Jacobi method needs N iterations. However, there are also matrices for which one method converges, but the other does not. We quote here without proof two more theorems formulating convergence conditions.

If A has positive elements in the main diagonal and all other elements are ≤ 0, then the Gauss-Seidel method converges if and only if the Jacobi method converges. If both methods converge, then the Gauss-Seidel method is asymptotically faster (*theorem of Stein and Rosenberg*).

If A is symmetric positive definite, then the Gauss-Seidel method converges.

4.3 Modern Iterative Solvers

Linear systems from flow simulation, structural analysis, financial mathematics, and many other fields may easily reach a size of several million unknowns. However, only a few elements per matrix row are different from zero. (Such a matrix is called *sparse*). Today, almost exclusively iterative methods are used to solve such systems. The classical methods (Jacobi, Gauss-Seidel) converge too slowly and therefore require too much computational effort.

These notes can only give an introductory overlook to some ideas used by modern iterative solvers.

4.3.1 Matrix Splitting, Preconditioning

Let us assume you want to solve the system

$$A\mathbf{x} = \mathbf{b} \ .$$

A clever idea: You replace the matrix A by another matrix $\tilde{A}$ for which you can the linear system more quickly. You can make it easy for yourself and choose for $\tilde{A}$ the unit matrix I, or the diagonal part of A, or selectively only certain matrix elements out of A.

Write $A = \tilde{A}+E$. This is called a *splitting* of A into an approximation, called the *preconditioner*, and a residual part E. You then reformulate the original system as a fixed point problem.

$$\begin{aligned} A\mathbf{x} &= \mathbf{b} \\ (\tilde{A}+E)\mathbf{x} &= \mathbf{b} \\ \tilde{A}\mathbf{x}+E\mathbf{x} &= \mathbf{b} \\ \tilde{A}\mathbf{x} &= \mathbf{b}-E\mathbf{x} \\ \mathbf{x} &= \tilde{A}^{-1}(\mathbf{b}-E\mathbf{x}) \end{aligned}$$

The Jacobi method uses this idea with $\tilde{A} = D$, the diagonal part. Alternatively, you will get $\tilde{A}$ for the Gauss-Seidel method if you set in A all elements above the main diagonal to zero.

In general, the better $\tilde{A}$ approximates the original matrix, the faster such an iterative procedure converges.

Particularly good splittings result from incomplete LU decomposition. These methods are discussed in section 3.7.4.

However, you should not implement the method in the fixed-point form above since one should explicitly compute the matrix $\tilde{A}^{-1}$ in the simplest cases only (such as $\tilde{A} = D$). An algebraically equivalent form but suitable for computers is

Basic iterative solver with $A = \tilde{A} + E$
For a suitable splitting $A = \tilde{A} + E$, an arbitrary initial vektor $\mathbf{x}^{(0)}$ and some given accuracy threshold $\epsilon > 0$ this algorithm approximately solves $A\mathbf{x} = \mathbf{b}$.

```
start with initial vector $\mathbf{x}^{(0)}$
set $\mathbf{r}^{(0)} = \mathbf{b} - A\mathbf{x}^{(0)}$
for $k = 0, 1, 2, \ldots$
  solve $\tilde{A}\mathbf{d}^{(k+1)} = \mathbf{r}^{(k)}$
  set $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{d}^{(k+1)}$
  set $\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - A\mathbf{d}^{(k+1)}$
until $\|\mathbf{r}^{(k+1)}\| < \epsilon$
result: approximate solution $\mathbf{x}^{(k+1)}$
```

In an iterative solver of this type, $\tilde{A}$ is called the *preconditioner*.

For a vector $\mathbf{x}$ and given A and $\mathbf{b}$, the expression $\mathbf{b} - A\mathbf{x}$ is called the *residual*. Thus, solving a linear system $A\mathbf{x} = \mathbf{b}$ is equivalent to finding an $\mathbf{x}$ with a vanishing residual. One can easily verify that the vectors $\mathbf{r}^{(k)}$ in the above basic scheme are indeed the respective residuals $\mathbf{r}^{(k)} = \mathbf{b} - A\mathbf{x}^{(k)}$. Thus, the stopping criterion of the method requires the residual norm to be smaller than a given bound.

Caution! A small residual does not automatically mean that also the error $\mathbf{x}_{\text{exc}} - \mathbf{x}^{(k)}$ between exact and approximate solution is small. For example, if A is symmetric, the following bounds hold.

$$\frac{\|\mathbf{r}^{(k)}\|_2}{|\lambda_{\max}|} \leq \|\mathbf{x}_{\text{exc}} - \mathbf{x}^{(k)}\|_2 \leq \frac{\|\mathbf{r}^{(k)}\|_2}{|\lambda_{\min}|}$$

where $\lambda_{\max}$ and $\lambda_{\min}$ denote A's largest and smallest eigenvalues, respectively. Therefore, if $\lambda_{\min}$ is close to zero, a small residual does not say much about the size of the error.

Likewise, small corrections $\mathbf{d}^{(k+1)} = \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}$ will not automatically guarantee the smallness of the error. However, modern iterative methods approximate the eigenvalues with little additional effort and thus provide reliable bounds for the error.

4.3.2 Minimizing the residual to accelerate convergence

One often observes with the above basic scheme that the vectors $\mathbf{d}^{(k)}$, by which the approximate solution vectors change per iteration, point in the right direction but not with the correct magnitude. Instead of changing the vector $\mathbf{x}^{(k)}$ by only the vector $\mathbf{d}^{(k+1)}$ per iteration step, one can therefore try to apply a multiple ω of this correction. (The SOR method already used a similar approach.)

If the approximation vector changes by $\omega\mathbf{d}^{(k+1)}$ at the step from k to $k+1$, then it is easy to see that the residual vector changes by $-\omega A\mathbf{d}^{(k+1)}$. Accordingly, one changes the basic scheme and sets

$$\begin{aligned}\mathbf{x}^{(k+1)} &= \mathbf{x}^{(k)} + \omega\mathbf{d}^{(k+1)} \\ \mathbf{r}^{(k+1)} &= \mathbf{r}^{(k)} - \omega A\mathbf{d}^{(k+1)}\end{aligned}$$

At each step, one chooses ω so that the magnitude $\|\mathbf{r}^{(k+1)}\|$ becomes as small as possible. How does this work? The usual procedure when looking for an extreme value is to differentiate and set the derivative to zero. (Here, $\|\cdot\|$ always denotes the 2-norm, that is, the Euclidean length of a vector.)

$$\begin{aligned}
\|\mathbf{r}^{(k+1)}\|^2 &= (\mathbf{r}^{(k+1)} \cdot \mathbf{r}^{(k+1)}) \\
&= \Big((\mathbf{r}^{(k)} - \omega A\mathbf{d}^{(k+1)}) \cdot (\mathbf{r}^{(k)} - \omega A\mathbf{d}^{(k+1)})\Big) \\
&= \Big(\mathbf{r}^{(k)} \cdot \mathbf{r}^{(k)} - 2\omega(\mathbf{r}^{(k)} \cdot A\mathbf{d}^{(k+1)}) + \omega^2(A\mathbf{d}^{(k+1)} \cdot A\mathbf{d}^{(k+1)})\Big)
\end{aligned}$$

The individual inner products are all just scalar constants. Differentiation with respect to ω and setting the derivative to zero yields

$$\begin{aligned}
0 &= \frac{d}{d\omega}\|\mathbf{r}^{(k+1)}\|^2 \\
&= \frac{d}{d\omega}\Big(\mathbf{r}^{(k)} \cdot \mathbf{r}^{(k)} - 2\omega(\mathbf{r}^{(k)} \cdot A\mathbf{d}^{(k+1)}) + \omega^2(A\mathbf{d}^{(k+1)} \cdot A\mathbf{d}^{(k+1)})\Big) \\
&= -2(\mathbf{r}^{(k)} \cdot A\mathbf{d}^{(k+1)}) + 2\omega(A\mathbf{d}^{(k+1)} \cdot A\mathbf{d}^{(k+1)}) \quad \text{, daraus} \\
\omega &= \frac{\mathbf{r}^{(k)} \cdot A\mathbf{d}^{(k+1)}}{A\mathbf{d}^{(k+1)} \cdot A\mathbf{d}^{(k+1)}}
\end{aligned}$$

4.3.3 Orthogonalizing search directions to accelerate convergence

We have set

$$\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \omega A\mathbf{d}^{(k+1)}$$

where ω is chosen optimally so that it minimizes the norm $\|\mathbf{r}^{(k+1)}\|$. Thus, any further change of the residual vector in direction $A\mathbf{d}^{(k+1)}$ will increase its norm. Now, if in the next iteration

$$\mathbf{r}^{(k+2)} = \mathbf{r}^{(k+1)} - \omega A\mathbf{d}^{(k+2)}$$

the correction $A\mathbf{d}^{(k+2)}$ contains a component in direction $A\mathbf{d}^{(k+1)}$, any amount of correction in that direction will deteriorate the approximation.

Therefore: If the residual is already minimized along some direction, it should not be changed in this direction any more. So we need a method that removes the undesired component from the new search direction $A\mathbf{d}^{(k+2)}$. This can be achieved by *orthogonalization.*

Let $\mathbf{p}$ and $\mathbf{q}$ be two vectors $\neq 0$. The component of $\mathbf{p}$ in direction $\mathbf{q}$ is given by

$$\left(\frac{\mathbf{p} \cdot \mathbf{q}}{\mathbf{q} \cdot \mathbf{q}}\right) \mathbf{q}$$

Thus, the vector

$$\mathbf{p} - \left(\frac{\mathbf{p} \cdot \mathbf{q}}{\mathbf{q} \cdot \mathbf{q}}\right) \mathbf{q}$$

no longer contains a component in direction $\mathbf{q}$, i.e., it is orthogonal to $\mathbf{q}$. (Special case: if $\mathbf{p}$ is a scalar multiple of $\mathbf{q}$, this calculation gives the zero vector.)

This procedure can successively remove from $\mathbf{p}$ components of several vectors.

Orthogonalization

Given m nonzero vectors $\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_m \in \mathbb{R}^n$ and a vector $\mathbf{p} \in \mathbb{R}^n$. This algorithm removes from $\mathbf{p}$ all components in directions $\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_m$.

```
for $i = 1 \ldots m$
  compute inner product $r_i = \mathbf{p} \cdot \mathbf{q}_i / \mathbf{q}_i \cdot \mathbf{q}_i$
  subtract component by $\mathbf{p} = \mathbf{p} - r_i \mathbf{q}_i$
```

An early application of these ideas (preconditioning, minimization, orthogonalization) is the ORTHOMIN algorithm (published in 1976 by P. K. W. Vinsome, then employed by a petroleum company). Since then, many methods based on similar principles have been developed. They all belong to the class of what is now called *Krylov-subspace methods*.

A very elegant and powerful method exists for symmetric positive definite matrices, the conjugate gradient method (Hestenes and Stiefel, 1952). Combined with suitable preconditioning, it is no routinely used for large linear systems.

For unsymmetric matrices, GMRES (for *generalized minimal residual method*), BiCG (for *biconjugate gradients*) or CGS (for *conjugate gradient squared*) are common methods.

5 Overdetermined Systems

A linear system $A\mathbf{x} = \mathbf{b}$ with more equations than unknowns is called *overdetermined.*

In such a case, A is a rectangular $n \times m$ matrix with $n > m$, i.e., more rows than columns. In general, such a system has no exact solution, but a uniquely determined "least wrong" approximate solution, the *least-squares solution.*

Regular case: m linearly independent columns in A, $m + 1$ linearly independent columns in the extended coefficients matrix$[A, \mathbf{b}]$.
Special cases:

- $\operatorname{rank} A = \operatorname{rank}[A, \mathbf{b}] = m \rightarrow$ a unique exact solution.

Systems without full column rank; the normal equations described below are not uniquely solvable.

- $\operatorname{rank} A = \operatorname{rank}[A, \mathbf{b}] < m \rightarrow$ infinitely many exact solutions;
- $\operatorname{rank} A < \operatorname{rank}[A, \mathbf{b}] < m \rightarrow$ no exact solution, infinitely many least-squares solutions.

Overdetermined systems occur, for example, in estimating parameters from experimental data. Typically, more measurements are available than parameters are searched for.

5.1 The Normal Equations

Overdetermines systems
Least squares solution: find that $\mathbf{x}$ which minimizes the Euclidean norm of the residual vector

$$\mathbf{r} = \mathbf{b} - A\mathbf{x} \,.$$

Leads to the *normal equations*

$$A^T A\mathbf{x} = A^T\mathbf{b}$$

Derivation 1: Differentiate $(\|\mathbf{r}\|_2)^2 = \mathbf{r}^T \cdot \mathbf{r}$ with respect to the components of $\mathbf{x}$; set the derivatives to zero.

Derivation 2: Assume that $\hat{\mathbf{x}}$ is a vector that fulfils $A^T(\mathbf{b} - A\hat{\mathbf{x}}) = 0$ (i.e., the normal equations). We will show: for every other vector $\mathbf{x} \neq \hat{\mathbf{x}}$ holds

$$\|\mathbf{b} - A\mathbf{x}\|_2 \geq \|\mathbf{b} - A\hat{\mathbf{x}}\|_2 \,.$$

This means that for no other vector a smaller residual exists[7]. In this sense $\hat{\mathbf{x}}$ is the optimal approximate or least-wrong solution to $A\mathbf{x} = \mathbf{b}$.

Proof. Set $\hat{\mathbf{r}} = \mathbf{b} - A\hat{\mathbf{x}}$ and $\mathbf{r} = \mathbf{b} - A\mathbf{x}$. Then, we may write

$$\mathbf{r} = \mathbf{b} - A\mathbf{x} = (\mathbf{b} - A\hat{\mathbf{x}}) + A(\hat{\mathbf{x}} - \mathbf{x}) = \hat{\mathbf{r}} + A(\hat{\mathbf{x}} - \mathbf{x})$$

With this definition, we calculate the inner product $\mathbf{r}^T \cdot \mathbf{r}$ and how it relates to $\hat{\mathbf{r}}^T \cdot \hat{\mathbf{r}}$.

$$\begin{aligned}
\mathbf{r}^T \cdot \mathbf{r} &= (\hat{\mathbf{r}} + A(\hat{\mathbf{x}} - \mathbf{x}))^T \cdot (\hat{\mathbf{r}} + A(\hat{\mathbf{x}} - \mathbf{x})) \\
&= \hat{\mathbf{r}}^T \cdot \hat{\mathbf{r}} + \hat{\mathbf{r}}^T \cdot (A(\hat{\mathbf{x}} - \mathbf{x})) + (A(\hat{\mathbf{x}} - \mathbf{x}))^T \cdot \hat{\mathbf{r}} + (A(\hat{\mathbf{x}} - \mathbf{x}))^T \cdot (A(\hat{\mathbf{x}} - \mathbf{x})) \\
&= \hat{\mathbf{r}}^T \cdot \hat{\mathbf{r}} + (\hat{\mathbf{r}}^T A) \cdot (\hat{\mathbf{x}} - \mathbf{x})) + (\hat{\mathbf{x}} - \mathbf{x})^T \cdot (A^T\hat{\mathbf{r}}) + (A(\hat{\mathbf{x}} - \mathbf{x}))^T \cdot (A(\hat{\mathbf{x}} - \mathbf{x}))
\end{aligned}$$

[7] In the regular case, all columns of A are linearly independent. Then strict inequality $\|\mathbf{b} - A\mathbf{x}\|_2 > \|\mathbf{b} - A\hat{\mathbf{x}}\|_2$ holds and $\hat{\mathbf{x}}$ is the uniquely determined least-squares solution.

Since by our assumptions, $\hat{\mathbf{x}}$ fulfills the normal equations, $A^T\hat{\mathbf{r}} = 0$, and, similarly, $\hat{\mathbf{r}}^T A = 0$. It remains

$$\mathbf{r}^T \cdot \mathbf{r} = \hat{\mathbf{r}}^T \cdot \hat{\mathbf{r}} + (A(\hat{\mathbf{x}} - \mathbf{x}))^T \cdot (A(\hat{\mathbf{x}} - \mathbf{x})) \ .$$

The last term, being an inner product of a vector with itself, is always ≥ 0[8]. This proves the statement.

Derivation 3 by geometric intuition for the case $\mathbf{x} \in \mathbb{R}^2$ and A a 3×2 matrix. The vectors in the set $\{A\mathbf{x} : \mathbf{x} \in \mathbb{R}^2\}$ span a plane in space (if the columns of A are linearly independent). On this plane, let $A\hat{\mathbf{x}}$ be the point with minimal distance to $\mathbf{b}$. The minimum is achieved for a distance vector standing orthogonal to the plane. Consequently, the residual vector $\mathbf{b} - A\hat{\mathbf{x}}$ is orthogonal to all vectors in the plane, that is, orthogonal to all vectors of type $A\mathbf{x}$. From

$$\forall \mathbf{x} : \quad (A\mathbf{x})^T \cdot (\mathbf{b} - A\hat{\mathbf{x}}) = \mathbf{x}^T \cdot (A^T(\mathbf{b} - A\hat{\mathbf{x}})) = 0$$

follows $A^T(\mathbf{b} - A\hat{\mathbf{x}}) = 0$, because the only vector orthogonal to all $\mathbf{x} \in \mathbb{R}^2$ is the zero vector.

5.2 Alternative Methods

The solution of overdetermined systems via the normal equations is the standard classical method but not necessarily the most favorable algorithm. It is merely the only one that, for small examples, can be done manually with familiar methods (matrix multiplication, elimination). Program packages and computational environments (like MATLAB) mostly use (some fine-tuned versions of) the QR decomposition. For systems without full column rank (be they overdetermined or not), the singular value decomposition (SVD) is advantageous. In this case, there are infinitely many solutions (exact or in the sense of the least squares), and SVD automatically yields the smallest one. On the other hand, the solutions with the most zero components can be read from the QR decomposition.

5.3 Examples of Overdetermined Systems

Given three linear equations in two unknowns,

$$\begin{array}{rcrcr} 2x & + & y & = & 19 \\ -4x & + & 4y & = & 13 \\ 4x & - & y & = & 17 \end{array}$$

In matrix-vector notation, this system reads

$$A\mathbf{x} = \mathbf{b} \text{ with } A = \begin{bmatrix} 2 & 1 \\ -4 & 4 \\ 4 & -1 \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 19 \\ 13 \\ 17 \end{bmatrix}$$

[8]If all columns of A are linearly independent then for $\mathbf{x} \neq \hat{\mathbf{x}}$ also $A(\hat{\mathbf{x}} - \mathbf{x}) \neq 0$; the last term therefore is strictly > 0.

Method of the Normal Equations

Form $A^T \cdot A$ and $A^T\mathbf{b}$:

$$A^T \cdot A = \begin{bmatrix} 2 & -4 & 4 \\ 1 & 4 & -1 \end{bmatrix} \cdot \begin{bmatrix} 2 & 1 \\ -4 & 4 \\ 4 & -1 \end{bmatrix} = \begin{bmatrix} 36 & -18 \\ -18 & 18 \end{bmatrix} = 18 \begin{bmatrix} 2 & -1 \\ -1 & 1 \end{bmatrix}$$

$$A^T\mathbf{b} = \begin{bmatrix} 2 & -4 & 4 \\ 1 & 4 & -1 \end{bmatrix} \cdot \begin{bmatrix} 19 \\ 13 \\ 17 \end{bmatrix} = \begin{bmatrix} 54 \\ 54 \end{bmatrix} = 18 \begin{bmatrix} 3 \\ 3 \end{bmatrix}$$

The normal equations (divided by 18 for simplicity) therefore are

$$\begin{bmatrix} 2 & -1 \\ -1 & 1 \end{bmatrix} \mathbf{x} = \begin{bmatrix} 3 \\ 3 \end{bmatrix} \text{, or written more explicitly} \quad \begin{array}{rcrcl} 2x & - & y & = & 3 \\ -x & + & y & = & 3 \end{array}$$

Adding the two equations yields $x = 6$ immediately, and from there, by substitution, $y = 9$.

Minimal error in various norms

The normal equations give $\mathbf{x} = [6; 9]$. Substituting these values in the original equations will show that this "solution" satisfies none of the three equations. The residual vector $\mathbf{r} = \mathbf{b} - A\mathbf{x}$ is

$$\begin{bmatrix} 19 \\ 13 \\ 17 \end{bmatrix} - \begin{bmatrix} 2 & 1 \\ -4 & 4 \\ 4 & -1 \end{bmatrix} \begin{bmatrix} 6 \\ 9 \end{bmatrix} = \begin{bmatrix} -2 \\ 1 \\ 2 \end{bmatrix}$$

The residual vector here has length 3, the minimally possible length—no other $\mathbf{x}$ will give a smaller residual vector. Vector size here is the Euclidean length, i. e., the 2-norm.

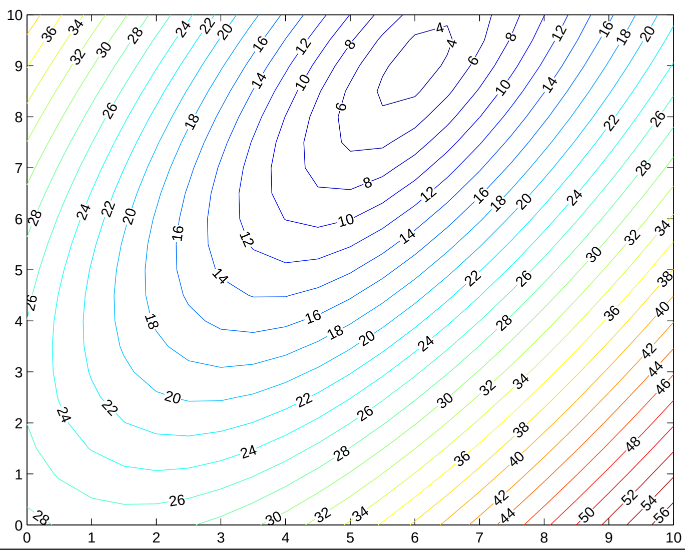
A small change in the vector $\mathbf{x}$, such as

$$\mathbf{x} = \begin{bmatrix} 6 \\ 8.8 \end{bmatrix} = \begin{bmatrix} 6 \\ 44/5 \end{bmatrix}$$

results in a residual vector

$$\mathbf{r} = \mathbf{b} - A\mathbf{x} = \begin{bmatrix} -9/5 \\ 9/5 \\ 9/5 \end{bmatrix} = \begin{bmatrix} -1.8 \\ 1.8 \\ 1.8 \end{bmatrix}$$

with Euclidean length $\|\mathbf{r}\|_2 = \frac{9}{5}\sqrt{3} = 3.1177$, clearly larger than the optimal value.


Isolines for the length of the residual vector. The minimum at $[6; 9]$ is clearly visible.

However, in the infinity norm (maximum absolute value among components), the error now is smaller: 1.8 instead of 2.

Let us try another vector $\mathbf{x}$:

$$\mathbf{x} = \begin{bmatrix} 6.15 \\ 9.4 \end{bmatrix}, \quad \text{corresponding residual vector } \mathbf{r} = \mathbf{b} - A\mathbf{x} = \begin{bmatrix} -2.7 \\ 0 \\ 1.8 \end{bmatrix}$$

$$\|\mathbf{r}\|_2 = 3.2450, \quad \|\mathbf{r}\|_\infty = 2.7$$

This vector is larger than the previous ones in the 2- and the ∞-norm. However, the sum of the magnitude of all components (the 1-norm) here is 4.5. The 1-norms of two other vectors are 5 and 5.4, respectively. Thus, in the 1-norm, the third choice of $\mathbf{x}$ is optimal.

Solution via QR-decomposition of A

$$A = Q \cdot R \text{ , in this example } \begin{bmatrix} 2 & 1 \\ -4 & 4 \\ 4 & -1 \end{bmatrix} = \begin{bmatrix} -1/3 & 2/3 & -2/3 \\ 2/3 & 2/3 & 1/3 \\ -2/3 & 1/3 & 2/3 \end{bmatrix} \cdot \begin{bmatrix} -6 & 3 \\ 0 & 3 \\ 0 & 0 \end{bmatrix}$$

The transformed system $R\mathbf{x} = Q^T\mathbf{b}$ is also overdetermined; written explicitly it reads

$$\begin{array}{rcrcr} -6x & + & 3y & = & -9 \\ & & 3y & = & 27 \\ & & 0 & = & 3 \end{array}$$

Solving the first two equations is straightforward because of their triangular form. Moreover, this solution will make the first two components of the residual vector zero. The last equation, however, does not depend on $\mathbf{x}$. No choice of $\mathbf{x}$ can change this equation's contribution to the residual vector.

Thus, in the transformed system $R\mathbf{x} = Q^T\mathbf{b}$, the solution from the first two equations is optimal. It is also the optimal one for the original system $A \cdot \mathbf{x} = \mathbf{b}$ because the transformation by the orthogonal matrix Q leaves the norm of the residual vector invariant.

Another example using singular value decomposition

Another overdetermined system is given here,

$$A\mathbf{x} = \mathbf{b} \text{ with } A = \begin{bmatrix} 14 & -2 \\ -4 & 22 \\ 16 & -13 \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} -80 \\ 40 \\ -145 \end{bmatrix}$$

The singular value decomposition for this matrix A is

$$A = U \cdot S \cdot V^T \ , \quad \begin{bmatrix} 14 & -2 \\ -4 & 22 \\ 16 & -13 \end{bmatrix} = \begin{bmatrix} -1/3 & -2/3 & -2/3 \\ 2/3 & -2/3 & 1/3 \\ -2/3 & -1/3 & 2/3 \end{bmatrix} \cdot \begin{bmatrix} 30 & 0 \\ 0 & 15 \\ 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} -3/5 & -4/5 \\ 4/5 & -3/5 \end{bmatrix}^T$$

The transformed system, in this case, is

$$S \cdot \mathbf{y} = U^T \cdot \mathbf{b} \ , \quad \begin{bmatrix} 30 & 0 \\ 0 & 15 \\ 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 150 \\ 75 \\ -30 \end{bmatrix}$$

Because S is a diagonal matrix, the optimal solution is easily determined: $\mathbf{y} = [5; 5]$. The corresponding solution for the original system is

$$\mathbf{x} = V \cdot \mathbf{y} = \begin{bmatrix} -3/5 & -4/5 \\ 4/5 & -3/5 \end{bmatrix} \cdot \begin{bmatrix} 5 \\ 5 \end{bmatrix} = \begin{bmatrix} -7 \\ 1 \end{bmatrix}$$

If the QR decomposition or the singular value decomposition is already available, the steps to solve a system of equations are relatively straightforward. The more difficult task, however, is to compute these decompositions. In this course, we cannot go into the details.

MATLAB commands

QR decomposition of a matrix A:	`[Q R]=qr(A)`
SVD, singular value decomposition $A = U \cdot S \cdot V^T$	`[U S V]=svd(A)`
Least-squares solution of an overdetermined linear system	
...by QR decomposition	`A\b`
...by SVD	`pinv(A)*b`

5.4 Fitting a Linear Model (a Plane) to Data Points in Space

This section uses another example to explain the optimal solution of overdetermined systems. It overlaps partially with Chapter 6, which discusses further methods for approximating multi-dimensional data.

For the (reasonably) realistic numerical values in this example, the comparison between the method of the normal equation and the QR decomposition also shows very clearly the different orders of magnitude in the intermediate results and the resulting influence of the rounding errors.

For the Blies (a tributary of the Saar), the flood levels at the Neunkirchen gauge are to be predicted from the water levels at the Ottweiler and Hangard gauges. The data of the peak water levels of 12 winter floods from the years 1963–1971 are available:

Wasserstand in cm												
Neunkirchen y	172	309	302	283	443	298	319	419	361	267	337	230
Ottweiler x_1	93	193	187	174	291	184	205	260	212	169	216	144
Hangard x_2	120	258	255	238	317	246	265	304	292	242	272	191

Daten-Quelle: U. Maniak, Hydrologie und Wasserwirtschaft, Springer, 1988

Let us assume that this data set follows a linear model $a_0 + a_1x_1 + a_2x_2 = y$. In this model, a_0, a_1, and a_2 are unknown coefficients that should be determined from the twelve data triplets. In a geometric interpretation, these triplets $(x_1|x_2|y)$ are points in space. The linear model then corresponds to a plane fitting the data points as well as possible.

Inserting the data values in the model equation yields

$$\begin{array}{ccccccc} a_0 & + & 93a_1 & + & 120a_2 & = & 172 \\ a_0 & + & 193a_1 & + & 258a_2 & = & 309 \\ a_0 & + & 144a_1 & + & 191a_2 & = & 230 \\ \vdots & & & & & & \vdots \\ a_0 & + & 187a_1 & + & 255a_2 & = & 302 \end{array}$$

This is an overdetermined linear system $A\mathbf{x} = \mathbf{b}$ with 12×3 matrix A and right-hand side $\mathbf{b}$,

$$A = \begin{bmatrix} 1 & 93 & 120 \\ 1 & 193 & 258 \\ 1 & 187 & 255 \\ \vdots & \vdots & \vdots \\ 1 & 144 & 191 \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} 172 \\ 309 \\ 302 \\ \vdots \\ 230 \end{bmatrix}$$

The classic standard method of least squares uses the normal equations $A^T \cdot A\mathbf{x} = A^T\mathbf{b}$, in this case,

$$\begin{bmatrix} 12 & 2328 & 3000 \\ 2328 & 480202 & 609985 \\ 3000 & 609985 & 780572 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 3740 \\ 766996 \\ 975996 \end{bmatrix}$$

The numerical values in this system vary greatly, ranging in order of magnitude from 10 to 10^6. However, the original data primarily falls within the low three-digit range. This significant jump in magnitude is typical for normal equations, indicating that this system will be sensitive to rounding errors.

The QR decomposition is numerically more advantageous, but it is not feasible to work it out with pen on paper for real-world data. However, with a computer, it is the preferred method. The transformed system $R\mathbf{x} = Q^T\mathbf{b}$ here is as follows.

$$\begin{bmatrix} -3.4641 & -672.0357 & -866.0254 \\ 0 & -169.0266 & -165.5656 \\ 0 & 0 & -56.2141 \\ 0 & 0 & 0 \\ \vdots & \vdots & \vdots \\ 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} -1079.6 \\ -245.14 \\ -7.2659 \\ -2.4029 \\ \vdots \\ -11.003 \end{bmatrix}$$

Unlike before, the matrix entries are now of the same order of magnitude as the original data.

The first three equations are in upper triangular form and can be solved exactly by back substitution. The remaining nine equations of the form $0 = -2.4029, \ldots$ are hopelessly unsolvable. They cause the nonzero components in the residual vector.

Loosely speaking: QR decomposition transforms an overdetermined system into an exactly solvable system in triangular form and an unsolvable remainder. Ignoring the unsolvable equations provides the best possible solution in terms of least squares.

Note: It is immediately apparent that the above recipe provides the solution with a minimal residual vector for the *transformed* system. However, the *original system* has a different residual vector. The real point of the procedure is that the transformation from one residual vector to the other is achieved by multiplication with an *orthogonal matrix.* Multiplication with an orthogonal matrix leaves the magnitude of a vector unchanged. Therefore, the residual vector of the original system also has a minimum magnitude.

In MATLAB, the standard equation solver command `A\b` automatically uses the QR method in the case of an overdetermined system. Here, it gives

```
>> A\b
ans =
   22.5505
    1.3237
    0.1293
```

The more complex solution with singular value decomposition (MATLAB: `pinv(A)*b`) gives the same result.

Therefore, the model for predicting the flood peak values is

$$y = 22.5505 + 1.3237x_1 + 0.1293x_2$$

It can be seen that the x_2 data have a good ten times less influence on the prediction model. (because the corresponding coefficient in the linear model is only 0.1293 compared to 1.3237). One possible interpretation would be that the water level in Neunkirchen depends mainly on the Ottweiler gauge and not really on the Hangard gauge.

How reliable is this model? Inserting the data points provides the residual vector.

Measured value y	172	309	302	283	443	298	319	419	361	267	337	230
Model prediction	161	311	303	284	449	298	328	406	341	278	344	238
Residual	11	-2	-1	-1	-6	0	-9	13	20	-11	-7	-8

The mean absolute error is 7.3 cm, but the maximum error is 20 cm.

A more precise assessment of the model concerning

- the justification of the assumed linear correlation,
- the impact of the individual model variables,
- the probability that the forecast value is accurate to a certain extent,
- the possibility of outliers,

requires methods of multivariate statistics and regression analysis (and probably also considerably more data).

6 Approximation of Data, Curve Fitting

6.1 Linear Data Models

An example of this has already been presented in Chapter 5.4. Lecture slides and exercises provide further examples. The best-fit model always results from the least squares approximation to an overdetermined system of equations.

However, the method of least squares doesn't always provide a plausible fit. A few grossly incorrect values in the data ("outliers") can significantly distort the result. Chapter 6.6 introduces a robust method. MATLAB offers various methods for *robust fitting* in its toolboxes.

6.2 Polynomial Regression

This is an important special case of linear data models where the formulas are somewhat simpler than in the general case.

A typical task for this section might be: Given measurement results for a set of temperature values T and the corresponding resistance values R of a temperature sensor. The relationship between R and T can be approximated by $R = a + bT + cT^2$. When data for precisely three (different) T values are available, the three parameters a, b, and c can be uniquely determined. However, it's sensible to conduct more measurements to reduce the impact of observational errors. The parameters a, b, and c are then determined by *curve fitting* (also called *regression*).

Polynomial Regression (Polynomial Fitting)
Given: $m+1$ pairs of values $(x_i, y_i), i = 0, \dots, m$
Wanted: $p(x)$, a polynomial of degree $n < m$, so that the sum of squared errors

$$\sum_{i=0}^{m} (p(x_i) - y_i)^2$$

is minimized. Loosely put: $y = p(x)$ approximates the data points as closely as possible.

Not always is a polynomial approach a suitable model, but often a seemingly more complex model can be reduced to a polynomial one (see the examples in the exercise section).

Direct approach: set up a polynomial with undetermined coefficients,

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x_n \ .$$

Substitute the given pairs of values for x and y; this leads to a system of $m+1$ linear equations in the $n+1$ unknown coefficients $a_0, a_1, \dots, a_n$.

If $n < m$, it's an *overdetermined system*.

- Classical solution: Approximation using the method of normal equations: manageable by hand for small examples; risk of rounding errors with large datasets.

  However, the normal equations are uniquely solvable only if at least $n+1$ of the x-values in the total of $m+1$ value pairs are different.

- Modern approach: QR decomposition. Practically feasible only on a computer. Better condition number, less susceptible to rounding errors.

This solution approach (assumption of undetermined coefficients, formulation of the overdetermined system, forming the normal equations) can be somewhat shortened for polynomial regression. If we set

$$s_0 = m + 1, \quad t_0 = \sum_{i=0}^{m} y_i$$

and

$$s_k = \sum_{i=0}^{m} x_i^k, \quad t_k = \sum_{i=0}^{m} x_i^k y_i \quad \text{für } k > 0,$$

we can express the normal equations, as can be easily derived, in the following form.

$$\begin{array}{ccccccccc} s_0a_0 & + & s_1a_1 & + & \ldots\ldots & + & s_na_n & = t_0 \\ s_1a_0 & + & s_2a_1 & + & \ldots\ldots & + & s_{n+1}a_n & = t_1 \\ & & & & \ldots\ldots & & & \\ s_na_0 & + & s_{n+1}a_1 & + & \ldots\ldots & + & s_{2n}a_n & = t_n \end{array}$$

The normal equations can be poorly conditioned for larger n. Better results in such cases can be achieved by using orthogonal polynomial expansions.
For example, using data points

$$(x, \exp(x)) \quad \text{for } x = 0;\ 0.01;\ 0.02; \ldots; 3.99; 4$$

and approximating the data with regression polynomials of various degrees (calculating with fourteen-digit precision), the accuracy of the approximation initially improves with increasing degree. However, from the thirteenth degree onwards, errors start to increase again. The polynomial of the twenty-fifth degree computed from the normal equations bears hardly any resemblance to the approximated function. Using orthogonal polynomials (Chebyshev polynomials), these numerical problems do not occur.

6.3 Line Fitting

An important special case of the above problem: Given $m + 1$ (more than two) data points, fitting a straight line with equation $y = a + bx$ to minimize the sum of squared errors in vertical direction.

$$a = \frac{s_2t_0 - s_1t_1}{s_0s_2 - s_1^2}$$

$$b = \frac{s_0t_1 - s_1t_0}{s_0s_2 - s_1^2}$$

Finding a best-fit straight line is sometimes also referred to as “simple linear regression.” However, keep in mind that the polynomial models from the previous section are also instances of “linear regression”. The adjective “linear” does not refer to the shape of the resulting function but to the fact that the model equations are *linear in the unknown parameters.*

6.4 Nonlinear Data Models

There is material on lecture slides and more detailed information in the exercise materials. Short version: Formulate the Jacobian matrix and iterate. In contrast to the Newton method you might already know, now the linear system with the Jacobian matrix is overdetermined and is approximately solved in the least squares sense.

This method is called the *Gauss-Newton method*.

In addition to the Gauss-Newton method for nonlinear fitting problems, there are other methods (e. g., Levenberg-Marquardt algorithm). This falls under the domain of optimization as a part of Applied Mathematics. This script does not delve deeper into these methods.

6.5 Why “Least Squares?”

The method of least squares minimizes the Euclidean length of the residual vector. However, one can measure the size of the residual vector in different ways and demand different minimal conditions accordingly. Important examples include: minimizing the sum of the *absolute values* of errors or minimizing the *maximum error* (‘Minimax-Approximation”). Two reasons advocate for the least squares method:

- Simple derivation and execution: the minimization problem can be solved with elementary differential calculus and leads to a simple algebraic problem.
- Statistical considerations: When the data are contaminated with independent, normally distributed random errors with the same standard deviation, least squares are optimal in a certain sense (more precisely: The method provides a *maximum likelihood* estimation of the parameters). Conversely, if the errors in the data are *not* normally distributed, least squares might be quite poor; see below.
- Further statistical properties: If an estimation of the accuracy of the data is available, one can infer the accuracy of the calculated model parameters.

  Assuming the data is scaled such that the variance of measurement errors equals 1. Let $C = (A^T A)^{-1}$ be the inverse matrix of the normal equations system. The diagonal elements of C represent the variances of the corresponding model parameters; the elements outside the main diagonal represent the corresponding covariances.

And what argues against it?

- The method reacts sensitively to ”outliers” in the data. Squaring the errors penalizes large deviations severely. Therefore, the least squares method is willing to distort a curve wildly to approximately reach a few data points that lie far outside. A few outliers in an otherwise reasonable dataset can cause a completely nonsensical approximation.

6.6 Line Fitting by Minimizing Absolute Deviation (Minimizing the 1-Norm)

Given: m pairs of values $(x_i, y_i), i = 1, \ldots, m$

Wanted: A straight line in the form $y = a + bx$ such that the sum of absolute errors

$$\sum_{i=1}^{m} |(a + bx_i) - y_i|$$

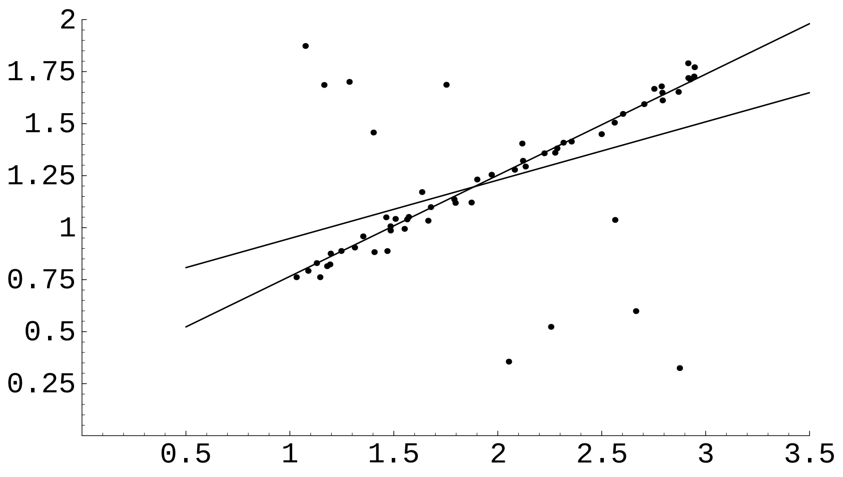


Figure 8: Fitting a line to data points. A few outliers can strongly deflect the line determined by the least squares method. Minimizing absolute errors results in a much more plausible line through the data.

is minimized.

The solution is[9] : For any given slope b, the parameter a is the median of a data field,

$$a = \text{median}\{y_i - bx_i\}$$

The parameter b is found as a solution to the equation

$$0 = \sum_{i=1}^{m} x_i \, \text{sgn}(y_i - a - bx_i)$$

(where sgn(0) should be interpreted as zero). If one substitutes a in this equation with the function $a(b)$ determined by the previous equation, one equation remains with one unknown. Interval bisection (see Chapter 1.7) is the appropriate solution method for this.

6.7 Line Fitting by Minimizing Normal Distances (Total Least Squares Method)

Material on this topic, titled *Total Least Squares*, is available in the lecture slides and exercise materials.

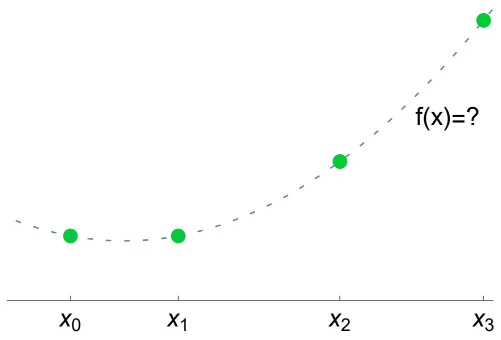
[9]For a more detailed explanation, see Section 15.7.3 in *Numerical Recipes: The Art of Scientific Computing*, Third Edition, by W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. Version 3.04 (2011).

7 Interpolation

Interpolation, Summary
Given: Data points
Wanted:

- A function that *passes through* the given data points.
- A value *between* the data points.
- Trend beyond the given data range: *Extrapolation* .

Applications: Intermediate values in tables, “nice-looking” curves for graphics…



If the values of a function f are given for a set of x-values, i.e., $f(x_i) = y_i$ for $i = 0, 1, \ldots, n$, but there is no analytical (formulaic) expression for f, one can instead look for a suitable, preferably simple function that takes the corresponding function values y_i for all given x_i values – called the *support points* or *knots* .

With the help of this function, values at any other points x can then be easily calculated. If you are looking for a function value for an x between x_0 and x_n, this is called *interpolation* ; if x is outside the given data range, the task is called *extrapolation* and is considerably riskier (as many pandemic forecasting gurus can now confirm).

Note the difference to the approximation tasks of the previous section: *Interpolation* establishes a function *through* given data points; *Approximation* finds a function that approximates the data points as closely as possible.

Older textbooks bring a multitude of interpolation formulas, associated with famous names of people who have long been dead (Newton, Bessel, Gauss, Lagrange…). Take this as an indication that interpolation methods were once of great importance. At a time when mathematical functions had to be taken from tables, trigonometric calculations were constantly associated with interpolation in tables. However, ten-dollar pocket calculators now offer trigonometric functions, and even less elementary functions are available in common computational environments. Interpolation in tables is not as important as it used to be.
Interpolation and extrapolation are still important as aids in solving other mathematical problems (numerical integration, numerical solution of differential equations).
Interpolation is very important in computer graphics and computer-aided design: your favorite hero in a video game exists as a set of data points; to make her look good on the screen, the computer draws attractive curves through the data. Usually, these are som sort of splines.

7.1 Polynomial Interpolation

Through two points, exactly one straight line passes. Through any three points, a parabola can be uniquely determined. In general: Through $n + 1$ (different) points, a polynomial of degree n is uniquely determined.

Polynomial Interpolation, Problem Description:
Given: $n+1$ value pairs $(x_i, y_i), i = 0, \dots, n$, where the x_i are pairwise distinct.
Wanted: the uniquely determined nth-degree polynomial p that passes through the given data points:

$$p(x_i) = y_i \quad \text{for } i = 0, \dots, n$$

Or: not seeking the interpolating polynomial itself, but only its value at a given point x.

Solution with the sledgehammer method: *Ansatz* of the polynomial with undetermined coefficients in standard form,

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n \ .$$

Inserting the given value pairs leads to a system of $n+1$ linear equations in the $n+1$ unknown coefficients $a_0, a_1, \dots, a_n$.

In matrix-vector form, the equations $p(x_i) = y_i$ have a simple structure:

$$\begin{bmatrix} 1 & x_0 & x_0^2 & x_0^3 & \dots & x_0^n \\ 1 & x_1 & x_1^2 & x_1^3 & \dots & x_1^n \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 1 & x_n & x_n^2 & x_n^3 & \dots & x_n^n \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}$$

Such a matrix, in whose columns the x_i are raised to powers $0, 1, 2 \dots$ in order, is called a *Vandermonde matrix* .

The good news: If all x_i values are different, this matrix is not singular, and the system of equations has a unique solution.

The bad news: The Vandermonde matrix can have a very high condition number. For polynomials of higher degrees, significant rounding errors occur. The computational effort to solve the system of equations increases with $O(n^3)$. All numerical analysis textbooks recommend alternative, more efficient methods. However, for small n and a fast computer, these arguments are not relevant. Even MATLAB's `polyfit` command uses this approach.

Furthermore, the interpolation polynomial can be explicitly written in a simple and elegant way. You are probably familiar with the formula from introductory mathematics courses:

Lagrange Interpolatig Polynomial
The interpolation polynomial passing through the $n+1$ value pairs (x_i, y_i), $i = 0, \dots, n$ is given by

$$p(x) = L_0(x) y_0 + L_1(x) y_1 + \dots + L_n(x) y_n,$$

where

$$L_i(x) = \frac{(x - x_0)(x - x_1)\dots(x - x_{i-1})(x - x_{i+1})\dots(x - x_n)}{(x_i - x_0)(x_i - x_1)\dots(x_i - x_{i-1})(x_i - x_{i+1})\dots(x_i - x_n)}$$

This representation is important for further considerations, but it is not the most favorable for programming or numerical evaluation.

In addition to the approach with the Vandermonde matrix and the Lagrange formula, you will now also learn the approach according to Newton.

Which computational path should you choose? *All roads lead to the polynomial*, in this case, to *the same* interpolation polynomial, just in different representations. Why should you learn additional computational paths then? *The journey is the destination*: In addition to practical applications, you encounter a mathematically profound connection: Polynomials behave in many ways like vectors; for mathematicians, they *are* vectors (elements of a vector space). That would go too far here, but we note:

Polynomials in Different Basis Representations A polynomial $p(x)$ can be expressed in different ways as a sum of terms of the form *coefficient times basis function.*

- Standard Basis

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

  p is a linear combination of the basis polynomials $1,\ x,\ x^2, \ldots, x^n$

- Lagrange Basis

$$p(x) = y_0 L_0(x) + y_1 L_1(x) + \cdots + y_n L_n(x)$$

  p is a linear combination of the Lagrange polynomials $L_0, L_1, \ldots, L_n$

- Newton Basis

$$p(x) = c_0 + c_1(x - x_0) + c_2(x - x_0)(x - x_1) + \cdots + c_n(x - x_0)\cdots(x - x_{n-1})$$

  p is a linear combination of the basis polynomials $1,\ (x - x_0)$, $(x - x_0)(x - x_1),\ \ldots,\ (x - x_0)(x - x_1)\cdots(x - x_{n-1})$

Newton's interpolation algorithm is a smarter variant of the initially presented brute-force method. Newton also uses an approach with undetermined coefficients but with different basis polynomials:

$$p(x) = c_0 + c_1(x - x_0) + c_2(x - x_0)(x - x_1) + \cdots + c_n(x - x_0)\cdots(x - x_n)$$

The coefficients $c_0, \ldots, c_n$ are sought. At first glance, the approach may seem more complicated than the standard form, but the resulting system of equations becomes simpler: The equations $p(x_i) = y_i$ now form a system of equations with a lower triangular matrix.

$$\begin{bmatrix} 1 & & & & 0 \\ 1 & (x_1 - x_0) & & & \\ 1 & (x_2 - x_0) & (x_2 - x_0)(x_2 - x_1) & & \\ \vdots & \vdots & & \ddots & \\ 1 & (x_n - x_0) & \cdots & & \prod_{i=0}^{n-1}(x_n - x_i) \end{bmatrix} \cdot \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ \\ c_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ \\ y_n \end{bmatrix} \tag{17}$$

The coefficients $c_0, \ldots, c_n$ can be calculated one after the other from top to bottom:

$$c_0 = y_0, \quad c_1 = \frac{y_1 - y_0}{x_1 - x_0}, \ldots$$

Continuing the calculation, you would notice: differences of y-values divided by differences of x-values occur continuously. This can be organized as an efficient calculation scheme: the *divided differences* method. This is what the next section is about.

7.2 Newton's Interpolation Polynomial

Newton's Interpolation Algorithm, Summary:

- Approach with *Newton basis functions*
- The *divided differences scheme* calculates the coefficients with little effort
- Efficient evaluation of the polynomial with *Horner's scheme*.

Values of a function are often given in tabular form, as in the following example:

Specific heat capacity of low-carbon steel in J/kg K for temperatures between 0 and 600C

T	cp
0	460.8
100	471.1
200	496.4
300	537.0
400	593.3
500	666.8
600	760.8

Interpolation polynomials are sought, which can be used to calculate intermediate values, for example, a cubic polynomial for the range $0 < T < 300$. (This example will be worked through later. It is possible but not necessarily wise to set up a single interpolation polynomial for the entire range $0 < T < 600$. From a practical point of view, linear interpolation between adjacent data points might be entirely sufficient. This depends on the quality of the data.)

The divided differences scheme is an optimized computational process for solving the system of equations (17). With pen and paper, it is best to organize the calculation in tabular form according to the following scheme:

$$
\begin{array}{cccccccc}
 & & & x_0 & y_0 & & & \\
 & & x_1 - x_0 & & & [x_0, x_1] & & \\
 & x_2 - x_0 & & x_1 & y_1 & & [x_0, x_1, x_2] & \\
 & & x_2 - x_1 & & & [x_1, x_2] & & \\
\dots & x_3 - x_1 & & x_2 & y_2 & & [x_1, x_2, x_3] & \dots \\
 & & x_3 - x_2 & & & [x_2, x_3] & & \\
 & x_4 - x_2 & & x_3 & y_3 & & [x_2, x_3, x_4] & \\
 & & x_4 - x_3 & & & [x_3, x_4] & & \\
 & & & x_4 & y_4 & & &
\end{array}
$$

Das Newton-Interpolationspolynom hat in diesem Fall die Form

There, the symbol in square brackets, $[x_0, x_1]$ between x_0 and x_1, denotes a divided difference defined by

$$[x_0, x_1] = \frac{y_1 - y_0}{x_1 - x_0}$$

This is the slope of the line through the two data points.

With this, we can already provide the formula for linear interpolation (first-degree interpolation polynomial):

$$p(x) = y_0 + (x - x_0)[x_0, x_1]$$

Higher divided differences are explained through the lower ones. A second difference $[x_0, x_1, x_2]$ is explained by

$$[x_0, x_1, x_2] = \frac{[x_1, x_2] - [x_0, x_1]}{x_2 - x_0}$$

and, in general, for $k > i$

$$[x_i, x_{i+1}, ..., x_{k+1}] = \frac{[x_{i+1}, ..., x_{k+1}] - [x_i, ..., x_k]}{x_{k+1} - x_i}$$

Then, $p(x)$ has the following representation, easily provable by induction:

$$\begin{aligned} p(x) = y_0 \quad + & \quad (x - x_0)[x_0, x_1] + (x - x_0)(x - x_1)[x_0, x_1, x_2] + \\ ... & \quad +(x - x_0)(x - x_1)...(x - x_{n-1})[x_0, x_1, ..., x_n] \end{aligned}$$

The formulas may look complicated, but the actual calculation process is very simple: First, calculate the entries in the left half of the table. In the right half of the table, obtain the divided differences following the rule "lower left minus upper left neighbor, divided by the symmetrically located entry from the left half of the table".

The coefficients of the Newton interpolation polynomial are located in the upper diagonal row (from the center top to the right diagonally downward) in this scheme.

The computed table for four data points from the above example looks like this:

```
            0    460.8
        100             0.103
    200     100  471.1         0.000750
300     100             0.253            0.000000050
    200     200  496.4         0.000765
        100             0.406
            300  537.0
```

In this case, the Newton interpolation polynomial has the form

$$p(x) = 460.8 + x \cdot 0.103 + x(x - 100) \cdot 0.000\,750 + x(x - 100)(x - 200) \cdot 0.000\,000\,050$$

It is not advisable to "simplify" this formula by expanding products like $x(x - 100)(x - 200)$ and combining like powers of x.

One can directly substitute x values into the above formula or write it in a less computationally expensive form (Horner's scheme), such as

$$p(x) = 460.8 + x(0.103 + (x - 100)(0.000\,750 + (x - 200) \cdot 0.000\,000\,050))$$

Rearranging it to $p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$ can introduce drastic rounding errors!

For equidistant x values, the left side of the table will have the same values column-wise, and the calculation process could be simplified. There are specially optimized formulas for interpolation with equidistant x values (e.g., in Bronstein); here, we limit ourselves to the general case.

The x values in this calculation table do not need to be sorted in size. Additionally, the table can be easily extended afterwards. The rest of the table does not need to be recalculated, and only one term is added to the interpolation polynomial, the others remain the same. This is an advantage of the Newton (and also the Neville) interpolation method over the Lagrange interpolation polynomial, where all terms change when a data point is added or changed.

For example, if you want to linearly approximate the given data in the range $200 \leq x \leq 600$, you set up the scheme

```
          200   496.4
      400             0.661
          600   760.8
```

and obtain

$$p(x) = 496.4 + (x - 200) \cdot 0.661 \ .$$

If you want to interpolate quadratically, you add a pair of values, complete the table

```
                200   496.4
            400             0.661
      100       600   760.8        0.00085
           -300             0.746
                300   537.0
```

and appends a term to the interpolation polynomial:

$$p(x) = 496.4 + (x - 200) \cdot 0.661 + (x - 200)(x - 600) \cdot 0.000\,85 \ .$$

However, in the standard form of polynomial representation, all coefficients change when moving from linear to quadratic interpolation. Linear and quadratic polynomials are given by:

$$\begin{aligned} p(x) &= 364.2 + 0.661 \cdot x \\ p(x) &= 466.2 - 0.019 \cdot x + 0.000\,85 \cdot x^2 \end{aligned}$$

Chapter 7.3 illustrates the Neville's method with a similar example. This method is useful when interpolating a single intermediate value, and only paper, pencil, and a cheap calculator are available.

Important Additional Information in the Exercise Materials

The materials for the 6th exercise explain the Vandermonde approach and the Newtonian interpolation method in detail. The approach with orthogonal polynomials is also presented there. Please refer to those materials!

7.3 Neville's Method

The Neville calculates the value of the interpolation polynomial at a given point x. (Sometimes confused with the Aitken method, which is quite similar.)

For $k = 0, 1, ..., n$, set

$$P_i(x) = y_i$$

Then P_i is the value of the uniquely determined polynomial of degree 0 passing through (x_i, y_i).

Now let P_{01} be the value (at the given point x) of the uniquely determined polynomial of the first degree (a straight line) passing through the points (x_0, y_0) and (x_1, y_1). And similarly: $P_{12}, P_{23}, \ldots$. Similarly, let P_{012} be the value of the parabola (at the given point x) passing through the points (x_0, y_0), (x_1, y_1), and (x_2, y_2). Continuing this schema for higher-order polynomials, we reach $P_{0,1,2\ldots n}$, the value of the polynomial through all given points and thus the sought answer.

The trick is that the individual P values can be easily calculated from the two previous ones. The formula is most easily written with the determinant of a 2×2 matrix:

$$P_{i(i+1)\ldots(i+m)} = \frac{1}{x_{i+m} - x_i} \begin{vmatrix} x - x_i & P_{i(i+1)\ldots(i+m-1)} \\ x - x_{i+m} & P_{(i+1)(i+2)\ldots(i+m)} \end{vmatrix}$$

By induction, it is easy to show that the P values defined in this way have exactly the claimed properties. To actually calculate, one then uses a table (matrix) of the form:

$$\begin{array}{llllll}
x_0: & y_0 = P_0 & & & \\
 & & P_{01} & & \\
x_1: & y_1 = P_1 & & P_{012} & \\
 & & P_{12} & & P_{0123} \\
x_2: & y_2 = P_2 & & P_{123} & \\
 & & P_{23} & & \\
x_3: & y_3 = P_3 & & &
\end{array}$$

This tableau is filled column by column from left to right. For example, the value P_{123} is obtained from the two left neighbors as

$$P_{123} = \frac{1}{x_3 - x_1} \begin{vmatrix} x - x_1 & P_{12} \\ x - x_3 & P_{23} \end{vmatrix} = \frac{(x - x_1)P_{23} - (x - x_3)P_{12}}{x_3 - x_1}$$

If you want to increase the interpolation order, you can append values for x_4 and y_4 at the bottom of the tableau and complete the missing values $P_{34}, P_{234}, P_{1234}$, and P_{01234} without having to recalculate the entire tableau (advantage over the Lagrange formula). By the way, the x_i do not need to be equidistant or ordered in any way.

Example: Density of Water

The density of water as a function of temperature is given in the following table for the range from 0 to 100 Celsius:

T	ρ	T	ρ
0	999.840	10	999.699
1	999.899	20	998.203
2	999.940	30	995.645
3	999.964	40	992.212
4	999.972	50	988.030
5	999.964	60	983.191
6	999.940	70	977.759
7	999.901	80	971.785
8	999.848	90	965.304
9	999.781	100	958.345

Density values for temperatures not listed in the table can be found through interpolation or approximation. Which method is appropriate?

- Individual values are needed: Neville's method.
- A formal relationship $T = T(\rho)$ in a narrow temperature interval is needed: Newtonian interpolation polynomial.
- A formal relationship over the entire range is sought: Approximation by a polynomial or a rational function.

Neville

The goal is to find $\rho(24)$. A calculated tableau looks like

$$\begin{array}{lllll}
10: & 999.699 & & & \\
 & & 997.605 & & \\
20: & 998.203 & & 997.307 & \\
 & & 997.180 & & 997.297 \\
30: & 995.645 & & 997.285 & \\
 & & 997.705 & & \\
40: & 992.212 & & &
\end{array}$$

The individual steps are as follows:

$$P_{01} = \frac{(x - x_0)P_1 - (x - x_1)P_0}{x_1 - x_0}, \quad P_{12} = \frac{(x - x_1)P_2 - (x - x_2)P_1}{x_2 - x_1}, \ldots$$

These are the values that result from linear interpolation. In many cases, linear interpolation will be sufficiently accurate; one could then be satisfied with the value P_{12}. The next column corresponds to quadratic interpolation according to the formulas

$$P_{012} = \frac{(x - x_0)P_{12} - (x - x_2)P_{01}}{x_2 - x_0}, \quad P_{123} = \frac{(x - x_1)P_{23} - (x - x_3)P_{12}}{x_3 - x_1}.$$

The values obtained from this differ only by 2 units in the hundredths place. The last value comes from cubic interpolation:

$$P_{0123} = \frac{(x - x_0)P_{123} - (x - x_3)P_{012}}{x_3 - x_0},$$

Computational programs that require material properties can either store extensive tables and linearly interpolate within them, or they can save storage space by using tables with fewer support points and higher-order interpolation. Very often, a significant part of the computation time is spent on the evaluation of material properties; an computationally efficient implementation is then significant.

7.4 Error Estimation of Polynomial Interpolation

Let's assume that a function $f : [a, b] \to \mathbb{R}$, $-\infty < a < b < \infty$, is given at the support points $\pi = \{a = x_0 < x_2 < \ldots < x_n = b\}$, and $p_n(x)$ is a polynomial of degree $n-1$ that passes through these support points. The question now is, how large is the error, or

$$e(x) = \sup_{x \in [a,b]} |p_n(x) - f(x)|.$$

The following theorem provides an estimate of the error, depending on the function f and its derivatives.

Error Estimation for Polynomial Interpolation
Assume f is $(n+1)$ times continuously differentiable, and the polynomial p_n passes through the points $\{(x_i, f(x_i)) : i = 0, \ldots, n\}$. Then, for each $x^* \in (a, b)$, there exists a point $\xi \in (a, b)$ such that

$$f(x^*) - p_n(x^*) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x^* - x_0)(x^* - x_1)\cdots(x^* - x_{n-1})(x^* - x_n).$$

Proof of the theorem: Let us consider the function

$$\Theta(x) := f(x) - p_n(x) - c(x^*)(x - x_0)(x - x_1)\cdots(x - x_{n-1})(x - x_n)$$

where

$$c(x^*) = \frac{f(x^*) - p_n(x^*)}{(x^* - x_0)(x^* - x_1)\cdots(x^* - x_{n-1})(x^* - x_n)}.$$

The function Θ is defined such that x^* is a root, i.e., $\Theta(x^*) = 0$. Since $f(x_k) = p(x_k)$ holds, the function Θ additionally has the roots $x_0, x_1, \ldots, x_n$, thus $n+2$ roots. Differentiating Θ (allowed as f is assumed to be differentiable $n+1$ times), we find, by Rolle's theorem, intermediate values $\xi_0, \xi_2, \ldots, \xi_n$ between the roots of Θ with $\Theta'(\xi_j) = 0$, $j = 1, \ldots, n$. Let's call these intermediate values $\xi_j^{(1)}$. Again applying Rolle's theorem, we know that there are n intermediate values $\xi_0, \xi_2, \ldots, \xi_{n-1}$ with $\Theta''(\xi_j) = 0$, $j = 0, \ldots, n-1$. Let's call these intermediate values $\xi_j^{(2)}$. This process can be continued until we have only one intermediate value $\xi = \xi_1^{(n+1)}$ such that

$$0 = \Theta^{(n+1)}(\xi) = f^{(n+1)}(\xi) - c(x^*)(n+1)!.$$

From this, after a simple rearrangement, follows

$$c(x^*) = \frac{f^{(n+1)}(\xi)}{(n+1)!}.$$

which is the estimate from the theorem.

Now, if we have an equidistant grid and $0 \le a < b \le 1$, then

$$|f(x^*) - p_n(x^*)| \le \sup_{\xi \in [a,b]} \frac{|f^{(n+1)}(\xi)|}{(n+1)!} \frac{(a-b)^n}{n^n} \le \sup_{\xi \in [a,b]} \frac{|f^{(n+1)}(\xi)|}{(n+1)! n^n}.$$

Here, we made the following estimate:

$$(x^* - x_0)(x^* - x_1) \cdots (x^* - x_{n-1})(x^* - x_n) \le \left(\frac{(a-b)}{n} \right)^n .$$

7.5 Interpolation or Approximation?

Polynomials of high degree tend to oscillate and are susceptible to rounding errors when evaluated in standard form. An interpolating polynomial through all data points in the example for the density of water from Section 7.3 yields completely nonsensical values.

The interpolation polynomial calculated and evaluated with the Matlab commands `polyfit` and `polyval` deviates significantly in this case, even at the original data points (where it should match exactly!): $\rho(0) = 1000.2$ instead of 999.8. Between the data points, it provides fantasy values in the range of $\pm 10^8$.
Numerically more accurate calculations of the interpolation polynomial would deliver better values at the original data points, but in between, it would still produce garbage.

A formal representation for the entire data range can be found with regression polynomials. A quadratic approximation provides an accuracy of ± 0.5,

$$\rho(T) = 1000.35 - 0.0614512\, T - 0.00364033\, T^2$$

a fourth-degree regression polynomial

$$\rho(T) = 999.867 + 0.0545396\, T - 0.00765475\, T^2 + 0.0000434548\, T^3 - 1.38985\, 10^{-7}\, T^4$$

deviates from the data points by a maximum of 0.03.

Rational functions are often better suited for approximation. For example, the approximation

$$\rho(T) = \frac{999.844 + 9.63049\, T - 0.0244379\, T^2}{1 + 0.00956721\, T - 0.0000163504\, T^2}$$

shows a maximum error of 0.004. Such approximations can also be found using the method of least squares: Assume with undetermined coefficients, substitute the data, transform it into an overdetermined linear system, and solve the normal equations. (Depending on the approach, *nonlinear* least squares problems may arise, which are much more difficult to handle.) The above approximation minimizes the maximum error (so-called *Minimax approximation*). For the construction of such approximations, refer to advanced literature (the computer algebra system Mathematica, for example, provides a package for this).

7.6 Further Interpolation Methods

Not all functions can be efficiently interpolated by polynomials. Other important methods include

- Rational interpolation
- Spline interpolation

- Trigonometric interpolation
- Interpolation in two or more dimensions

Rational functions are quotients of two polynomials. Bulirsch and Stoer (see their book Numerical Mathematics 1) have found an algorithm that works with a tableau similar to Aitken-Neville interpolation. Most functions can be better approximated by rational functions than by polynomials. Even your calculator likely evaluates mathematical functions like cosine and sine in the form of rational expressions.

Spline functions play an important role wherever computers need to draw a "nice-looking" curve or surface through given points.

Trigonometric interpolation uses sums of sine and cosine functions or complex exponential functions to fit a curve through a set of data points. Closely related and highly important is the method of Fast Fourier Transform (*FFT*).

7.7 Spline Interpolation

Interpolating polynomials of high degree are computationally expensive, often very sensitive to minor changes in the data (coordinates of support points), and tend to overshoot. If one wants to connect points with a "nice-looking" curve, especially for graphical purposes, the idea is to piece together individual polynomials of lower degree. Such functions are generally referred to as splines.

Splines
A spline function is a piecewise-defined function defined on subintervals, whose parts are continuous or even differentiable at the interval boundaries.

Cubic (natural) Splines
A cubic spline $s(x)$ through the $n+1$ data pairs (x_i, y_i), $i = 0, \ldots, n$ is characterized as follows:

In each interval (x_{i-1}, x_i), $s(x)$ is a cubic polynomial.

At the interval boundaries, the function values, the first, and the second derivatives match on both sides.

Additional condition: (natural spline) At the left and right edges, $s''(x) = 0$.

Among all twice-differentiable interpolating functions $f(x)$, natural cubic splines have the smallest

$$\int_{x_0}^{x_n} \left(f''(x)\right)^2 dx$$

However, this integral measures the roughness or total curvature of the function in a certain sense. Thus, overshooting is hardly possible.

However, natural cubic splines cannot draw loop or circular curves. For this purpose, *parametric* splines are used, which represent both x and y values of a curve as a spline function of a parameter. The method for drawing curves in Excel works with such splines.

Not all types of splines interpolate points. *Bézier* splines use some of the points to influence the shape of the curve. Drawing curves in MS-Paint works in this way.

Error Estimation:

Let's assume that a function $f : [a, b] \to \mathbb{R}$, $-\infty < a < b < \infty$, is given at the support points $\Pi = \{a = x_0 < x_2 < \ldots < x_n = b\}$, and $s_n(x)$ is a cubic spline with exactly these support points $\{(x_i, f(x_i)) : i = 0, \ldots, n\}$. The question now is, how large is

$$e(x) = \sup_{x\in[a,b]} |s_n(x) - f(x)|.$$

Assuming that f is four times continuously differentiable, with $f''(a) = f''(b) = 0$.

- Then, there exists exactly one natural cubic spline s_n.
- The following error estimates hold:

$$\begin{aligned}
\sup_{x\in[a,b]} |f(x) - s_n(x)| &\leq cM_4h^4, && (18)\\
\sup_{x\in[a,b]} |f'(x) - s'_n(x)| &\leq cM_4h^3, && (19)\\
\sup_{x\in[a,b]} |f''(x) - s''_n(x)| &\leq cM_4h^2, && (20)\\
\sup_{x\in[a,b]} |f'''(x) - s'''_n(x)| &\leq cM_4h, && (21)
\end{aligned}$$

  where $h = \max(|x_j - x_{j-1}|)$ and $M_4 = \sup_{x\in[a,b]} |f^{IV}(x)|$.

8 Numerical Integration

Since ancient times, mathematics has been concerned with the calculation of the area of curvilinearly bounded surfaces. Such tasks require, in modern terminology, the evaluation of certain integrals.

The *squaring of the circle* is impossible with compass and straightedge, the tools of ancient geometers. The term "squaring" as a synonym for numerical area or integral calculation has remained. Numerical integration is also called *numerical quadrature*; the corresponding formulas are called *quadrature formulas*.

Functions with elementary antiderivatives can be listed on a few pages of a formula collection. Beyond that, integrals can only be calculated numerically[10].

Even if a function is given not as a symbolic expression but as a table or the result of a computer program, numerical integration is employed.

8.1 Newton-Cotes Type Integration Formulas

Newton-Cotes Formulas
Given: A function $f(x)$ in an interval (a,b) by its values f_i at $n+1$ equidistant support points,

$$f_i = f(a+ih), \quad \text{with } h = \frac{b-a}{n}, \quad i = 0, \ldots, n.$$

Sought: An approximation for the integral $\int_a^b f(x)dx$.
Principle: Interpolate $f(x)$ by a polynomial $p(x)$. Approximate the integral of f by the integral of p. The approximation is given as a weighted sum of the f_i,

$$\int_a^b f(x)dx \approx (b-a)\sum_{i=0}^{n} \alpha_i f_i$$

with fixed weights α_i

Examples: Trapezoidal rule

$$\int_a^b f(x)dx \approx \frac{b-a}{2}\,(f(a)+f(b))$$

Quadratic interpolation (Local: "Kepler's barrel rule," international: "Simpson's rule")

$$\int_a^b f(x)dx \approx \frac{b-a}{6}\left(f(a) + 4f(\frac{a+b}{2}) + f(b)\right)$$

Johannes Kepler came up with this approximation formula using function values at three equidistant support points when he thought about the volume of some barrels of wine he was buying in Linz in 1612 (how much he paid for it is not recorded).

[10] However, if an integral occurs frequently, it is simply *defined* as its own function, tabulated, and special approximation formulas and series expansions are found for its evaluation. Examples: The integral $\int_0^\infty e^{-x^2}dx$ has no elementary antiderivative, but it is so important that a special function, the Gaussian error function, is defined for it. Also, for the arc length of an ellipse or the period of a mathematical pendulum at large amplitudes, integrals are to be evaluated for which special functions (elliptic functions) are defined.

The trapezoidal rule, of course, gives exact values for linear functions. Simpson's rule, based on quadratic interpolation, clearly calculates exact values for integrals of parabolas, but, surprisingly, even for polynomials of the third degree. Errors can be estimated by Taylor series expansion and depend on the higher derivatives of the function f.

Usually, these formulas are not used for the entire interval $[a, b]$ but are divided into a series of smaller intervals. The formulas are then applied in each interval, and the partial results are summed. This is referred to as *composite* Newton-Cotes formulas.

Given: For a function $f(x)$ in an interval (a, b), values f_i at $n+1$ equidistant support points,

$$f_i = f(a + ih), \quad \text{with } h = \frac{b-a}{n}, \quad i = 0, \ldots, n.$$

Composite Trapezoidal Rule

$$\int_a^b f(x)dx = \frac{h}{2}(f_0 + 2f_1 + 2f_2 + \cdots + 2f_{n-1} + f_n) + E$$

Composite Simpson's Rule

$$\int_a^b f(x)dx = \frac{h}{3}(f_0 + 4f_1 + 2f_2 + 4f_3 + \cdots + 2f_{n-2} + 4f_{n-1} + f_n) + E$$

Only for even n!

The error terms are

$$E = \frac{a-b}{12}h^2 f''(\xi) \quad \text{for some } \xi \in [a, b]$$

for the composite trapezoidal rule and

$$E = \frac{a-b}{180}h^4 f''''(\xi) \quad \text{for some } \xi \in [a, b]$$

for the composite Simpson's rule

8.2 Romberg Method, Gaussian Quadrature Formulas

The Romberg method applies the composite trapezoidal rule multiple times with different step sizes. It extrapolates the results to obtain an even more accurate value. Detailed treatment in the exercise materials, Unit 6.

Gaussian quadrature formulas will not be covered this year (2023, 2024). In contrast to Newton-Cotes-type formulas, which use equidistant support points, Gaussian formulas choose support points at optimal positions to achieve the highest possible error order.

9 Eigenvalues and Eigenvectors

9.1 Introduction, Definition, Fundamentals

Eigenvalue Problem
Given an $n \times n$ matrix A. Wanted

- a vector $\mathbf{x}$ different from the zero vector and
- a scalar λ (including the case $\lambda = 0$),

which satisfy the equation

$$A\mathbf{x} = \lambda\mathbf{x} \tag{22}$$

Such a λ is called an *eigenvalue* of A, a corresponding $\mathbf{x}$ is called an *eigenvector* of A for the eigenvalue λ.

The situation of "Matrix times eigenvector equals zero times vector," i.e., $A\mathbf{x} = 0\mathbf{x}$, can indeed occur. In such a case, $\lambda = 0$ is an eigenvalue of A.

However, we explicitly require that $\mathbf{x}$ must not be the zero vector 0 because the equation $A \cdot 0 = \lambda \cdot 0$ would be satisfied for any λ—this would not be a meaningful definition of λ.

Intuitive Interpretation

The main job of a matrix is to multiply vectors. When it does that, the resulting vector usually has a different length and points in a different direction than the input vector. However, every matrix has very special "own" vectors for which it changes the length but leaves the direction unchanged (if $\lambda > 0$) or exactly reverses it (if $\lambda < 0$). It can also happen (if $\lambda = 0$) that an eigenvector is turned into the zero vector. Regular matrices, however, do not do that—only singular matrices are so mean.

Example:
The matrix $A = \begin{bmatrix} -1 & 0 \\ 1 & 2 \end{bmatrix}$ transforms $\mathbf{u} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$ into

$$A\mathbf{u} = \begin{bmatrix} -1 & 0 \\ 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} -1 \\ 5 \end{bmatrix},$$

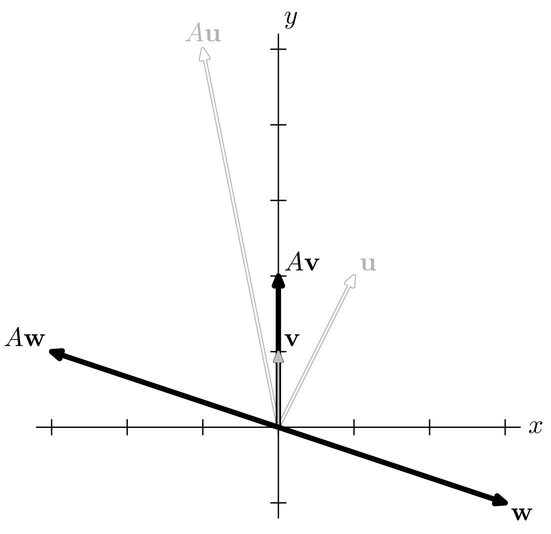
different length and direction—no eigenvector.
On the other hand, the vector $\mathbf{v} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ is transformed into

$$A\mathbf{v} = \begin{bmatrix} -1 & 0 \\ 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 2 \end{bmatrix},$$

same direction, double length—
eigenvector for the eigenvalue 2.
The vector $\mathbf{w} = \begin{bmatrix} 3 \\ -1 \end{bmatrix}$ becomes

$$A\mathbf{w} = \begin{bmatrix} -1 & 0 \\ 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} 3 \\ -1 \end{bmatrix} = \begin{bmatrix} -3 \\ 1 \end{bmatrix}$$

same length, reversed direction—
eigenvector for the eigenvalue -1.



Multiple Eigenvectors for One Eigenvalue

If $\mathbf{x}$ is an eigenvector of A for the eigenvalue λ, then any multiple $\mu\mathbf{x}$, $\mu \neq 0$, is also an eigenvector of A for the same eigenvalue λ. This is because A, λ, and $\mathbf{x}$ satisfy the equation (22), so we have

$$A(\mu\mathbf{x}) = \mu(A\mathbf{x}) = \mu(\lambda\mathbf{x}) = \lambda(\mu\mathbf{x}).$$

Eigen vectors are uniquely determined up to a scalar factor.

By rearranging (22), we get

$$\begin{aligned} A\mathbf{x} &= \lambda\mathbf{x} \\ A\mathbf{x} - \lambda\mathbf{x} &= 0 \\ (A - \lambda I)\mathbf{x} &= 0 \end{aligned} \tag{23}$$

where I is the identity matrix. The vector $\mathbf{x} = 0$ (the zero vector) is always a solution of Equation (23), but we are not interested in that. We want there to be other non-trivial solutions. This is only the case when

$$\det(A - \lambda I) = 0. \tag{24}$$

Expanding the determinant yields a polynomial of degree n in λ. This polynomial is called the *characteristic polynomial* of A. Every polynomial of degree n has exactly n real or complex roots (according to the Fundamental Theorem of Algebra; counting multiple roots according to their multiplicity). It follows that every $n \times n$ matrix has exactly n (real or complex, possibly counted with multiplicity) eigenvalues.

The eigenvalues of a matrix A can be calculated as the roots (zeroes) of its characteristic polynomial. This is a classical method but generally practical only for small matrices ($2 \times 2,\ 3 \times 3$).

Example calculation: We calculate the eigenvalues of the matrix A from the previous example (Page 75).

$$\begin{aligned} \det(A - \lambda I) &= \det\left(\begin{bmatrix} -1 & 0 \\ 1 & 2 \end{bmatrix} - \lambda \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right) = \det\begin{bmatrix} -1-\lambda & 0 \\ 1 & 2-\lambda \end{bmatrix}, \\ &= (-1-\lambda)\cdot(2-\lambda) - 0\cdot 1 = \lambda^2 - \lambda - 2 \end{aligned}$$

The roots of the characteristic polynomial $p(\lambda) = \lambda^2 - \lambda - 2$, and therefore the eigenvalues of A, are $\lambda_1 = -1\,,\ \lambda_2 = 2$.

Another example: Let's calculate the eigenvalues of the matrix $A = \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}$ as roots of the characteristic polynomial.

$$\det(A - \lambda I) = \det\left(\begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix} - \lambda \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\right) = \det\begin{bmatrix} 8-\lambda & 1 & 6 \\ 3 & 5-\lambda & 7 \\ 4 & 9 & 2-\lambda \end{bmatrix}$$

Calculating the determinant, for example, using the Rule of Sarrus, gives

$$\begin{aligned} \det(A - \lambda I) &= (8-\lambda)(5-\lambda)(2-\lambda) + 28 + 162 - 24(5-\lambda) - 63(8-\lambda) - 3(2-\lambda) \\ &= -360 + 24\lambda + 15\lambda^2 - \lambda^3 \end{aligned}$$

Therefore, the characteristic polynomial of A is $p(\lambda) = -360 + 24\lambda + 15\lambda^2 - \lambda^3$, and its roots (zeroes) can be found using methods from Chapter 1. The three roots, and thus the three eigenvalues of A, are $\lambda_1 = 15, \lambda_{2,3} = \pm 4.898\,98$.
For larger matrices, this calculation method is too cumbersome and susceptible to rounding errors.

If you know an eigenvalue λ, you can find a corresponding eigenvector as a solution to the system of equations (23).

For $\lambda = 15$ in the example above, we are looking for a solution to the system

$$\begin{bmatrix} -7 & 1 & 6 \\ 3 & -10 & 7 \\ 4 & 9 & -13 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} .$$

In this system, the sum of the first and second equations equals the negative of the third. If you are wondering how to determine the three unknowns $x_1,\ x_2,\ x_3$ with only two linearly independent equations, you haven't grasped the point: *The system must not have a unique solution!* That's why we ensured, with the determinant condition (24), that besides the trivial solution $x_1 = x_2 = x_3 = 0$, there are other meaningful solutions. We can, for example, choose $x_1 = 1$ freely and substitute it into the first two equations, then determine x_2 and x_3:

$$\begin{array}{rcrcr} x_2 & + & 6x_3 & = & 7 \\ -10x_2 & + & 7x_3 & = & -3 \end{array} \qquad \text{Solution: } x_2 = 1,\ x_3 = 1$$

9.2 Eigenvalue problem $A\mathrm{x} = \lambda\mathrm{x}$: Further properties

Matrices in physical applications are usually symmetric (stiffness matrix, stress tensor, inertia tensor, etc.). The corresponding eigenvalues correspond to physical quantities (vibration frequencies, principal stresses, moments of inertia, etc.) that are expected to be real. It is reassuring to know that mathematics guarantees:

All eigenvalues of a symmetric matrix are real.

For a symmetric matrix A, the eigenvalue problem simplifies considerably. There is a rich and elegant mathematical theory and a wealth of methods for dealing with it.

The eigenvectors of a symmetric matrix form an orthogonal system.

If the eigenvectors of a symmetric matrix A are also normalized to length 1, they can be combined in an orthogonal matrix Q. Then, it holds:

$$Q^T A Q = D \ \text{ with } D \text{ a diagonal matrix of eigenvalues} .$$

Eigenvalues can be "shifted"

If λ is an eigenvalue of A and $\mathbf{x}$ is a corresponding eigenvector, then, for any $s \in \mathbb{R}$, the value $\lambda + s$ is an eigenvalue of $A + sI$ with the same $\mathbf{x}$ as the corresponding eigenvector.

Eigenvalues of the inverse matrix are inverses of the eigenvalues

If λ is an eigenvalue of A with the corresponding eigenvector $\mathbf{x}$, then $1/\lambda$ is an eigenvalue of A^{-1} with the same eigenvector $\mathbf{x}$.

Similarity transformation

If X is an invertible matrix[11] , then A and $X^{-1}AX$ have the same eigenvalues. The transformation from A to $X^{-1}AX$ is called *similarity transformation* .

Modern computational methods cleverly utilize similarity transformations, the shifting of eigenvalues, and the computation of inverse eigenvalues. Without such tricks, the methods would be significantly slower.

Diagonalization

If an $n \times n$ matrix A has exactly n linearly independent eigenvectors, then they can be arranged as column vectors into an $n \times n$ matrix V. The similarity transformation with V produces a diagonal matrix D; its main diagonal contains the eigenvalues.

$$V^{-1}AV = D$$

Example: The matrix $A = \begin{bmatrix} -1 & 0 \\ 1 & 2 \end{bmatrix}$ from the example on page 75 has eigenvectors $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$ and $\begin{bmatrix} 3 \\ -1 \end{bmatrix}$. Here,

$$V = \begin{bmatrix} 0 & 3 \\ 1 & -1 \end{bmatrix} \text{ and } V^{-1} = \begin{bmatrix} 1/3 & 1 \\ 1/3 & 0 \end{bmatrix}, \quad V^{-1}AV = \begin{bmatrix} 1/3 & 1 \\ 1/3 & 0 \end{bmatrix} \cdot \begin{bmatrix} -1 & 0 \\ 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} 0 & 3 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & -1 \end{bmatrix}$$

The MATLAB command for computing eigenvalues and eigenvectors is `[V,D] = eig(A)`. The underlying computational method performs iterative similarity transformations, gradually approaching the transformed matrix to a diagonal matrix. As the final result, it provides the diagonal matrix D and the matrix V of the similarity transformation. Then, in the main diagonal of D, the eigenvalues of A are given, and the corresponding eigenvectors are in the columns of V.

It may happen that an $n \times n$ matrix A does not have n linearly independent eigenvectors. This complicates the situation. Fortunately, as mentioned earlier, symmetric matrices occur in important applications. For them, there is always a transformation matrix made up of linearly independent eigenvectors. As an additional bonus: the transformation matrix can be chosen as an *orthogonal* matrix Q.

In short:

Symmetric matrices can be diagonalized by orthogonal transformations:

$$Q^T AQ = D \ .$$

The columns of Q are eigenvectors of A with the corresponding diagonal elements of D as eigenvalues.

9.3 Power Iteration

The computation of eigenvalues and eigenvectors is of significance in many applications, ranging from mechanics (vibrations) to economics (average prices for maximum profit) and to the evaluation of the importance of websites (PageRank of a homepage as an eigenvector of the Google matrix, a simple example is given in the exercises).

[11]equivalent to: non-singular matrix, regular matrix

It is not always necessary to know all eigenvalues; often, it suffices to compute the largest eigenvalue or the one with the largest absolute value.

In exceptional cases, there may be multiple eigenvalues with the same absolute value. However, among all eigenvalues, *exactly one* with the maximum absolute value is called the *dominant* eigenvalue. Such an eigenvalue λ and an associated eigenvector can be found as a fixed point of the system

$$\mathbf{x} = \frac{1}{\lambda} A\mathbf{x}$$

using a simple iterative method: the so-called power iteration (or *power method*).

Power iteration (derived from earlier approaches) goes back to Richard von Mises[12] and Hilda Geiringer[13] In brief: Start with any vector $\mathbf{x}^{(0)}$ and multiply it with A. Scale the result suitably by dividing it by a factor $\lambda^{(1)}$ and call it $\mathbf{x}^{(1)}$. Continue this process until the $\mathbf{x}^{(k)}$ changes negligibly. Then, $\lambda^{(k)}$ is the eigenvalue with the largest absolute value, and $\mathbf{x}^{(k)}$ is an associated eigenvector.

Power Iteration for the Largest Eigenvalue
(by von Mises and Geiringer, also called *power method*)
Given an $n \times n$ matrix A, an initial vector $\mathbf{x}^{(0)} \neq 0$, and a fixed choice of $i \in \{1, \ldots, n\}$

```
Iterate for $k = 1, 2, \ldots$
  compute $\mathbf{y}^{(k)} = A\mathbf{x}^{(k-1)}$
  set $\lambda^{(k)} = y_i^{(k)}$ (the $i$-th component of $\mathbf{y}^{(k)}$)
  set $\mathbf{x}^{(k)} = \mathbf{y}^{(k)}/\lambda^{(k)}$ (scaling)
```

With paper and pencil, the multiplication $A\mathbf{x}$ is of deadly complexity; however, computers find this method quite easy. It converges if there is a dominant eigenvalue, and the initial vector has a component in the direction of the associated eigenvector.

Normally, it is irrelevant which component of $\mathbf{x}^{(k)}$ is chosen for scaling, except the eigenvector has exactly 0 for that component. An alternative scaling rule avoids specifying a particular component; this is simpler for computer programs:

Find the component $y_i^{(k)}$ with the largest magnitude and set $\lambda^{(k)} = y_i^{(k)}$.

By applying the power method to the matrix A^{-1}, one can determine the *smallest-magnitude* eigenvalue. In general, one can find the eigenvalue that is closest to a value μ by applying the power method to $(A-\mu I)^{-1}$ (*inverse iteration*). However, A^{-1} or $(A-\mu I)^{-1}$ is not explicitly calculated. Instead, the iteration step $\mathbf{y}^{(k)} = A^{-1}\mathbf{x}^{(k-1)}$ is reformulated as $A\mathbf{y}^{(k)} = \mathbf{x}^{(k-1)}$, and $\mathbf{y}^{(k)}$ is determined as the solution of this system. If μ is a good approximation to λ_i, the method converges rapidly.

For large, sparsely populated symmetric matrices, there is a particularly efficient method, especially when only the largest or some of the largest (or smallest) eigenvalues are sought—the Lanczos method. At its core are iterations of the form $\mathbf{y}^{(k)} = A\mathbf{x}^{(k-1)}$.

There are also MATLAB commands for this. For example, `d = eigs(A)` for sparsely populated matrices returns the six largest-magnitude eigenvalues and their corresponding eigenvectors.

[12]Richard von Mises, 1883 (Lemberg, Austria; now Lviv, Ukraine) – 1953 (Boston, USA). Mathematician, studied in Vienna, professor in Strasbourg, Dresden, Berlin, Istanbul, and from 1939 at Harvard University; groundbreaking work in almost all areas of applied mathematics, as well as fluid mechanics, especially with regard to aircraft construction (holds the first lecture on motor flight in 1913!)

[13]Hilda Geiringer (Vienna, 1893 – Santa Barbara, California, 1973), studied in Vienna, worked in Berlin as an assistant to von Mises at the Institute for Applied Mathematics on probability theory and statistics, emigrated in 1933, taught in Brussels, Istanbul, and various American universities.

9.4 Simple Forms

Many important methods for eigenvalue computation transform a matrix A through a sequence of similarity transformations $X^{-1}AX$ into a simple form and then compute the eigenvalues. (Eigenvalues do not change under similarity transformations.) The standard method is the so-called QR algorithm.

Simple Forms

- Diagonal matrix
- Triangular matrix
- Tridiagonal matrix

For diagonal and triangular matrices, the eigenvalues are precisely the diagonal elements.

For a tridiagonal matrix, the characteristic polynomial can be evaluated easily without having to write it out explicitly. For a symmetric tridiagonal matrix T,

$$T = \begin{bmatrix} a_1 & b_1 & & \cdots & 0 \\ b_1 & a_2 & b_2 & & \vdots \\ & \ddots & \ddots & \ddots & \\ \vdots & & \ddots & \ddots & b_{n-1} \\ 0 & \cdots & & b_{n-1} & a_n \end{bmatrix},$$

expanding the determinant $\det(T - xI)$ shows that the value $p(x)$ of the characteristic polynomial can be recursively determined as follows:

```
set $p_0(x) = 1, p_{-1}(x) = 0.$
For $k = 1, \dots, n$
  $p_k(x) = (a_k - x)p_{k-1}(x) - b_{k-1}^2 p_{k-2}(x)$
Result $p(x) = p_n(x).$
```

A method for finding roots, such as bisection, yields the eigenvalues as roots of the characteristic polynomial.

10 Ordinary Differential Equations

Note: This Chapter covers the topic briefly and at a glance. The exercise materials provide more in-depth explanations. Please also have a look at the lecture slides. They contain additional material, including pictures, explanations, and summaries.

10.1 Problem Statement, Examples

10.1.1 First-Order Differential Equations

Initial Value Problem for an explicit 1st-order ordinary differential equation.
Given: an equation

$$y' = f(x, y) \ .$$

Wanted: a function $y : x \mapsto y(x)$ that fulfills

$$\begin{aligned} y'(x) &= f(x, y(x)) && \text{differential equation} \\ y(x_0) &= y_0 && \text{initial condition} \end{aligned}$$

The function f maps a pair (x, y) from a domain $D \subseteq \mathbb{R}^2$ to $f(x, y) \in \mathbb{R}$. In mathematical notation,

$$f : D \subseteq \mathbb{R}^2 \to \mathbb{R}, \ (x, y) \mapsto f(x, y) \ .$$

If f is continuous in x and satisfies a Lipschitz condition for y, then a unique solution exists in some neighborhood of the starting point x_0.

The notation $y : x \mapsto y(x)$ or, slightly inconsistent but widely used, $y = y(x)$ represents the typical unknown function and the variable it depends on. In many applications, however, differential equations describe *temporal* changes. In this case, the unknown function y is a function of time, denoted as $y(t)$, or, when x denotes a position in space, $x(t)$ describse the motion. In physics, you are familiar with the notation $\dot{x}(t)$ for derivatives with respect to time.

In the notation with $y = y(t)$, the problem statement reads as follows:

Find a function $y(t)$ that satisfies

$$\begin{aligned} \dot{y} &= f(t, y) \\ y(t_0) &= y_0 \end{aligned}$$

So, do not be confused: depending on the context, the unknown function, the *dependent variable*, is denoted as y or x or (as in the second example) s, depending on x or t, and derivatives are indicated by dots or dashes.

Example 1: Exponential Growth of a Population Suppose a population of bacteria is in a nutrient liquid and has a size of $y(t)$ at time $t > 0$. At $t = 0$, the population had a size of $y(0) = y_0$. Assuming that, starting from time t, the population will increase by $\Delta y = y(t + \Delta t) - y(t)$ members after a time interval Δt. It is reasonable to assume that for small time intervals Δt, the increase is approximately proportional to the population size $y(t)$ and the time interval Δt. Therefore,

$$\Delta y = y(t + \Delta t) - y(t) \sim a \cdot y(t) \cdot \Delta t,$$

with a proportionality constant $a > 0$.

Now, calculating the derivative using the differential quotient, we get

$$\dot{y}(t) = \lim_{\Delta t \to 0} \frac{1}{\Delta t} (y(t + \Delta t) - y(t)) = \lim_{\Delta t \to 0} \frac{1}{\Delta t} a \cdot y(t) \cdot \Delta t = a \cdot y(t).$$

In addition to the differential equation, the function $y(t)$ must satisfy the *initial condition* $y(0) = y_0$. This *initial value problem* is described by the following system:

$$\begin{array}{rcl} \dot{y}(t) & = & ay(t) \\ y(0) & = & y_0 \end{array}$$

This system describes the growth law of the bacterial population. You are probably familiar with the solution: it is *exponential growth.* The solution to the initial value problem is

$$y(t) = y_0 \exp(at).$$

The same differential equation models a savings account, a loan, or radioactive decay (with $a < 0$).

Bounded Growth of a Population:

If resources are not unlimited, we speak of bounded growth. In bounded growth, the rate of change is proportional to the so-called saturation deficit (the difference between the upper (or lower) limit S and the old stock $y(t)$). This leads to the following differential equation:

$$\dot{y}(t) = k(S - y(t)), \quad t > 0.$$

The corresponding solution is:

$$y(t) = S - ce^{-kt}, \quad t \geq 0,\, c = S - y(0).$$

10.1.2 Higher Order and Systems of Differential Equations

Example 2: Motion in the Earth's Gravitational Field: A body with mass m is in the Earth's gravitational field (mass M). If s is the distance body—Earth center, then the gravitational force according to the law of gravity is given by:

$$K = \gamma \frac{M\,m}{s^2}, \quad \gamma = \text{gravitational constant.}$$

At time $t = 0$, the distance of the body is s_0. It moves with the initial velocity v_0, upwards if v_0 is positive and downwards if v_0 is negative. After t time units, it has the distance $s(t)$ from the Earth's center. Air friction is neglected. According to Newton's law, the *second-order* differential equation is:

$$m\ddot{s}(t) = -\gamma \frac{M\,m}{s(t)^2}. \tag{25}$$

This differential equation has many solutions. To specify one, additional initial conditions (initial position, initial velocity) are required:

$$s(0) = s_0, \quad \dot{s}(0) = v_0.$$

Here, an explicit solution in the form $s(t) = \dots$ cannot be given. However, *numerical methods* can easily provide solutions with any desired accuracy.

Only the simplest differential equations can be solved analytically. In practice, one is mostly reliant on numerical solution methods.

In the differential equation 25, *second* derivatives appear; it is a *second-order differential equation.* If we introduce, in addition to the displacement $s(t)$, the velocity $v(t)$ as a second sought function, this equation can be equivalently written as a *system of two first-order differential equations*:

$$\begin{aligned}\dot{s}(t) &= v(t)\\ \dot{v}(t) &= -\gamma\frac{M}{s(t)^2}.\end{aligned}$$

Justification: The first equation defines the velocity v as the time derivative of the displacement s; the second equation formulates, because $\dot{v}(t) = \ddot{s}(t)$, the force law 25.

The general problem is formulated as follows:

System of n ordinary differential equations 1st order, initial value problem

Seeking n functions $y_1(x), \dots, y_n(x)$, which satisfy the system

$$\left.\begin{array}{rcl} y_1' &=& f_1(x, y_1, \dots, y_n)\\ y_2' &=& f_2(x, y_1, \dots, y_n)\\ \vdots && \vdots\\ y_n' &=& f_n(x, y_1, \dots, y_n)\end{array}\right\} \text{differential equations}$$

$$\left.\begin{array}{rcl} y_1(x_0) &=& y_{10}\\ y_2(x_0) &=& y_{20}\\ \vdots && \vdots\\ y_n(x_0) &=& y_{n0}\end{array}\right\} \text{initial conditions}$$

For $n = 2$, a notation with two functions $y(x)$ and $z(x)$ (or, as in the previous example, $s(t)$ and $v(t)$) is simpler than the index notation $y_1(x)$ and $y_2(x)$: differential equations

$$\begin{array}{rcl} y' &=& f(x, y)\\ z' &=& g(x, y)\end{array}$$

However, with more equations, letters quickly run out, and the formulas for computational methods become confusing. A systematic notation that also simplifies computer programs combines functions vectorially:

$$\mathbf{y}(x) = \begin{bmatrix} y_1(x)\\ y_2(x)\\ \vdots\\ y_n(x)\end{bmatrix}, \quad \mathbf{y}'(x) = \begin{bmatrix} y_1'(x)\\ y_2'(x)\\ \vdots\\ y_n'(x)\end{bmatrix}, \quad \mathbf{f}(x, \mathbf{y}) = \begin{bmatrix} f_1(x, y_1, \dots, y_n)\\ f_2(x, y_1, \dots, y_n)\\ \vdots\\ f_n(x, y_1, \dots, y_n)\end{bmatrix}$$

Then the initial value problem is simply

$$\begin{aligned} \mathbf{y}' &= \mathbf{f}(x, \mathbf{y}) \\ \mathbf{y}(x_0) &= \mathbf{y}_0 \end{aligned}$$

Ordinary differential equation of higher order
An ODE of higher order can be transformed into an equivalent system of 1st order ODEs by introducing auxiliary functions.
This transformation is often necessary in practical problems (and also a popular exam question). The exercise materials provide detailed instructions and examples.

10.2 Numerical Methods

Explicit One-Step Methods

Important methods are the Euler polygon method (*because it is the simplest: 1st order*), Heun's method, and the modified Euler method (*because they are more accurate: 2nd order*), as well as the classical Runge-Kutta method (*because it is often used in practice; 4th order*).

For the numerical solution of an ODE, an explicit one-step method determines, based on the initial conditions, a sequence of value pairs $(x_0, y_0), (x_1, y_1), (x_2, y_2), \ldots$, which should approximate the behavior of the sought function $y = y(x)$. Scheme:

```
Choose step size $h$ and a maximum number of steps $N$;
set $x_0$ and $y_0$ according to initial conditions;
for $i = 0, 1, \ldots, N$
    $x_{i+1} = x_i + h$ ;
    $y_{i+1} = y_i + hF(x_i, y_i, h)$ .
```

The function $F(x, y, h)$ is called the *method function* of the respective method. Geometrically interpreted, $F(x, y, h)$ gives the *direction of progression* . In the classical Euler method (Euler polygon method), it is equal to the tangent slope at the starting point,

$$F(x, y, h) = f(x, y),$$

in the modified Euler method

$$F(x, y, h) = f\left(x + \frac{h}{2}, y + \frac{h}{2} f(x, y)\right),$$

in Heun's method

$$F(x, y, h) = \frac{1}{2}(k_1 + k_2)$$

with

$$\begin{aligned} k_1 &= f(x, y) \\ k_2 &= f(x + h, y + hf(x, y)), \end{aligned}$$

and in the classical Runge-Kutta method

$$F(x, y, h) = \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

with

$$\begin{aligned}
k_1 &= f(x,y) \\
k_2 &= f\left(x+\frac{h}{2}, y+\frac{h}{2}k_1\right) \\
k_3 &= f\left(x+\frac{h}{2}, y+\frac{h}{2}k_2\right) \\
k_4 &= f(x+h, y+hk_3).
\end{aligned}$$

10.3 Illustrations for a Simple Initial Value Problem

Geometric Interpretation
An ordinary differential equation provides a direction field

Given the differential equation

$$y' = \frac{xy}{4} - 1.$$

It assigns a slope y' to each point (x,y) in the plane. If we consider these slopes as direction vectors (with a uniform length), we obtain a *direction field*, as shown in Figure 9.

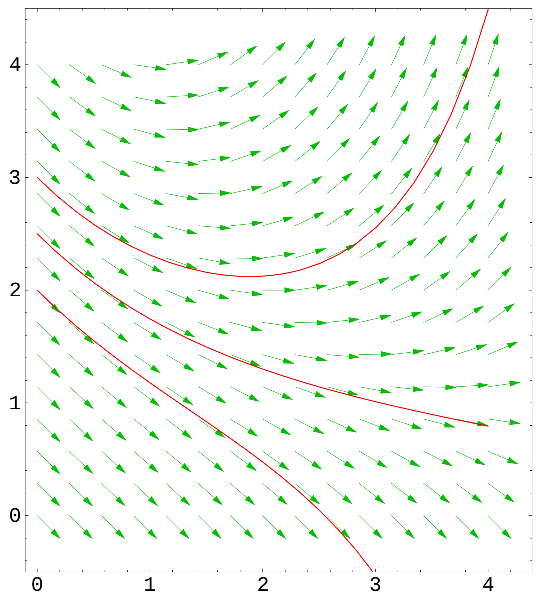


Figure 9: The differential equation provides a direction field. Three solutions with different initial conditions are plotted.

A solution curve starts at a point (x_0, y_0) determined by the initial condition, heading in the direction indicated there. In its further course, the curve always follows the direction arrows, adapting opportunistically to the prevailing direction at each point it reaches.

A one-step method, such as Euler's method, also starts at the point (x_0, y_0). Euler's method also precisely follows the direction given at (x_0, y_0) and stubbornly adheres to this decision for a step size h. It reaches a point (x_1, y_1) with this strategy. Only there does it reconsider its path and reorients itself, again precisely following the prevailing direction at (x_1, y_1). The smaller the step size, i.e., the more often the method adapts to the prevailing conditions, the better it should follow the exact solution curve. Figure 10 illustrates these ideas. A clarification of the idea and an examination of the errors are presented in the next chapter.

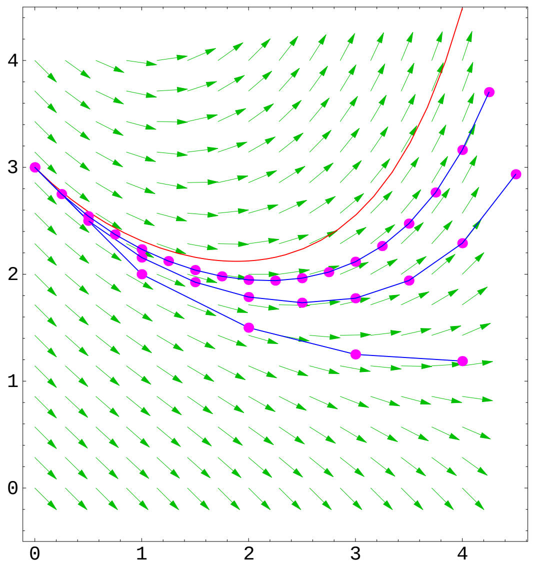


Figure 10: For the differential equation $y' = xy/4 - 1$ with initial condition $y(0) = 3$, the exact solution as well as three approximations using the Euler's polygonal method with step sizes $h = 1; \frac{1}{2}; \frac{1}{4}$ are plotted.

The Euler's polygonal method, however, is shortsighted in the sense that it simply chooses the direction given at the starting point as the progression direction (method function). The modified Euler method attempts to act more foresightedly: it first advances half a step, explores the direction given there, and chooses it as the progression direction (method function). See Figures 11 and 12 for illustrations.

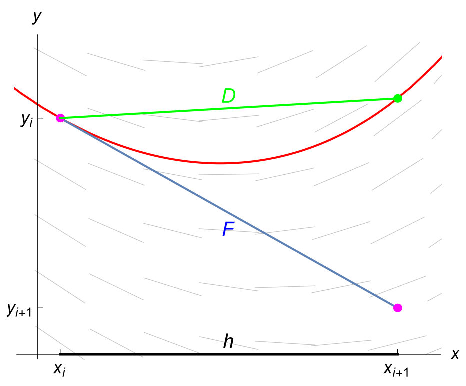


Figure 11: Method function F of the explicit Euler method and exact direction D.

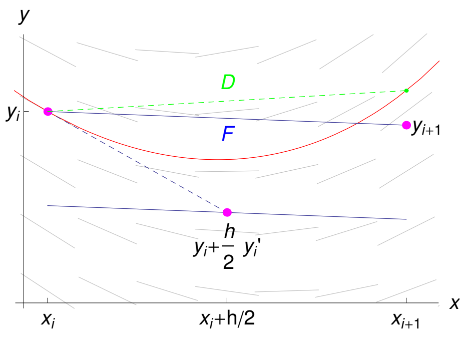


Figure 12: Choice of the method function F (geometric: progression direction) in the modified Euler method. F approximates the exact difference quotient D better than in the explicit Euler method, compare Figure 11.

Similarly, the method of Heun proceeds. It takes a step in the same direction as the ordinary polygonal method, explores the direction at the target point, and chooses as the final progression direction the average of the directions at the starting and (preliminary) target points. See Figure 13

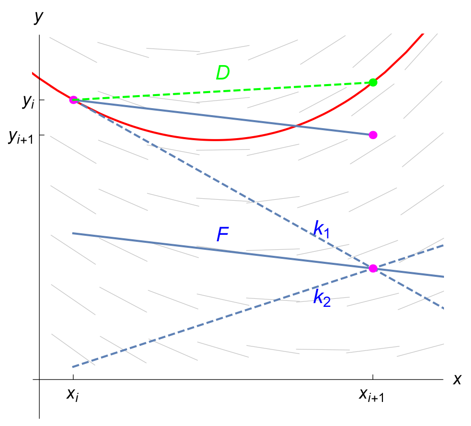


Figure 13: Choice of the method function F (geometric: progression direction) in the Heun method: average of initial direction and direction at the (approximate) end point. F approximates the exact difference quotient similarly accurately to the modified Euler method D.

There are still other ways to improve the simple Euler polygonal method in this way. Different names are used in the literature for these methods. The following method, which is attributed to Heun and is occasionally referred to as Heun's method:

$$F(x, y, h) = \frac{1}{4}(k_1 + 3k_2)$$

with

$$\begin{aligned} k_1 &= f(x, y) \\ k_2 &= f(x + \frac{2}{3}h, y + \frac{2}{3}hf(x, y)) \end{aligned}$$

The classical Runge-Kutta method perfects the interplay of tentative probing and exploring the direction: it calculates four different slopes $k_1, \ldots, k_4$, one at the starting point, two in the middle, and one at the target, and chooses as the method function a weighted average of the directions calculated in this way.

In general, methods of this kind are referred to as Runge-Kutta methods (hence the addition ”classical” above). Modern Runge-Kutta methods try to achieve a high order with minimal computational effort while providing estimates of the error size or the optimal step size.

10.4 Discretization Error

When a numerical method is supposed to calculate approximate values y_i of the solution y of an initial value problem, questions about the size of the error and the number of correct digits in the result are important.

Two types of errors need to be distinguished in the one-step method:

- In each individual step, the method function chooses a progression direction that usually does not exactly hit the value of the solution. This directional difference is the *local discretization error* .

- The effects of the local discretization errors in individual steps accumulate. As a result, the numerical solution deviates from the exact solution. Ultimately, of course, this error, the *global discretization error* , is of interest. It can be estimated with the local discretization error.

Figures 11, 12, and 13 show different local discretization errors. The direction chosen by the method function F is plotted, as well as (for the exact solution starting from y_i) the exact direction determined by the difference quotient D.

Both F and D depend on x_i, y_i, and the step size h and approach the tangent slope $f(x_i, y_i)$ for $h \to 0$. It holds

$$\lim_{h \to 0} D(x_m, y_m, h) = f(x_m, y_m).$$

The deviation d of the direction determined by the method function F from the exact direction D

$$d(x_m, y_m, h) = |F(x_m, y_m, h) - D(x_m, y_m, h)|$$

is called the *local discretization error* . Naturally, this error should become smaller as the step size h is chosen to be smaller. Information about how much d depends on the step size is provided by the *order of the method* .

Order of a One-Step Method
The largest natural number p with

$$d(x_m, y_m, h) = O(h^p)$$

is called the order of the method.

Order 1: Error d proportional to step size
Order 2: Error d proportional to the square of the step size
And so on.

The order of the local discretization error is important because it can be estimated using mathematical methods (such as Taylor series expansion). The effects of the local discretization errors in individual steps accumulate. As a result, the numerical solution deviates from the exact solution. Ultimately, of course, this error, the *global discretization error* , is of interest. It can be estimated using the local discretization error.

Local and Global Discretization Error
The error d between the method function F and the exact direction D

$$d(x_m, y_m, h) = |F(x_m, y_m, h) - D(x_m, y_m, h)|$$

is called the *local discretization error* at the point (x_m, y_m).
If Y is the exact solution to the initial value problem

$$y' = f(x, y), \quad y(x_0) = y_0 \ ,$$

and y_m is the approximate solution at the point x_m, the error

$$g(x_m, h) = |y_m - Y(x_m)|$$

is called the *global discretization error* .

We demand from any one-step method that at a given point x, the error $g(x, h)$ goes to zero as h approaches zero and call the largest natural number p satisfying

$$g(x, h) = |y(x, h) - Y(x)| = O(h^p)$$

the convergence order of the method. The following relationship holds between the local and global discretization errors:

Convergence of the One-Step Method
If the local discretization error is of order $p \geq 1$ and F satisfies a Lipschitz condition, the one-step method converges with order p.

Order of Individual Methods
Order 1: Euler's Polygon Method
Order 2: Modified Euler Method, Heun's Method
Order 4: Classical Runge-Kutta Method

10.5 When to Use Which Method

For example, if you search for `ode45` in the MATLAB help, MATLAB lists the following differential equation solvers: `ode23`, `ode45`, `ode113`, `ode15s`, `ode23s`, `ode23t`, `ode23tb`. Which method should you use? The first criterion is the convergence order (higher means more accuracy). However, if the expected solution is not smooth (for example, if f has jump discontinuities, corners, or higher derivatives do not exist), a method with lower convergence order may work faster and require less computational power than a method with higher convergence order.

Solver Selection Criteria

- Convergence Order - Accuracy of the Approximation: MATLAB's `ode45` is a 5th-order method, but only if the function you pass to `ode45` is sufficiently differentiable. If not, `ode23` might work more efficiently.
- Stiffness of the System: If the system is stiff, you should use `ode15s`, `ode23t`, or `ode23tb`.
- Complex Right-hand Side: Explicit solvers (`ode45`, `ode23`, `ode23`, `ode113`) often compute faster because, unlike implicit methods, they do not have to solve equation systems in every step. MATLAB's help recommends: *"ode113 can be more efficient than ode45 [...] when the ODE function is expensive to evaluate"*.

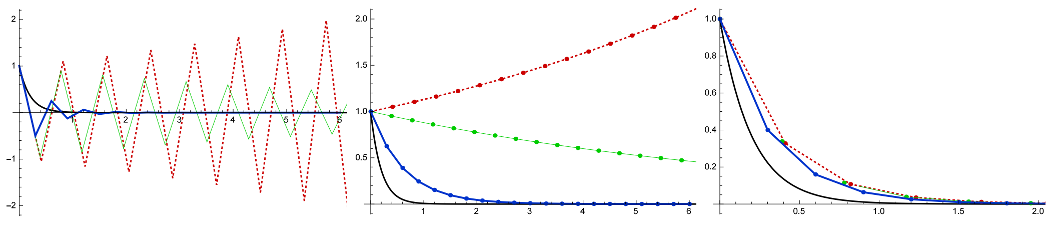
The following sections address issues in the application of numerical solution methods. Important topics include:

- Explicit versus implicit methods
- Stability
- Stiffness

First, let me explain the difference between explicit and implicit methods in this context. Consider the following differential equation:

$$\begin{cases} y'(x) &= -5\,y(x), \\ y(0) &= 1. \end{cases}$$

In the following plots, the exact solution e^{-5x} (black curve) has been approximated using various methods with step sizes of 0.3 (blue), 0.39 (green), and 0.41 (red). From left to right: explicit Euler method, Heun's method, implicit Euler method.



Upon inspecting the graphs, it is evident that the explicit Euler method oscillates and diverges for a step size of 0.41. Similarly, Heun's method also diverges at this step size. In this context, these methods are considered *unstable*.

For step sizes < 0.4, the methods roughly capture the qualitative behavior of the exact solution: exponential decay. This behavior is referred to as *stable*.

A method is called stable if it qualitatively computes exponentially decaying solutions correctly. Stability can depend on the step size.

- There are various definitions for the stability of a numerical method.
- In general, one can remember: Implicit methods are always stable, explicit methods are stable only for small step sizes.
- Stability problems mainly arise in *stiff differential equations*. For such problems, implicit methods are better suited.

If one calculates the individual steps of the explicit Euler method, one can understand what happens. The computation steps are as follows:

$$\begin{array}{lcl} y_1 & = & y_0 + h \cdot (-5) y_0 = (1-5h) y_0, \\ y_2 & = & y_1 + h \cdot (-5) y_1 = (1-5h) y_1 = (1-5h)^2 y_0, \\ y_3 & = & (1-5h) y_2 = \ldots = (1-5h)^3 y_0, \\ \ldots & \ldots & \ldots \\ y_k & = & (1-5h) y_{k-1} = \ldots = (1-5h)^k y_0. \end{array}$$

The explicit Euler method calculates the next value from the previous value by multiplying it with $(1-5h)$.

For $h = 0.41$, $(1-5h) = -1.05$, and the sequence $(1-5h)^k$ tends to $\pm\infty$ with alternating signs as $k \to \infty$.

On the other hand, for $h = 0.39$, $(1-5h) = -0.95$, and the sequence $(1-5h)^k$ oscillates between positive and negative values but tends to zero as $k \to \infty$.

Analyzing the implicit Euler method leads to the following calculation:

$$\begin{array}{lcllcl} y_1 & = & y_0 + h \cdot (-5) y_1, & \to & y_1 = \dfrac{y_0}{(1+5h)}, \\ y_2 & = & y_1 + h \cdot (-5) y_2, & \to & y_2 = \dfrac{y_1}{(1+5h)} = \dfrac{y_0}{(1+5h)^2}, \\ y_3 & = & y_2 + h \cdot (-5) y_3, & \to & y_3 = \dfrac{y_2}{(1+5h)} = \dfrac{y_0}{(1+5h)^3}, \\ \ldots & \ldots & \ldots \\ y_k & = & \cdots = \dfrac{y_{k-1}}{(1+5h)} = \dfrac{y_0}{(1+5h)^k}. \end{array}$$

For $h = 0.41$, $\frac{1}{(1+5h)} = 0.327$, and the sequence $\frac{1}{(1+5h)^k}$ monotonically tends to zero as $k \to \infty$.

In fact, for all $h > 0$, $\frac{1}{(1+5h)^k}$ monotonically tends to zero.

Analyzing the Heun method, the sequence is given by:

$$\left\{ \left(1 - 5h + \frac{(5h)^2}{2}\right)^k y_0 : k \in \mathbb{N} \right\}.$$

For $h = 0.41$, the expression within the brackets is 1.051, and therefore, the sequence monotonically increases and diverges.

For $h = 0.39$, the expression within the brackets is 0.951, and the sequence converges to zero.

Stability region

For systematic stability investigations, it is useful to consider a more general model equation $y' = \lambda y$ with parameter λ. Complex values of λ are also examined; the corresponding exact solutions are damped sinusoidal oscillations. Thus, this simple model equation describes the most important processes that occur in practical applications, depending on the λ value.

Depending on the combination of h and λ, methods can be stable or not. However, it depends only on the value of the product $\xi = h \cdot \lambda$. Therefore, one defines:

The stability region of a method is the set of complex numbers $\xi = h \cdot \lambda$ for which, when solving the test equation

$$y' = \lambda y, \quad y(0) = 1$$

with a fixed step size h, it provides a bounded sequence of approximations.

In the case of the explicit Euler method, we have

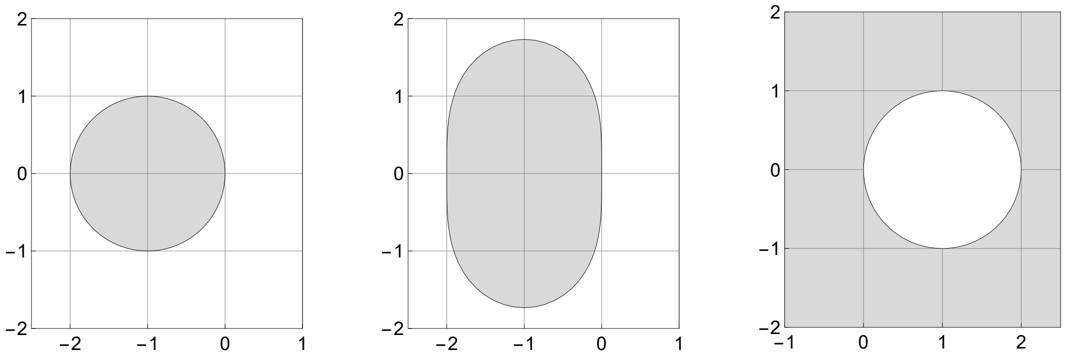
$$\begin{aligned} \mathcal{B} &= \left\{\lambda \in \mathbb{C} : \lim_{k\to\infty} y_k^\lambda < \infty\right\} = \left\{\lambda \in \mathbb{C} : \lim_{k\to\infty} (1+\lambda)^k < \infty\right\} \\ &= \{\lambda \in \mathbb{C} : |1+\lambda| \le 1\} = \left\{a + ib = \lambda \in \mathbb{C} : (a+1)^2 + b^2 \le 1\right\}. \end{aligned}$$

For the implicit Euler method, we have

$$\begin{aligned} \mathcal{B} &= \left\{\lambda \in \mathbb{C} : \lim_{k\to\infty} y_k^\lambda < \infty\right\} = \left\{\lambda \in \mathbb{C} : \lim_{k\to\infty} (1+\lambda)^{-k} < \infty\right\} \\ &= \left\{a + ib = \lambda \in \mathbb{C} : (a-1)^2 + b^2 > 1\right\}. \end{aligned}$$

The stability regions of the explicit, modified, and implicit Euler methods are depicted here (from left to right) as gray areas in the complex plane. The Heun method has the same stability region as the modified Euler method.

Practically relevant is the left half-plane[14]; it corresponds to decaying exponential and oscillatory processes. For stability, λh must be in the gray regions.



For explicit methods, there are limitations: you must choose a small enough step size; otherwise, λh is outside the stability region. For implicit methods, the entire left half-plane is in the stability region, and you can choose the step size arbitrarily large for $\lambda \le 0$ (decaying exponential and oscillatory processes).

[14] In the right half-plane, exponentially increasing processes occur, leading to exponential growth in the errors of all numerical methods. The question of stability is not meaningful here.

10.5.1 Stiffness

Differential equations describing chemical or physical processes often have solutions that consist of components decaying at very different rates. This occurs when sub-processes run at very different speeds. Consider the following simple example:

$$\begin{aligned} \dot{y}_1(t) &= -y_1(t) + 50y_2(t), \\ \dot{y}_2(t) &= -70y_2(t), \end{aligned}$$

with initial values $y_1(0) = 1$ and $y_2(0) = 10$. Since the matrix

$$\begin{pmatrix} -1 & 50 \\ 0 & -70 \end{pmatrix}$$

has eigenvalues $\lambda_1 = -1$ and $\lambda_2 = -70$ with eigenvectors $v_1 = (1, 0)^T$ and $v_2 = -(50, 96)^T$, the solution is given by

$$y_1(t) = 8.24638e^{-t} - 7.2464e^{-70t}, \quad y_2(t) = 10e^{-70t}.$$

To compute the fastest decaying component with an accuracy of 10^{-3} using numerical solutions, the step size must be chosen so that e^{-70h} agrees with $F(-70h)$ to five decimal places. But after a relatively short time, e^{-70t} has practically completely decayed compared to e^{-t}. Nevertheless, the solver still needs to compute with a very small step size, even though the e^{-70t} component is essentially gone.

Definition 1: A linear system

$$\begin{cases} \dot{x}(t) &= Ax(t), \\ x(0) &= x_0. \end{cases}$$

is called stiff if the quantity

$$S := \frac{\max\{\Re(\lambda_j) : \lambda_j \text{ ist Eigenwert von } A \text{ mit negativen Realteil}\}}{\min\{\Re(\lambda_j) : \lambda_j \text{ ist Eigenwert von } A \text{ mit negativen Realteil}\}}$$

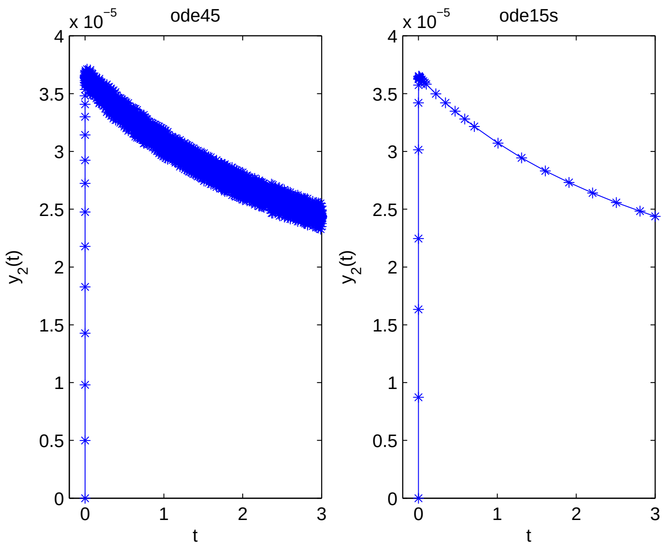
is on the order of (or greater than) 1000.

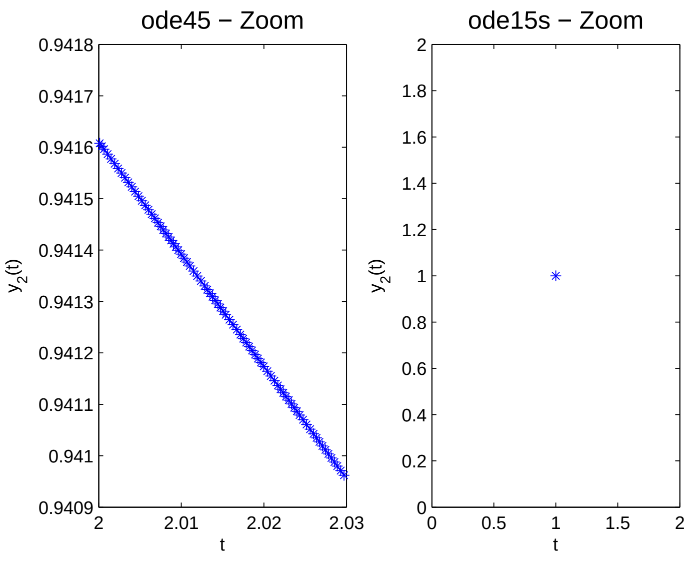
Remark 1: Only eigenvalues with a negative real part are considered since they correspond to components that decay to 0 for large t. Eigenvalues with a positive real part lead to components that do not converge.

Another (non-constructed) example is the following differential equation, the *Robertson differential equation*: Let $y(0) = (1, 0, 0)$ and

$$\begin{aligned} \dot{y}_1(t) &= -0.04\,y_1(t) \\ &\quad +10^4 y_2(t)\,y_3(t), \\ \dot{y}_2(t) &= 0.04\,y_1(t) - 10^4 y_2(t)\,y_3(t) \\ &\quad -3 \times 10^7 y_2^2(t), \\ \dot{y}_3(t) &= 3 \times 10^7 y_2^2(t). \end{aligned}$$

The following figures show how the differential equation is numerically solved once with the `ode45` solver and once with the `ode15s` solver. Although `ode45` uses a higher-order method, the solver converges worse than the implicit Euler method used by the `ode15s` solver. In the


ode45
ode15s
x 10^−5
y_2(t)
t


ode45 − Zoom
ode15s − Zoom
y_2(t)
t

second figure, a subinterval was selected. In the time period during which the `ode45` solver calculated 81 points, the `ode15s` solver calculated one point.

How to determine if it makes sense to use a simple but implicit solver

Given is the second-order linear differential equation

$$\left\{\begin{array}{rcl} \dot{x}(t) & = & \begin{pmatrix} 0.5 & -0.5 \\ 10 & 10 \end{pmatrix} x(t), \\ x(0) & = & \begin{pmatrix} 1 \\ 1 \end{pmatrix}. \end{array}\right. \tag{26}$$

Solving this with the `ode45` solver on the time interval $[0, 20]$, MATLAB discretizes time into 265 points, while the `ode15s` solver computes the function at only 65 points with the same accuracy. Examining the system more closely, it is observed that the matrix has two eigenvalues, -0.5 and -10. As the `ode45` solver is an explicit method, the time steps towards the eigenvalue -10 need to be very small due to stability issues. On the other hand, the solution in this direction is very smooth and converges rapidly to zero, suggesting that larger steps could be taken. The `ode15s` solver, being an implicit solver, can adapt the step size to the problem.

Most systems are nonlinear, so how can the concept of stiffness be transferred to a nonlinear system?

Let

$$\left\{\begin{array}{rcl} \dot{x}(t) & = & f(x(t)), \\ x(0) & = & x_0. \end{array}\right. \tag{27}$$

Define stiffness for the linearized system by analyzing the *local* behavior of the exact solution $x(t)$ in the vicinity of a point t_0. Let

$$x(t) \quad = \quad x(t_0) + z(t), \quad t_0 \le t \le t_0 + h,$$

be the linearized system at the point t_0. Expanding the system (27) using Taylor approximation yields

$$\dot{x}(t) \quad = \quad f(x(t_0)) + \nabla f(x(t_0))(x(t_0) - x(t)) + O(x(t_0) - x(t)).$$

Substituting $x(t)$ with the approximation $x(t_0) + z(t)$, we get

$$\dot{x}(t) \quad = \quad f(x(t_0)) + \nabla f(x(t_0))z(t) + O(x(t_0) - x(t)).$$

On the other hand, $\dot{x}(t) = \dot{x}(t_0) + \dot{z}(t) = f(x(t_0)) + \dot{z}(t)$. Substituting this above, we get

$$f(x(t_0)) + \dot{z}(t) \quad = \quad f(x(t_0)) + \nabla f(x(t_0))z(t) + O(x(t_0) - x(t)),$$

and consequently,

$$\dot{z}(t) \quad = \quad \nabla f(x(t_0))z(t).$$

The infinitesimal change at time t_0 thus solves the above linear system. Moreover, for a linear system, stiffness can be defined as mentioned above. This definition can be extended to nonlinear systems through the given calculation.

Definition 2: Let

$$A(x_0) := \frac{\partial f}{\partial x}\Big|_{x=x_0}.$$

and

$$S(x_0) := \frac{\max\{\Re(\lambda_j) : \lambda_j \text{ is an eigenvalue of } A(x_0) \text{ with negative real part }\}}{\min\{\Re(\lambda_j) : \lambda_j \text{ is an eigenvalue of } A(x_0) \text{ with negative real part }\}}$$

We call a system stiff at x_0 if $S(x_0)$ is on the order of 1000 or greater.

Example 3: Robertson Differential Equation Let $y(0) = (1, 0, 0)$ and

$$\begin{aligned} \dot{y}_1(t) &= -0.04\, y_1(t) + 10^4 y_2(t)\, y_3(t), \\ \dot{y}_2(t) &= 0.04\, y_1(t) - 10^4 y_2(t)\, y_3(t) - 3 \times 10^7 y_2^2(t), \\ \dot{y}_3(t) &= 3 \times 10^7 y_2^2(t). \end{aligned}$$

This means with $\mathbf{y} = (y_1, y_2, y_3)^T$, we can also write

$$\dot{\mathbf{y}}(t) = f(\mathbf{y}(t)),$$

with

$$f(\mathbf{x}) = f\left(\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}\right) = \begin{pmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \\ f_3(\mathbf{x}) \end{pmatrix} = \begin{pmatrix} -0.04\, x_1 + 10^4 x_2\, x_3 \\ 0.04\, x_1 - 10^4 x_2\, x_3 - 3 \times 10^7 x_2^2, \\ 3 \times 10^7 x_2^2 \end{pmatrix}$$

The Jacobian matrix is now

$$\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} = \begin{pmatrix} \frac{df_1}{dx_1} & \frac{df_1}{dx_2} & \frac{df_1}{dx_3} \\ \frac{df_2}{dx_1} & \frac{df_2}{dx_2} & \frac{df_3}{dx_3} \\ \frac{df_3}{dx_1} & \frac{df_3}{dx_2} & \frac{df_3}{dx_3} \end{pmatrix} = \begin{pmatrix} -0.04 & 10^4\, x_3 & 10^4\, x_2 \\ 0.04 & -10^4 x_3 - 6 \cdot 10^7\, x_2 & -10^4 x_2 \\ 0 & 6 \cdot 10^7\, x_2 & 0 \end{pmatrix}$$

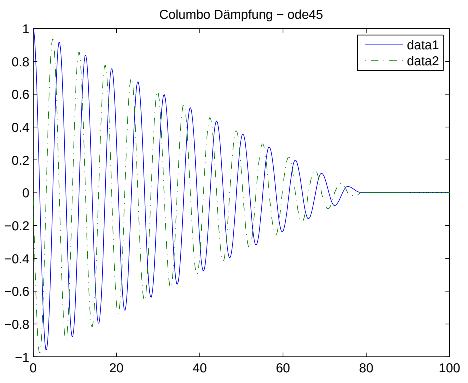
Computing the eigenvalues at the point $\mathbf{x} = (1, 0, 0)^T$ yields the values $-0.04, 0, 0$ in Mathematika, at point $\mathbf{x} = (1, 0.05, 0.05)^T$, the values are $-3 \cdot 10^6, -500, 1.4 \cdot 10^{-14}$, at point $\mathbf{x} = (1, 0.1, 0.1)^T$, the values are $-6.0 \cdot 10^6, -1000, 5.26 \cdot 10^{-14}$), and $\mathbf{x} = (1, 1, 1)^T$ yields the values $-6 \cdot 10^7, -10000, 1.94 \cdot 10^{-12}$. The ratio between the smallest and largest negative real eigenvalue is respectively

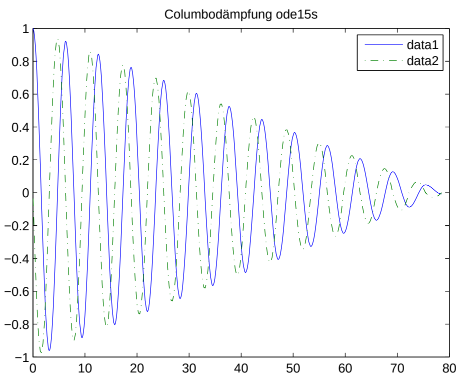
$$\begin{aligned} S((1, 0, 0)^T) &\sim \infty, \\ S((1, 0.1, 0.1)^T) &\sim \frac{3 \cdot 10^6}{500} \sim 10^4, \\ S((1, 0.1, 0.1)^T) &\sim \frac{6.0 \cdot 10^6}{1000} \sim 10^3, \\ S((1, 1, 1)^T) &\sim \frac{6 \cdot 10^7}{10000} \sim 5 \cdot 10^2. \end{aligned}$$

Example 4: Coulomb Damping Consider the following second-order differential equation:

$$m\ddot{x} + cx = -\mu m g \,\operatorname{sign}(\dot{x}). \tag{28}$$

The left figure shows the solution obtained with the ode45 solver. The number of support points was 112221, the right figure shows the solution obtained with the ode15s solver. The number of support points was 945. The reason is the function $x \mapsto \operatorname{sign}(x)$, which is not differentiable at 0.





One difficulty with implicit systems is the evaluation of the left-hand side. Suppose one computes the example above with the implicit Euler scheme. So, at each time step, the following system of equations is solved:

$$\hat{\mathbf{y}}_k \quad = \quad \begin{pmatrix} \hat{y}_k^1 \\ \hat{y}_k^2 \end{pmatrix} = \hat{\mathbf{y}}_{k-1} + h \begin{pmatrix} \hat{y}_k^2 \\ -c\,\hat{y}_k^1 - \nu mg\,\mathrm{sign}(c\,\hat{y}_k^1) \end{pmatrix}.$$

This system must be solved for $\hat{\mathbf{y}}_k = (\hat{y}_k^1, \hat{y}_k^2)^T$. Typically, this is done with fixed-point iteration, taking into account that the function $x \mapsto \mathrm{sign}(x)$ is not Lipschitz continuous.

11 Fourier Series, Fourier Analysis, Fast Fourier Transform (FFT)

Es gibt im Hauptteil des Skriptums (noch) kein eigenes Kapitel dazu. Der Stoff wird anhand der Vorlesungsfolien (klick!) und im Übungsteil L 10 erklärt und erarbeitet.

L 1 First Lab Unit

Four important rules:
1. It never works the first time.
2. It probably won't work the second time either.
3. It works better when all cables are plugged in.
4. If nothing else works, read the manual.

L 1.1 MATLAB

The exercises begin with an introduction to the computational and programming environment MATLAB (short for "MATrix LABoratory"). This program is a standard tool in the academic and industrial sectors for technical and scientific calculations. For many numerical tasks, MATLAB provides solution functions and methods for visualization. At the same time, it is a modern programming language in which you can develop your own applications.

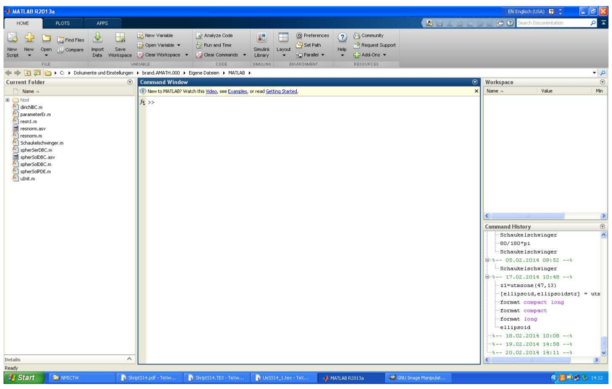
This exercise unit aims to provide you with a quick introduction.

Meanwhile, the programming language Python, along with its additional packages (numpy, scipy), offers almost the same functionality as MATLAB. All exercise tasks can be solved equally well in Python. If you are familiar with Python, numerical computing in MATLAB will be easy for you, and vice versa.

Several other free programming and computing environments also function more or less similarly to MATLAB: Scilab, Octave, FreeMat, Julia, R. The same principle applies: If you are proficient in MATLAB, transitioning to these software packages will be straightforward (and vice versa).

L 1.2 The Beginning

Start MATLAB on your computers like a typical MS-Windows application by double-clicking on the MATLAB icon. Once the MATLAB user interface (MATLAB *Desktop*) appears on the screen, you are ready for the first lesson.



```
New to MATLAB? Watch this Video, see Examples, or read Getting Started.
>>
```

```
Schaukelschwinger
80/180*pi
Schaukelschwinger
%-- 05.02.2014 09:52 --%
Schaukelschwinger
%-- 17.02.2014 10:48 --%
z1=utmzone(47,13)
[ellipsoid,ellipsoidstr] = utm
format compact long
format compact
format long
ellipsoid
%-- 18.02.2014 10:08 --%
%-- 19.02.2014 14:58 --%
%-- 20.02.2014 14:11 --%
```

You see the MATLAB user interface with the *Command Window* as the large window in the middle.

If the MATLAB user interface looks different from the illustration here—there are many configuration options—first restore the default standard layout. Locate the *Layout* icon in the middle of the top bar and click on *Layout–SELECT LAYOUT–Default.*

Feel free to try some of the other options offered if you like. To make the interface look exactly as shown here, set *Layout–SHOW–Command History–Docked.*

But finally, please restore the default layout by selecting *Default.* Then you will see, from left to right, four windows: *Current Folder,* *Command Window, Workspace, Command History.* We will deal with the additional windows later.

The *Command Window* is like a large blank sheet in front of you. However, unlike in a word processing program, you cannot simply write anywhere but only in the current command line. It is marked on the screen by the command prompt (*prompt*) `>>`.

Start by entering the following commands. What you need to type is also marked in the documentation with a preceding `>>`. Below is the response from MATLAB (what you should see on the screen after pressing the Enter key).

```
>> 3-2

ans =

     1
```

Enter 3-2. The result is saved under the default name "ans."

```
>> ans

ans =

     1
```

You can retrieve the result under the label "ans" at any time...

```
>> zwa=ans+ans

zwa =

     2
```

... or use it in further mathematical expressions. You can also assign the value of an expression to a variable.

```
>> y=2^2 + log(pi)*sin(zwa);
>> y

y =

    5.0409

>>
```

You calculate $2^2 + \log(\pi) \cdot \sin(2)$. A **semicolon** at the end suppresses result output and line break. MATLAB still remembers y. You can retrieve the value of y anytime by simply typing "y."

```
>> format long g
>> y
y =
          5.04089993961332
>> y^10
ans =
          10594507.4098595
>>
>> format long e
>> y^10
ans =
     1.059450740985953e+07
>> format short g
>> y^10
ans =
   1.0595e+07

>>
```

MATLAB computes with approximately sixteen-digit precision. In the default setting, it displays only four decimal places. You can set different display formats. The `e`-format always uses exponential notation, and the `g`-format writes the result as a floating-point number without an exponent if possible.
`format compact` suppresses blank lines in the output, allowing more output to fit on one screen. `format loose` adds blank lines for better readability.

```
>> theta=acos(-1)

theta =

    3.1416
```

MATLAB knows trigonometric functions. This is the arc cosine of -1 (**in radians!**)

```
>> help elfun

  Elementary math functions.

  Trigonometric.
    sin         - Sine.
    .
    .
    .
```

MATLAB knows a large number of so-called elementary functions. You can call up a list of available functions in this way. Additional help can be requested for each individual function.

Try to find out through the various options of MATLAB help: What is the full name of the function `sinh`?

Assignment: Use the *help browser* and draw a function graph The extensive MATLAB help (the *help browser*) provides more information on `sinh` and also shows how to plot the function.

Follow the instructions in the help to let MATLAB draw a function graph of `sinh`.

L 1.2.1 Repeat Input, Copy Previous Commands

If you make a mistake somewhere and need to repeat the input, you don't have to type everything again. You can use the $\boxed{\uparrow}$ key to recall previously entered commands and correct them if necessary.

If you enter the first few letters of a previous command and then press $\boxed{\uparrow}$, MATLAB remembers and completes those commands that began with those letters.

You can also copy commands from the lower-right window, the *Command History*, with a right-click copy, and paste them into the *Command Window* with a right-click paste. Even faster is right-click-*Evaluate Selection* or highlight-$\boxed{\text{F9}}$

Assignment: Draw more familiar functions: sine, cosine, exponential function, following the pattern of the previous section. Don't type everything again. Use the arrow key and/or the *Command History* to retrieve previous commands and only correct the function names.

L 1.2.2 Suppress Output, Clear Screen, Emergency Brake

Enter the following command:

```
>>  0:10000

ans =

  Columns 1 through 13

     0      1      2      3
  .
  .
  .
        9990         9991

  Columns 9997 through 10001

        9996         9997

>>
```

This command creates a long list. It serves here only as an example of an instruction that floods the screen with a large amount of output. Such commands should be terminated with a semicolon. Internally, calculations will be performed, but screen output will be suppressed.

```
>>  0:10000;
>>
```

That's better.

However, if an instruction refuses to end: Pressing the $\boxed{\text{Ctrl C}}$ key terminates the ongoing MATLAB computation. You can also use it to delete started command lines.

If MATLAB still does not respond to $\boxed{\text{Ctrl C}}$, it can only be stopped via the *Windows Task Manager*. In that case, the entire session with all previously calculated values is lost.

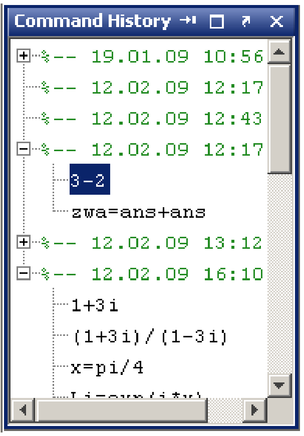
If you want to clear all input and output in the *Command Window*, use the command

```
>> clc
>>
```

Now the table is cleared.

L 1.3 Saving Commands in a Script File

If you've been working diligently so far, you may want to save your work. The window at the bottom left, the *Command History*, has faithfully preserved all commands. (If you don't see this window, you must set *Layout–SHOW–Command History–Docked*!)



```
⊞%-- 19.01.09 10:56
 %-- 12.02.09 12:17
 %-- 12.02.09 12:43
⊟%-- 12.02.09 12:17
   3-2
   zwa=ans+ans
⊞%-- 12.02.09 13:12
⊟%-- 12.02.09 16:10
   1+3i
   (1+3i)/(1-3i)
   x=pi/4
```

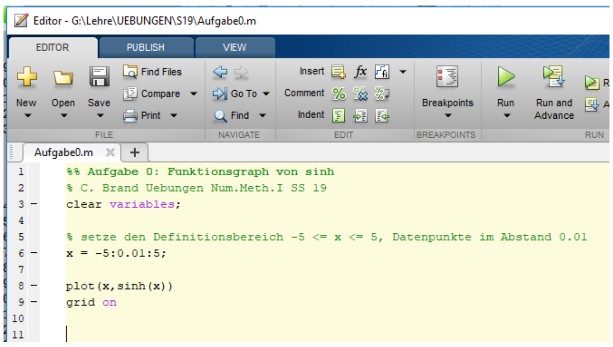
Scroll through the commands in the *Command History*. You can see all the commands you entered. If you scroll further back in time, you will find commands from earlier sessions, marked with date and time. They can be opened or closed by clicking.

Suppose you want to save the commands for the first exercise (plotting the graph of the hyperbolic sine function). Follow these steps:

Highlight (click!) the first command; hold down the shift key $\boxed{\Uparrow\quad}$ and scroll down in the command list; highlight (click!) the last command. The highlighted commands are now dark blue. Right-click→*Create Script*. The Editor window opens.

In the Editor, you can edit commands, unlike in the *Command Window*. You can insert and delete characters anywhere.

It is recommended to insert *comments* (starting with %) in your script. Your file could then look like this:


Editor - G:\Lehre\UEBUNGEN\S19\Aufgabe0.m

```
%% Aufgabe 0: Funktionsgraph von sinh
% C. Brand Uebungen Num.Meth.I SS 19
clear variables;

% setze den Definitionsbereich -5 <= x <= 5, Datenpunkte im Abstand 0.01
x = -5:0.01:5;

plot(x,sinh(x))
grid on

```

Save your file in the usual Windows way (Menu *Save – Save as...*) – preferably directly to a USB stick you have brought with you. MATLAB script files automatically get the `.m` extension.

With the start symbol in the menu bar, you can run your commands as a *script .m file.* This works as if you were entering the commands directly into the *Command Window.* The corresponding output appears in the *Command Window.*

However, the script file should actually contain all commands necessary to solve a task. A typical beginner's mistake is that the script uses variables that are not defined in the script. It works because you defined the variables beforehand, and therefore the variables are already present in the workspace. But if you or someone else starts the script in a new MATLAB session, an error message will occur.

Ensure that all required variables are defined in the script!
Tip: At the beginning of the script, use the command `clear variables`. This clears all variables in the workspace, and you immediately notice if you forgot definitions in the script.

Task 1: Script file for function plot

In MATLAB, choose a function from the areas of *Trigonometry, Exponents and Logarithms*, or if you are particularly curious, *Special Functions* that you are not familiar with. Create a script that plots the graph of this function. (Refer to the screenshot on page L-5 for guidance.)

Tip: Create a separate subdirectory for the MATLAB files you create in these exercises.
If you are working on university computers, it is best to save on a USB drive you brought with you!

In the Editor window, you will see a green "Run" arrow symbol in the middle at the top if the script file is saved without errors. Clicking on "Run" executes all commands in the script again.

(If a dialog box appears with a text like *File D:/work/Aufgabe1.m is not found in the current folder blabla blabla*, simply click on "Change Folder".)

L 1.4 Plotting Functions of the Form $y = f(x)$

In this section, you will learn to create simple function plots in the xy-plane using the `plot` command. (The MATLAB Help refers to it as “linear 2D-plots”.)

Work through the following example in the *Command Window*. The goal is to plot the graphs of two functions,

$$y = 3\cos x \quad \text{and } z = \log x,$$

for the domain $0 < x \leq 25$.

```
>> x=linspace(0.1,25,50)
```

This creates a row vector x containing 50 equidistant values in the interval $[0.1, 25]$. The output on the screen will be correspondingly long.

```
x =

  Columns 1 through 7

    0.1000    0.6082    1.1163
1.6245    2.1327    2.6408
3.1490
  .......

  Columns 43 through 49

   21.4429   21.9510   22.4592
22.9673   23.4755   23.9837
24.4918

  Column 50

   25.0000
```

You can see that indeed 50 values are generated. In MATLAB’s perspective, x is considered a matrix with one row and 50 columns (hence “Columns” in the output).

```
>> x=linspace(0.1,25,100);
```

Repeat the input (using the arrow key), but this time calculate 100 values and add a semicolon at the end: This suppresses the output, keeping the screen more organized. By the way, even if you don’t specify the number of values (using the command `x=linspace(0.1,25);`), MATLAB will default to generating 100 values.

```
>> x=linspace(0.1,25,100);
```

This command generates 100 equidistant values in the interval $[0.1, 25]$.

```
>> y = 3*cos(x);
```

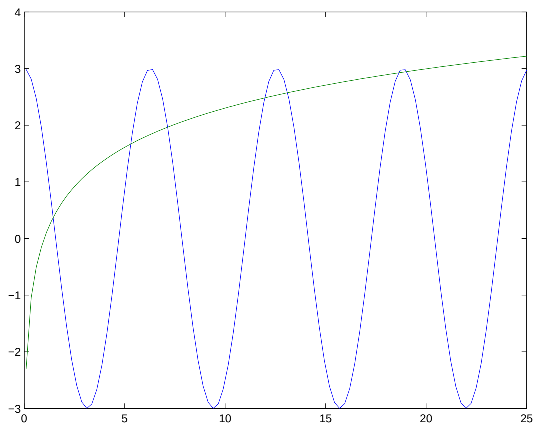
This command calculates the corresponding y value for each component in the vector x.

```
>> plot(x,y)
```

And this gives you a plot of the function graph.

```
>> z=log(x);
>> plot(x,y,x,z)
```

Generate another vector representing the function values of $\log x$. You can plot both function graphs in one plot. Note that for the second function, you need to specify the x values again, even though both function graphs are based on the same x values.

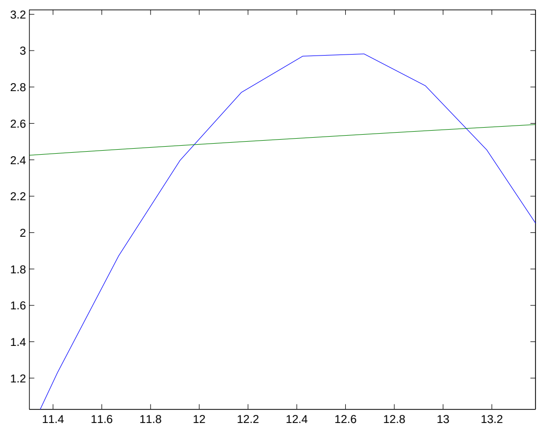


The graph should correspond to the above figure. The intersection points of the two curves correspond to solutions of the equation

$$3\cos x = \log x.$$

Compare this with Figure 1 and Chapter 1.4 in the lecture notes, where this example is discussed in detail.

If you click on the zoom-in button (with the magnifying glass symbol), you can then use the mouse to enlarge a region of the graph. You can approximate the solutions of the equation $3\cos x = \log x$ from the graphical representation. The example below shows two solutions near $x = 11.9$ and $x = 13.1$.



However, the magnification also shows that the graph does not represent the "real" functions but rather connects the individual data points of the vectors x, y, and z with straight lines. Therefore, you can only represent the solutions of the equation $3\cos x = \log x$ within the accuracy of the graph. Additionally, even if you zoom in, the axis labels display no more than three or four decimal places.

L 1.5 Additional Exercise Examples

You can work on the following examples during this exercise session or complete them on your own computer. Save the tasks as script files on a USB drive so that you can present your examples during consultations.

Task 2: Solutions of $3\cos x = \log x$

Create a more detailed version of the function graphs above with 1000 data points. Give your plot a title and label the axes. (Consult the MATLAB Help, ask your instructor, or seek help from a helpful neighbor to learn how to do this!)

Determine all solutions of the equation from the graph.

Note: Start with `x=linspace(0.1,25,1000);`

> If you have the documents in paper form or can write into your digital copy, enter answers here. In any case, include the required answers as comment lines in your script.

The solutions are (to three decimal places):

____

____

Task 3: Quadratic Equation

Find the smaller root of

$$x^2 - 12345678x + 9 = 0$$

Use calculations in the *Command Window* and save the solution in an organized and commented script file.

(Use the display format `format long e`)

- The common solution formula $x_{1,2} = -p/2 \pm \sqrt{p^2/4 - q}$ yields

  ____

- The numerically correct calculation of the *smaller* root of a quadratic equation first computes the *larger* root x_{gr} with the standard formula. The smaller root x_{kl} is then obtained as

  $$x_{\text{kl}} = \frac{q}{x_{\text{gr}}}$$

  Result: ____

- Solving for the linear term,

  $$x = \frac{x^2 + 9}{12345678}$$

  Fixed-point iteration in the form

```
>> x=0;
>> x = (x^2+9)/12345678
x =
       7.290000597780049e-007
```

  Repeat until the value no longer changes

Result: ______________________________

Task 4: Fixed-Point Iteration:

As in the previous task, find a fixed point of the function $\phi(x) = 1/\exp x$ by repeated evaluation. Start with 5, precision `format long e`.

Result: ______________________________

Also, examine how the accuracy increases from one step to the next. Can you find a rule like "on average, x iterations are needed per decimal place of accuracy?"

Task 5: Square Root Calculation:

Even the ancient Babylonians calculated square roots $\sqrt{a}$ using the iteration (often referred to as the Heron method)

$$x^{(0)} = a; \quad x^{(k+1)} = \frac{1}{2}\left(x^{(k)} + \frac{a}{x^{(k)}}\right) \text{ for } \quad k = 0, 1, 2, \ldots$$

Compute $\sqrt{2}$ by repeatedly evaluating the iteration rule (precision `format long e`). Also, test other square root calculations, e.g., $\sqrt{2024}$, $\sqrt{4711}$, $\sqrt{0.815}, \ldots$.

For all examples, examine how the number of correct digits increases from one step to the next. Which of the following rules best describes the convergence behavior?

1. The error is halved with each iteration.
2. Two correct decimal places are gained per iteration.
3. Three correct decimal places are gained per iteration.
4. The number of correct digits approximately doubles per iteration.

Task 6: Newton's Method:

Use Newton's method to find the root of $f(x) = x \tan x - 1$ near the initial value $x^{(0)} = 1$. Instructions:

```
>> x=1;
>> f = x*tan(x)-1
f =
    5.574077246549023e-001
>> fstr = ...
fstr =
    4.982926545469661e+000
>> x = x - f/fstr
x =
    8.881364757099055e-001
```

Note: The derivative of $\tan x$ is $1/\cos^2 x$.

Repeat until the result is accurate to the full number of decimal places. Check the number of correct decimal places in each iteration step based on the final result. Investigate the convergence behavior (increasing number of correct digits) by creating a table:

Step	Correct Decimal Places
0	0
1	
2	
⋮	

How can the convergence behavior be described?

Executing a command like `x = x - f/fstr` repeatedly in a program suggests the use of a loop. MATLAB, of course, supports such control structures. If you are familiar with them, write a `for`-loop in your script. Otherwise, simply repeat the instruction using copy/paste.

Here is how the code in your script file could look like if you use a loop:

```
format long e
x=1
for i=1:7
    f = x*tan(x)-1;
    fstr =  ...
    x = x-f/fstr
end
```

Task 7: Kepler's Equation

The lecture slides for the first lecture illustrate the Kepler equation as an example:

$$x - \epsilon \sin x = m$$

(it relates various parameters of an elliptical orbit - but you don't need to know that!) Assuming $\epsilon = 1/10$ and $m = 2$ are given; x is sought. Various approaches are possible:

- Reading from a suitable graphical representation.
- Through fixed-point iteration.
- Newton's method.
- Secant method.
- Interval halving.

Choose two of them and write a script. Your script should output a sequence of approximation values. Numerical values are provided on the lecture slides for comparison.

Analyze the convergence behavior of your method and indicate how the error reduces from one step to the next.

L 1.6 Drawing Curves of the Type $\mathrm{x} = \mathrm{f}(\theta); \mathrm{y} = \mathrm{g}(\theta)$

A function $f : x \mapsto f(x)$ assigns exactly one y-value to each x-value. However, a circle cannot be described in this way because there are two y-values for the same x-value.

In such cases, curves can be represented in parametric form. Both the x- and y-values are functions of a parameter (here called θ).

Let's draw a circle with a radius of 1. To ensure that the circle looks as expected, we generate 100 points lying on the circle. We use the following relationships:

$$x = \cos\theta, \qquad y = \sin\theta, \qquad 0 \le \theta \le 2\pi$$

The sequence of MATLAB commands to solve this task could look like this:

```
>> theta = linspace(0, 2*pi, 100);
```
Generates 100 equidistant values between 0 and 2π.

```
>> x=cos(theta);
>> y=sin(theta);
```
Generates the x- and y-coordinates of the 100 points.

```
>> plot(x,y);
```
Draw the lines connecting Point 1 to Point 2 to ...Point 100.

Additional commands to format your drawing are:

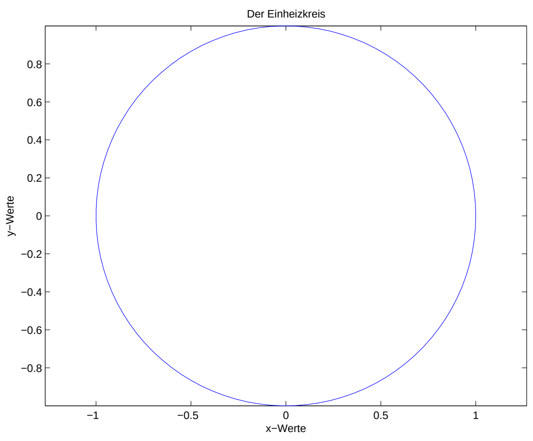
```
>> axis('equal');
```

Sets the scaling of both axes to be equal.

```
>> xlabel('x-Values')
>> ylabel('y-Values')
>> title('The␣Unit␣Circle')
```

Labels the x- and y-axis and titles your work. (Whether or not a semicolon is used as a terminator makes no difference for these commands.)

Your work should now look like this:



You can also interactively format the graphic as you are used to from other Windows applications. Above, next to the printer icon, you will find the “Edit plot” button. Alternatively, you can choose “Tools–Edit Plot” from the menu. After that, you can double-click on axes or other elements of the plot to further edit them. Other buttons allow you to insert text, arrows, or lines.

It is also important to know how to save the graphic. Under the “File–Export” menu, you can save the graphic in various formats.

L 1.7 Element-wise Arithmetic Operations

You may not have been aware: when evaluating function terms, you have applied arithmetic operations to data vectors. The command `x = cos(theta)` works for both scalar θ and a data vector with n elements. Addition and subtraction can also be applied equally to scalar and vector operands.

However, for multiplication, division, and the power function, you must use the element-wise operators `.*` `./` `.^` when linking data vectors. Reason: The "ordinary" operator symbols `*` `/` `^` link matrices and vectors in MATLAB according to the rules of matrix algebra. You *do not* want to apply these rules when inserting data vectors into function terms.

Initially, it can be difficult to understand when MATLAB interprets arithmetic operations element-wise on its own and when you need to explicitly use the dot.

> **Rule of thumb:** Always use element-wise operators in function terms when linking data vectors.

Tasks 8 and 10 require this distinction. Here are some additional hints:

```
>> t = linspace(0, 2*pi, 100);
>> x = sin(t)*2 - sin(t*2);
```

First function term in Task 8: Here, the "ordinary" `*` operator is sufficient because it only appears in operations of the type "Scalar times Vector."

```
>> x = sin(t).*2 - sin(t.*2);
```

However, it would not be wrong to use the dot notation. The result remains the same.

```
>> r = 2 - 2*sin(t) + sin(t) .* abs(cos(t)).^0.5 ./ (sin(t)+7/5)
```

Another function term in Task 8: But here, a dot operator must be used everywhere vectors are linked!

L 1.8 Additional Exercise Examples

Document your exercise examples as script M-files. Save drawings as `JPEG` files, then these files can be viewed with any image viewing program outside MATLAB. If you save drawings as `.fig` files – this is MATLAB's default – then you can only open these drawings within MATLAB.

Task 8: For Valentine's Day (even though it's already over, better late than never)

Draw the cardioid or heart curve. This is a curve given by the following parameterization for $0 \leq t < 2\pi$.

$$\begin{aligned} x &= 2\sin(t) - \sin(2t) \\ y &= 2\cos(t) - \cos(2t) \end{aligned}$$

The name cardioid comes from Giovanni di Castiglione, an Italian mathematician in the 18th century. For me, it looks more like an apple without a stem. A more beautiful heart curve equation is given here[15]:

$$\begin{aligned} r &= 2 - 2\sin t + \sin t \frac{\sqrt{|\cos t|}}{\sin(t) + 7/5} \\ x &= r\cos(t) \\ y &= r\sin(t) \end{aligned}$$

Note: For $0 \leq t < 2\pi$, use at least 500 subpoints and make sure to use `.*` and `./` at the right places!

Task 9: 3D Plots

Draw a spiral using `plot3(x, y, z)`:

$$x(t) = \sin(t), \quad y(t) = \cos(t), \quad z(t) = t, \quad 0 \leq t \leq 20.$$

Use the "Rotate 3D" button to create drawings of the curve from two different perspectives.

Task 10: For the Fish

Discuss the function

$$y = 3 - \frac{1}{2}\arctan\left(\frac{2x-5}{x-2}\right) + \frac{1}{10}\sin(7x) \quad 0 \leq x \leq 4$$

based on a graph. Where are the supremum and infimum located? In which intervals is the function continuous and differentiable? At which point does a shark fin protrude from the waves?

If the last question seems meaningless to you, you may have plotted the function incorrectly. Note that when calculating the y-values, all arithmetic operations must be performed elementwise (for each individual x-value). If one of the operands in an operation is a scalar and the other is a vector, it works automatically, as in the statement `2*x-5`. If both operands are vectors, you must use the elementwise operators `.*` and `./`. For example, the expression $(2x-5)/(x-2)$ must be written as `(2*x-5)./(x-2)`.

[15]Source: `http://mathworld.wolfram.com/HeartCurve.html`

L 2 Second Lab Unit

Contents of the second lab unit:

- Row and column vectors
- Matrices, mappings
- Functions
  - Programming functions as function M-files
  - Fixed points and roots of functions
  - Commands `fzero, roots`
  - Function handles (*anonymous functions*)
- Control structures
  - Loops
  - Branching
- Fixed-point iteration, one- and multi-dimensional

L 2.1 Row and Column Vectors

You have already worked with vectors in the previous unit: when plotting function graphs, you created ranges for x and y values as row vectors. MATLAB distinguishes between row and column vectors.

Here is a brief summary of creating and operating with vectors. Go through this guide, and you will learn the essential commands. Consider the result of each operation and think about the mathematical rules MATLAB used!

- Standard vector operations (addition `+`, subtraction `-`, scalar multiplication `*`, inner product `*` of row and column vectors, vector transposition `'`).
- Element-wise multiplication `.*` and division `./` for vectors of the same size. The result is a vector of the same size; corresponding elements are multiplied or divided.
- When one operand is a row vector and the other is a column vector, addition or subtraction creates a matrix following the pattern “each element with each.”
- Regarding multiplication, it depends: when the left operand is a row vector and the right operand is a column vector, MATLAB calculates the inner product. When the left operand is a column vector and the right operand is a row vector, it creates a matrix following the pattern “each element with each” (also known as outer, Kronecker, or tensor product - the scientific community does not entirely agree on the terminology).
- Division `/` of two row vectors or two column vectors results in completely bizarre outcomes – we won’t even attempt to explain what MATLAB calculates in such cases.

Creating Vectors

```
>> x=[1 2 3]
x =
     1     2     3
```

x is a row vector with three elements.
Vectors are enclosed in square brackets `[ ]`.

```
>> x=[1, 2, 3]
x =
     1     2     3
```

```
>> y=[2;1;5]
y =
     2
     1
     5
```

y is a column vector with three elements. The semicolons ; separate the rows in the vector.

```
>> y = [2
1
5]
y =
     2
     1
     5
```

Instead of semicolons, you can also start a new line in the *command window* or in files.

Arithmetic Operations

```
>> z=[2 1 0];
>> a=x+z

a =

     3     3     3
```

You can add or subtract two row vectors or two column vectors – this corresponds to the standard rules of vector arithmetic.

```
>> b=2*a

b =

     4     4     0
```

Scalar times vector, that's a standard rule.

```
>> x/2
ans =
    0.5000    1.0000    1.5000
```

Division of vector by scalar, also standard.

```
>> 2/x
Error using  /
Matrix dimensions must agree.
```

Not possible in this order.

```
>> 2./x
ans =
    2.0000    1.0000    0.6667
```

This works: Element-wise division.

```
>> b=x+y
b =
     3     4     5
     2     3     4
     6     7     8
```

Addition of row plus column vector results in a matrix – This is not exactly standard, but MATLAB's creative extension of the plus operator.

Attention for older MATLAB versions! Versions before R2016b do not allow adding row to column vectors. The above command yields an error message.

```
 >> b=x+y
Error using +
Matrix dimensions must agree.
```

The creative interpretation of the standard rules in newer versions applies to many arithmetic and logical operators. Rule: If it seems somehow meaningful, MATLAB extends the operands so that an element-wise operation becomes possible[16].

MATLAB introduced this automatic extension in 2016 with the reasoning: [17] *"MATLAB has a long history of inventing notation that became widely accepted"* – The comments in the blog are rather controversial. Not all users appreciate such a loose and original handling of the arithmetic rules.

> Caution when performing vector operations! If the formats of row and column vectors do not adhere to the rules of standard vector and matrix algebra, MATLAB often finds a creative but usually unintended interpretation of the instructions.

```
>> a=x.*z

a =

     2     2     0
```

You can multiply (or divide) two vectors of the same size element-wise; array operator `.*` (or `./`)

```
>> x*y
ans =
    19
```

Scalar (or dot) product. Standard rule for row times column vector.

```
>> y*x
ans =
     2     4     6
     1     2     3
     5    10    15
```

Column times row vector yields a matrix. Known as outer, tensor, or Kronecker product; this is also a standard operation.

```
>> x.*y
ans =
     2     4     6
     1     2     3
     5    10    15
```

Element-wise multiplication of row and column vectors yields the same matrix as above – MATLAB's creative interpretation.

Transposing Vectors

```
>> x
x =
     1     2     3
>> x'
ans =
     1
     2
     3
>> y
y =
     2
     1
     5
>> y'
ans =
     2     1     5
```

It's high time for you to get to know the `'` operator. This operator allows you to transpose vectors: row vectors become column vectors, and vice versa.

[16]For a more detailed explanation, refer to the MATLAB documentation on *Compatible Array Sizes for Basic Operations*

[17]`https://blogs.mathworks.com/loren/2016/10/24/matlab-arithmetic-expands-in-r2016b/`

By the way, in the Matlab desktop, you can see the “Workspace” tab on the left in the middle, and if you click on it, a window will appear. (The window may already be open without you clicking on it.) All variables used so far are listed there. The matrix symbol next to the variable name reminds you that MATLAB interprets all variables as matrices - scalars as 1×1 matrices, row vectors with n elements as $1 \times n$ matrices, and column vectors with m entries as $m \times 1$ matrices. Double-clicking on a row of this list opens a window, the “Array Editor,” which displays the values of the variables in tabular form. The values can also be changed (double-click on the corresponding cell in the spreadsheet).

Row vectors with regular entries

Here are some examples of creating row vectors:

```
>> x=linspace(0,10,5)

x =

         0    2.5000    5.0000
7.5000   10.0000
```

This command creates a row vector of length 5, whose elements are equidistant in the interval [0,10]. Without the third argument, a vector of length 100 is always created.

```
>> x=0:2.5:10
x =
         0    2.5000    5.0000
7.5000   10.0000
```

The same row vector can also be created using the colon operator following the pattern `StartValue:StepSize:EndValue` For the desired vector length, `linspace` is simpler; if the step size is specified, the colon variant is more convenient.

```
>> x=1:6
x =
     1      2      3      4      5
6
```

With a step size of 1, the colon instruction is particularly simple.

L 2.2 Matrices

The name MATLAB stands for *matrix laboratory.* This signals to you: Working with matrices is a central theme in this computing environment. Here’s a brief introduction to creating and performing basic arithmetic operations on matrices.

Hopefully, you still remember the rules of matrix arithmetic. Before you continue - are you aware of how matrix multiplication works? You should be able to calculate it on paper with a pen:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 5 & 6 & 7 \\ 8 & 9 & 0 \end{bmatrix} = \begin{bmatrix} 21 & 24 & \dots \\ 47 & \dots & ? \end{bmatrix}$$

Creating Matrices

```
>> A=[1 2; 3 4]
A =
     1     2
     3     4
>> B = [ 5, 6, 7
8, 9, 0]
B =
     5     6     7
     8     9     0
```

Input: Spaces or commas separate components in a row; semicolon or new line separates rows.

Arithmetic Operations, Accessing Elements

MATLAB offers the following operations similar to vectors:

`+ , -`	element-wise addition, subtraction
`*`	multiplies according to the rules of linear algebra
`.* , ./`	element-wise multiplication, division
`A'`	transpose matrix `A`

In arithmetic operations between matrices, it is important that the matrix dimensions match each other. Try for the matrices `A` and `B` that you just created:

```
>> C=A*B
C =
    21    24     7
    47    54    21
```

Matrix multiplication according to the rules of linear algebra.

```
>> C=B*A
Error using  *
Incorrect dimensions ...
```

Inner dimensions must match.

Matrix A is a 2×2 matrix, and B is a 2×3 matrix. In $A \cdot B$, the inner dimension is 2, as in $(2 \times 2) \cdot (2 \times 3)$. However, in $B \cdot A$ with $(2 \times 3) \cdot (2 \times 2)$, the inner dimensions do not match: $2 \neq 3$.

Element-wise operations can only be applied to matrices of the same size.

```
>> C=A+B
Arrays have incompatible sizes for this operation.
```

```
>> A(1,2)
ans =
     2
```

Accessing element a_{12}

```
>> B(1:2, 2:3)
ans =
     6     7
     9     0
```

Accessing elements in rows 1 to 2, columns 2 to 3

L 2.3 The Colon Operator

The colon (:) operator is one of the most useful operators in dealing with matrices and vectors. It creates vectors, index ranges, and iteration indices. As an index expression, it can extract individual rows or columns from matrices.

In the MATLAB help, you can find information under the keyword *colon*.

Syntax Examples

Creating Simple Number Sequences

`x=3:8` generates the row vector `[3, 4, 5, 6, 7, 8]`

`x=0:2:6` generates the row vector `[0, 2, 4, 6]`

`x=10:-1:0` generates the countdown vector `[10, 9, …, 3, 2, 1, 0]`

We will use these syntax examples for counting loops in Chapter L 2.6.

Index Expressions, Accessing Vector and Matrix Elements

As a matrix or vector index, the colon means “all rows” or “all columns.”

`A(:,4)` all rows, column 4; the entire fourth column of A

`A(3,:)` row 3, all columns; the entire third row of A

`x(3:7)` uses the vector `3:7 = [3, 4, 5, 6, 7]` as an access index; retrieves the sub-vector consisting of the x-components 3 to 7

`A(2:4,3:5)` the submatrix consisting of rows 2 to 4 and columns 3 to 5 of A

Be careful not to confuse `x(3:7)`, `x(3,7)`, and `x([3,7])`! The first expression is a subvector of `x` (as mentioned above); the second expression denotes the matrix element x_{37}; the third uses the vector `[3, 7]` as an access index, thus retrieving the 3rd and 7th components of the vector x. This results in a vector of length 2 with the components `[x(3), x(7)]`.

L 2.4 Matrices and Transformations

A matrix swiftly will take care
to move some points from here to there.

Linear mappings between vector spaces $\mathbb{R}^m \to \mathbb{R}^n$ correspond exactly to matrix-vector multiplications $\mathbf{y} = A \cdot \mathbf{x}$ with $n \times m$ matrices A.

Linear relationships between input and output variables are fundamental building blocks for any kind of data analysis and modeling. It can become fascinating and challenging for high-dimensional vector spaces. However, in 2- and 3-dimensional spaces, mappings can still be well visualized. Once you have worked through the examples here, generalizing to higher-dimensional spaces should not be a big step. Download the file `cat.dat` from the exercise homepage to your working directory.

```
>> X=readmatrix('cat.dat');
>> size(X)
ans =
   119      2
>> X=X';
>> size(X)
ans =
     2   119
```

You read a dataset from a text file as a matrix. X has 119 rows and 2 columns. For further processing, we transpose rows and columns. So now X has 2 rows and 119 columns.

For MATLAB versions before R2019a: The command `X=readmatrix('cat.dat')` does not exist yet. Download the script file datenX.m and run it. After that, the matrix `X` is loaded into the workspace, and you can continue with the commands `size(X)` and `X=X';`.

The column vectors in `X` correspond to 119 points in 2-dimensional space. If you plot these points, you can better visualize the data.

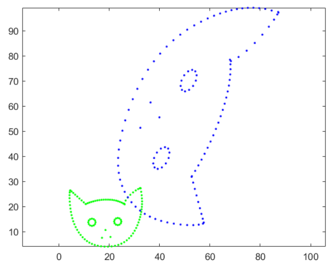
```
>> plot(X(1,:),X(2,:),'b.')
>> axis equal
```

Arnold's cat[18] is looking at you. Now, let's consider the matrix

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

How does this matrix act on points in $\mathbb{R}^2$? You can simply apply the 2×2 matrix A to the 119 column vectors in the matrix X by multiplication: $Y = A \cdot X$ is a 2×119 matrix, whose column vectors are the column vectors of X multiplied by A. Let's draw the input and output points for visualization:

```
A=[ 1 2; 3 0];
Y=A*X;
plot(X(1,:),X(2,:),'g.',Y(1,:),Y(2,:),'b.')
axis equal
```



You see: A twists, enlarges, and distorts the cat. It transforms input vectors into image vectors that (except in special cases) have different length and direction. (By now, you might get the meaning of the little rhyme at the beginning of this Section L 2.4!)

In this example, all vectors are more or less elongated but not arbitrarily large. The left ear tip, point $\approx [32; 27]$, is mapped to $[86; 96]$, which corresponds to an elongation by a factor of ≈ 3.1.

```
>> x=[32; 27];
>> y=A*x
y =
    86
    96
>> norm(y)/norm(x)
ans =
    3.0784
```

This way, you can calculate elongation factors (in the 2-norm). Try other values for x. For which vector do you find the largest elongation factor? Conversely, can you find a vector with a particularly small elongation factor?

[18]The mathematician Vladimir Igorewitsch Arnold (1937–2010) used this cat illustration in textbooks on classical mechanics to illustrate the properties of transformations.

The norm of a matrix provides the maximum stretch factor

(This applies to 1-, 2-, or ∞-norm; depending on the norm used, the values may be slightly different. For the matrix A in this example, $\|A\|_2 = 3.5266$. Because this cannot be easily calculated from the matrix elements, one also works with the 1-norm (maximum column sum: 4, the vector [1; 0] elongates by this value in the 1-norm) or the ∞-norm (maximum row sum: 3, the vector [1;1] elongates by this value in the ∞-norm)).

```
>> [norm(A) norm(A,1) norm(A,'inf')]
ans =
    3.2566    4.0000    3.0000
```

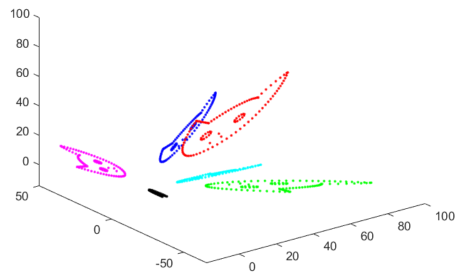
These are three matrix norms of A

However, matrices can also represent linear mappings between vector spaces of different dimensions. Now, consider the matrices

$$B = \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad C = \frac{1}{15}\begin{bmatrix} -2 & 2 & 14 \\ -2 & -13 & -16 \\ 14 & 16 & 7 \end{bmatrix}.$$

Apply B to the data matrix X first and then apply C several times, and see Arnold's cat wander through space...

```
Z=B*X;
plot3(Z(1,:),Z(2,:),Z(3,:),'b.')
hold on
Z=C*Z;
plot3(Z(1,:),Z(2,:),Z(3,:),'r.')
Z=C*Z;
...
plot3(Z(1,:),Z(2,:),Z(3,:),'k.')
hold off
```



Task 11:

Given are the 2×119 data matrix X as above and matrices

$$D = \begin{bmatrix} 1 & 1 \\ -\frac{1}{4} & \frac{3}{4} \end{bmatrix}, \quad E = \begin{bmatrix} 1 & 2 \\ -\frac{1}{4} & \frac{3}{4} \end{bmatrix}, \quad F = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ -1 & \frac{1}{2} \end{bmatrix}, \quad G = \begin{bmatrix} \frac{1}{2} & 0 \\ -\frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

Apply the corresponding mappings to the data matrix several times and plot the image points. For which matrices

- does the fixed-point iteration converge?
- is there a contracting mapping?
  A contracting mapping automatically guarantees convergence of the fixed-point iteration. However, for some non-contracting mappings, fixed-point iteration can still converge.
- is there a symplectic mapping?
  These are mappings that distort areas but leave the area unchanged. You can only estimate this from the graphical representations. But perhaps you will also find out: in addition to the norm, which measures the elongation factor, there is another important matrix quantity that determines the area (or more generally: volume) enlargement factor.

Symplectic mappings are an important topic in theoretical mechanics and for Austrian female cyclists[19].

L 2.5 Functions

In MATLAB, you can define your own functions and save them in so-called function M-files. For simple function terms, it is also possible to use function inliners, so-called *anonymous functions*.

This section explains the following points:

- A simple function file
- Zeros with `fzero` and `roots`
- Function inliners (*anonymous functions*)

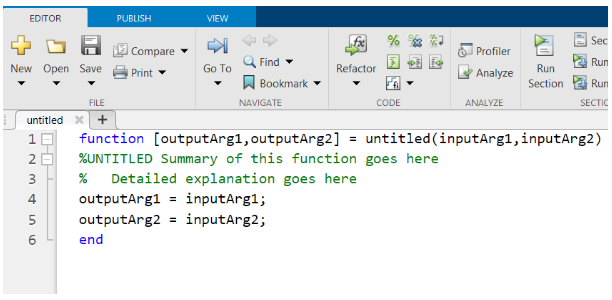
L 2.5.1 A simple function file

Let's consider the function

$$f : x \mapsto y = 3\cos x - \log x \ .$$

You have already encountered it in the lecture and in the script.

In the MATLAB window, select "New–Function" in the upper left corner. This opens a window of the MATLAB Editor, in which a sample function is already provided.



```
function [outputArg1,outputArg2] = untitled(inputArg1,inputArg2)
%UNTITLED Summary of this function goes here
%   Detailed explanation goes here
outputArg1 = inputArg1;
outputArg2 = inputArg2;
end
```

Complete and create a source code. Orient yourself on the following pattern:

```
function [ y ] = myf(x)
%MYF My first try to code a function
%  This is the function from the Lecture Notes
page U22
%  Jane Doe, Summer Term 2024
y=3*cos(x)-log(x);
end
```

The first line of a function file defines the input and output parameters as well as the name of a function. In this case, the function is named **myf**, has a variable x as an argument (input), and returns a function value y (output). The arguments x and y can be scalars or arrays (vectors, matrices). You can use your own function names, but please use only permissible names (no umlauts and special characters, no digits at the beginning).

Save your function. Make sure that the file is saved under the name `myf.m`. It is advisable to create a separate subdirectory for the MATLAB files you create in these exercises.

Depending on the directory in which you saved the file, you may have to do a little extra work:

[19]Olympic gold medalist Anna Kiesenhofer, Integrable systems on b-symplectic manifolds. (Thesis, 2016)

Script and function files can only be called in the *command window* if they are in the current folder.
In the MATLAB interface, in the left window "Current Folder", navigate to the directory where you saved your function.

Tipp: It's faster if, in the Editor window, after saving the function, you click on the green "Run" arrow icon. If a dialog box appears with a text like *File D:/work/myf.m is not found in the current folder blabla blabla*, simply click on "Change Folder" and ignore further messages (*mein.f requires more input...*)—the directory has changed, that's all we wanted.

The *add to path* option is not recommended. With name collisions of files, you might lose track of which file is actually being executed.

Switch back to the MATLAB *Command Window*. First, set the display of more decimal places and eliminate additional blank lines:

```
>> format long g
>> format compact
>>
```

Other options:
short g gives fewer digits.
short or long e gives exponential notation.
format loose inserts blank lines.

Now, test if the function can be called:

```
>> y=myf(1)
y =
   1.62090691760442
```

You have evaluated the function at the point 1.

If you have entered comments diligently, now there is a reward: Enter the following:

```
>> help myf
myf My first try to code a function
  This is the function from the Lecture Notes page U22
  Jane Doe, Summer Term 2024
```

And something else is great about this function: it not only provides a result for a scalar value but automatically for an entire list of values. Remember: The statement *start value*:*step size*:*end value* is (alternatively to `linspace`) a second way to generate row vectors.

```
>> x=0:1:10

x =

     0      1      2      3      4
5      6      7      8      9     10
```

Values from 0 to ten. Also, x=0:10 would have worked (If step size is omitted, MATLAB automatically takes 1)

The function `myf` can be applied element-wise to the row vector x.

```
>> myf(x)
ans =   Inf    1.62090691760442
-1.94158769020137 ...
```

One small flaw: $f(x) = 3\cos x - \log x$ is not defined for $x = 0$. The function value $f(0)$ is stored as `Inf`.

L 2.5.2 Roots calculated by Newton's Method

For this, you also need to code the derivative of the function f as a function file. For this time only, these notes will help you with differentiation:

$$f'(x) = -3\sin x - \frac{1}{x}$$

```
function y = mydf(x)
%MYDF Derivative of the function myf
% it's so boring to write comments,
% but it pays off
% when you try to understand
% what this is supposed to be
% a week later
y=-3*sin(x)-1./x;
end
```

Open a new M-file, program the derivative, and save it under the name `mydf.m`.

```
>> mydf(1)
ans =
  -3.52441295442369
>> x=1:3
x =
     1     2     3
>> mydf(x)
ans =
  -3.52441295442369
-3.22789228047704
-0.75669335751293
```

Check if the derivative can be evaluated correctly.

A small but important detail: If you program division as `1/x`, the evaluation works smoothly for *scalar* x, but not if x is a vector! At first, it is often difficult to understand when MATLAB automatically interprets multiplication and division element-wise, and when you need to explicitly use the dot notation.

> **Rule:** (Was already present in the previous unit[20])
> In function files, always use element-wise operators `.* ./ .^`

Then, according to the rule, we should have used `.*` for the first multiplication above:

```
y=-3.*sin(x)-1./x;
```

In this case, it doesn't matter whether you write `.*` or `*`: if one of the operands is a scalar, element-wise calculation is automatically performed.

Back to Newton's method. If necessary, inform yourself in the Lecture Notes, Chapter 1.10, about the computational procedure.

```
>> x=1
x =
     1
>> x=x-myf(x)/mydf(x)
x =
   1.45990834177644
```

Start with an initial value of 1 and perform one step of the Newton method.

Repeat the Newton step until the value of the root no longer changes!

Compare your values with the following table and observe the convergence behavior:

[20] *repetitio est mater studiorum*, meaning "repetition is the mother of boredom" or something like that...

```
1                    correct digits: 1
1.45990834177644                     1.4
1.44725583798192                     1.44725
1.44725861727779                     1.447258617277
1.44725861727790                     all
```

In each step, the number of correct digits is at least twice as large as in the previous step. This extraordinarily rapid convergence (quadratic convergence) is characteristic of the Newton method.

L 2.5.3 Roots with the Secant Method

```
>> xalt = 2
xalt =
     2
>> x = 1
x =
     1
```

If you don't know what this is about, consult the script, Chapter 1.9, or lecture materials.

```
>> xneu = x - myf(x)*(x-xalt)/(myf(x)-myf(xalt))
xneu =
   1.45499210414322
```

```
>> xalt = x; x = xneu;
```

With these commands, you reset the two initial values of the secant method to the two last calculated values. Now you can reevaluate the formula (but don't type it again, use the arrow key!)

```
>> xneu = x - myf(x)*(x-xalt)/(myf(x)-myf(xalt))
xneu =
   1.44716725175157
```

Repeat the evaluation of the secant method until the value of the root no longer changes!

Compare your values with the following table and observe the convergence behavior:

```
1                    correct digits: 1
1.45499210414322                     1.4
1.44716725175157                     1.447
1.44725862822699                     1.4472586
1.44725861727792                     1.4472586172779
1.44725861727790                     all
```

The number of correct digits, according to theory, is the sum of the correct digits of the two previous approximations. This would typically mean 0, 1, 1, 2, 3, 5, 8, 13, ... exact digits (does this sequence sound familiar to you?) Accordingly, the number of correct digits should increase by about 60% per step.

In fact, the number doubles in the first steps (1-2-4-8), and finally still increases by 75% (from 8 to 14 digits). The secant method converges faster here than it should according to the rules of theory.[21]

L 2.5.4 Roots with `fzero`

MATLAB has built-in *numerical* methods for finding roots, a combination of bisection, secant, and inverse quadratic interpolation. The function `fzero` calls these methods.

For our example $f(x) = 3\cos(x) - \log(x)$, the call is:

[21] *the Code is more what you'd call "guidelines" than actual rules. Welcome aboard the Black Pearl, Miss Turner* (Pirates of the Carribean, 2003)

```
>> fzero(@myf,1)
ans =
   1.44725861727790
>>
```

`fzero` stands for "find zero."

Important: You need to prefix the function name (here: `myf`) with the "function handle" `@` (MATLAB calls `@` a function handle). Function names with the handle `@` in front are a separate data type; they allow functions to be passed to other functions. Imagine `@` is the handle you use to pass a cup of coffee on.

However, `fzero` has its pitfalls: it finds a point as close as possible to a sign change of the function. For continuous functions (Intermediate Value Theorem!), this is also a value near a root. For discontinuous functions, `fzero` may return values corresponding to singularities of the function. For example, the tangent at $\pi/2$

```
>> fzero(@tan,1)
ans =
   1.57079632679490
>>
```

This is not a root of the tangent function.

`fzero` also cannot find multiple roots (of even order), where the function touches but does not intersect the x-axis.

Function handles can do even more. Without writing a script, you can define a function as a one-liner. The next section explains this.

L 2.5.5 Anonymous Functions, *Anonymous Functions*

For a one-line function term, it is not necessary to write separate function files. Our example function f with the function term $f(x) = 3\cos(x) - \log(x)$ and its derivative $f'(x) = -3\sin x - \frac{1}{x}$ can also be defined as so-called *anonymous functions*. These are functions that are not stored in a program file but are associated with a variable of type *function handle* (recognizable by the "function handle" `@` with a function argument in parentheses). For our examples:

```
>> f = @(x) 3*cos(x) -log(x)
f =
  function_handle with value:
    @(x)3*cos(x)-log(x)

>> df = @(x) -3*sin(x)-1./x
df =
  function_handle with value:
    @(x)-3*sin(x)-1./x
```

You can evaluate these one-liner functions:

```
>> f(1)
ans =
    1.6209
```

or directly apply `fzero`. Be sure to try it out! Note that because `f` is already of type "function handle," no `@` handle is allowed in the argument of `fzero`.

```
>> fzero(f,1)
ans =
    1.4473
>> fzero(f,10)
ans =
   11.9702
```

The steps of a Newton method in the *command window* could look like this:

```
>> x=1;
>> x=x-f(x)/df(x)
x =
   1.459908341776445
>> x=x-f(x)/df(x)
x =
   1.447255837981919
>> x=x-f(x)/df(x)
x =
   1.447258617277790
>> x=x-f(x)/df(x)
x =
   1.447258617277903
```

One-liner functions (*anonymous functions*) are useful when the function term consists of a single executable command.
The designation of a one-liner function is a variable of type “function handle”; it does *not* refer to a function file.

L 2.5.6 Roots with `roots`

Solutions (also called roots) of *polynomial* equations are easier to find for MATLAB. Let’s consider an example:

$$p(x) = x^3 - 2x - 5$$

A polynomial is determined by specifying its coefficients. In this example, they are

$$1; 0; -2; -5$$

because $p(x) = 1 \cdot x^3 + 0 \cdot x^2 - 2 \cdot x - 5 \cdot x^0$. You pass the list of coefficients as a vector to `roots` like `[1 0 -2 -5]`, and `roots` provides *all* (including complex) roots of the polynomial.

```
>> roots([1 0 -2 -5])
ans =
  2.09455148154233
 -1.04727574077116 + 1.13593988908893i
 -1.04727574077116 - 1.13593988908893i
>>
```

Example

Task 12:

What is the molar volume of nitrogen at 20 °C and 1 bar according to the Van der Waals equation?

The equation of state

$$\left(p + \frac{a}{V_{mol}^2}\right)(V_{mol} - b) = RT$$

describes the relationship between pressure p, molar volume V_{mol}, and temperature T. For nitrogen, the constants a and b have the values

$$a = 0.129\,\mathrm{Pa\,m^6/mol^2}, \quad b = 38.6 \times 10^{-6}\,\mathrm{m^3/mol}.$$

The molar gas constant is $R = 8.3145\,\mathrm{J/molK}$. After substituting the numerical values, the equation for V_{mol} is:

$$\left(100000 + \frac{0.129}{V_{mol}^2}\right)(V_{mol} - 0.0000386) = 2437.4$$

Solve this problem using the secant method, the Newton method, and `fzero`. The equation can also be transformed into polynomial form. Use `roots` for the solution. The script describes a transformation into a fixed-point equation in Chapter 1.6. Solve the problem also with fixed-point iteration!

Task 13:

Consider the polynomial

$$p(x) = -64 + 176x - 188x^2 + 101x^3 - 29x^4 + \frac{17x^5}{4} - \frac{x^6}{4}$$

Write a function M-file for this polynomial. Remember to use element-wise operators `.^`. Don't forget the semicolon at the end of the line (otherwise, the M-file will produce a lot of unnecessary output in the *Command Window*).

Plot the polynomial for $0 < x < 5$. Find roots using the Newton method, with `fzero` (you need to set suitable initial values), and with `roots` .

Why doesn't `fzero` find all the roots?

Check, as in previous exercises, the convergence rate of the Newton method for all three roots. Does the convergence rate behave according to the rules?

Change the coefficient of x^3 in the polynomial by 1%, 0.1%, 0.01%. Determine roots with `roots` and, for each original real root, provide the following information: Does this real root still exist? If yes, how much has its value changed relative to the original? Or has a multiple root split into several different real roots?

This task illustrates: Numerical computation of multiple roots is sensitive to small changes in the polynomial and rounding errors. Roots can vanish, shift, or multiply. It is referred to as a *ill-conditioned problem.*

Ill-conditioned problem: Small changes in the data and/or rounding errors produce drastic changes in the result.

L 2.6 Control Structures

In the previous lab unit, you performed fixed-point iterations or steps of Newton's method or the secant method by entering commands manually in the *Command Window*. You monitored the output and stopped the process when the results no longer changed. You can also formulate this process as a program. MATLAB provides standard control structures (similar to Java or C++). For starters, `for` loops and `if` branches, as presented in this section, should suffice. The MATLAB help will provide more information, including `while` loops.

Loops

A `for` loop typically has the form:

```
for index = startvalue:endvalue
    statement
    statement
    ...
end
```

The index then iterates through the values `startvalue, startvalue+1, startvalue+2, ... endvalue`; it does not need to be incremented as in Java or C++. A more general form of the loop header is:

```
for index = startvalue:step:endvalue
```

For example, a loop counter s that traverses the values $1; 0.9; 0.8; \dots 0$ with a step size of -0.1:

```
for s = 1:-0.1:0
```

Branching

A conditional branch with `if` has the form:

```
if expression
    statement
    statement
    ...
end
```

MATLAB evaluates `expression`, and if the result is logically `true` or (difference from Java!) not equal to 0, it executes the following statements until `end`. Nested `if` statements are possible, and each level must be closed with the corresponding `end`.

While Java strictly requires that only a logical expression be placed in an `if` statement, MATLAB even allows vectors or matrices. If `expression` is not a scalar, each component must be `true` or $\neq 0$.

The more general form with `elseif` and/or `else` is as follows:

```
if expression1
    statements1
elseif expression2
    statements2
else
    statements3
end
```

Sample Program

The following sample program for fixed-point iteration introduces loops and branches. It performs fixed-point iteration according to the rule

$$x^{(k+1)} = \phi(x^{(k)})$$

for the initial value $x^{(0)}$. You can download the program from the exercise homepage:

```
function x = fixpunkt(phi, x0)
% FIXPUNKT: Demo program for fixed-point iteration
% The method iterates according to the rule
%       x_i+1 = phi(x_i)
% until changes fall below a tolerance threshold
% or the maximum number of iterations is exceeded.
%
% Input:    phi ... Function whose fixed point is sought
%                   (with function handle @)
%           x0  ... Initial value or vector
% Output:   x   ... If convergence: fixed point,
%                   otherwise: NaN
% Example:  fixpunkt(@cos, 1)
%
% ----------------------------------------NMI,SS09-18 C.Brand
itmax = 100;                % maximum number of iterations
errlim = 1.e-9;             % error threshold
for i = 1:itmax
    x = phi(x0);            % function evaluation
    if norm(x - x0) < errlim % termination criterion: 2-norm of the error
        return
    end
    x0 = x;
end
x = NaN;
end
```

L 2.7 Fixed Point Iteration One- and Multi-dimensional

One-dimensional

In Task 5, you calculated the square root of a as the fixed point of the function

$$y = \frac{1}{2}\left(x + \frac{a}{x}\right)$$

Using loops and the fixed-point template program, we will repeat the process and generalize it to the multi-dimensional case.

```
>> heron = @(x)(x+13/x)/2;
```

Define the iteration function as an *anonymous function*. Here, $\sqrt{13}$ is to be calculated. Alternatively, you can also write a function file for it.

```
>> heron(2)
ans =
    4.2500
```

Make sure to test if the function can be called and produces the expected result.

```
>> x=1;
>> x=heron(x)
x =
     7
```

This is how a fixed-point iteration looks like in “manual mode”: repeated calling of the function in the command line.

```
>> x=heron(x)
x =
    4.4286
>> x=heron(x)
x =
    3.6820
>> x=heron(x)
x =
    3.6063
```

```
>> fixedpoint(heron,1)
ans =
    3.6056
```

And this is how the fixed-point template program calculates the same result.

Multi-dimensional Fixed-Point Iteration, Vector-valued Functions

This program can also perform multi-dimensional fixed-point iteration. In the lecture, the fixed point of a vector-valued function $\mathbf{\Phi} : \mathbb{R}^2 \to \mathbb{R}^2$ was discussed as an example:

$$\begin{aligned} x_1 &= \frac{1}{4}(x_2 - x_1 x_2 + 1) \\ x_2 &= \frac{1}{6}(x_1 - \log(x_1 x_2) + 2) \end{aligned}$$

The function $\mathbf{\Phi}(\mathbf{x})$ can be implemented as a function file as follows:

```
function y = phi(x)
x1=x(1);
x2=x(2);
y = [ (x2 - x1.*x2 + 1)/4
      (x1-log(x1.*x2)+2)/6 ];
end
```

(Note: A one-line *anonymous function* is also possible, but for a slightly more extensive function term, the function file provides a clearer implementation!)

Calling the fixed-point function for the function file (remember the `@` before the filename!):

```
>> fixedpoint(@phi,[1;1])

ans =

    0.3534
    0.6400
```

L 2.8 Additional Tasks for the Next Exercise Session

Task 14:

Formulate the iteration rule of the Newton method for solving

$$\sin x = x/2 \ .$$

Program the corresponding iteration function and call the fixed-point template program. Use it to find all positive solutions of the equation.

Task 15:

Find the roots of the function

$$f(x) = e^x - 3x^2$$

using the Newton method and simple fixed-point iteration (several transformations are possible).

Compare (with the same initial values) the number of iterations. Estimate the reduction factor C in methods with linear convergence. Does your fixed-point formulation find all roots?

Task 16:

The lecture notes discuss the equation

$$r = K \frac{q-1}{1-q^{-n}} \quad \text{for } r = 900, K = 100,000, n = 180.$$

Write a MATLAB program that, given input values r, K, and n, finds the solution q. (Assumption: $n \cdot r > K$ and hence $q > 1$) You can choose the method you prefer.

Hints:

- The discount factor q realistically lies in the range $1 < q < 1.05$, corresponding to a monthly interest rate between 0% and 5%. For $q = 1$, the equation is not defined (denominator becomes 0), so a starting value of $q = 1$ does not make sense.
- The fixed-point form (solving for q in the numerator) converges slowly but is quickly implemented and relatively insensitive to the choice of the starting value (as long as $q > 1$).
- Newton's method and MATLAB's `fzero` require good initial values.

Task 17:

Formulate, analogous to the fixed-point template program, a function M-file for the bisection method. The call

```
Bisection(@yourFunction, x0, x1)
```

should find a root of the function, starting from the initial values $x^{(0)}$ and $x^{(1)}$. Test the method by searching for roots of the following function:

$$f(x) = x^2 - 3\tan x + 1$$

Initial intervals with sign changes are: $[0, 1]$, $[1, 2]$, $[4.5, 4.7]$. Does bisection find a root for all three intervals? What does `fzero` find when using the above interval boundaries as starting values?

Task 18:

The lecture slides for the first lecture describe the Illinois variant of the false position method. This method generally exhibits the good convergence properties of the secant method while simultaneously ensuring the safe inclusion of the root in the current interval. If you implement this method, you have a very useful general-purpose root-finding program.

Follow the structure of the fixed-point template program and write a function M-file for the Illinois variant of the false position method. The call

```
Illinois(@yourFunction, x0, x1)
```

should find a root of the function, starting from the initial values $x^{(0)}$ and $x^{(1)}$. Test the method by searching for all positive solutions of the following equation:

$$\sin x = x/2$$

For comparison, also calculate the solution using `fzero`.

Task 19:

Consider the system of equations

$$\begin{array}{rcrcrcr} 8x_1 & + & x_2 & - & x_3 & = & 8\ , \\ 2x_1 & + & x_2 & + & 9x_3 & = & 12\ , \\ x_1 & - & 7x_2 & + & 2x_3 & = & -4\ . \end{array}$$

Formulate a fixed-point iteration. Consider from which equation to express x_1, x_2, and x_3. Hint: Express variables from the equation where they have the strongest influence (the largest coefficient). Test the method.

Task 20:

(If you have gone through the materials, this task is already solved)

Consider a system of two equations in two unknowns:

$$\begin{aligned} f(x,y) = 4x - y + xy - 1 &= 0 \\ g(x,y) = -x + 6y + \log xy - 2 &= 0 \end{aligned}$$

Formulate a fixed-point iteration for it and test!

L 3 Third Lab Unit

Content of the third lab unit:

- Script and function files, Live Scripts
- Arithmetic operations when inputting matrices and vectors
- Backslash solves systems of equations
- Inverse problem, regularization
- Newton's method for systems
- Local functions
- `fsolve`
- Contour and surface plots

L 3.1 Script and Function Files: Summary

We have already worked with both: See sections L 1.3 and L 2.5 . Important differences exist in the usage and scope of variables. Here is a brief review and summary.

- Scripts. They have neither input arguments nor return values. They access data and variables from the current workspace. Command lines in a script file act as if they were entered directly in the *Command Window*.

  A common mistake is using variables from the current workspace in the script but not defining them themselves. You won't notice anything during the current MATLAB session. Only at the next restart will MATLAB report: `Undefined function or variable`. Therefore, at the very beginning of the script, type the command `clear variables`. This will clear all variables in the workspace and immediately alert you if you have forgotten definitions in the script. See section L 1.3.
- Function files. They take input values (arguments) and return results. Variables declared within the function are local, meaning they exist only within the function.
- Local functions: In a function file, additional functions ('sub-functions') can appear. The first function (the 'main function') can be started from the command line in the *Command Window* or via command lines in a script. All other functions (order doesn't matter) are only local—meaning available only to other functions from the same file. It can be quite useful to structure a program with sub-functions. Further information: MATLAB help, keyword 'local functions'
- Script files can also contain local functions (from MATLAB R2016b onwards); however, they must begin after the last line of script code. Search for help under 'Add Functions to Scripts'
- Nested functions (for advanced users): these are functions declared within a parent function. This allows you to control access to variables even more precisely. Details can be found in the MATLAB help, keyword 'nested functions'.
- Function one-liners (anonymous functions) are practical when the function term consists of a single executable command. Introduced in section L 2.5.5.
- Live Scripts and Functions (For advanced users): This type of program files can represent source code and output together, format accompanying text and comments neatly, you can incorporate switches and control panels for interactive work, and more. Such files have the extension `.mlx` as opposed to the extension `.m` for "ordinary" scripts.

If you want to make your files particularly appealing, you can try clicking on *Save/Save as/Save as type: Matlab Live Code files (*.mlx)* in the editor and convert your file into a Live Script. You can find more information and examples in the MATLAB help, keyword *Live Scripts and Functions*. A nice elaboration[22] of Exercise 8 can be found here (click!)

L 3.2 Arithmetic Expressions When Inputting Vectors and Matrices

A reminder from the last session: When entering vectors and matrices, comma, semicolon, space, and line break act as delimiters.

You can specify not only numerical values but also arithmetic expressions:

```
>> x=[1+2, 2+3 3+4]
x =
     3     5     7
```

Row vectors: Space or comma separates components. Here (not recommended), once using comma, once using space.

```
>> y=[1-2 3 + 4 5 +6 7+ 8]
y =
    -1     7     5     6    15
```

Be careful with spaces around + and −: they can be interpreted as signs, arithmetic operations, or delimiters.

Typical Mistake with Arithmetic Expressions in a Vector: Space before operator unintentionally acts as a delimiter. Recommendation: When using arithmetic operations, either no spaces or spaces around operators on both sides.

Rules for plus or minus in terms:

- No space on both sides: arithmetic operation is performed (`+` and `-` act as *binary operators*)
- Space on both sides: also interpreted as a binary operation.
- Space on the left, none on the right: Space on the left acts as a delimiter between matrix elements, operators `+` and `-` act as *unary operators* (as *signs* of the following term).

  Often happens unintentionally, resulting in error messages or incorrect results.
- Space on the right, none on the left: binary operation (but if you intentionally do this, it's pretty behaviorally original).

L 3.3 Systems of Equations: MATLAB's Slanted Writing

MATLAB uses – quite normally – the slash as a division operator: `x = 3/4` unsurprisingly yields `x = 0.75000`.

You may not be familiar with the reverse slash, the backslash operator \ :

```
>> x=3\4
x =
    1.3333
```

[22] by Chr. Tuschl, thank you!

Also, `\` divides, but in the form `x = denominator\numerator`. It actually makes sense if you imagine `\` as a downward-slanting fraction bar (or tilt your screen to the left for a moment), with the left term *under* and the right term *over* the fraction bar.

Quite creative is MATLAB's interpretation of the backslash in systems of equations. The most important in short:

`x = A\b` solves the equation system $A\mathbf{x} = \mathbf{b}$ (or "solves")

Not every result calculated by MATLAB in this way is the solution of an equation system in the true sense – hence the quotation marks around "solves".

No one wants to remember in detail what MATLAB does in the various special cases. Just so you can see and be warned about what can happen, here's a list.

The command `x = A\b` provides for an equation system $A\mathbf{x} = \mathbf{b}$

- for non-singular $n \times n$ matrices, the unique solution;
- for singular $n \times n$ matrices, a warning message and, if there are solutions, a solution with as many zero components as possible. If there is no solution, MATLAB usually provides nonsensical numerical values.
- for an $n \times m$ matrix with $n < m$ (*underdetermined* system of equations), if there are solutions, a solution with as many zero components as possible.
- for an $n \times m$ matrix with $n > m$ (*overdetermined* system of equations), the "least wrong solution". This is the vector $\mathbf{x}$ for which $A\mathbf{x} - \mathbf{b}$ has the smallest 2-norm. This is called the *least squares solution*.
- Special cases are $n \times m$ matrices with $n \neq m$ and rank $< \min(m, n)$. Matlab warns: "Warning: Rank deficient" and provides a least squares solution.

Systems of equations with multiple right sides

For the same matrix A and multiple right sides $\mathbf{b}, \mathbf{c}, \mathbf{d}, \ldots$ the equation systems $A\mathbf{x} = \mathbf{b}, A\mathbf{y} = \mathbf{c}, A\mathbf{z} = \mathbf{d}, \ldots$ can be solved together. Combine all right sides as column vectors in a matrix B: `B = [b,c,d]`. MATLAB's `\` provides a matrix X, whose columns contain the solution vectors: $X = [\mathbf{x}, \mathbf{y}, \mathbf{z}]$.

`X = A\B` solves (or "solves") the equation systems $A \cdot X = B$

Slashes between matrices, general case

The summary:

Use the "upward slash" `/` between scalars as division operator and the "backslash `\` between matrices and vectors for solving linear equation systems.

If you want to know more about MATLAB's handling of `/` and `\`, read on...

You could also write $x = 3/4$ in the form $x = 3 \cdot 4^{-1}$, or $x = 4^{-1} \cdot 3$. Nobody does that for scalar terms. However, for matrices, the notation with inverses is standard, for example $A \cdot B^{-1}$ or $A^{-1} \cdot B$. MATLAB allows for the fraction notation:

$$A \cdot B^{-1} = A/B \quad \text{und } A^{-1} \cdot B = A\backslash B$$

Consider `/B` as B^{-1} because B lies "under" the fraction bar, and correspondingly `A\` as A^{-1}. The slashes intentionally appear *left* of B and *right* of A – matrix multiplication is not commutative! Depending on which side you want to multiply with an inverse, you use `/` or `\`.

MATLAB does not actually compute the inverse matrices and multiply with them. Explicitly calculating an inverse is computationally expensive and prone to rounding errors. In reality, MATLAB solves linear equations using the slash notation. This is algebraically equivalent to multiplication with the inverse only for nonsingular square matrices.

Equation systems instead of inverse matrices

When multiplication with the inverse matrix appears in mathematical expressions, you don't actually need the inverse in explicit form for computational evaluation[23].

For numerical computation, there is a huge difference in terms of computational effort and accuracy whether you calculate an inverse and multiply with it, or reformulate to solve equation systems.

> Avoid explicit multiplication with an inverse matrix! It causes unnecessary computational effort and undesirable rounding errors.
> `X = A\B` calculates $A^{-1} \cdot B$ as the solution of the equation system $A \cdot X = B$
> `X = A/B` calculates $A \cdot B^{-1}$ as the solution of the equation system $X \cdot B = A$

To explain why multiplication with the inverse can be reduced to solving an equation system, here are the corresponding transformations. Pay attention, you must multiply "on the correct side" so that the inverses vanish: in the first case, multiply A from the left, and the other time with B from the right. (Matrix multiplication is not commutative!)

$$\begin{aligned} X &= A^{-1} \cdot B \qquad &| \; A\cdot \\ A \cdot X &= A \cdot A^{-1} \cdot B \\ A \cdot X &= B \end{aligned} \qquad\qquad \begin{aligned} X &= A \cdot B^{-1} \qquad &| \; \cdot B \\ X \cdot B &= A \cdot B^{-1} \cdot B \\ X \cdot B &= A \end{aligned}$$

L 3.4 Inverse Problems, Regularization

A matrix' action on a vector's state
the inverse will annihilate.

Section L 2.4 explained: The matrix-vector multiplication $\mathbf{y} = A \cdot \mathbf{x}$ defines a transformation *input data* $\rightarrow$ *image data*. Often, the problem occurs in the reverse form: Given the result vector $\mathbf{y}$, which vector $\mathbf{x}$ leads to this result?

If the inverse matrix exists, the theoretical answer is simple:

$$\mathbf{y} = A \cdot \mathbf{x} \qquad \Leftrightarrow \qquad \mathbf{x} = A^{-1} \cdot \mathbf{y} \quad ,$$

However, it may be that several $\mathbf{x}$ vectors have the same result vector $\mathbf{y}$, or there is no $\mathbf{x}$ vector that leads exactly to the result vector $\mathbf{y}$. Then it becomes difficult. But even if an inverse theoretically exists, it may be practically impossible to perform the reverse transformation.

> **Inverse Problem:** Outcome known, cause sought. Often difficult.

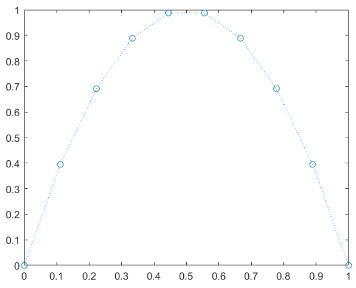
Here we present such a situation and show how to deal with it.

Task 21: Ill-conditioned inverse problem

[23] To prevent you from even attempting to calculate an inverse, these exercise materials (for now) intentionally omit the MATLAB command for calculating inverses.

Generate a time series $\mathbf{x}(t)$ with n data points; here with a parabolic shape; plot the data.

```
n=10;
t = linspace(0,1,n)';  % t column vector!
x = 4*t.*(1-t);
plot(t,x,'o:')
```

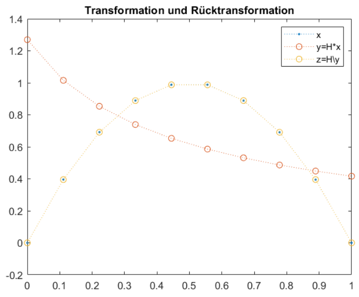


Apply a transformation to $\mathbf{x}$, the so-called Hilbert matrix: $\mathbf{y} = H \cdot \mathbf{x}$. The matrix is simple, its inverse (it contains only integer elements) can be explicitly given. In MATLAB, the command `hilb(n)` generates it. For example, for $n = 4$

$$H = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \end{bmatrix} \qquad H^{-1} = \begin{bmatrix} 16 & -120 & 240 & -140 \\ -120 & 1200 & -2700 & 1680 \\ 240 & -2700 & 6480 & -4200 \\ -140 & 1680 & -4200 & 2800 \end{bmatrix}$$

Apply the reverse transformation to the result vector $\mathbf{y}$ immediately. That is, solve the equation system $H \cdot \mathbf{z} = \mathbf{y}$. Theoretically[24] , $\mathbf{z}$ must of course be identical to the original vector $\mathbf{x}$. Therefore, plot $\mathbf{x}, \mathbf{y}$ and $\mathbf{z}$. In MATLAB, you can do this as follows:

```
H = hilb(n);
y = H*x;
z = H\y;
plot(t,x,'.:',t,y,'o:',t,z,'o:')
legend('x','y=H*x','z=H\y')
title('Transformation und Rücktransformation')
```



In the figure here, for $n = 10$ data points, the $\mathbf{x}$ and $\mathbf{z}$ data points perfectly overlap.

Repeat the task for larger n. At what point does the reverse transformation yield significantly different values, and when does it become catastrophically unusable?

The Hilbert matrix is just one example of transformations leading to extremely poorly conditioned inverse problems.

Despite this, you should perform the reverse transformation for $n = 50$. In such cases, regularization methods are applied. Here's the recipe for Tikhonov regularization.

Instead of

```
    z = H\y;
```

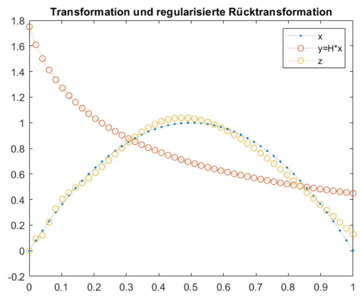
use the following for the reverse transformation

```
    alpha= …
    z = (H'*H + alpha*eye(n))\H'*y;
```

[24]Old wisdom: *In theory, there is no difference between theory and practice. In practice, there is.*

for a *very* small $\alpha > 0$.

For what value of α can you transform and reverse-transform $n = 50$ data points so that $\mathbf{x}$ and $\mathbf{z}$ match as closely as possible in the plot? Here on the right, the match is not particularly good yet. With a well-chosen α, it should be significantly better.



Regularization for inverse problems occurs in many application areas. This is a highly topical issue, although there are not even German Wikipedia entries for it[25].

The exercise materials do not delve into the theory[26], they just want to show here: even theoretically challenging methods can be executed with simple MATLAB commands.

L 3.5 Newton's Method for Systems: Instructions

Solve the following nonlinear system of equations using the Newton method.

$$\begin{aligned} 2x_1 - x_2 &= e^{-x_1} \\ x_1 + 2\sin x_2 &= \cos x_2 \end{aligned}$$

Here follows a step-by-step guide. At the end of this section, there's also a link to a completed solution. Urgent advice: work through this guide first (especially if you think that ready-made sample solutions save time and effort…)

Transform Equations into Zero-Problem

First, transform the equations into the form $\mathbf{f}(\mathbf{x}) = \mathbf{0}$.

$$\begin{aligned} 2x_1 - x_2 - e^{-x_1} &= 0 \\ x_1 + 2\sin x_2 - \cos x_2 &= 0 \end{aligned}$$

Compute Jacobian Matrix

Compute the Jacobian matrix: $D_{\mathbf{f}}(\mathbf{x}) = \begin{bmatrix} 2 + e^{-x_1} & -1 \\ 1 & 2\cos x_2 + \sin x_2 \end{bmatrix}$

Choose Initial Vector

Start with the initial vector $\mathbf{x}^{(0)} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$. Evaluate $\mathbf{f}$ and $D_{\mathbf{f}}$ for $\mathbf{x}^{(0)}$ – even better: let MATLAB do it (will be explained shortly).

[25]see the entries *Regularization (mathematics)* and *Tikhonov regularization* in English Wikipedia

[26]Explanation of the meaning of α (for interested parties only): The result vector $\mathbf{y}$ is not completely accurately determinable in practice. If one allows relative errors of the order of α in the result $\mathbf{y}$, then there are infinitely many initial vectors $\mathbf{x}$ that could have produced the result vector $\mathbf{y}$ within the accuracy range. The above recipe finds among all these vectors the one with the smallest 2-norm.

Although this is not necessarily the "correct" initial vector, it is the "most plausible" under certain additional assumptions.

Compute Correction Vector

Compute a correction vector $\Delta\mathbf{x}^{(0)}$ as a solution of a linear equation system. Its matrix is the Jacobian matrix, and on the right side stands $-\mathbf{f}(\mathbf{x}^{(0)})$.

Please refer to the theory in the script. There you will find:

> Iteration scheme
>
> $$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \Delta\mathbf{x}^{(k)}$$
>
> with $\Delta\mathbf{x}^{(k)}$ as a solution of $D_f(\mathbf{x}^{(k)})\Delta\mathbf{x}^{(k)} = -\mathbf{f}(\mathbf{x}^{(k)})$

MATLAB solves linear equation systems with the backslash operator `\`. You would solve a system $A\mathbf{x} = \mathbf{b}$ in the form `x = A\b`. For the Newton method, you can combine equation solving `Deltax = Df\(-f)` and correction step `xnew = x + Deltax` as a one-liner: `xnew = x - Df\f`

Programming

Program $\mathbf{f}(\mathbf{x})$ and $D_\mathbf{f}(\mathbf{x})$ as MATLAB functions and save them as files `f.m` and `Df.m`.

Sections L 2.1 and L 2.2 briefly explained inputting vectors and matrices. You can separate the matrix elements in a row by commas or spaces. Semicolons or the beginning of a new line in the source code separate the rows in matrices and column vectors.

Complete the following two function files!

```
function y = f(x)
y= [ ... %first component
     ... %second component
   ];
end
```

Vector-valued function: Vector as input, vector as output!

```
function dy = Df(x)
dy= [ ... %first row
      ... %second row
   ];
end
```

Matrix-valued function: Vector as input, matrix as output!

Check in the *Command Window* if the two function files can be called correctly. This should come out:

```
>> x=[1; 1];
>> f(x)
ans =
    0.6321
    2.1426
>> Df(x)
ans =
    2.3679   -1.0000
    1.0000    1.9221
```

Attention, set spaces correctly when programming the Jacobian matrix! Section L 3.2 already points this out; here it is again:

Spaces around $+$ and $-$ in terms in matrices:
CORRECT

- No spaces. Example `[ a+b, x+y ]`
- Spaces on both sides. Example `[ a + b, x + y ]`

WRONG:

- Space on one side. Example `[ a +b, x +y ]`

Manual Iteration in the Command Window

Once the function and Jacobian matrix are correctly programmed, test the Newton iteration in the *Command Window*. Start the search with $\mathbf{x} = \begin{bmatrix}1\\1\end{bmatrix}$.

```
>> x=[1; 1];
>> x = x - Df(x)\f(x)
x =
         0.395158347619808
         0.199928444993336
>> x = x - Df(x)\f(x)
x =
         0.449399668569469
         0.261761474532574
>> x = x - Df(x)\f(x)
x =
         0.449562385013556
         0.261217530152151
>> x = x - Df(x)\f(x)
x =
         0.449562377948136
         0.261217503068493
```

The initial value is moderately accurate. However, the second iteration already provides three significant figures; from then on, quadratic convergence behavior can be observed.

L 3.6 Local Functions

The two function files along with the Newton iteration are packed into a downloadable file. This program shows you the use of local functions.

Whether you organize your program in one single file with local or anonymous functions, or whether you prefer to divide your solution into scripts and functions in separate files, is up to you! All variants have advantages and disadvantages.

Get the file and execute it. For a neatly formatted version, you can also convert the file into a live script or use MATLAB's *publish* feature. See MATLAB Help, keyword *Publish and Share MATLAB Code*. The output should look like this:

```
>> NewtonMusterAufgabe
Convergence after 4 iterations
Root:
         0.449562377948136
         0.261217503068493
```

With this, it should be easy to solve the following task:

Task 22:

Exercise 20 asks for a solution to the system of equations

$$\begin{aligned} f(x,y) = 4x - y + xy - 1 &= 0 \\ g(x,y) = -x + 6y + \log xy - 2 &= 0 \end{aligned}$$

using the Newton method. Refer to the example in the previous section and to the template program `NewtonMusterAufgabe.m`. This example is also worked out in the script (page 26).

Task 23:

Solve the following nonlinear system of equations using the Newton method. Initial vector $[1;-1;0]$. Iterate until consecutive solutions in the ∞-norm, which is in MATLAB `norm(…,inf)`, match to 10^{-12}.

$$\begin{aligned} x^2 + \sin^2 y + z &= 1 \\ e^x + e^{-x} - yz &= 2 \\ x + y + z^2 &= 0 \end{aligned}$$

Task 24:

Given the function equations

$$\begin{aligned} &y = 2x^2 - 1 \\ &y = \sin(3x) \quad \text{for } -1 \le x \le 1 \quad . \end{aligned}$$

Draw both curves and read off the coordinates of the intersection points to one decimal place from the graph.

Transform both equations into $\cdots = 0$ and calculate the solution of the nonlinear 2×2 system using the Newton method.

Use the approximations from the graphical representation as initial values. Convergence criterion: solutions should not differ by more than 10^{-9} in any component.

Task 25:

In an aqueous solution of sodium acetate, a nonlinear system of four equations determines the concentration of the ions H^+, OH^-, Ac^-, and the undissociated acid HAc. Given $K_w = 10^{-14}, K_s = 10^{-4.75}$ and a variable concentration C.

$$\begin{array}{rcll} H^+ \cdot OH^- &=& K_w & \text{Ion product of water} \\ H^+ \cdot Ac^- &=& K_s \cdot \mathrm{HAc} & \text{Acid dissociation equilibrium} \\ \mathrm{HAc} + Ac^- &=& C & \text{Acetate total concentration} \\ Ac^- + OH^- &=& C + H^+ & \text{Electroneutrality} \end{array}$$

If you denote the unknown concentrations H^+, OH^-, Ac^-, HAc with variables x_1, x_2, x_3, x_4, the problem can be reformulated as follows:

$$\mathbf{f}(\mathbf{x}) = 0 \quad \text{with} \quad \mathbf{f}(\mathbf{x}) = \begin{bmatrix} -K_w + x_1 x_2 \\ x_1 x_3 - K_s x_4 \\ -C + x_3 + x_4 \\ C + x_1 - x_2 - x_3 \end{bmatrix}$$

Write a MATLAB function `f(x,C)` that calculates the vector $\mathbf{f}(\mathbf{x})$ for an input vector $\mathbf{x} = (x_1, x_2, x_3, x_4)$ and given concentration C. ($K_w = 10^{-14}, K_s = 10^{-4.75}$ are fixed.) Also, write the corresponding function for the Jacobian matrix and a Newton method for root finding. Once you find a solution, $-\log_{10} x_1$ is the pH value.

Good initial values are important in this example. Recommendation:

$$\mathbf{x}^{(0)} = [10^{-7}; 10^{-7}; C; 0]$$

For comparison, test values for $C = 0.01$

```
>> x0=[1.e-7;1.e-7;0.01;0];
>> f(x0,0.01)
ans =
  1.0e-008 *

  -0.00000000000000
   0.10000000000000
                  0
                  0

>> df(x0)
ans =

   0.00000010000000   0.00000010000000                  0                  0
   0.01000000000000                  0   0.00000010000000  -0.00001778279410
                  0                  0   1.00000000000000   1.00000000000000
   1.00000000000000  -1.00000000000000  -1.00000000000000                  0

>> x0=x0-df(x0)\f(x0,0.01)
x0 =

   0.00000000035638
   0.00000019964362
   0.00999980071276
   0.00000019928724
```

L 3.7 Solving Systems with *fsolve*

Systems of equations in the form $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ can be solved by MATLAB's *Optimization Toolbox* using the `fsolve` command. The call works similar to `fzero`.

Assuming you have programmed the function from Section L 3.5 as a function file `f.m`. Then the call and result are

```
>> fsolve(@f,[1,1])

Equation solved.

fsolve completed because the vector of function values is near zero
as measured by the default value of the function tolerance, and
the problem appears regular as measured by the gradient.

<stopping criteria details>

ans =
 0.449562378440135 0.261217525788617
```

You may suspect from the extensive output: it wasn't that simple. And indeed, it isn't – we cannot cover the underlying methods within the scope of this exercise.

From the eighth decimal place onwards, the results differ from those of the Newton method. If you value high accuracy, you would need to adjust the default settings of `fsolve`[27].

[27]Challenge: who can find settings so that `fsolve` provides exactly the same values to all 16 double digits as the Newton method? I tried and failed. C.B.

Task 25 cannot be solved with default settings. (`fsolve` claims “Equation solved” as well, but the result is completely off.)

Task 26:

Solve Exercises 22, 23, and 24 with `fsolve`. Set the display format in the *command window* to `format long g`, and compare the accuracy with the results of the Newton method.

L 3.8 Isoline and Surface Plots

Task 27: Plotting a function $z = f(x, y)$ with isolines

Plot the function

$$f : \mathbb{R}^2 \to \mathbb{R}, \quad z = xe^{-x^2-y^2}$$

as an isoline plot. The easiest way to do this is with the command `ezcontour`: (pronounced as American-English *ea-sy-contour*)

```
>> ezcontour('x*exp(-x^2␣-␣y^2)')
```

Attention, don't forget the single quotes '

However, if you want to specify more precisely how and what you want to plot, use the "correct" command `contour`. The MATLAB help under the keyword `contour` explains how to do this. You can work according to a sample example given there (depending on your MATLAB version, scroll three, four figures down):

For an isoline plot (contour plot) of the function $z = xe^{-x^2-y^2}$ in the range $-2 \le x \le 2, -2 \le y \le 3$, first generate x and y values on a grid in the xy plane.

```
>> x = -2:0.2:2;
>> y = -2:0.2:3;
>> [X,Y] = meshgrid(x,y);
```

Note: In English texts, you often see `.2` (without a leading 0 before the decimal point), where we would write 0.2. The MATLAB expression `-2:.2:2` is also correct.

For the (x, y) value pairs, compute a matrix Z using the command:

```
>> Z = X.*exp(-X.^2-Y.^2);
```

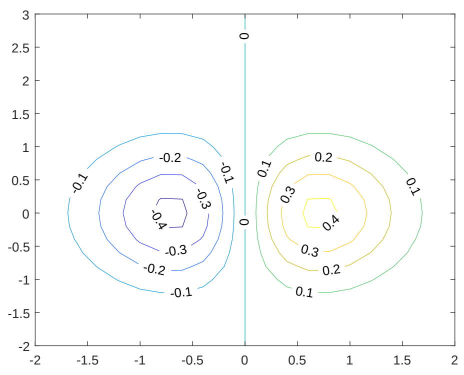
Next, generate the contour plot:

```
>> contour(X,Y,Z,'ShowText','on')
>> colormap cool
```

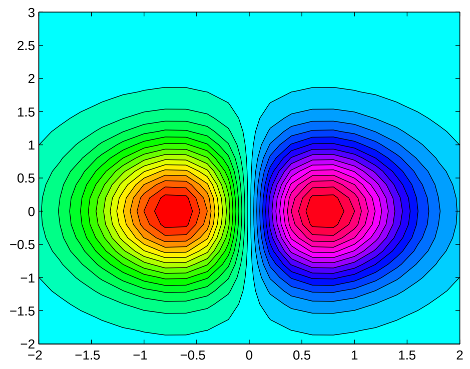
There are various variants of the contour command. You can find more examples in the MATLAB help (commands contourf, contour3)

This is how the contour plot should look like, generated with the above commands.

Other variants of the `contour` command: `contour(X,Y,Z,30)` generates 30 contour lines; `contour(X,Y,Z,-1:0.1:1)` generates contour lines for values from -1 to 1 in steps of 0.1.



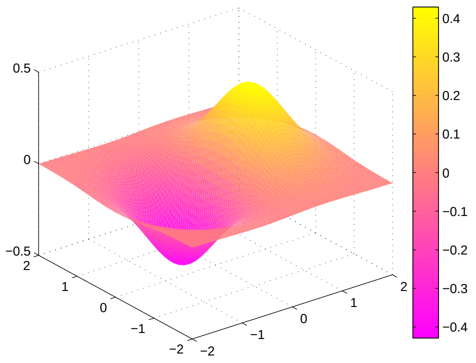
Explore the options of the `contour` command. Your task is to find out which commands make the plot look as close as possible to the one shown here:



Task 28: Representation of a function $z = f(x, y)$ as a surface in space

Very similar to `contour`, the command `surf` works. Look it up in the MATLAB help and draw the function from Exercise 27 as a surface in space.

Your task is to find out which commands make the plot look as close as possible to the one shown here (data range, resolution, color scheme, color bar, etc.).



L 3.9 Graphical search for roots for nonlinear equation systems in two variables

For a function $f : \mathbb{R} \to \mathbb{R}$ in the form $y = f(x)$, roots can be read from the function graph. You already know this from middle school and the first lecture.

A function $f : \mathbb{R}^2 \to \mathbb{R}$ can be represented in the form $z = f(x, y)$ as a surface in space or as an isoline graph, allowing you to graphically determine the location of roots as well. In this case, a root is a value pair (x, y) that satisfies the equation $f(x, y) = 0$.

For a system of two nonlinear equations in the form:

$$\begin{aligned} f_1(x, y) &= 0 \\ f_2(x, y) &= 0 \end{aligned}$$

you can define the function

$$f(x, y) = [f_1(x, y)]^2 + [f_2(x, y)]^2$$

. This function equals 0 if and only if f_1 and f_2 are both equal to 0. Since it is the sum of two squares, f cannot be less than 0 by definition. Therefore, the roots of f are also global minima of f and solutions of the nonlinear equation system $f_1 = 0$, $f_2 = 0$.

The next two tasks illustrate this relationship.

Introduction to Exercises 29 and 30

The function $f : \mathbb{R}^2 \to \mathbb{R}$, given by

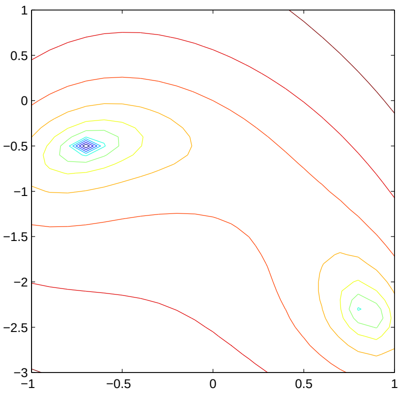
$$f(x, y) = \left[x^2 + \sin y\right]^2 + \left[e^x + y\right]^2$$

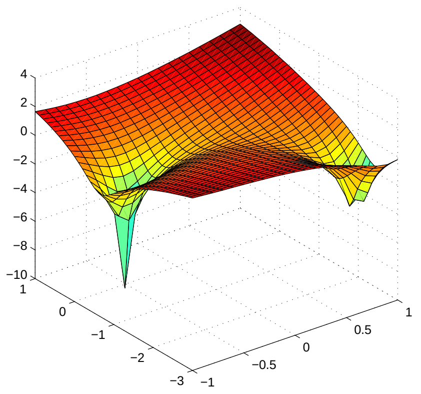
has a minimum value of 0 for exactly those value pairs $(x; y)$ that are solutions of the following nonlinear equation system:

$$\begin{aligned} x^2 + \sin y &= 0 \\ e^x + y &= 0 \end{aligned}$$

Task 29: Roots of a function in two variables (Part 1)

For $\log f$, the logarithm of the function given above, draw a `contour` and a `surf` graph in the range $-1 < x < 1$ and $-3 < y < 1$ similar to the figure. (In the logarithmic representation, the location of the minimum is more clearly visible than in the direct representation of f.) Use approximately 20 grid points in the x direction and approximately 40 grid points in the y direction.





Task 30: Roots of a function in two variables (Part 2)

Solve the nonlinear equation system given above using the Newton-Raphson method. Find both solutions in the range $-1 < x < 1$ and $-3 < y < 1$. You can derive suitable initial values from the graphical representation (Exercise 29). Termination criterion: Successive iterations differ, measured as the sum of the absolute differences of all component pairs, by less than 10^{-8}.

L 4 Fourth Lab Unit

Content of the fourth lab unit:

- Systems with unique, non-unique or no solution
- Sensitivity to errors, matrix norms, condition number
- Determinant, complexity
- Fitting models to data
- Linear data models

A Guide for this Lab Unit

In the current lecture notes, Chapter 3 deals with systems of linear equations with the same number of equations as unknowns. In the slides for the 4th lecture, you should review the sections

1. Solvability, Sensitivity to Errors, and
3. Direct Methods

for a recap.

In particular, the script chapters 3.2, 3.4, 3.6, 3.7.1, and 3.8 provide material directly related to the exercise tasks 31 – 35.

The other tasks in this unit deal with estimating the parameters of a model from datasets; for example: estimating the parameters of an exponential growth model from the number of infected persons, instructions for this in exercise materials L 4.4.2.

For this, you should review the topic

5. Overdetermined Systems

in the lecture slides and skim through Chapter 5 in the lecture script. We will cover the theory in more detail later, but for practical purposes, it's enough to know:

- A system with more equations than unknowns is usually not exactly solvable.
- An approximate solution (the "least incorrect solution") can be determined using the method of least squares.
- MATLAB's operator \ finds the least squares approximate solution for overdetermined systems of equations.

L 4.1 Systems of Equations, Solution Manifolds

Is it always possible to solve a system of linear equations with two equations and two unknowns?

> The rule of thumb "With as many equations as unknowns, there is always a solution" is WRONG. WRONG! **WRONG!!**

For the three systems of equations

$$\begin{aligned} x+y&=2 \\ 2x+2y&=4 \end{aligned} \qquad \begin{aligned} x+y&=2 \\ 2x+2y&=3 \end{aligned} \qquad \begin{aligned} x+y&=2 \\ x+2y&=3 \end{aligned}$$

your naked eye should clearly see: In the first and the third case, $x = 1,\ y = 1$ are solutions. The second system is unsolvable. However, the first system has infinitely many other solutions.

Review what you have learned in Mathematics 1. In the lecture notes, Chapter 3.4, you will find further information. You should be familiar with the following topics:

- Matrix rank, rank of the augmented matrix, MATLAB commands `rank(A)`, `rank([A,b])`, case distinctions for solvability.
- Row echelon form of a system of equations, MATLAB command `rref([A,b])`.
- General solution of the homogeneous system, null space, MATLAB command `null(A)`.
- Special solution of the inhomogeneous system, MATLAB commands `A\b` for full rank, `pinv(A)*b` for solvable systems with rank less than row number.
- For unsolvable systems, `pinv(A)*b` provides the "least wrong" answer (with the smallest error in the 2-norm).

Task 31: Systems of Equations, Solution Manifolds

Which of the following systems of equations $A\mathbf{x} = \mathbf{b}$ have a unique solution, a non-unique solution, no solution? If possible, provide the (or a) solution. In the case of non-unique solutions: how many free parameters does the solution set have?

Further explanations, steps
You can determine the possible cases for solvability based on the rank of the matrix and the rank of the augmented matrix, using commands like `rank(A)`, `rank([A,b])`. Alternatively, you can also use `rref([A,b])`.

If a unique solution exists, `A\b` is the appropriate command.

If a solution set exists, `pinv(A)*b` provides a possible solution, specifically the one with the smallest absolute value. The standard command `A\b` may also work; it provides a solution vector with as many components as possible equal to 0.

You can derive the complete representation of a solution manifold from the result of `rref([A,b])`, but some further manipulation is required.

Alternative: The general solution consists of a special solution (determined with `pinv(A)*b`) plus a linear combination of the column vectors of the null space (determined with `null(A)`).

1.
$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

2.
$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 2 \\ 0 \\ 1 \end{bmatrix}$$

3.
$$A = \begin{bmatrix} 10 & 9 & 8 & 7 \\ 6 & 5 & 4 & 3 \\ 2 & 1 & 0 & -1 \\ -2 & -3 & -4 & -5 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

4.
$$A = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \\ 1 & 4 & 9 & 16 \\ 1 & 8 & 27 & 64 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Task 32: LR Decomposition

The script covers Gaussian elimination, LR decomposition, forward elimination, and backward substitution in Chapter 3. It also lists JAVA codes. Here is a porting to MATLAB. By working through this task, you will have implemented your first own equation solver.

(Warning: The programs are intended to illustrate the basic form of the methods. They are not yet suitable for practical use; it would be necessary to add: pivoting, safeguards against singular matrices, division by zero, etc.

Use the example calculation provided in the script, Section 3.6: Consider the equation system $A \cdot \mathbf{x} = \mathbf{b}$ with
$$A = \begin{bmatrix} 5 & 6 & 7 \\ 10 & 20 & 23 \\ 15 & 50 & 67 \end{bmatrix} \quad , \quad \mathbf{b} = \begin{bmatrix} 6 \\ 6 \\ 14 \end{bmatrix}$$
You can thus trace all intermediate results.

Do not type manually as this is error-prone. Use copy-paste from the PDF file!

Step 1: LR decomposition. The two matrices L and R are not separately stored for efficiency reasons but replace the A matrix. The L matrix fills the lower triangle below the main diagonal excluding the one on the diagonal; The R matrix fills the upper triangle.

```
%% Decomposition A = LR
% Fails if zero occurs on the diagonal!
for k=1:n
    for i=k+1:n
        A(i,k)=A(i,k)/A(k,k);
        for j=k+1:n
            A(i,j) = A(i,j)-A(i,k)*A(k,j);
        end
    end
end
```

The decomposition is complete; compare (and note: the two output matrices overwrite the corresponding areas in A):

$$A \text{ (Input)} \begin{bmatrix} 5 & 6 & 7 \\ 10 & 20 & 23 \\ 15 & 50 & 67 \end{bmatrix} \quad , \quad L \cdot R \text{ (Output)} \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 4 & 1 \end{bmatrix} \cdot \begin{bmatrix} 5 & 6 & 7 \\ 0 & 8 & 9 \\ 0 & 0 & 10 \end{bmatrix}$$

Now comes the right-hand side b into play: The forward elimination solves $Ly = b$ and thus calculates the intermediate result y. In the code, y is stored as x immediately.

```
%% Vorwärts-Elimination Ly=b
x = b;
for i=1:n
    for j=1:i-1
        x(i) = x(i) - A(i,j)*x(j);
    end
end
```

The intermediate result y or x corresponds to the last column after the elimination process is completed, before the back substitution. Compare it in the script:

$$[A\,\mathbf{b}]^{(2)} = \begin{bmatrix} 5 & 6 & 7 & 6 \\ 0 & 8 & 9 & -6 \\ 0 & 0 & 10 & 20 \end{bmatrix}$$

```
%% Rücksubstitution Rx=y
for i=n:-1:1
    for j=i+1:n
        x(i) = x(i)-A(i,j)*x(j);
    end
    x(i) = x(i)/A(i,i);
end
```

Now x is the solution of the system.

How good is this very naive implementation compared to "correct" equation-solving programs? Test this with randomly generated systems

```
n=4;
A=floor(10*rand(n));
b=floor(10*rand(n,1));
```

and compare MATLAB's result `x=A\b` with our toy program. Most of the time the results should be the same. Test many systems and indicate: in what percentage of cases does our naive program fail? Why?

L 4.2 Sensitivity to Errors in the Elimination Method

Task 33: Sensitivity to Errors

Use Matlab's downhill dash to solve the system of equations $A\mathbf{x} = \mathbf{b}$ (it is the system of equations number 4 from exercise 31),

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \\ 1 & 4 & 9 & 16 \\ 1 & 8 & 27 & 64 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Now add to the matrix artificial error matrices of the form $\delta A =$ `0.01*rand(4)` and compare the solution with that of the undisturbed system.

Calculate the relative errors of the matrix data[28] $||\delta A||/||A||$ and the solution $||\delta \mathbf{x}||/||\mathbf{x}||$. Test some cases, also with smaller or larger perturbations, and estimate the *maximum ratio* of relative error of the solution to relative error in the data.

What does the value `cond(A)` mean in this context?

Now change the matrix of the system of equations to

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

(Hint: `hilb(4)` delivers exactly this matrix!) and repeat your investigations.

L 4.3 Development of Determinant, Complexity

Task 34: Determinant

Given are

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, C = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \\ 1 & 4 & 9 & 16 \\ 1 & 8 & 27 & 64 \end{bmatrix}, D = \begin{bmatrix} 10 & 9 & 8 & 7 \\ 6 & 5 & 4 & 3 \\ 2 & 1 & 0 & -1 \\ -2 & -3 & -4 & -5 \end{bmatrix}$$

Calculate the determinants of these matrices in various ways:

1. Using the Matlab function `det( )`
2. through LR decomposition, Matlab function `lu( )`, from the product of the diagonal elements of R[29]. The command `diag(R)` extracts the main diagonal of a matrix as a vector, and the command `prod` multiplies all elements.
3. Even the primitive LR program from Exercise 32 delivers the determinant as the product of the diagonal elements (even with correct sign, as it does not perform pivot search). Test!
4. The following function implements the standard method for calculating the determinant by expanding into subdeterminants. (Copy-paste from PDF, can also be downloaded from the exercise homepage)

```
function d = mydet(A)
n = length(A);
if n==1
    d = A(1,1);
else
    d = 0;
    sig = 1;
    for i=1:n
        d = d + sig*A(1,i)*mydet(A(2:n,[1:i-1,i+1:n]));
```

[28] You need matrix norms for this! Inform yourself in the script and lecture slides!

[29] Be careful, Matlab's `lu()` may swap rows of the matrix during computation (pivoting). With each row exchange, the sign of the determinant changes. So if `lu()` swapped an odd number of rows, the product of the diagonal elements has the wrong sign. In the context of this task, signs are arbitrary.

```
            sig = -sig;
        end
    end
```

5. Repeat the known standard methods (Rule of Sarrus, expansion into subdeterminants) and calculate by hand for A, B, C.

   Beware! The rule of Sarrus (sometimes called “basketweave method”, or *„Jägerzaunregel“* in German) only applies to 2×2 and 3×3 matrices. Anyone who wants to calculate 4×4 determinants this way is wrong. WRONG! **WRONG!!**

Task 35:

A computational method that calculates correctly but takes forever to do so is of theoretical value only. Calculating the determinant by expanding into subdeterminants is an example of this.

The function `mydet` from Exercise 34 implements the standard method for recursively calculating the determinant by expanding into subdeterminants.

Test the computation time of `mydet` compared to the MATLAB standard command `det`. Simple test matrices are provided, for example, by the function `magic(n)`.

MATLAB provides a stopwatch-like functionality with the commands `tic` and `toc` to measure computation time. You could use code like the following:

```
>> A = magic(4)
A =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
>> tic; mydet(A); toc
Elapsed time is 0.000203 seconds.
```

Now try larger matrices than in Exercise 34. Up to what n can the determinant be calculated

1. in less than ten seconds,
2. in less than one minute
3. in less than ten minutes

Of course, this depends on the performance of your computer. Based on your time measurements and the table in the script, Chapter 3.7.1, estimate how long the calculation for $n = 15$ and $n = 20$ would take.

L 4.4 Fitting Model Functions to Data

Given are data points, sought are parameters for a model that describes these data. In the simplest case: a “nice curve” that fits the course of the data points well.

Statistically meaningful is this only if (significantly) more data points are given than model parameters sought. This leads to overdetermined systems of equations.

Materials and links can be found in the introduction. Once again, the important statement in advance:

MATLAB's downhill dash \ finds for overdetermined systems of equations (which are usually not solvable exactly) the "least wrong" result.

More precisely: MATLAB's downhill dash \ finds for overdetermined systems of equations the best possible approximation in terms of least squares.

MATLAB offers many tools in its *curve fitting toolbox* to fit curves or surfaces to given data. Before delving into it, you should understand the basic ideas and methods. That's what this lab unit is about.

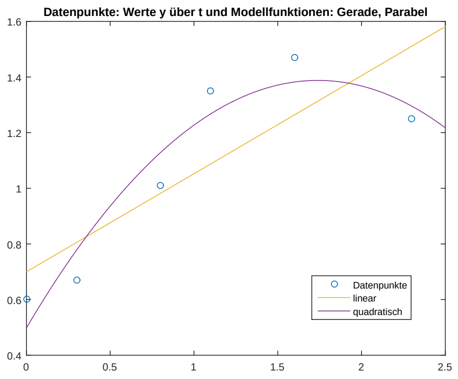
L 4.4.1 One Variable: Example from MATLAB Help

The MATLAB help (R2019–R2021) discusses examples under MATLAB › Data Import and Analysis › Descriptive Statistics › Programmatic Fitting › MATLAB Functions for Polynomial Models (type *"Programmatic Fitting"* into the search window of the help, then you will quickly find this example.)

Here is the example from the section *MATLAB Functions for Polynomial Models* prepared for you, so that you can solve the following task based on the information from the MATLAB help and additional information:[30]

Given are measurement values y at different times t as column vectors:

```
t = [0 0.3 0.8 1.1 1.6 2.3]';
y = [0.6 0.67 1.01 1.35 1.47 1.25]';
```



You are supposed to find a function that models the relationship $y = y(t)$ well. Simple approaches are: a linear function, or a more general polynomial, for example:

$$y = a_0 + a_1 t \qquad \text{or}$$
$$y = a_0 + a_1 t + a_2 t^2$$

For this, MATLAB provides the commands `polyfit` and `polyval` and the *Basic Fitting User Interface*. Tip: use `plot` to draw the points. Click on Tools › Basic Fitting in the *figure* window and try it out!

The polynomial functions do not fit the data very well in this example. Instead, try a linear model[31] with more general functions, namely

$$y(t) = a_0 + a_1 e^{-t} + a_2 t e^{-t}$$

[30] Actually, you only need to copy the example commands from the MATLAB help into your own script file. But please, do not turn off your brain while using Ctrl+C and Ctrl+V!

[31] *Linear* here does not refer to the functions used in the model: e^{-t} and te^{-t}. *Linear model* means that these functions are *linearly combined*: so a model of the form "Coefficient 1 times Function 1 plus Coefficient 2 times Function 2…"

Procedure: Substitute the given data points into the model function. For each pair of values, you get a (linear!) equation in the three unknowns a_0, a_1, a_2.

$$\begin{aligned} 0.60 &= a_0 + a_1 e^{0.0} + a_2 \cdot 0.0 \cdot e^{0.0} \\ 0.67 &= a_0 + a_1 e^{-0.3} + a_2 \cdot 0.3 \cdot e^{-0.3} \\ 1.01 &= a_0 + a_1 e^{-0.8} + a_2 \cdot 0.8 \cdot e^{-0.8} \\ \ldots & \quad \ldots \end{aligned}$$

In your MATLAB script, create the coefficient matrix X (the MATLAB help says: *form the design matrix*). Caution: `t` and `y` must be column vectors!

```
X = [ones(size(t))  exp(-t)  t.*exp(-t)];
```

Understanding the structure of this matrix is crucial. The coefficients of a_0 in the above system of equations are always ones, so the first column of X contains only ones, like an elementary school report card. The coefficients of a_1 are e^{-t} values, so the second column of X contains a vector of e^{-t} values, and so on.

Please think along: Of course, the first column of X is not always automatically a primary school report card – it depends on the chosen basis functions and their order. But as a simple rule applies: "The columns of X contain the values of the basis functions."

The right side consists simply of the y-values. The model coefficients a_0, a_1, a_2 are computed by the backslash operator $\backslash$ as the best possible approximation from the overdetermined system of equations:

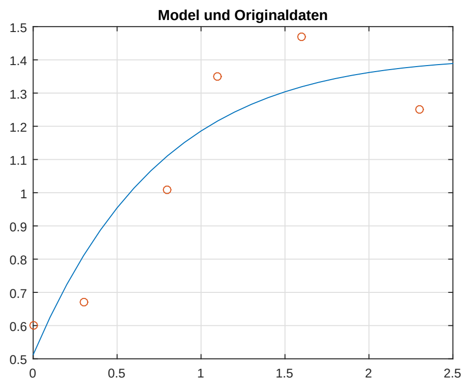
```
a = X\y
a =

    1.3983
   -0.8860
    0.3085
```

Thus, the best possible data model is given by

$$y = 1.3983 - 0.8860\, e^{-t} + 0.3085\, te^{-t}.$$

Now you can evaluate your model function (with sufficiently high resolution, i.e., at many t-points at close intervals) and plot it together with the original data.

```
T = (0:0.1:2.5)';
Y = [ones(size(T))  exp(-T)  T.*exp(-T)]*a;
plot(T,Y,'-',t,y,'o'), grid on
title('Model and Original Data')
```



Also, please think along here: the insertion of t-data into the model is elegantly done again in matrix-vector form: The matrix Y is structured similarly to the X matrix used previously. Multiplying Y by coefficient vector **a** yields model values.

Alternative evaluation: (in reality the same calculation as above, just written out individually for the components of `a` – perhaps it's easier to understand this way)

```
Y = [ones(size(T))  exp(-T)  T.*exp(-T)]*a;  % Evaluation in matrix-vector form
%
Y = a(1) + exp(-T)*a(2) + T.*exp(-T)*a(3); % Alternative: component-wise writing
```

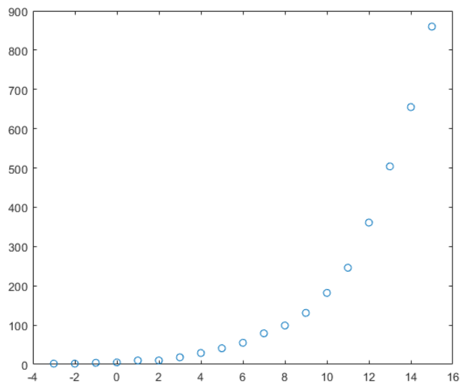
L 4.4.2 Fitting an Exponential Growth Model

Given are (t, y)-value pairs. (MATLAB source code, copy-pasteable, you just need to fix the single quotes at `'o'`)

```
t = -3:15;
y = [2 2 4 5 10 10 18 29 41 55 79 ...
99 131 182 246 361 504 655 860];
plot(t,y,'o')
```

The parameters a and b of an exponential growth model are sought

$$y = a \exp(bt) \ .$$



It concerns the COVID-19 cases in Austria (cumulated), with the time axis t in days from February 26th to March 15th, 2020.

If you insert the y and t values into the model equation here, just like in section L 4.4.1, you will get one equation for the unknown parameters a and b per data pair.

$$\begin{aligned} 2 &= a \exp(-3b) \\ 2 &= a \exp(-2b) \\ 4 &= a \exp(-b) \\ &\vdots \qquad \vdots \\ 860 &= a \exp(15b) \end{aligned}$$

However, unlike before, the equations here are nonlinear in a and b. We will deal with truly nonlinear overdetermined systems later. But this system can be simplified into a linear problem.

This simple trick shows you how to get rid of a nonlinear system...

Take the logarithm of the equations:

$$\begin{aligned} \log 2 &= \log a - 3b \\ \log 2 &= \log a - 2b \\ \log 4 &= \log a - b \\ &\vdots \qquad \vdots \\ \log 860 &= \log a - 15b \end{aligned}$$

(Of course, here log stands for the natural logarithm!) Also, denote $\log a = \bar{a}$, then we have an overdetermined linear system in the unknowns $\bar{a}$ and b.

In MATLAB, transform the data vectors from row to column vectors. You can then generate the coefficient matrix X and the right-hand side of this system as follows:

```
t = t';
y = y';
X = [ones(size(t)) t];
rhs = log(y);
```

The backslash operator provides the least squares estimate for the unknown parameters $\bar{a} = \log a$ and b.

```
ab=X\rhs
```

For the given data, `ab(1)` $= \bar{a} = \log a = 1.7602$, and thus the parameter $a = \exp(\bar{a}) = 5.8135$. This is the (model-estimated) initial value for $y(0)$, where $t = 0$ is February 29th.

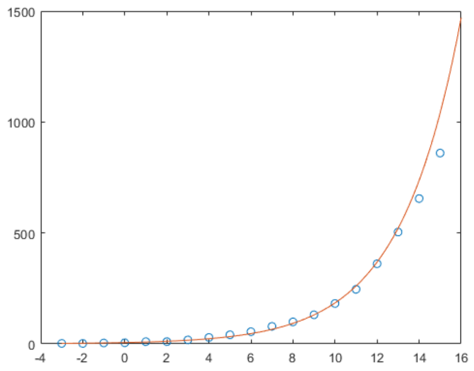
The value `ab(2)` $= b = 0.3458$ is the growth constant in the exponential model. Multiplying the y-values by the factor $\exp(b) = 1.4132$ per day corresponds to a daily increase of 41%. The doubling time T_2 in days is obtained from

$$T_2 = \frac{\log 2}{b} = 2.0043 \quad .$$

Do you still remember, that was the worrying data situation a few years ago. Let's take a closer look at the data and the estimated growth model.

We need time points in finer resolution for the time axis and let's also extrapolate one day into the future. `linspace` provides a hundred time points, which is sufficient. For this T-vector, we evaluate the model.

```
a = exp(ab(1));
b = ab(2);
T = linspace(-3,16)';
Y = a*exp(b*T);
plot(t,y,'o',T,Y)
```



Fortunately, the last two data points already lie below the model curve.

Caution! Based on this alone (with data as of March 2020), no well-founded prediction could have been derived. The model and our data adjustment are too simplified for that. What could be said at that time: An exponential growth model with the parameters calculated here looks plausible in the graphical representation—except for the last two data points—**if nothing else slows down the growth**, this model predicts catastrophic case numbers in two weeks.

In retrospect, the model predictions can be checked. At the end of this section, a graphical representation is added up to early April.

But first, let's try a slightly refined model calculation. Because our logarithmic linearization trick causes a statistical error: the parameter estimation minimizes the errors in the logarithmic data. As a result, the (less informative) data points at the beginning are weighted more heavily than the (more recent) data points towards the end of the time interval.

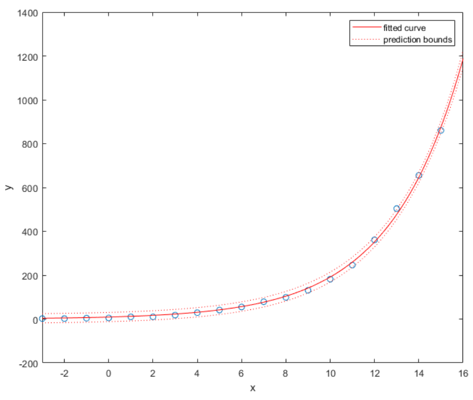
We will delve deeper into the direct adjustment of the nonlinear data model $y = a\exp(bt)$ later, but for now, let MATLAB do the work. If you have the *Curve Fitting Toolbox* installed, the command
`model = fit(t,y,'exp1')` Caution, it's 1, one, not lowercase L in `'exp1'` !)
and yields the result

```
model =
     General model Exp1:
     model(x) = a*exp(b*x)
     Coefficients (with 95% confidence bounds):
       a =       9.162  (7.889, 10.44)
       b =       0.304  (0.294, 0.314)
```

The estimated parameters are accompanied by a 95% confidence interval. **Caution!** These statements assume that the data actually obey an exponential law and that the individual data points deviate from the exponential curve only randomly with a certain probability distribution. So the statement is: *If* exponential growth were present, *then* these would be the estimated parameters.

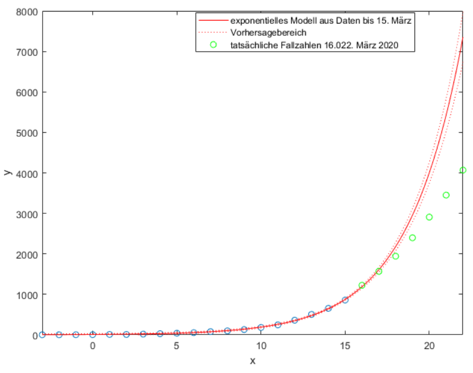
When the model is calculated via MATLAB's `fit` command, there are additional options for the graphical representation: The keyword `'predobs'` also draws uncertainty bounds. The `xlim([-3,16])` command sets the x- (here t-) axis range; a larger end value would extrapolate the model further into the future.

```
plot(t,y,'o')
hold on
xlim([-3,16])
plot(model,'predobs')
hold off
```



Again, **caution!** The tight uncertainty bounds should not imply high accuracy or reliability. The curves are based on the assumption that the course really follows an exponential model.

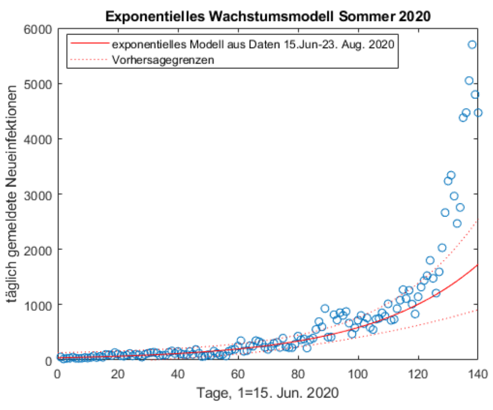
We did not extrapolate the model further into the future in the above graph because the model assumptions were strongly simplified and uncertain. In fact, the first lockdown was able to slow down exponential growth. The graph on the right, supplemented with more data points, shows that a few days after the start of the lockdown, the reported case numbers deviate from the exponential model curve. Another growth model sets in.



Task 36: COVID-19 in Summer and Autumn 2020

You can download data on daily new infections in Austria, covering a period of 140 days (June 15 to Nov 1, 2020), here.

Calculate the parameters of an exponential growth model from the first 70 data points, both in simple form (linear fit to the logarithmic data) and with MATLAB's *curve fitting toolbox*, using the `fit` command. Compare the adjusted exponential model with the actual data. Present your model calculation roughly as shown in the graph on the right.



Interpretation: Until about day 120, the actual case numbers are somewhat within the prediction limits, but then the data break out upwards. Obviously, at this point (mid-October), the mode of transmission had changed significantly. Consequently, lockdown measures had to be reintroduced from the beginning of November.

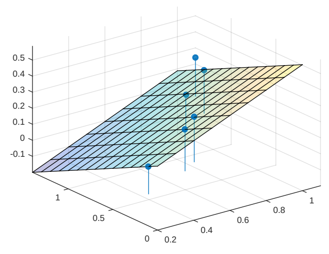
L 4.4.3 Multiple Variables

The MATLAB help shows *Multiple Regression* as the next example under
MATLAB 》Data Import and Analysis 》Descriptive Statistics 》Programmatic Fitting 》Multiple Regression.

Description and illustration can also be found in the lecture slides for Lecture 4, penultimate slide. The script also explains in Chapter 5.4: Fitting a linear model (a plane fit).

Here is a sample solution to the exercise from the penultimate lecture slide. In comparison to the solution in the MATLAB help, it also shows how to draw the plane and the original data.

```
%% Data points: Variables x1, x2, and measured value y
x1 = [.2 .5 .6 .8 1.0 1.1]';
x2 = [.1 .3 .4 .9 1.1 1.4]';
y  = [.17 .26 .28 .23 .27 .24]';

%% Overdetermined system matrix
% for model y = a0 + a1*x1 + a2*x2
X = [ones(size(x1))  x1  x2];

%% Solution: Coefficient vector
a = X\y
%% Evaluation and drawing on a fine grid
[XX,YY]=meshgrid(0.2:0.1:1,0:0.1:1.4);
ZZ= a(1) + a(2)*XX + a(3)*YY;

surf(XX,YY,ZZ,'FaceAlpha',0.3),
axis tight, hold on
% Original data
stem3(x1,x2,y,'.','MarkerSize',25)
hold off
```



L 4.5 More Tasks on Linear Data Models

Task 37: Floods

For the Blies (a tributary of the Saar), the flood levels at the Neunkirchen gauge are to be predicted from the water levels at the Ottweiler gauge and the Hangard gauge. The data of the peak water levels from 12 winter floods from 1963 to 1971 are available: (You can download the data from the exercise homepage)

Water level in cm												
Neunkirchen y	172	309	302	283	443	298	319	419	361	267	337	230
Ottweiler x_1	93	193	187	174	291	184	205	260	212	169	216	144
Hangard x_2	120	258	255	238	317	246	265	304	292	242	272	191

Data Source: U. Maniak, Hydrologie und Wasserwirtschaft, Springer, 1988

We assume the linear model $a_0 + a_1x_1 + a_2x_2 = y$ for the data. In this approach, a_0, a_1, and a_2 are unknown coefficients that should be determined as accurately as possible from the twelve given sets of values.

Calculate the coefficients of the linear model and also provide the difference vector between model predictions and measurements. (This example is worked out in detail in Chapter 5.4 of the script)

Task 38: Jack Sparrow's Compass

Whether you have attended a geophysics lecture or watched Johnny Depp in *Pirates of the Caribbean*, you may have learned that a compass needle doesn't always point to the north. This deviation is called magnetic declination, and its knowledge is still essential for every helmsman who navigates a vessel. Therefore, the Central Institute for Meteorology and Geodynamics (now known as *Geosphere Austria* since 2023) regularly provided current magnetic declination values for all state capitals. However, since around 2017, this table is no longer supplied by Geosphere Austria, probably because the importance of marine navigation in Austria has decreased.

The values of the declinations for the individual state capitals (eastern declination based on the mid-2008):

```
Declination values of the individual state capitals
(eastern declination based on mid-2008)

City          Longitude  Latitude  Declination
                 x          y         D
Vienna         16.37      48.20      3.0000
Eisenstadt     16.52      47.85      3.0333
St. Pölten     15.63      48.20      2.9000
Graz           15.45      47.07      2.7833
Linz           14.30      48.30      2.5500
Klagenfurt     14.31      46.62      2.5000
Salzburg       13.03      47.80      2.2500
Innsbruck      11.40      47.27      1.8833
Bregenz         9.77      47.50      1.4000
```

Find an adjustment for these data in the form

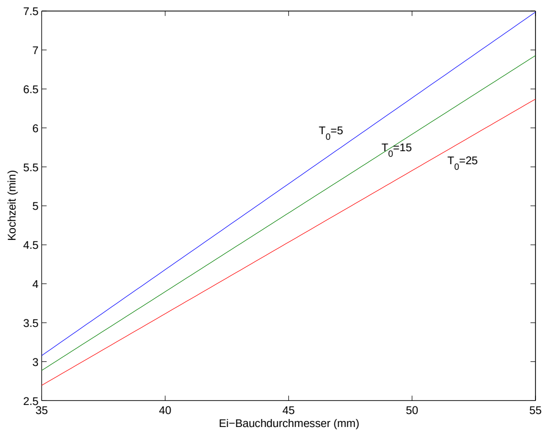
$$z(x, y) = a_1 + a_2x + a_3x^2 + a_4y$$

Give the difference $D - z(x, y)$ between the actual declination values and the adjusted values. Which value is approximated most badly? Modern models for earth's magnetic field (International Geomagnetic Reference Field IGRF, World Magnetic Model WMM) essentially use the same approach: They fit a model to measurement data. However, unlike our polynomial approach for 9 data points, they process huge amounts of data and use spherical harmonic functions as an approach.

Task 39: For Softies

The Viennese physicist Werner Gruber deals with the thermodynamics of egg cooking and provides the values for the cooking time of a perfectly soft breakfast egg depending on the diameter d (not his, that of the egg) and the initial temperature T_0 (You can download the data from the exercise homepage)

Diameter d (mm)	Initial Temperature T_0 (C)	Cooking Time (min:sec)
35.0	4	3:10
40.0	4	4:10
45.0	4	5:20
50.0	4	6:30
35.0	20	3:00
40.0	20	3:40
45.0	20	4:40
50.0	20	5:50



Find an adjustment for these data in the form

$$t(d, T_0) = a_1 + a_2 d + a_3 T_0 + a_4 d T_0$$

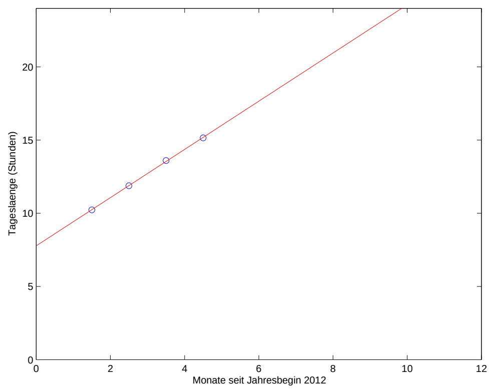
Draw a diagram similar to the one shown here (x-axis: egg diameter, y-axis: cooking time), in which you enter the cooking times for initial temperatures of 5, 15, 25 degrees.

Task 40: The Empire on Which the Sun Never Sets

An exercise on data models using polynomials and more general model functions.

The following table provides the length of the day (from sunrise to sunset) in hours for four days from February to May in Leoben. A trend towards longer days is clearly visible.

Date			Months since beginning of year	Day length in hours
15.	Feb.	2019	1.5	10.23
15.	Mar.	2019	2.5	11.88
15.	Apr.	2019	3.5	13.60
15.	May.	2019	4.5	15.15



Based on the graph of this data, financial advisors and other talented salespeople could convincingly explain to you that the sun will no longer set by the end of October.

Before you invest in solar stocks, create models for this dataset yourself:

1. Simple linear regression: $y = a_0 + a_1 x$
2. Polynomial approach $y = a_0 + a_1 x + a_2 x^2 + a_3 x^3$
3. Data model $y = a_1 + a_2 \cos\left(x\frac{\pi}{6}\right) + a_3 \sin\left(x\frac{\pi}{6}\right)$

Display the data points and the three models in a graph (axis range as above). Answer the following questions (by reading from the graphical representation): When does Model 1 predict the sun will not set in Leoben (day length 24 hours)? When does Model 2 predict eternal night (day length 0 hours)? When is the summer solstice (maximum day length) according to model 3?

Only Model 3 can make reliable predictions because it uses additional knowledge about the nature of the data: it appropriately models the periodic fluctuations of the sun's path by sine and cosine terms.

L 5 Fifth Lab Unit

Content of the fifth lab unit:

- Nonlinear Data Models
- Brief review: linear and nonlinear data models
- Bonus material: MATLAB tools for fitting functions to data
- Alternatives to least squares minimization: Robust regression, total least squares

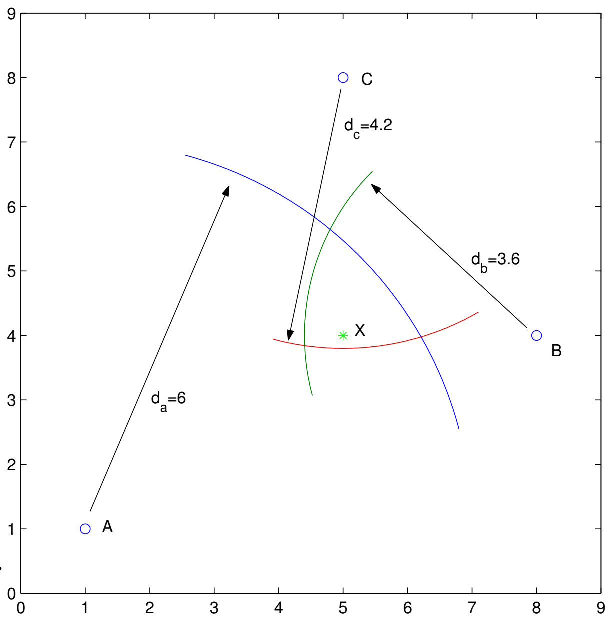
L 5.1 Overdetermined Nonlinear Systems, Gauss-Newton Method

The Gauss-Newton method finds approximate solutions (in the sense of least squares) for overdetermined nonlinear systems. The basic idea is to iteratively calculate correction terms using the *Jacobian matrix* of the nonlinear system, similar to solving nonlinear equation systems with the Newton method. Exercises 41 and 42 explain the calculation process in detail.

Task 41: Trilateration to Determine Position

The distances from three fixed points A, B, C in the xy-plane to an unknown point X are (somewhat inaccurately) known. The goal is to determine the position as accurately as possible.

```
Point x  y   Distance
      1  1     6
      8  4     3.6
      5  8     4.2
```



The corresponding equations are:

$$\begin{aligned}\sqrt{(x_1-1)^2+(x_2-1)^2} &= 6\\ \sqrt{(x_1-8)^2+(x_2-4)^2} &= 3.6\\ \sqrt{(x_1-5)^2+(x_2-8)^2} &= 4.2\end{aligned}$$

The three equations correspond to three circles in $\mathbb{R}^2$. They do not have a common intersection point. Accordingly, there is no exact solution to the overdetermined system of equations.

Write a MATLAB program that performs the least squares fitting for the location.

Note: This task is the 2-dimensional simplification of position determination in space. GPS technology is based on these geometric principles. A GPS receiver measures distances (more precisely: signal travel times) to multiple satellites. The signals are highly noisy; only curve fitting and additional data filtering ensure a position accurate to a few meters.

Summary: Task and Solution Approach

Given: Overdetermined nonlinear system

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}\ , \quad \mathbf{x} \in \mathbb{R}^n\ , \quad \mathbf{f}(\mathbf{x}) \in \mathbb{R}^m, \quad m > n$$

Iterative solution of the linearized system: Starting from initial vector $\mathbf{x}^{(0)}$, determine a correction $\Delta\mathbf{x}$.

The computational rule of the Newton method for $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ yields an overdetermined linear system with the Jacobian matrix D_f

$$D_f \cdot \Delta\mathbf{x} = -\mathbf{f}(\mathbf{x})$$

Improved solution $\mathbf{x}^{(1)} = \mathbf{x}^{(0)} + \Delta\mathbf{x}$.

Specifically for the present task:

$$\mathbf{f}(\mathbf{x}) = \begin{bmatrix} \sqrt{(x_1-1)^2+(x_2-1)^2} - 6 \\ \sqrt{(x_1-8)^2+(x_2-4)^2} - 3.6 \\ \sqrt{(x_1-5)^2+(x_2-8)^2} - 4.2 \end{bmatrix}, \qquad D_f = \begin{bmatrix} \frac{x_1-1}{\sqrt{(x_1-1)^2+(x_2-1)^2}} & \frac{x_2-1}{\sqrt{(x_1-1)^2+(x_2-1)^2}} \\ \frac{x_1-8}{\sqrt{(x_1-8)^2+(x_2-4)^2}} & \frac{x_2-4}{\sqrt{(x_1-8)^2+(x_2-4)^2}} \\ \frac{x_1-5}{\sqrt{(x_1-5)^2+(x_2-8)^2}} & \frac{x_2-8}{\sqrt{(x_1-5)^2+(x_2-8)^2}} \end{bmatrix}$$

With the initial vector $\mathbf{x} = \begin{bmatrix} 5 \\ 4 \end{bmatrix}$, we obtain

$$\mathbf{f}\left(\begin{bmatrix} 5 \\ 4 \end{bmatrix}\right) = \begin{bmatrix} -1 \\ -3/5 \\ -1/5 \end{bmatrix}, \ D_f = \begin{bmatrix} \frac{4}{5} & \frac{3}{5} \\ -1 & 0 \\ 0 & -1 \end{bmatrix}, \quad \text{linear system} \begin{bmatrix} \frac{4}{5} & \frac{3}{5} \\ -1 & 0 \\ 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 3/5 \\ 1/5 \end{bmatrix}$$

Results in $\Delta x_1 = 1/25, \Delta x_2 = 7/25 \quad \longrightarrow$ improved position $[5.04; 4.28]$.

Repeat these calculations and iterate until the position no longer changes within a reasonably chosen error threshold. Compare this to exercise section L 3.5: the calculation process and especially the implementation in MATLAB are almost identical. The only difference is that the matrix of the equation system is no longer square and not constant; it depends on the currently approximate values for $\mathbf{x}$.

Of course, this recipe doesn't provide information about the theory of overdetermined linear systems or the properties of the computed least squares solution. However, the idea of approximating a nonlinear system by linearizing it with the Jacobian matrix will constantly appear in practical applications.

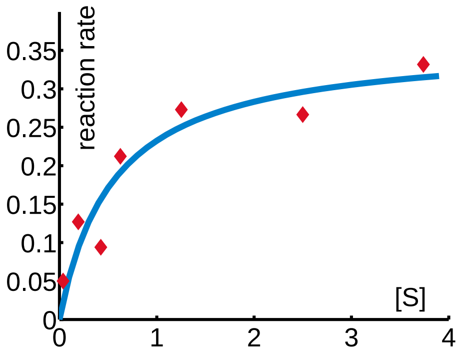
Task 42: Gauss-Newton Method on Wikipedia

The information for the following example comes from the English and French Wikipedia pages (keywords *Gauss-Newton algorithm* and *Algorithme de Gauss-Newton*). Solve the task with the following instructions in MATLAB. [32]

[32] Anyone reading this example on the English or French Wikipedia page and clicking on "more details" in the curve modeling section will find ready-made MATLAB code for solving the example and plotting the curve. However, understanding someone else's code is not always straightforward.

Biological experiments on the relationship between substance concentration x and reaction rate y in an enzyme-mediated reaction have yielded the following data:

x	0.038	0.194	0.425	0.626	1.253	2.500	3.740
y	0.05	0.127	0.094	0.2122	0.2729	0.2665	0.3317



The goal is to find a curve (model function) of the form

$$y = a\frac{x}{b+x}$$

that best approximates the data in the least squares sense. The parameters a and b need to be determined.
Substituting the data results in seven nonlinear equations in the two unknowns a, b.

$$a\frac{0.038}{b+0.038} - 0.05 = 0$$
$$a\frac{0.194}{b+0.194} - 0.127 = 0$$
$$\vdots$$
$$a\frac{3.740}{b+3.740} - 0.3317 = 0$$

The equation system in vector form:

$$\mathbf{f}(\mathbf{x}) = 0 \text{ with } \mathbf{f} : \mathbb{R}^2 \to \mathbb{R}^7$$

The Jacobian matrix D_f of this system is a 7×2 matrix. Row i contains the partial derivatives of the i-th equation with respect to the unknowns a and b.

$$(D_f)_{i1} = \frac{x_i}{b+x_i}, \quad (D_f)_{i2} = -\frac{ax_i}{(b+x_i)^2}$$

The further calculation process proceeds entirely analogously to the instructions in the exercise section L 3.5 and the model solution on page 26 in the script: Choose the initial value `[a;b]=[0.9;0.2]`, evaluate $\mathbf{f}$ and D_f. The least squares approximation from the overdetermined equation system $D_f\mathbf{\Delta x} = -\mathbf{f}$ provides the correction vector $\mathbf{\Delta x}$.

Note: With a square Jacobian matrix, the MATLAB command `Df(x0)\f(x0)` gives the solution to the equation system; if the Jacobian matrix has more rows than columns (overdetermined system), the same command *does not* provide the solution (there is no exact solution!), but rather an adjustment with the smallest residual error (the “least incorrect answer”)!

Bonus Question: Our Newton template program finds the improved approximation with the MATLAB command

```
 x = x0 -  Df(x0)\f(x0);    % Newton step
```

The Wikipedia example calls the Jacobian matrix J_f and uses the following command for the same step (written in MATLAB code)

```
 x = x0 - (J'*J)\(J'*f(x0));
```

Why use `Df(x0)\f(x0)` in one case and `(J'*J)\(J'*f(x0))` in the other? What is being calculated in each case? Advantages and disadvantages of the two variants in comparison?

Task 43: Calorimeter Experiment

The dataset `Kalorimeter.dat` (click!)`Kalorimeter.dat` contains pairs of values (time t in minutes, temperature T in Celsius) from a calorimeter experiment, measured from the introduction of a test object. The temperature initially decreases rapidly, then rises again and gradually approaches the ambient temperature.

The course of temperature T as a function of time t is to be described by a model of the form

$$T(t) = T_0 + C_1 \exp(-\lambda_1 t) + C_2 \exp(-\lambda_2 t)$$

Determine the parameters of this nonlinear model based on the initial values

$$T_0 = 6 \qquad C_1 = -1 \qquad \lambda_1 = 1/10 \qquad C_2 = 20 \qquad \lambda_2 = 2$$

(three iterations of the Newton method are sufficient). Plot data points and model function.

L 5.2 Brief Review: Linear and Nonlinear Data Models

The tasks for fitting functions to data in the previous unit were limited to *linear models*. In this context, "linear" means that the sought fit f is a *linear combination* of n basis functions $\phi_1, \phi_2, \ldots, \phi_n$ in the form

$$f = a_1\phi_1 + a_2\phi_2 + \cdots + a_n\phi_n \quad .$$

Important: The basis functions $\phi_1, \ldots, \phi_n$ can be arbitrary, even nonlinear functions — it's the sought coefficients $a_1, \ldots, a_n$ that appear linearly in the model!

Data fitting with linear models results in an overdetermined System of linear equations for the unknown coefficients $a_1, \ldots, a_n$. The matrix columns contain the values of the basis functions $\phi_1, \ldots, \phi_n$ at the given data points.

Confusingly, the term "linear regression" is also used for fitting a straight line. "Polynomial regression" refers to fitting a polynomial to given data, but it is still a linear data model.

Differentiate between:

- Type of fitting functions: Linear ("linear regression"), Polynomial ("polynomial regression"), Trigonometric or Exponential functions...

- Linking of fitting functions: Linear, nonlinear. This includes the terms linear/nonlinear data model.

Compare: Linear Models

Task 38: Approach $z(x, y) = a_1 + a_2x + a_3x^2 + a_4y$.
Here, the basis functions are $\phi_1(x, y) = 1, \phi_2(x, y) = x, \phi_3(x, y) = x^2, \phi_4(x, y) = y$.

Task 39: Approach $t(d, T_0) = a_1 + a_2 d + a_3 T_0 + a_4 d T_0$.
Basis functions are $\phi_1(d, T_0) = 1, \phi_2(d, T_0) = d, \phi_3(d, T_0) = T_0, \phi_4(d, T_0) = d \cdot T_0$.

Task 40: Approach $y = a_1 + a_2 \cos\left(x\frac{\pi}{6}\right) + a_3 \sin\left(x\frac{\pi}{6}\right)$
Basis functions are $\phi_1(x) = 1, \phi_2(x) = \cos\left(x\frac{\pi}{6}\right), \phi_3(x) = \sin\left(x\frac{\pi}{6}\right)$.

Nonlinear Models

Task 42: Approach $y = a\frac{x}{b+x}$. Here, the sought coefficients a and b are nonlinearly linked!

Task 43: Approach $T(t) = T_0 + C_1 \exp(-\lambda_1 t) + C_2 \exp(-\lambda_2 t)$.
If λ_1 and λ_2 were known, it would be a linear model with parameters T_0, C_1, and C_2. Because both λ_i are also sought, the task becomes nonlinear.

Task 41: Ansatz for distancd d between pointd P and X: $d = \sqrt{(x_1 - p_1)^2 + (x_2 - p_2)^2}$; several data sets for point coordinates $P = (p_1, p_2)$ and corresponding distances d are given. Wanted: coordinates for $X = (x_1, x_2)$.

Data fitting with nonlinear models leads to an overdetermined nonlinear system of equations. The Gauss-Newton method solves this system iteratively; unlike the standard Newton method, the Jacobian matrix here is not quadratic; each iteration step solves an overdetermined linear system of equations.

Chapter L 5.1 in these materials presents a standard solution approach for nonlinear fitting. The following section L 5.3 introduces additional possibilities and presents MATLAB tools. You might find these tools easier and more intuitive to use than the solution approach from Chapter L 5.1.

L 5.3 MATLAB Tools for Fitting Functions to Data

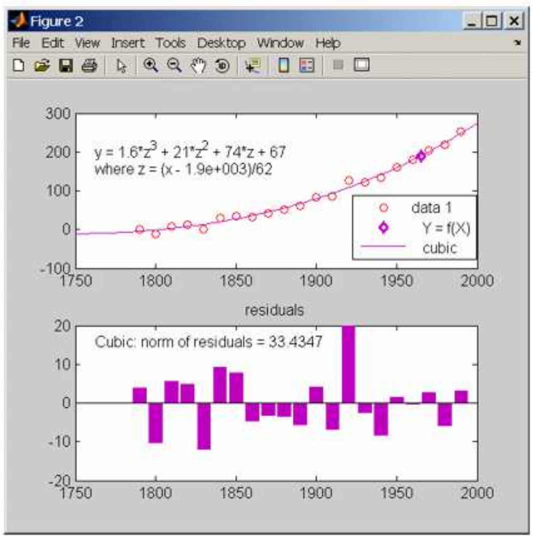
(Bonus material for interested readers: Compared to older versions, MATLAB now offers great new tools. Check them out, it's worth it!)

The MATLAB basic equipment includes the *Basic Fitting Tool*. You should definitely be familiar with it. Get acquainted with it right away: Find the corresponding instructions in the MATLAB Help (Version 2021b) under MATLAB › Data Import and Analysis › Descriptive Statistics › Interactive Fitting or (better, as this path may differ in older versions) directly search for the keyword *Interactive Fitting*.

There, you will find a worked example that explains the use of the *Basic Fitting Tool* for interactively fitting curves to data points.

Work through the example in the MATLAB Help and graphically represent the approximation, similar to the figure on the side.

In addition, MATLAB offers many tools for data fitting in the *Statistics and Machine Learning Toolbox* and the *Curve Fitting Toolbox*, far more than we can cover in these exercises. Tasks 44 and 46 exemplify the possibilities provided by the commands `cftool` and `polytool`. Task 47 mentions the command `robustfit`.



Task 44: Basic Fitting Tool compared to Curve Fitting Toolbox

Here, you solve the nonlinear least squares problem from Task 42 using MATLAB toolboxes.

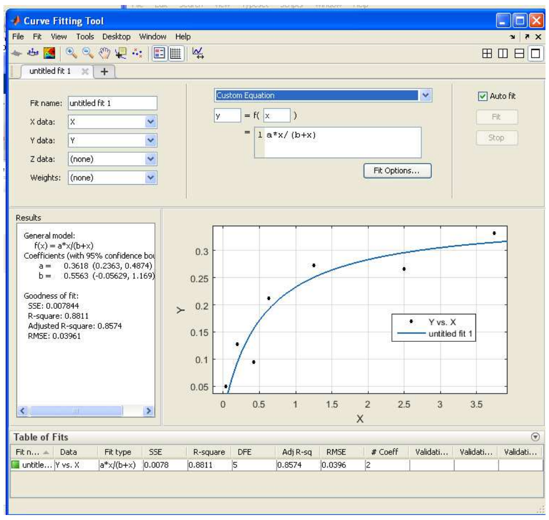
Use MATLAB's *basic fitting tool* to fit a straight line, a quadratic, and then a cubic polynomial through the following data points and plot the results.

```
X = [0.038 0.194 0.425  0.626  1.253  2.500  3.740];
Y = [0.05  0.127 0.094 0.2122 0.2729 0.2665 0.3317];
```

These are the data points from Task 42; there you will also find a figure with a well-fitted curve. Which other options of the *basic fitting tool* provide plausible curves? What do the error messages mean that you get from degree 7 onwards?

It turns out: No polynomial can satisfactorily approximate the data points. This is partly due to the fact that the data points are widely scattered due to measurement uncertainty, but also because in the measured experiment, the y values asymptotically approach a constant value as x increases. No polynomial can describe such asymptotic behavior.

The command `cftool` opens the *Curve Fitting App*. Here, you can fit not only polynomials but also other model functions. Try it out: Select the appropriate data vectors under *X Data* and *Y Data* in the upper-left corner from the variables present in the workspace. In the middle at the top, you can choose the function type. Try *Polynomial* first with different polynomial degrees. Then select *Custom Equation* and enter the function term `a*x/(b+x)`. (This is the model function from Task 42!) Your fit should look like the one shown here.



Fit name: untitled fit 1
X data: X
Y data: Y
Z data: (none)
Weights: (none)

Custom Equation
y = f(x)
= a*x/ (b+x)

General model:
f(x) = a*x/(b+x)
Coefficients (with 95% confidence bou
a = 0.3618 (0.2363, 0.4874)
b = 0.5563 (-0.05629, 1.169)

Goodness of fit:
SSE: 0.007844
R-square: 0.8811
Adjusted R-square: 0.8574
RMSE: 0.03961

Table of Fits

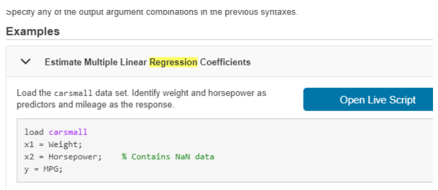
Fit n...	Data	Fit type	SSE	R-square	DFE	Adj R-sq	RMSE	# Coeff	Validati...	Validati...	Validati...
🟩 untitle...	Y vs. X	a*x/(b+x)	0.0078	0.8811	5	0.8574	0.0396	2			

In addition to the calculated values of the coefficients a and b, MATLAB also provides a confidence interval. Interpretation: If the data actually follow a model of the form $y = ax/(b+x)$, but the y data are randomly noisy, MATLAB calculates *estimates* $\hat{a}$ and $\hat{b}$ for the parameters a and b. For example:

```
Coefficients (with 95% confidence bounds):
       a =      0.3618  (0.2363, 0.4874)
       b =      0.5563  (-0.05629, 1.169)
```

The interpretation "The true values of a and b are within the calculated interval with 95% certainty" is not correct; the true values of a and b are either inside or not—there is no room for probabilities. It is the other way around: the calculated bounds of the confidence interval are uncertain. Each independent repetition of the measurement provides (for the same model parameters a and b) a new set of data points with different random errors. MATLAB's estimation method calculates confidence intervals with slightly different limits for each repetition. Most of the time (in 95% of cases), MATLAB estimates the error limits correctly, but in 5% of cases, the calculated confidence interval does not cover the true values.

Task 45: The `regress` Command from the Statistics Toolbox



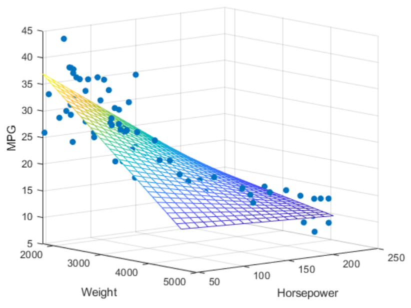
Specify any of the output argument combinations in the previous syntaxes.

Examples

Estimate Multiple Linear Regression Coefficients

Load the `carsmall` data set. Identify weight and horsepower as predictors and mileage as the response.

```
load carsmall
x1 = Weight;
x2 = Horsepower;    % Contains NaN data
y = MPG;
```



The MATLAB Help on the keyword "regress" shows an example with data on the weight, motor power, and fuel consumption of cars, finding a fitting function. Compare it with Task 39: there, a data model of the same kind is calculated, and the form of the matrix is the same.

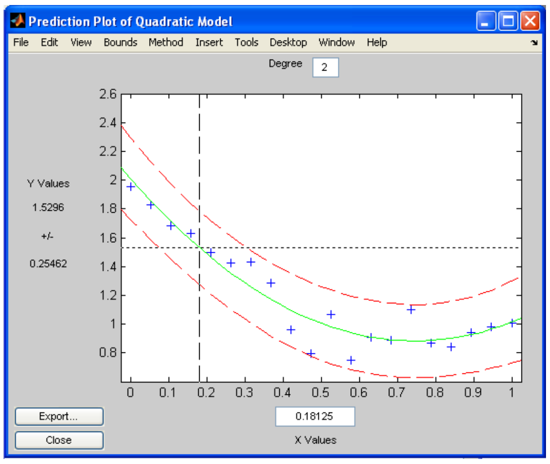
You can click on *Open Live Script* in the MATLAB Help, which will take you to the *Live Editor*, offering more possibilities than the standard editor. The file extension `*.mls` indicates Live Script files.
If you are curious, try out the Live Editor. If you want to stay in your familiar working environment, save this sample example using *save as* as a file type *MATLAB Code Files*, file extension `*.m`.

Orient yourself on this pattern and use the `regress` command to create a fit and a similar graph for the data from Task 39 (Diameter, Initial Temperature, Cooking Time) (with cooking time as the z-axis).

Task 46: The `polytool` Command from the Statistics Toolbox

The following MATLAB command lines generate data points along the function $y = 2-3x+2x^2$ that are perturbed by random normally distributed disturbances.

```
n=20;
x=linspace(0,1,n);
y=2 - 3*x + 2*x.^2 + 0.1*randn(1,n);
```


Prediction Plot of Quadratic Model
File Edit View Bounds Method Insert Tools Desktop Window Help
Degree 2
Y Values
1.5296
+/-
0.25462
Export...
Close
0.18125
X Values

The command `polytool(x,y)` opens a window similar to the basic fitting tool. Use it to create an image like the adjacent figure and explain:

- What do the different curves mean?
- What is the purpose of the movable crosshair, and what are the displayed X-Values and Y-Values?
- When you click on Export..., what do “Parameters” and “Parameters CI” mean?

L 5.4 Alternative Fitting Methods: Robust Regression, Total Least Squares

Task 47: Robust Regression

Linear regression using the method of least squares is sensitive to outliers in the data points. The following MATLAB commands generate a dataset of points that—except for an outlier—show an approximately linear trend.

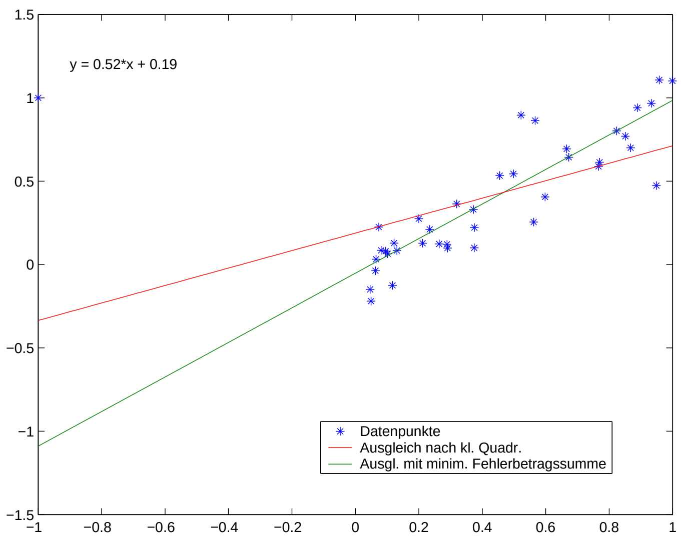
```
x = rand(40,1);
y = x + 0.2*randn(40,1);
x(1) = -1; y(1)=1;
```

The function ‘randn‘ generates normally distributed deviations!

Write a script file that generates such a dataset and perform the following steps:

1. Plot the data points using the symbol `'*'` in a diagram.
2. Chapter 6.3 of the lecture script describes how to determine the parameters a and b of the regression line $y = a + bx$ using the method of least squares. Implement these formulas in your script file and calculate a and b.
3. Use the tools in the “Tools–Basic Fitting” menu to plot a regression line. Display the formula in the diagram and compare it with the previously calculated values of a and b.
4. Minimizing the *squares* of errors is the standard procedure for regression. Why does it not make sense to minimize the *sum* of errors? Chapter 6.6 of the lecture script describes a regression method that minimizes the *absolute sum* of errors. There is a sample program file `linregrob.m` for this purpose. Plot the determined line in the diagram.

Your diagram should look something like this:



Task 48: Robust Regression with `robustfit`

By the way, the MATLAB help for the command `robustfit` from the Statistics Toolbox provides an example very similar to Exercise 47. The `robustfit` command offers more options compared to the file `linregrob.m`. If you're interested, you can also solve Exercise 47 with `robustfit` and compare the results with the `linregrob.m` results. Additionally, `cftool` from the *Curve Fitting Toolbox*, see Exercise 44, supports robust fitting: choose *Polynomial, Degree 1, Robust Bisquare* or *LAR*.

Task 49: Total Least Squares

The standard method for linear regression minimizes the sum of squared deviations *in the y-direction*. If only the y values in the data are affected by random, normally distributed errors, this method statistically provides an optimal estimate for the regression line.

However, if the x values are also subject to errors (simplest case: randomly, independently, normally distributed, mean 0, same standard deviation as the y values), it is more meaningful to minimize the sum of squares of the actual geometric distances between the line and the data points.

The *Total Least Squares* (TLS) method minimizes the sum of squares of the normal distances between the line and the data points. It uses singular value decomposition.

Procedure for data $x = [x_1 x_2 \dots x_n], y = [y_1 y_2, \dots y_n]$:

- Determine the *center of mass* $[\bar{x}, \bar{y}]$ of the data.

$$\bar{x} = \frac{1}{n} \sum_{i=1,n} x_i, \quad \bar{y} = \frac{1}{n} \sum_{i=1,n} y_i$$

- Shift the data

$$\Delta x_i = x_i - \bar{x}, \quad \Delta y_i = y_i - \bar{y}$$

- Perform singular value decomposition

$$U \cdot S \cdot V^T = \begin{bmatrix} \Delta x_1 & \Delta y_1 \\ \vdots & \vdots \\ \Delta x_n & \Delta y_n \end{bmatrix}$$

The corresponding MATLAB command is

```
>> [U S V]=svd([delx dely])
```

Enter delx and dely as column vectors!

- The TLS line passes through the *center of mass* $[\bar{x}, \bar{y}]$ in the direction of the first column vector of V and has the second column vector of V as its normal vector. The normal form of the equation of the line is $ax + by = c$. Here, $\begin{bmatrix} a \\ b \end{bmatrix}$ is the second column vector of V, and $c = a\bar{x} + b\bar{y}$ is obtained by substituting the center of mass into the equation of the line.

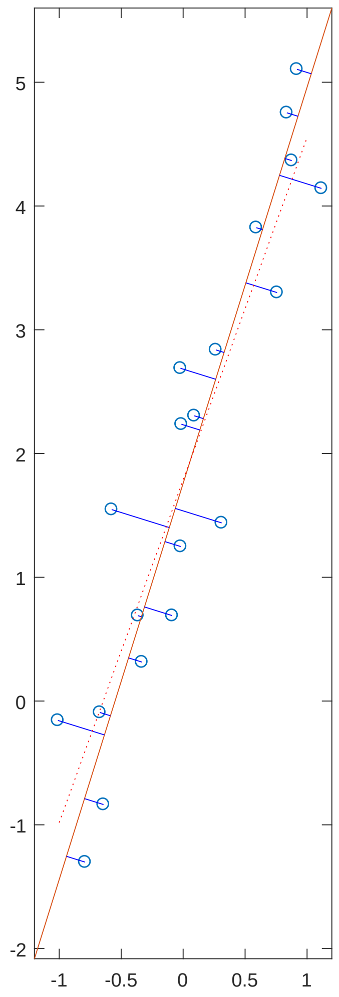
The Matlab commands

```
n=20;
x = linspace(-1,1,n)';
y = 3*x +2;
y = y + 0.2*randn(n,1);
x = x + 0.2*randn(n,1);
```

generate points on the line $y = 3x + 2$ with errors in the x and y directions. Find the regression line with the classical approach and with TLS.
Repeat the calculation several times (at least 10 times). You are testing different noisy datasets for the same relationship $y = 3x + 2$ and estimating the values of k and d in $y = kx + d$ from the error-prone data. Which method, on average, provides better estimates for the slope k (here 3) and the intercept d (here 2) of the line?

Your calculations should show: Estimates with classical least squares regression, on average, yield too small values for k, while TLS estimates, on average, are close to the correct value. It is said that the TLS estimator is *unbiased*, while the other estimator is *biased*. The choice of the correct method for fitting a regression line also depends on assumptions about the nature of the randomly distributed errors.

The figure on the right illustrates the idea of Total Least Squares approximation: The normal distances to the regression line (solid line) are drawn for the data points. The classical least squares regression line is drawn dashed. It underestimates the slope, while the TLS regression line provides a better estimate.



Task 50: TLS Planar Fit

When fitting a regression plane to data points $\in \mathbb{R}^3$, and the data in the x, y, and z directions are affected by random errors (assumption: independent normally distributed errors, mean 0, same standard deviation), then TLS fitting (optimizing normal distances to the regression plane) provides a better estimate than classical least squares fitting (optimizing errors in the z direction).

The procedure is analogous to the previous task. Procedure for data $x = [x_1 x_2 \ldots x_n], y = [y_1 y_2, \ldots y_n], z = [z_1 z_2, \ldots z_n]$:

- Determine the *center of mass* $[\bar{x}, \bar{y}, \bar{z}]$ of the data.
- Shift the coordinate origin to the center of mass

$$\Delta x_i = x_i - \bar{x}, \quad \Delta y_i = y_i - \bar{y}, \quad \Delta z_i = z_i - \bar{z}$$

- Perform singular value decomposition

$$U \cdot S \cdot V^T = \begin{bmatrix} \Delta x_1 & \Delta y_1 & \Delta z_1 \\ \vdots & \vdots & \vdots \\ \Delta x_n & \Delta y_n & \Delta z_n \end{bmatrix}$$

- The TLS fitting plane passes through the *center of mass* $[\bar{x}, \bar{y}, \bar{z}]$ and is spanned by the first two column vectors of V. It has the last column vector of V as its normal vector. The standard form of the plane equation is

$$ax + by + cz = d \ .$$

  Here, $\begin{bmatrix} a \\ b \\ c \end{bmatrix}$ is the last column vector of V, and $d = a\bar{x} + b\bar{y} + c\bar{z}$ is obtained by substituting the center of mass into the plane equation.

Calculate the TLS fitting plane for the data from Section L 4.4.3

```
%% Data points: Variables x1, x2, and measured value y
x1 = [.2 .5 .6 .8 1.0 1.1]';
x2 = [.1 .3 .4 .9 1.1 1.4]';
y  = [.17 .26 .28 .23 .27 .24]';
```

and compare it with the classical least squares fitting plane (solution from Section L 4.4.3).

L 6 Sixth Lab Unit

Content of the sixth exercise unit:

- Singular value decomposition, data compression
- Polynomial Interpolation
- Numerical Integration
- First Test: Sample questions

L 6.1 Singular Value Decomposition for Data Compression

The seventh Lecture 2024 contains material on Singular Value Decomposition and image compression. Here is a step-by-step guide. The task is as follows:

- Choose a nice photo
- Compress it
- Present the original and compressed versions side by side and compare the data sizes

The idea explained here, approximation through matrices with low rank, is used in signal processing, image processing, and the big data domain. (If you're interested: search Wikipedia for the keyword "Singular Value Decomposition - Applications".)

Task 51: How to compress a cat

No, we don't want to harm the cat. Therefore, choose a nice photo yourself. First, read the image file in MATLAB. For simplicity, we work in black and white.

The following lines of code perform a standard conversion from color to black and white.
It's best to copy-paste the code; you only need to replace the single quotes; and you should use your own image file here.

```
photoarray = imread('katze2.jpg');
redpix = single(photoarray(:,:,1))/255;
greenpix = single(photoarray(:,:,2))/255;
bluepix = single(photoarray(:,:,3))/255;
graypix = 0.299*redpix + 0.587*greenpix + 0.114*bluepix;
imagesc(graypix)
colormap(gray)
axis off image
```

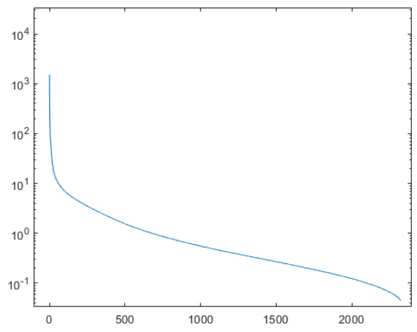
Now calculate the Singular Value Decomposition of the pixel matrix.

```
 [U,S,V] = svd(graypix);
```

The diagonal of the S-matrix contains the singular values. In this context (very simplified), large singular values belong to important data components. Small singular values belong to less important data material. Therefore, we plot the singular values here.

```
semilogy(diag(S))
```

In the semi-logarithmic representation, it is clearly visible: a drop by two orders of magnitude within the first 100 singular values, followed by a slower decay.
Depending on which image you are working with, your plot may look different. If this steep drop also occurs in your data, the chances are good for data compression.

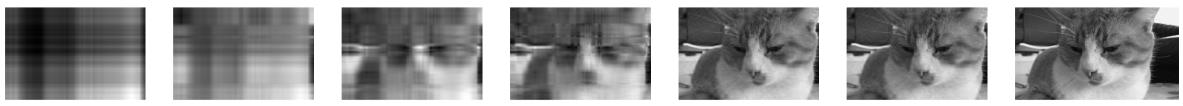


The recipe for SVD compression: keep only a few column vectors of U and V that belong to the largest singular values. From this, you can approximately reconstruct the original image using the recipe:

Column vectors of U times S times transposed columns of V

For n columns, this recipe yields a matrix with rank n. (One speaks of a "rank-n approximation". The Singular Value Decomposition provides the best possible rank-n approximation, measured in the 2-norm.) Let's try this for different n values and define a function that also displays the reconstructed image.

```
reconstruct = @(n) imagesc(U(:,1:n)*S(1:n,1:n)*V(:,1:n)');

colormap(gray)
reconstruct(1); axis off image
reconstruct(2); axis off image
reconstruct(4); axis off image
reconstruct(8); axis off image
...
reconstruct(64); axis off image
```



And so the little kitten becomes more visible...Choose an n for your image that provides satisfactory image quality. Discard the unnecessary U and V columns. How many bytes of storage do the data now require?

How much storage space does the original file need? Save the original grayscale image as a JPEG file. The following commands save the file and display file information.

```
imwrite(graypix, 'graysamp.jpg', 'jpg')
d = dir('graysamp.jpg'); d.bytes
```

Of course, the JPEG algorithm uses much smarter tricks, but in principle, it also looks for low-dimensional approximations of data matrices. Nonetheless, we get a comparable compression ratio with acceptable image quality.

L 6.2 Polynomial Interpolation

Learn more about this topic: Slides from the 6th lecture and Chapter 7 in the script. Feel free to also consult the Wikipedia article on Polynomial Interpolation.

Often, values of a function are given in tabular form, as in the following dataset:

T	cp
300	537.0
400	593.3
500	666.8
600	760.8

Specific heat capacity of low-carbon steel in J/kg K for temperatures between 300 and 600°C

Table 1: Sample dataset

Finding a Polynomial through Data Points. Possible approaches:

- Approach of the polynomial with undetermined coefficients: $p(x) = a + bx + cx^2 + dx^3$, substitute the data points, set up and solve the system of equations. See Task 52.
- Lagrange interpolation formula. Known from the Mathematics I lecture. Also covered in this script. Advantage: Polynomial can be directly written; Change of y-data is easily possible. Disadvantage: not the most favorable form for computer-assisted evaluation.
- The script explains two additional calculation schemes (Neville and Newton methods, divided differences). These methods are optimized for manual calculations and come from a time before modern computers. Manual calculation is not as important today.
- Interpolation in Newton form is also advantageous on the computer, especially in real-time applications, where computational speed is crucial. This form requires the fewest computational operations compared to other methods when programmed skillfully, see Task 55.

L 6.2.1 Approach with Different Polynomial Types: Standard Approach, Polynomials in Newton Form, Discrete Orthogonal Polynomials

Task 52: Interpolation: Approach with the Vandermonde Matrix

Determine the coefficients a, b, c, d of the cubic interpolation polynomial for the dataset in Table 1 using the standard approach $p(x) = a + bx + cx^2 + dx^3$.

Consider: Substituting the data points leads to a system of equations with the matrix

$$\begin{bmatrix} 1 & 300 & 300^2 & 300^3 \\ 1 & 400 & 400^2 & 400^3 \\ 1 & 500 & 500^2 & 500^3 \\ 1 & 600 & 600^2 & 600^3 \end{bmatrix}.$$

A matrix whose rows sequentially contain the powers of x values is called a *Vandermonde matrix*. The good news: If all x values are different, the system of equations is uniquely solvable.
The bad news: This matrix can have a very high condition number. For polynomials of higher degrees, serious rounding errors may occur. MATLAB's command `polyfit` uses this approach (with all its advantages and disadvantages).

Task 53: Interpolation: Approach with Polynomials in Newton Form

You should work through this task with pen and paper. It explains the important properties of the Newton interpolation method.

You can also set up the cubic polynomial in the following form:

$$p(x) = a_0 + a_1(x-300) + a_2(x-300)(x-400) + a_3(x-300)(x-400)(x-500)$$

Insert the value pairs from the (T, c_p) dataset in Table 1 here. Write down the associated coefficient matrix. A start has already been made:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 100 & 0 & 0 \\ 1 & 200 & \ddots & \\ 1 & 300 & \dots & \dots \end{bmatrix}.$$

Conveniently, many zeros occur here. What is the name for the special form of such a matrix?

Determine the coefficients a_0 to a_3. Why (with pen and paper) is solving the system significantly simpler than solving a Vandermonde system?

The computational effort to solve an $n \times n$ Vandermonde equation system grows with $O(n^3)$. With which power does the computational effort grow for n data points in the Newton approach?

Now, leave out the last term $a_3(x-300)(x-400)(x-500)$ and consider the quadratic polynomial

$$p(x) = a_0 + a_1(x-300) + a_2(x-300)(x-400)$$

Is this polynomial also a type of interpolation polynomial? What data can it describe?

And if you also leave out the a_2 term? What does

$$p(x) = a_0 + a_1(x-300) \quad ?$$

Now, you want to include an additional data pair in the interpolation:

```
T      cp
200    496.4
```

Add this data pair *at the bottom* of the data table and add another term to the Newton approach

$$a_4(x-300)(x-400)(x-500)(x-600)$$

What changes in the matrix of the equation system? What changes in the result, i.e., in the values of the coefficients $a_0, a_1, \dots$?

The individual terms in the Newton interpolation polynomial describe linear, quadratic, cubic, etc., interpolation between two, three, four, etc., data pairs.

Each additional data pair adds another term to the Newton interpolation polynomial; the other terms remain unchanged.

Compare this to the polynomial approach in standard form $p(x) = a_0 + a_1 x + a_2 x^2 + \dots$, as used in Task 52. In that case, the linear, quadratic, cubic, etc., interpolation polynomials would have completely different coefficients.

Task 54: Interpolation: Approach with Discrete Orthogonal Polynomials

Also for this task, you should perform the calculations with pen and paper. The goal here is for you to see that, in addition to the Lagrange and Newton approaches, there are other polynomial basis functions.

As a third approach, choose

$$p(x) = a_0 p_0 + a_1 p_1 + a_2 p_2 + a_3 p_3$$

with basis polynomials

$$\begin{aligned} p_0(x) &= 1, \\ p_1(x) &= \frac{1}{50}(x - 450), \\ p_2(x) &= \frac{1}{10\,000}\left[(x - 450)^2 - 12500\right], \\ p_3(x) &= \frac{1}{300\,000}(x - 450)\left[(x - 450)^2 - 20500\right] \end{aligned}$$

And how do you arrive at these polynomials? The rules for constructing the polynomials $p_0, p_1, \dots$ are not particularly complicated, but for the learning effect in this task, they are not so important. Take the polynomials as given. The task aims to make you aware of the following:

- Besides the standard basis $1, x, x^2, x^3, \dots$, there are other, often more favorable, basis polynomials to represent the space of all polynomial functions.
- Orthogonality is an important property of vectors; this property can be transferred to polynomials.

Task 57 uses the sample programs `orth_polyfit.m` and `orth_polyval.m` for the construction of such polynomials, but they do not go into detail on how this works. However, what you can see in the sample programs is that only a few lines of code are needed to construct these polynomials, no more than for the Newton or Vandermonde approach.

Insert the data pairs from the (T, c_p) dataset in Table 1. Write down the corresponding coefficient matrix A. It looks harmless and uninteresting:

$$A = \begin{bmatrix} 1 & -3 & 1 & \dots \\ 1 & -1 & & \\ 1 & & \ddots & \\ \vdots & & & \end{bmatrix}.$$

But it has an important property: $A^T \cdot A$ is a diagonal matrix. In other words, the column vectors of A are orthogonal to each other. Thus, A almost fulfills the definition of an orthogonal matrix – What important property is missing?

The quasi-orthogonality of A simplifies the solution of the system for the coefficients: Set up A; calculate $A^T \cdot A$ and $A^T\mathbf{b}$. What simple step is still missing for the solution?

The basis polynomials p_0, p_1, p_2, p_3 are chosen here so that their values for the x data vector (which are precisely the column vectors in the A matrix) are orthogonal to each other. Polynomials with such a property are called *orthogonal polynomials*. Specifically, these are *discrete orthogonal polynomials* because orthogonality refers to the discrete data points on the x-axis.

If you are now asking: Are there also polynomials or functions that are orthogonal not only for some discrete x values but for all x in a certain range? Yes, indeed! The term ”orthogonality” can be transferred from vectors to functions.

The important property with an approach using orthogonal polynomials is: If you leave out the last term, you get a quadratic polynomial. Unlike the Newton approach, it no longer interpolates data points, but it is the best approximation polynomial (in the least-squares sense) to the data.

Similarly, if you leave out the penultimate term, with $p(x) = a_0p_0(x) + a_1p_1(x)$ you get the best linear polynomial (the least squares fit) to the data.

The individual terms in an approach with orthogonal polynomials describe in sequence the best possible linear, quadratic, cubic, etc., approximation of the data pairs. Ultimately, with the same number of terms as data pairs, the interpolation polynomial is obtained.

Task 55: Interpolation: Newton’s Divided-Difference Scheme

This task is aimed at people who are inclined towards programming and want to know how to implement interpolation methods as computationally efficiently as possible. As long as you have MATLAB as a computing environment, you don’t need to deal with this. But here you will learn clever programming tricks that can be used elsewhere.

Specifically, this is about efficient computation of tables, where intermediate results are overwritten as soon as they are no longer needed to save memory space.

Page 65 in the script shows a triangular table for the cubic interpolation polynomial according to Newton.

In many numerical methods, tables need to be filled in triangular form. Not all entries are needed in the end. Therefore, the computation can be cleverly organized so that only one column of the table with currently needed values is stored. The newly calculated entries then overwrite no longer necessary intermediate results. In the end, a vector with the required final results remains.

The following code segment calculates, from data vectors x and y, the vector dd as the upper diagonal of the divided-difference scheme:

```
dd = y;
for i=2:n
    for k=n:-1:i
        dd(k) = (dd(k)-dd(k-1))/(x(k)-x(k-i+1));
    end
end
```

Implement this code as a function `dd=newtonpoly(x,y)` and Verify for the worked example in the script (Page 65) or for the data from the previous task in a single run in debug mode that all intermediate results are calculated in dd and only then overwritten when they are no longer necessary for further computation.

The computationally inefficient evaluation of an interpolation polynomial in Newton’s form at the point z, when the coefficients from the divided-difference scheme are stored in a vector dd and the x support points in x, is accomplished by a loop of the form

```
y = dd(n);
for i=n-1:-1:1
    y = y.*(z-x(i)) + dd(i);
end
```

Implement and test a function `y = newtonval(x, dd, z)`, which, for given x data points and divided-difference coefficients dd, evaluates the polynomial at the point z.

Compare the application of the functions `newtonpoly` and `newtonval` with the MATLAB commands `polyfit` and `polyval`. MATLAB uses the Vandermonde approach; for the data from the script, there should be no difference.

L 6.2.2 Quality of Interpolation. Warning about High Degree. Runge's Phenomenon

Task 56: Runge's Phenomenon, Chebyshev Nodes

In the pre-computer era, no one would easily think of using interpolation polynomials of the tenth or twentieth degree. Today, a simple MATLAB command is sufficient to use polynomials of the hundredth degree. But just because it's easy, is it also reasonable? This task explores the question.

Compute interpolation polynomials for the function

$$y = \frac{1}{1+x^2}, \qquad -5 \leq x \leq 5$$

at 5, 10, 15, and 20 equidistant support points. The easiest way is to use MATLAB's `polyfit, polyval` or the code from Task 55. Draw the function and polynomials. Interpret the graph, also experiment with higher polynomial degrees.

Read up on the lecture slides and Wikipedia under the keyword "Runge's Phenomenon" [33] and then answer: Does polynomial interpolation automatically imply: "the higher the degree, the better the approximation?"

The Weierstrass approximation theorem (Karl Weierstrass, 1885) states that any continuous function on a closed interval can be approximated arbitrarily closely by polynomials. Why do the approximations get worse in this example? Did Weierstrass make a mistake? – Of course not, Weierstrass was a mathematician. The problem in this case is the equidistant support points. Most functions (apart from completely harmless sin, cos, exp functions) resist being forced into equidistant support points. It is possible, of course, but the function reacts stubbornly and deviates significantly from the intended course between the support points. See Runge's Phenomenon.

Especially towards the edge of the x range, there are strong oscillations. Therefore, it is better to place the x values denser towards the edge. Particularly favorable in the interval $[-5;\ 5]$ is the arrangement of support points according to the formula

$$x_i = 5\cos\left(\frac{2i-1}{2n}\pi\right), \quad i = 1, \ldots, n \qquad \text{(Chebyshev Nodes)}.$$

Interpolate the function again with 5, 10, 15, and 20 support points, but they are not equidistantly distributed, but according to the above formula. Experiment with higher polynomial

[33] Regarding pronunciation: Carl Runge (1856–1927) was a German mathematician, his name rhymes with "Junge," not with the music style "Grunge".

degrees as well and then answer: Does polynomial interpolation perhaps imply: "the higher the degree, the better the approximation?"

Task 57: Approximation, Very High Polynomial Degree

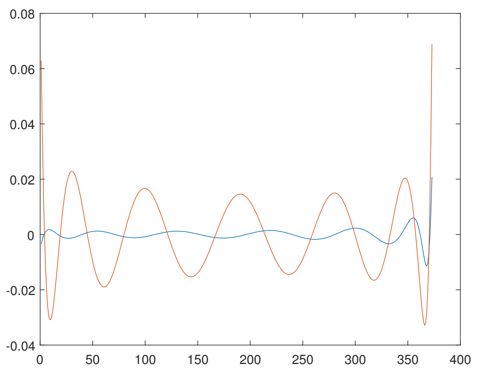
Even though it has been emphasized so far that polynomials of high degree should be used with great caution, there are applications (e.g. image processing, noise filtering of data, etc.) where approximations with high degree make sense. Here we examine, using a simple data set, which numerical methods cope well with a high degree. The file `SatPressH2O.m` (download from exercise homepage) contains real data for the saturation vapor pressure $p(T)$ of water as a function of temperature T.

Suppose you want to find a highly accurate approximation polynomial for these data[34].

Compare three methods:

1. Classical approach with Vandermonde matrix and forming the normal equations
2. QR decomposition of the Vandermonde matrix – done by the MATLAB commands `polyfit, polyval`
3. Approach with discrete orthogonal polynomials. There are files `orth_polyfit.m` and `orth_polyval.m` for download; these files work analogously to `polyfit, polyval` but use discrete orthogonal polynomials.

Start with the approximation by a polynomial of the 10th degree and plot the error in the data points (data value - polynomial value). Increase the polynomial degree step by step and determine if and how much the error decreases. Depending on the method, meaningful approximations can only be found up to a certain degree. Up to which degree can the method of normal equations, the QR decomposition, and the orthogonal polynomials be numerically stable?



Here is an example plot: It shows, for a polynomial degree of 12, the error of the approximation polynomial from the QR decomposition (MATLAB commands `polyfit, polyval`) in blue, and the corresponding error with the method of normal equations in orange. It is significantly larger, which is due only to rounding errors. At degree 11, there is hardly any difference. So,

[34]For professional use, the data should be suitably transformed and scaled; it also practically makes little sense to demand accuracy in the microbar or nanocelsius range; no one can measure temperatures or pressures with such precision in practical use. But it's about the principle; the results are typical for other functions or datasets as well.

you can forget about normal equations for this dataset from degree 12 onwards. How about the other two methods at higher degrees?

L 6.3 Numerical Integration

See the slides of the 6th and 7th lecture. In the script under Kapitel 8.

L 6.3.1 Integration of Tabulated Values

If only discrete data points of a function are available, its integral can be approximated using various numerical quadrature formulas.

Here are the tabulated values for the specific heat capacity of low-carbon steel.

T	cp
0	460.8
100	471.1
200	496.4
300	537.0
400	593.3
500	666.8
600	760.8

Specific heat capacity of low-carbon steel in J/kg K for temperatures between 0 and 600C

Task 58: Integral of Specific Heat Capacity

The integral of specific heat capacity is the enthalpy. Calculate, for the given data, the integral from $T = 0$ to $T = 600$ using the following methods:

- Simpson's rule (3 data points at $T = 0, 300, 600$)
- 3/8 rule (4 data points at $T = 0, 200, 400, 600$)
- Composite trapezoidal rule ($h = 100$)
- Composite Simpson's rule ($h = 100$)

Note: In MATLAB, the multiplication of function values by weighting factors can be elegantly expressed as the scalar product of two vectors. As an example, using the 3/8 rule:

```
b=600;
a=0;
f= [460.8  496.4  593.3  760.8];
w=[1 3 3 1]/8;
int38=(b-a)*f*w'
```

L 6.3.2 Numerical Integration of Functions

When a function is not integrable in closed form, or the expression becomes too complicated, numerical integration (also known as *numerical quadrature*) is possible. The MATLAB command `q = quad(fun,a,b)` calculates the integral of f in the range from a to b using an adaptive Simpson's rule. The function f can be given *inline*,

```
>> q=quad('1./x',1,2)
q =
    0.6931
>>
```

Note: Element-wise vector operations ./, .^, etc. are used.

as an anonymous function or as a function handle, just like when using `fzero()`.

Task 59: Agnesi's Curve

On May 16th, the 304th birthday of Maria Gaetana Agnesi, an Italian mathematician who studied curves of the form

$$y = \frac{a^3}{x^2 + a^2}$$

is celebrated. In her honor, you are to numerically integrate this curve. Numerically calculate the integral

$$\int_{-1}^{1} \frac{1}{x^2+1} dx \qquad \text{(exact value: } \frac{\pi}{2}\text{)}$$

(a) using the composite trapezoidal rule with three different step sizes: $h = \frac{4}{10}, \frac{2}{10}, \frac{1}{10}$. Compare with the exact value $\frac{\pi}{2}$ and indicate: with half the step size, the error is reduced by a factor of ...

(b) with MATLAB's `quad` command

The Romberg method (Task 60) is not only interesting as a precise numerical integration method but also illustrates a general principle: calculate an approximate value multiple times with different step sizes $h_1, h_2, \dots$, extrapolate from this data to $h = 0$. This general idea is called *Richardson extrapolation.* Its application to the composite trapezoidal rule is the Romberg method.

Task 60: Agnesi's Curve with Romberg Integration

Romberg method: Denote the three results from point a of the previous task (trapezoidal rule with halved step size) as $T_{1,1}, T_{1,2}, T_{1,3}$. Compute a triangular array according to the scheme

$$\begin{array}{ccc} T_{1,1} & & \\ & T_{2,2} & \\ T_{1,2} & & T_{3,3} \\ & T_{2,3} & \\ T_{1,3} & & \end{array}$$

using the formula

$$T_{2,2} = \frac{4T_{1,2} - T_{1,1}}{3} \quad , \quad T_{2,3} = \frac{4T_{1,3} - T_{1,2}}{3} \text{ and } , \quad T_{3,3} = \frac{16T_{2,3} - T_{2,2}}{15}$$

The results become increasingly accurate. Compare with the exact result!

And now the general case (bonus task): the formula for the Romberg method with k results $T_{1,1}, T_{1,2}, \dots T_{1,k}$ from the trapezoidal rule with halved step size is:

$$T_{i+1,j} = \frac{4^i T_{i,j} - T_{i,j-1}}{4^i - 1}$$

Here, the index i refers to the columns $1, 2, \ldots k$ of the triangular array, index j denotes the upward-sloping rows.

$$\begin{array}{ccccc}
T_{1,1} & & & & \\
 & T_{2,2} & & & \\
T_{1,2} & & T_{3,3} & & \\
 & T_{2,3} & \vdots & \ldots & T_{k,k} \\
T_{1,3} & \vdots & T_{3,k} & & \\
\vdots & T_{2,k} & & & \\
T_{1,k} & & & &
\end{array}$$

You can calculate this triangular array in a memory-efficient way in a similar manner to the divided differences of the Newton scheme in Task 55 – if you manage to do that, it would be an extra bonus!

The $T_{i,j}$ values approximate with increasing accuracy; when values no longer change (within an error bound), it can be assumed that a sufficiently accurate result has been achieved. Adaptive quadrature formulas essentially work on this principle.

L 6.4 Analytical Integration of Functions

Bonus material for interested readers. MATLAB can symbolically integrate, not only differentiate. It's good to know, and we invite you to go through this section.

While there are fixed rules for differentiation that lead step by step to the result, analytical (also called symbolic) integration is much more complicated. There are integration rules only for relatively simple functions; beyond that, there is a collection of tricks and "recipes". Often, it cannot be foreseen in advance whether a given function has an antiderivative and which method will yield a result.

Formerly, extensive integral tables were used for reference; now, computer algebra systems take on this task. They are now very powerful and can calculate integrals for many types of functions. The relevant command in MATLAB's *Symbolic Math Toolbox* is `int()`.

To warm up:

```
>> syms x                                    Define x as a symbolic variable.
```

With that, you can refresh your mathematics knowledge: What is $\int x^2\,dx$?

```
>> int(x^2)
ans =
1/3*x^3
```

Did MATLAB give the correct answer? To understand this, let me tell an old joke…

> Two mathematicians are sitting at the bar. The first one laments that the average person knows little about basic mathematics. The second one disagrees and claims that most people can cope with a reasonable amount of math.
>
> The first mathematician goes off to the washroom and the second calls over the waitress in his absence. He tells her that in a few minutes, after his friend has returned, he will call her over and ask her a question. All she has to do is answer "one-third x cubed". She repeats "one thir – dex cue?" He repeats patiently: "One-third x cubed." She says, "one thir dex cuebd?" Yes, that's right, he says. So she agrees and goes off mumbling to herself, "one thir dex cuebd…"

The first guy returns, and the second proposes a bet to prove his point, that most people do know something about basic math. He says he will ask the blonde waitress an integral, and the first laughingly agrees. The second man calls over the waitress and asks "what is the integral of x squared?"

The waitress says, "one-third x cubed," and while walking away, turns back, shouting over her shoulder, "Plus C!"

Try more integrals:

```
>> syms x n t
```

Define a few more symbolic variables.

```
>> int(x^n)
ans =
piecewise([n = -1, log(x)], [n <> -1, x^(n + 1)/(n + 1)])
>>
```

It looks complicated, but it is MATLAB's correct interpretation of the rule

$$\int x^n\,dx = \begin{cases} \log x & (n=-1) \\ \frac{x^{n+1}}{n+1} & (n \neq -1) \end{cases} + C$$

```
>> int(sin(t))
ans =
-cos(t)
>>
```

Common functions can be integrated...

```
>> int(exp(-x^2))
ans =
(pi^(1/2)*erf(x))/2
>>
```

...although it can quickly become complicated. Here, the integral cannot be expressed using elementary functions. However, there is the special function erf (created specifically for integrals of this type).

```
>> int(cos(n*t))
ans =
1/n*sin(n*t)
>>
```

Did MATLAB integrate here with respect to n oder t?

MATLAB tries to guess the integration variable. It assumes that letters at the end of the alphabet are variables, letters further ahead are rather parameters or constants. It therefore interpret `int(cos(n*t))` as $\int \cos(nt)\,dt$ and not as $\int \cos(nt)\,dn$. In case of doubt, `int()` accepts a second argument:

```
>> int(log(x)*cos(t),x)
ans =
x*cos(t)*(log(x) - 1)
>>
```

That's $\int \log x \cos t\,dx$

```
>>int(log(x)*cos(t),t)
ans =
log(x)*sin(t)
>>
```

That's $\int \log x \cos t\,dt$

More features: `int(S,a,b)` gives the definite integral of S over the interval $[a, b]$. The command `int(S,v,a,b)` also specifies v as the variable in the integral.

Try MATLAB's symbolic integration with the following examples!q

$$\int \frac{1}{1+x^2}\,dx \qquad \int \sqrt{1-ax^2}\,dx$$

$$\int e^{-tx^2}\,dt \qquad \int_1^2 \frac{1}{x}\,dx$$

L 6.5 First Test: Sample Questions

The first test consists of four subtasks randomly chosen from the following topics:

- Drawing curves and surfaces. Tasks 27–30 from the 3rd exercise are of this type.
- Solvability and solution sets of linear systems of equations. The sample test already includes several variations for this topic. Task 31 from the 4th exercise is also typical. This should be sufficient; these materials do not provide additional sample tasks for this.
- Nonlinear systems of equations. (Similar to tasks 23–25 from the 3rd exercise.)
- Data fitting, linear and nonlinear models. (Similar to tasks 38–43 from the 4th and 5th exercises; using various MATLAB toolboxes is not required.)

Since the sample test in the Moodle does not cover all types of tasks, there are additional sample tasks here. No submission in the Moodle course is required for these.

Solving Nonlinear System

Solve the following nonlinear system of equations using the Newton method. Start vector: $[1; -1; 0]$. Also, solve it using the MATLAB function `fsolve`.

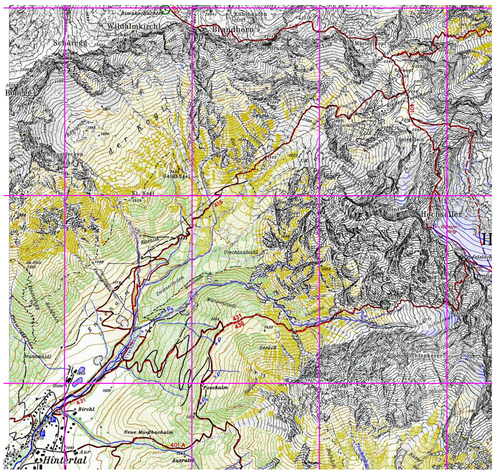
$$\begin{aligned} x^2 + \sin^2 y + z &= 1 \\ e^x + e^{-x} - yz &= 2 \\ x + y + z^2 &= 0 \end{aligned}$$

Drawing Curves and Surfaces

The first equation of the above system defines a surface in the xyz-space. Draw this surface in the range $x \in [-1; 1]$ and $y \in [-1; 1]$. Also, create a contour plot. Read the maximum z-value in this range from the plot.

The second equation can be transformed into $z = f(x, y)$. Draw a contour plot of f in the range $x \in [0; 2]$ and $y \in [-2; 0]$.

Adapting Linear Data Models: Georeferencing



Download the data and a sample program from here. The MATLAB commands

```
imdata = imread('HohTs.jpg');
image(imdata)
```

read an image file and display it in the *figure* window.
GPS data (geographical longitude and latitude) should be marked on this image. For this, you need an assignment from pixel coordinates in the image to geographical coordinates.

The assignment

$$\text{Image Coordinates} \longleftrightarrow \text{World Coordinates}$$

is called *image registration*. It uses *control points*: pixel points in the image with known world coordinates, here geographical longitude λ and latitude ϕ. Data for 9 control points are included in the sample program.

Find an adjustment of the form

$$x = a_0 + a_1\lambda + a_2\phi$$
$$y = b_0 + b_1\lambda + b_2\phi$$

Image Coord		World Coord	
x	y	λ	ϕ
369	2299	12.983	47.417
1143	2299	13.000	47.417
1917	2299	13.017	47.417
$\vdots$	$\vdots$	$\vdots$	$\vdots$

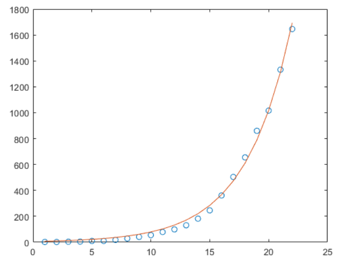
What is the maximum deviation of the image coordinates x, y between the model and original data? (Ideally, the deviation should not exceed 1–2 pixels. However, the world coordinates here were rounded to one decimal place too much.)

The MATLAB command `load trackpnts.mat;` loads a long list of $\lambda\phi$ coordinates of a GPS track. Convert these coordinates to pixel xy coordinates and plot them.

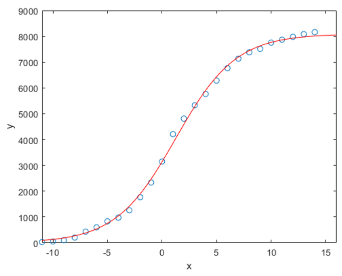
Adapting Nonlinear Data Models

Data points and model equations are given, parameters are sought. For the data and a sample program, see `GaussNewtonCoronaFit.m`

$$y = ae^{bx}$$
$$a, b = ?$$



$$y = \frac{a}{1 + e^{-b(x-c)}}$$
$$a, b, c = ?$$



(See also the lecture slides of the 6th lecture)

L 7 Seventh Lab Unit

Content of the seventh exercise unit:

- Eigenvalue Problems
  - Intuitive Explanation: `eigshow`
  - Vector Iteration, QR Iteration
  - Some Applications
- Numerical Methods for First Order Ordinary Differential Equations
  - MATLAB Solvers
  - Classical Euler Method and Other Simple Explicit Methods

L 7.1 Eigenvalues and Eigenvectors

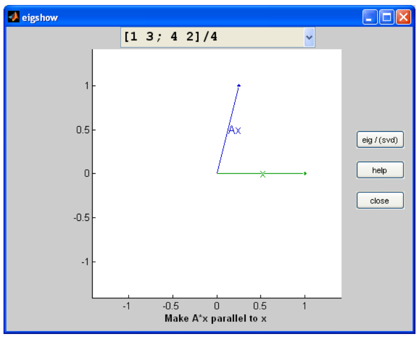
What you need in terms of theoretical foundations for this unit is summarized in Section 9.1 (click!) in the current script. Also refer to the slides of the 8th lecture (click!) for more information. Find definitions and basic properties there! The exercise tasks establish the connection to typical applications.

MATLAB Commands:

`d = eig(A)` yields vector of eigenvalues

`[V,D] = eig(A)` Columns of V are eigenvectors, diagonal elements of D are eigenvalues.

An interactive and educational example regarding eigenvectors is provided by the MATLAB demo program `eigshow`. Download the materials for the 7th exercise (click!) and start `eigshow.m`. Tip: At `http://blogs.mathworks.com/cleve/2013/07/08/eigshow-week-1`, Cleve Moler, one of the MathWorks founders, personally explains what eigshow is about.



Move the vector $\mathbf{x}$ and observe how $A\mathbf{x}$ changes accordingly. Rotate vector $\mathbf{x}$ so that it aligns with $A\mathbf{x}$. Any such vector $\mathbf{x}$ is an *eigenvector* of matrix A.
In this case, the result $A\mathbf{x}$ is a multiple of the initial vector, thus satisfying the equation

$$A\mathbf{x} = \lambda\mathbf{x} \ .$$

The proportionality factor λ is the *eigenvalue* of A corresponding to $\mathbf{x}$.

You can choose matrix A from a list in the window above. The default is

$$A = \frac{1}{4}\begin{bmatrix} 1 & 3 \\ 4 & 2 \end{bmatrix} \ .$$

- For the above matrix A, find eigenvectors. Read off the components of $\mathbf{x}$ and estimate λ. How many different values of λ are there?
- Compare with the result of the command `[V,D] = eig(A)`.
- Try other matrices from the list. You cannot always find eigenvectors. There are examples where $\mathbf{x}$ and $\mathbf{Ax}$ do not lie on a line in *any* direction. What does `[V,D] = eig(A)` provide in such cases?

- In the normal case (for a regular matrix A), $A\mathbf{x}$ describes an ellipse as you move $\mathbf{x}$. Singular matrices are the exception. They have $\lambda = 0$ as an eigenvalue. What does the path of $A\mathbf{x}$ look like then? Find an example matrix from the list.

Task 61: Stress Tensor, Principal Axis Transformation

This task tests whether you can call MATLAB commands and interpret their results.

Pure compressive or tensile forces act normal to a surface element, pure shear forces act parallel to it. In an elastically deformed body, the force on a surface element can act in some skewed manner. The stress tensor establishes the relationship between surface normal direction and force direction:

$$\boxed{\text{Force Vector} = \text{Stress Tensor times Normal Vector of the Surface Element}}$$

In a suitably rotated coordinate system, the stress tensor assumes diagonal form. The axes of this coordinate system are the *principal axes* of the stress tensor. They point in the direction of its eigenvectors. The diagonalized stress tensor contains the eigenvalues on the main diagonal.

The stress tensor in a material is (in arbitrary units)

$$p = \begin{bmatrix} 5 & 10 & -8 \\ 10 & 2 & 2 \\ -8 & 2 & 11 \end{bmatrix}.$$

a) Which force vector acts on a surface element parallel to the xy-plane? — to the plane $x + y + z = 0$? (You don't need eigenvalues for this.)

b) In which directions do the principal axes of the stress tensor point (`format rat` provides "nice" numbers)? What is the tensor transformed into diagonal form?

c) Pure tensile stress acts in two directions, pure compressive stress in one (sign convention for the diagonal terms of the stress tensor: compression is negative[35]. In which direction does
— the greater tensile stress act?
— the smaller tensile stress act?
— the compressive stress act?

[35]as it is in life: exam pressure, performance pressure, success pressure, …

Task 62: Power Method (also: power iteration, vector iteration)

What happens when you repeatedly multiply a vector by the same matrix? Test for the vector $x = (1, 0, 0)^T$ and the matrix A

$$A = \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}$$

When you iterate the command `x = A*x` ten times, the numbers in x become quite large. You are actually computing $A^{10}\mathbf{x}$, the tenth power of A times `x`. That's where the name "power method" comes from. Another name for this method is "vector iteration".

Now, modify the procedure: divide the vector x by its first component after multiplication by A in each step. This does not change the direction of x, but keeps the numerical values of the components within reasonable bounds. You can go through this using `>> y=A*x; x = y/y(1)` in the workspace. What do you observe? What value does y_1 converge to?

Starting with the initial vector $x = (1, 0, 0)^T$,

```
>> y=A*x; x = y/y(1)
```

What value does y_1 converge to?

Compare: For this matrix, the lecture notes (click!) on page 76 calculate the eigenvalues from the characteristic polynomial.

In the lecture notes on page 79, power iteration (in a somewhat more general form) is given as pseudocode. It is also considered that simple scaling `x = y/y(1)` does not work if $y(1) = 0$.

The example matrix $A = \begin{bmatrix} -1 & 0 \\ 1 & 2 \end{bmatrix}$ in the script on page 75 is such a case where scaling with the first component is not helpful. Do the calculations as above, then you will see why.

A computer program scales best with the component that has the largest absolute value. The MATLAB command `[m,i]=max(abs(x))` finds the index i of this component in vector x.
A simple MATLAB implementation would be:

```
for k=1:100
    x1 = A*x0;
    [m,i]=max(abs(x1));
    lam = x1(i);

    x1 = x1/lam;
    if(norm(lam - lam0)<eps)
        break
    end
    x0=x1;
    lam0=lam;
end
```

Specific Task Write a script that generates a 10×10 matrix A with randomly chosen integer elements $a_{ij} \in \{0, \dots, 9\}$ [36].

Calculate the largest eigenvalue and a corresponding eigenvector using vector iteration. Choose a termination criterion with accuracy $\epsilon < 10^{-6}$.

Also, let MATLAB's `eig` command calculate the eigenvalue and eigenvector and compare the results. Note that MATLAB calculates eigenvectors as unit vectors in the 2-norm, while the vector iteration shown above scales to 1 in the infinity norm. To compare the eigenvectors, you need to scale MATLAB's eigenvector the same way as the one calculated from the vector iteration!

Task 63: A Top-10 Algorithm: Eigenvalues from QR Iteration

Eigenvalue problems for 2×2 or 3×3 matrices have encountered in the principal axis transformation of the stress tensor (compare task 61). In addition, until about 1920, from the perspective of technical-scientific applications, there was no particular motivation to calculate eigenvalues of larger matrices. But then the importance of eigenvalue problems dramatically increased: quantum theory, matrix mechanics, finite element methods, and, most recently, data science came into play.

The classical approach via the characteristic polynomial is unsuitable for larger matrices, among other things, because the roots can be extremely sensitive to small disturbances in the polynomial coefficients (compare exercise example 13).

Around 1960, the QR algorithm emerged, making it into the top 10 algorithms of greatest influence on science and technology in the 20th century[37][38]. What also excites the experts is: it is actually a completely new approach and not just (as we have often encountered here) a refined elaboration of the ideas of Gauss or Newton.

You have already encountered the QR decomposition of a matrix when solving overdetermined systems of equations. (Compare slides for the 5th lecture). So far, the materials have only referred to MATLAB's command `[Q R]=qr(A)` for computational execution. But the QR decomposition can be implemented in a few lines of code. This should also be addressed in this task. (Compare exercise task 32, which deals with the code lines for LU decomposition).

This task allows you to experiment with the basic version of the QR algorithm. In its simplest form, it decomposes a matrix into the product of an orthogonal and an upper triangular matrix, $A = Q \cdot R$. It then multiplies the factors in reverse order. These two steps are iterated.

Specific Task Write a script that generates a *symmetric*[39] 10×10 matrix A with randomly chosen integer elements $a_{ij} \in \{0, \dots, 8\}$ using a simple approach:

```
A = randi(5,10)-1;
A=A+A';
```

[36] We use random matrices here only as test problems for the vector iteration algorithm. This is by no means the most interesting application. In a random conversation in Princeton in 1972, the mathematician Hugh Montgomery and the physicist Freeman Dyson discovered a surprising connection between zeros of the Riemann zeta function, eigenvalues of random matrices, and quantum physical systems *(Montgomery's pair correlation conjecture)*

[37] J. Dongarra and F. Sullivan, "Guest Editors Introduction to the top 10 algorithms" in Computing in Science & Engineering, vol. 2, no. 01, pp. 22-23, 2000.

[38] B. Parlett, "The QR Algorithm" in Computing in Science & Engineering, vol. 2, no. 01, pp. 38-42, 2000.

[39] The QR algorithm can also handle nonsymmetric matrices; our basic version works better for symmetric matrices.

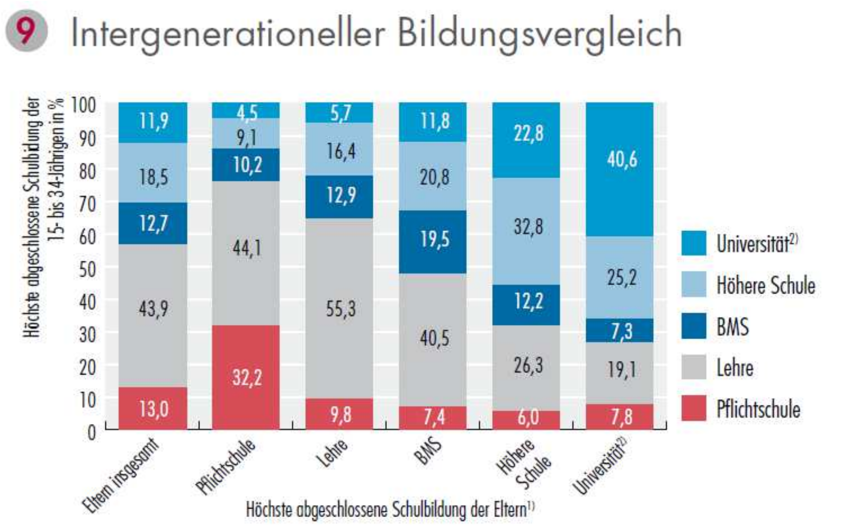
Let MATLAB determine the eigenvalues of A; store the result vector. Now repeat the following steps 100 times:

```
[Q, R]=qr(A);
A=R*Q;
```

- For several attempts with different random matrices, examine the structure of the resulting matrix: where do values approximately close to zero occur? Where do they mostly occur?
- Compare the diagonal elements of the resulting matrix with the eigenvalues of the original matrix. What can you observe?
- Replace MATLAB's `qr` command with the call to `myQR` (in the material for exercise 7). This is a simple implementation of the QR decomposition. Check: This should not change the statements about the two previous points.

Task 64: Intergenerational Educational Mobility

The level of education that children and young people receive depends heavily on their social background. The graph[40] shows data from Austria.



Q: STATISTIK AUSTRIA, Mikrozens.-Arbeitskräfteerhebung Ad-hoc-Modul „Eintritt junger Menschen in den Arbeitsmarkt" – 2. Quart. 2009. Bevölkerung in Privathaushalten (15- bis 34-Jährige). – Pers. nicht in Ausbild. – 1) Höchste abgeschl. Schulbildung d. Eltern: Haben Mutter und Vater nicht denselben Ausbildungsabschluss, wird jeweils die höhere Ausbildung verwendet. – 2) Inkl. hochschulverwandte Lehranstalten.

Create a matrix $A = [a_{ij}]$ from this data, where the element a_{ij} indicates:

> Of children whose parents have education level j, the proportion a_{ij} attain education level i.

[40]Source: *Education in Numbers 2010/11, Key Indicators and Analyses.* Statistics Austria, Vienna, 2012

Example: Children from academic families ($j = 5$) attain an apprenticeship ($i = 2$) as the highest level of education with 19.1% probability; therefore, $a_{25} = 0.191$.

The bar on the far left in the graph represents the current state (education level of young people overall). Put this data into a vector $\mathbf{x}^{(1)}$. Assume that the overall education level of parents is described by a vector $\mathbf{x}^{(0)}$ (this data cannot be directly read from the graph).

a) Justify: The overall education level of the next generation is calculated by matrix-vector multiplication $\mathbf{x}^{(1)} = A \cdot \mathbf{x}^{(0)}$.

b) Calculate $\mathbf{x}^{(0)}$ from the given data for A and $\mathbf{x}^{(1)}$.

c) Calculate, starting from $\mathbf{x}^{(1)}$, the state after 1, 2, and 3 more generations;

d) A *stable state* exists when nothing changes from one generation to the next. Justify: this is an eigenvector of A. Calculate the stable state.

The calculation of the stable state can be performed using vector iteration (Task 62) or with MATLAB's `eig` command.

Note: The percentages in the graph are rounded, so not all columns sum up to exactly 100%, resulting in small inaccuracies.

Task 65: Vibration Equation

The eigenmodes and corresponding frequencies of a vibrating string can be approximately determined from the eigenvalue problem

$$A\mathbf{x} = \lambda\mathbf{x}$$

One imagines the string divided into n individual masses, like beads on a string. For large n, the bead string approximates a continuous mass distribution. The $n \times n$ matrix A has a simple tridiagonal form.

$$A = \begin{bmatrix} 2 & -1 & 0 & 0 & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & 0 \\ 0 & -1 & 2 & -1 & \dots & 0 \\ \vdots & & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & -1 & 2 & -1 \\ 0 & \dots & & 0 & -1 & 2 \end{bmatrix}.$$

If the string is tensioned with length ℓ, total mass m, and force τ, then the eigenfrequency ν (in Hz) associated with the eigenvalue λ of this matrix is approximately determined by:

$$\nu = \frac{n+1}{2\pi}\sqrt{\lambda\frac{\tau}{m\ell}}$$

The smallest eigenvalue of A corresponds to the frequency of the fundamental vibration, while the other eigenvalues correspond to overtones. In practice, the fundamental vibration and several low-order overtones are relevant. The larger n is, the more accurate the frequencies calculated from the "bead-string approximation" are.

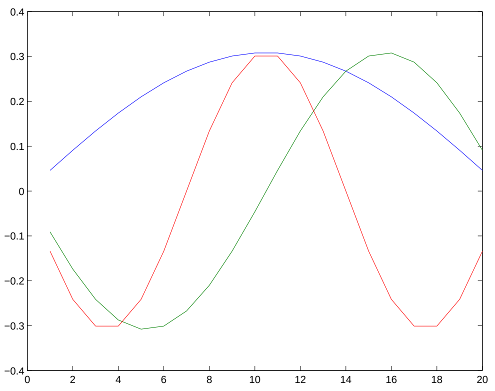
In this simple case, eigenfrequencies and vibration modes can also be specified by exact formulas. However, slightly more general problems, such as non-uniform mass distribution along the string, are not analytically solvable. The analysis of the vibration behavior of components is a typical application for numerical simulation.

See also lecture slide 21 of the 8th lecture, which shows a bead-string model of a freely swinging chain.

Your Task: For the E-string of a concert guitar, we have: free length $\ell = 65\,\mathrm{cm}$, mass $m = 3.58\,\mathrm{g}$, tension force $\tau = 63.11\,\mathrm{N}$. With these data, the frequency equation above becomes

$$\nu = 164.7\frac{n+1}{2\pi}\sqrt{\lambda}$$

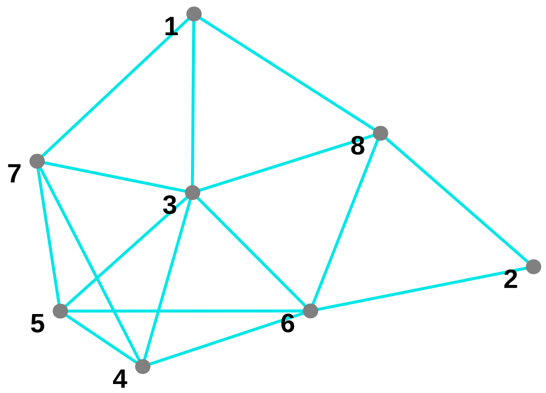
For $n = 5, 10, 20$, create the matrix A and calculate the frequencies of the fundamental vibration. You should observe that the calculated approximations converge rapidly. (However, for an accuracy of two decimal places, you would need to choose $n = 80$!)



For $n = 20$, plot the eigenvectors corresponding to the lowest three eigenvalues; they represent the corresponding vibration modes of the string (fundamental vibration, first and second overtone). See the figure!

Task 66: Accessibility in a Network

A network (traffic connections, linked pages on the internet, social networks..., mathematically: a graph) can be described by its adjacency matrix.



$$A = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ \vdots & & & & & & & \vdots \end{bmatrix}$$

(a) For the network shown here, construct the (01)-adjacency matrix A (see Lecture 8) and calculate an eigenvector for the largest eigenvalue. According to the eigenvector data, which node in the network is the best connected, and which one is the worst?

(b) Given a vector $\mathbf{x}^{(0)}$ consisting of all ones. What measure of "connectivity" or "accessibility" is defined by $\mathbf{x}^{(1)} = A\mathbf{x}^{(0)}$?

(c) Starting with the vector $\mathbf{x}^{(0)}$ from (b), perform ten steps of the power method (vector iteration). What do you approximately obtain?

The idea of mathematically modeling links between web pages in this way was explored by a doctoral student (a certain Larry Page) at Stanford University in 1997. Although he has not yet completed his dissertation, if you're interested, Google what became of him and his research project...

The slides for Lecture 8 show a similar network. For a practical example, refer to Wikipedia under the keyword Page Rank. In the materials for Exercise 7, you will find the file `pagerank.m`, which performs the calculations outlined in the Wikipedia article.

L 7.2 Ordinary Differential Equations of First Order

Task:
Explicit ordinary differential equation of 1st order with initial condition
Given a function $f(x, y)$. We seek a function $y(x)$. It should satisfy

$y'(x) = f(x, y(x))$	Differential equation
$y(x_0) = y_0$	Initial condition

The slides for Lecture 9 (click!) show the geometric interpretation of this problem as a direction field in $\mathbb{R}^2$ with solution functions following the direction field.

In the current exercise materials, you will find the MATLAB files `BeispieleGDG.m` and `GDGDemo.m`. Download these script files. They are presented in a clear and readable format if you use MATLAB's Publish function or open the files as live scripts!

The file `BeispieleGDG.m` initially shows the symbolic solution of differential equations as well. We focus on the numerical solution because useful symbolic solutions exist only in simple cases.

L 7.2.1 MATLAB Solvers for Ordinary Differential Equations

MATLAB provides ready-made solvers for initial value problems.

There are seven solvers available: `ode23`, `ode23s`, `ode23t`, `ode23tb`, `ode45`, `ode113`, `ode15s`. You can read a more detailed description in the MATLAB help.

The function `ode45` is the standard method that you would try first for a "new" initial value problem. It is the implementation of an explicit single-step Runge-Kutta method (error order 5 and error control with order 4).

Here's a small example demonstrating the use of `ode45`. Suppose you want to solve the initial value problem for $y = y(x)$:

$$y' = g(x, y) = 2x(1 + y^2), \quad y(0) = 0,$$

in the interval $[0, 1]$ using this solver. You save the function g in a file called `g.m`:

```
function ydot=g(x,y)
ydot=2*x*(1+y^2);
end
```

and then you put together the solution of the initial value problem in a script file (`test.m`):

```
y0=0.0;
xspan=[0, 1];
ode45(@g,xspan,y0)
```

It's that simple! You get a graphical representation of the solution. (Even shorter is the implementation in the sample file `BeispieleGDG.m`!)

You get vectors of x and y values instead of graphical output if you use the following syntax:

```
[X, Y] = ode45(@g,xspan,y0);
```

However, MATLAB chooses the location of the x values itself (to keep the errors below a certain threshold). If you want to compute the solution numerically for specific x values chosen by you, you write, for example:

```
sol=ode45(@g,xspan,y0);
x = [0 0.25 0.5 0.75 1];
y = deval(sol,x)
```

Task 67:

Compute numerically with MATLAB for the following given functions $y = y(x)$ the solution of the initial value problem. Graphically represent the solution.

a) $y' = y$ with $y(0) = 1$, solution sought in the range $0 \le x \le 2$.

This is the classic differential equation of the exponential function, exact solution is $y = e^x$. Also, represent (in a separate diagram) the error between numerical and exact solution.

b) $y' = xy/4 - 1$ with three different initial conditions: $y(0) = 2;\ 2.5;\ 3$, solutions sought in the range $0 \le x \le 4$.

Compare: in the script solutions of this equation are depicted in the direction field.

c) $y' = \dfrac{1}{x^2}$ with $y(-1) = 1$, solutions sought in the range $-1 \le x \le 1$. Check: The function $y = -\dfrac{1}{x}$ fulfills the differential equation and initial condition within its domain. However, in the numerical solution, you have to be prepared for problems. The function y' has a special property here, which does not allow a solution throughout the interval $-1 \le x \le 1$ – what property is that? Compare with the numerical solution.

L 7.2.2 Explicit Single-step (Runge-Kutta-Type) Methods

In the numerical solution of an ordinary differential equation, one determines, starting from the initial conditions, a sequence of value pairs $(x_0, y_0), (x_1, y_1), (x_2, y_2), \ldots$, which should approximate the values $y(x)$of the sought function y. Scheme:

```
Choose step size $h$ and maximum number of steps $N$;
set $x_0$ and $y_0$ according to initial conditions;
for $i = 0, 1, \ldots, N-1$
    $x_{i+1} = x_i + h$ ;
    $y_{i+1} = y_i + hF(x_i, y_i, h)$ .
```

The function $F(x, y, h)$ is called the *method function* of the respective method. Geometrically interpreted, $F(x, y, h)$ gives the *progress direction*. The different methods differ in the method function, that is, in the definition of the progress direction. Only in the explicit Euler method is the progress direction at the point $(x; y)$ also equal to the slope $y'(x, y)$, that is

$$F(x, y, h) = f(x, y),$$

Other method functions take some intermediate steps to better adapt the progress direction to the course of the solution. The slides of the 8th lecture depict various single-step (actually, Runge-Kutta-type)[41]

For the modified Euler method, we have

$$F(x,y,h) = f\left(x + \frac{h}{2}, y + \frac{h}{2} f(x,y)\right),$$

for the Heun's method

$$F(x,y,h) = \frac{1}{2}(k_1 + k_2)$$

with

$$\begin{aligned} k_1 &= f(x,y) \\ k_2 &= f(x+h, y + hf(x,y)). \end{aligned}$$

There is a sample program `GDGdemo.m` available for download!

Tip: Let Cleve Moler, one of the MathWorks founders, explain to you personally how the Runge-Kutta method computes the growth of a flame!

L 7.2.3 Manual Calculation and Simple Programs for Understanding

"I only understand a numerical method when I have miscalculated it myself."

The numerical solution of a differential equation by manual calculation nowadays serves only to illustrate the computational methods. The practical implementation is left to the computer. However, you must demonstrate your understanding of the computational methods, at least during the lecture exam, by working out some calculation steps on paper.

The slides of the 9th lecture (click!) show in detail the individual calculation steps of the methods you are supposed to work out here. The sample program `GDGdemo.m` implements the calculation steps in MATLAB.

An example is the initial value problem for the function $y(x)$

$$\begin{aligned} y' &= 4xy + 3 \\ y(0) &= 0 \end{aligned}$$

In this case, we have $f(x,y) = 4xy + 3$ and $x_0 = y_0 = 0$. We want to determine the value of the function $y(x)$ approximately at the points $x_1 = 0.2$; $x_2 = 0.4$; $x_3 = 0.6$ with a step size of $h = 0.2$. The exact solution is known, but not given by elementary functions; to six decimal places, $y(0.6) = 2.973\,414$.

A tabular calculation scheme is helpful. Note: The penultimate column always computes the method function, the last column corresponds to the general step $y_{i+1} = y_i + hF(x_i, y_i, h)$, in simple words: "new y-value = old value plus h times progress direction F"

[41] The nomenclature here corresponds to the one used mostly in German textboks (*Explizite Einschrittverfahren*); in the English literature, the methods treated here would be seen as special cases of *linear multistep methods*. However, in German, *Mehrschrittverfahren* have a different meaning. Referring to the methods treated here as *Runge-Kutta-type methods* would conform to both English and German usage.

For the explicit Euler method:

i	x_i	y_i	$F = f(x, y) = 4xy + 3$	$y + hF$
0				
1				
2				

Modified Euler method:

i	x_i	y_i	$k_1 = f(x, y)$	$y + \frac{h}{2}k_1$	$F = f(x + \frac{h}{2}, y + \frac{h}{2}k_1)$	$y + hF$
0						
1						
2						

Method Heun:

x_i	y_i	$k_1 = f(x, y)$	$y + hk_1$	$k_2 = f(x + h, y + hk_1)$	$F = (k_1 + k_2)/2$	$y + hF$

Task 68: Programming Simple One-Step Methods, Recognizing Order of Error

Use the sample program `GDGdemo.m` and calculate for the example given above the value $y(0.6)$ with different step sizes and different methods:

1. Compute with $h = 0.2$ and the three methods mentioned above. (Compare with the values from manual calculation.) Also, implement the three-stage method

$$\begin{aligned} k_1 &= f(x, y) \\ k_2 &= f(x + \frac{h}{2}, y + \frac{h}{2}k_1) \\ k_3 &= f(x + h, y - hk_1 + 2hk_2) \\ F &= \frac{1}{6}(k_1 + 4k_2 + k_3) \end{aligned}$$

2. Now compute with a finer step size $h = 0.05$. What is the error in each case (difference between the approximation and the exact solution $y(0.6) = 2.973414$?

3. Halve the step size, so compute with $h = 0.025$ and compare the errors. By what factor has the error been reduced in each case?

The various MATLAB solvers for initial value problems work essentially like the demo program `GDGdemo.m`, but

- the method function determines the progression direction from several optimally chosen intermediate points,
- the (global) error is estimated,
- the step size is automatically adjusted.
- for *stiff problems*, there are methods (`ode23s, ode15s`) with better stability behavior.

L 8 Eighth Lab Unit

Content of the eighth exercise unit:

- Setting options in MATLAB's `ode45`
- Stability, implicit methods
- Systems of first-order differential equations

L 8.1 Ordinary Differential Equations of First Order (Continuation of Lab Unit 7.2)

Task 69: Setting Error Bounds

MATLAB's `ode45` command solves many simple problems satisfactorily with the default settings. However, sometimes the default options are not sufficient. You can make extensive adjustments. This exercise shows you how to influence the computational accuracy.

Consider the initial value problem for $y = y(x)$:

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

with

$$f(x, y) = -200xy^2, \quad y(-3) = \frac{1}{901}.$$

Compute using the MATLAB solver `ode45` the solution for $x = -2, -1, 0$.

Error bounds: Use the commands

```
options=odeset('reltol',1.e-8,'abstol',1.e-8);
sol=ode45(@g,xspan,y0,options)
```

to change the given error bounds. Also, try error bounds $10^{-10}, 10^{-12}, 10^{-14}$. Calculate the value $y(0)$ in each case.

For this problem, the exact solution can be easily stated:

$$y_{\text{exact}}(x) = \frac{1}{100x^2 + 1}$$

Compare the numerically computed values $y(0)$ with the exact solution $y_{\text{exact}}(0) = 1$. Calculate the errors (relative and absolute errors are equal here!). Are the desired error limits reached?

L 8.1.1 Stability of a Method; Implicit Methods

Computational methods should determine the behavior of the solution, if not exactly, at least *qualitatively* correctly.

Specifically, if the exact solution is an exponentially decaying function, then the numerical solution should also decay. Unfortunately, when the step size is too large, the *explicit* methods presented so far calculate exponentially *growing* solutions, which is pretty much the opposite of what should happen.

A method is called *stable* with step size h if, for the model problem

$$y' = -y \qquad y(0) = 1$$

the numerical solution converges to zero as x increases.

Now you will learn about *implicit* methods; these make errors of similar magnitude to explicit methods at each individual computation step. However, with explicit methods, errors accumulate exponentially from one step to the next; with implicit methods, errors decrease in sum and do not disrupt the qualitative behavior of the solution.

Task 70: Stability of Explicit One-Step Methods

Modify the template program `GDGdemo.m` and test the one-step methods discussed so far (explicit Euler, modified Euler, Heun, and the three-step method from Exercise 68) for different step sizes h for the initial value problem for $y(x)$

$$\begin{aligned} y' &= -y \\ y(0) &= 1 \end{aligned}$$

Compute up to $x = 10$, and find for each method h values for which the method computes just decaying or slightly swelling solutions – these are h values just within or just outside the stability limit.

Compare the illustrations on the penultimate lecture slide, 9th lecture!

Task 71: Implicit Euler Method

Here, the method function is given by

$$F(x, y, h) = f(x + h, y(x + h))$$

and the new approximation value is therefore

$$y(x + h) = y(x) + hf(x + h, y(x + h))$$

Unlike the explicit method, the sought value cannot be found directly: You can only calculate $y(x + h)$ on the left side if you already know it, because you have to substitute it on the right side of the equation. It's a bit of a circular argument...

Mathematically speaking, it is an *implicit equation* for $y(x + h)$. It can be easily solved for our test problem with the simple function $f(x, y) = -y$.

Refer to the template program `GDGdemo.m` and implement the implicit Euler method for the test problem

$$\begin{aligned} y' &= -y \\ y(0) &= 1 \end{aligned}$$

Examine the stability of this method. Are there step sizes h for which the numerical solution swells?

L 8.2 Systems of First-Order Differential Equations

L 8.2.1 Example: SIR Model for the Spread of an Epidemic

Background Information

The SIR model (susceptible-infected-removed model) describes, in a highly simplified form, the spread of infectious diseases with immunity formation.

The model divides a total population into three groups, S, I, and R. The first group, S for susceptible, contains susceptible individuals: they are not infected but can be infected at any time. Group I consists of the infected. In the third group are all those who can no longer infect themselves or others. It would be nice if we could say R for recovered, but R stands for removed because the model combines recovered and deceased individuals in this group.

The creators of this model, biochemist William Ogilvy Kermack and military physician Anderson Gray McKendrick, were able to model the data from a plague epidemic in Bombay 1905/06 very well despite the simplicity of the model. Just over 100 years later, in 2014, Harko and co-authors published an exact analytical solution to this system of differential equations. However, in most cases, these equations are solved numerically, and that's what we'll do here.

You can easily find detailed information about this model in the current situation. For now, it is sufficient to know that it is formulated as a system of three first-order differential equations:

$$\begin{aligned}\frac{dS}{dt} &= -\frac{\beta IS}{N}\\ \frac{dI}{dt} &= \frac{\beta IS}{N} - \gamma I\\ \frac{dR}{dt} &= \gamma I\end{aligned}$$

The functions $S(t)$, $I(t)$, and $R(t)$ describe the size of the respective groups as a function of time t.

The total population size $N = S(t) + I(t) + R(t)$ remains constant in it. The general demographic development, i.e., births and deaths not caused by the epidemic, is not modeled. However, this could be taken into account with a few additional terms.

The parameter β indicates how many close contacts an individual has on average per day (with only one contact between an S and an I individual leading to an infection; contacts of type S-S, S-R, I-I, and I-R remain without consequences).

The parameter γ is easier to interpret in the form of its reciprocal as

$$\frac{1}{\gamma} \quad \ldots \text{ Typical duration of the infection.}$$

The basis reproduction number

$$R_0 = \frac{\beta}{\gamma} \qquad .$$

This value indicates how many more people an infected person infects on average during the initial phase of the epidemic (when hardly anyone in the general population is yet immune).

Numerical Solution in MATLAB

There is a ready-made template program for this, `SIRmodel.m`, but the essential lines of program code are so short that we will present them here alongside the system of differential equations:

System of equations

$$\begin{aligned}\frac{dS}{dt} &= -\frac{\beta I S}{N}\\ \frac{dI}{dt} &= \frac{\beta I S}{N} - \gamma I\\ \frac{dR}{dt} &= \gamma I\end{aligned}$$

$0 \le t \le 100$

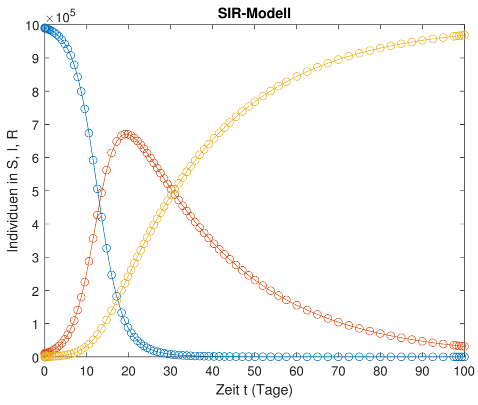
$S(0) = 999, \quad I(0) = 1, \quad R(0) = 0.$

Implementation

```
f = @(t,y)[
    - beta*y(2)*y(1)/N;
      beta*y(2)*y(1)/N - gamma*y(2);
      gamma*y(2) ];
```

```
ode45(f,[0 100],[999, 1,0])
```

The call to `ode45` in its simplest form, with parameters $\beta = 0.4$, $\gamma = 0.04$, $N = 10000000$, yields the adjacent figure.



Task 72: Exercise on the SIR Model

Apply an SIR model with plausible parameters to the situation in Austria in the summer and autumn of 2021. Set the start time $t = 0$ to May 17; at that time, $I(0) \approx 10000$, $S(0) \approx 6400000$ (8 million minus 1 million vaccinated and 600000 recovered), $R(0) = N - S(0) - I(0)$. Choose $\gamma = 1/14$, which corresponds to an average duration of the infectious phase of 14 days. Select various values for β so that the effective reproduction number $R_{\text{eff}} = \beta/\gamma \cdot S/N$ is between ≈ 0.8 (the value in May 2021) and 1.4 (the value reached by R_{eff} in January 2022).

Draw the course of the I and R variables for several R_{eff} scenarios for the next few months (excluding S, as it would not be visible on the scale of the plot: the S group is so large that the I and R components would be barely visible).

You can compare this simple SIR model with the actual course: In May and June 2021, $R_{\text{eff}} < 1$, resulting in a steady decline in I during this period. From July to November, $R_{\text{eff}} \approx 1.2$. During this period, a new wave built up, peaking in mid-November with $I \approx 150000$ and only being stopped by a lockdown.

L 8.2.2 Self-programmed One-Step Methods

The equations of motion for position and velocity and a simple force law are as follows:

Time derivative of position = velocity
Time derivative of velocity = acceleration
Acceleration is proportional to force
Restoring force proportional to displacement

If the time-dependent functions $y_1(t)$ and $y_2(t)$ denote the position and velocity, respectively, then the mathematical formulation of the above relationships in the simplest case is

$$\begin{aligned} \dot{y}_1(t) &= y_2(t) \\ \dot{y}_2(t) &= -y_1(t) \end{aligned}$$

This is a *system of two first-order differential equations.* The good news: Both self-programmed one-step methods following the pattern of the program `GDGdemo.m` and MATLAB solvers work - with few and obvious changes - also for systems.

Change 1 You formulate the right-hand side of the system as a vector-valued function. Specifically, in `GDGdemo.m`, you set

```
    function ystrich = f(x,y)
        ystrich = [ y(2)
                   -y(1)];
    end
```

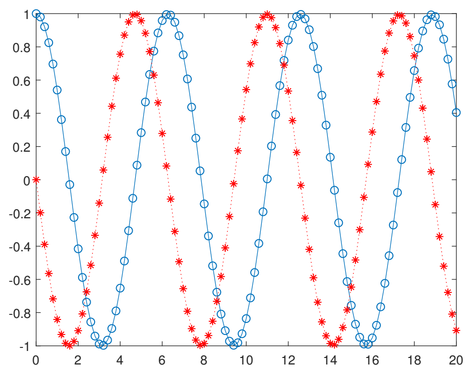
Change 2 Since it is a *two* first-order differential equations system, you also need two initial conditions; specifically initial position $y_1(0) = 1$ and initial velocity $y_2(0) = 0$. Set in `GDGdemo.m`

```
%% Set initial values, x-interval, and step size
x = 0;
y = [1;0]
% End value of x and step size
x_end = 20;
h = 0.2;
```

Change 3 The result `y` is a two-row matrix; the first row contains the computed approximate solution for the position $y_1(t)$, and the second row contains the corresponding approximations to the velocity $y_2(t)$. You plot both functions in one plot, for example, as follows:

```
plot(xLsg, yLsg(1,:),'-o',xLsg, yLsg(2,:),'-or' );
```

Your plot should look like this when `GDGdemo.m` calculates with the three-step method from Exercise 68 (the other methods yield similar plots):



MATLAB Solvers

You can adopt the function `ystrich` from above. Setting initial values and calling the solver is done in the simplest form as follows: y0=[1; 0]; xspan=[0, 20]; ode45(@f,xspan,y0)

`ode45` generates a plot similar to the one above.

If you use the other possible variants of the `ode45` command, [X, Y] = ode45(@f,xspan,y0); or sol=ode45(@g,xspan,y0); x = linspace(0,20); y = deval(sol,x) you can control the form and representation of the result specifically.

Task 73: Population Dynamics

A mathematical model that represents the development of the population of two species ("predator" and "prey") in a simplified manner is expressed by the system of nonlinear differential equations

$$\dot{y}_1(t) = k_1 y_1(t) - k_2 y_1(t) y_2(t)$$
$$\dot{y}_2(t) = k_3 y_1(t) y_2(t) - k_4 y_2(t)$$

where $y_1(t)$ and $y_2(t)$ represent the temporal population of prey and predators, respectively.

Solve this system for $0 \leq t \leq 4$ assuming that the initial population of prey is 1000, predators is 200, and the constants are $k_1 = 3, k_2 = 0.002, k_3 = 0.0006$, and $k_4 = 0.5$. Method: `ode45`. To illustrate the results, graphically represent the solution, i.e., the number of both populations versus time.

Repeat the calculation for an initial population of 2000 prey and 1500 predators.

Task 74: Butterfly Effect

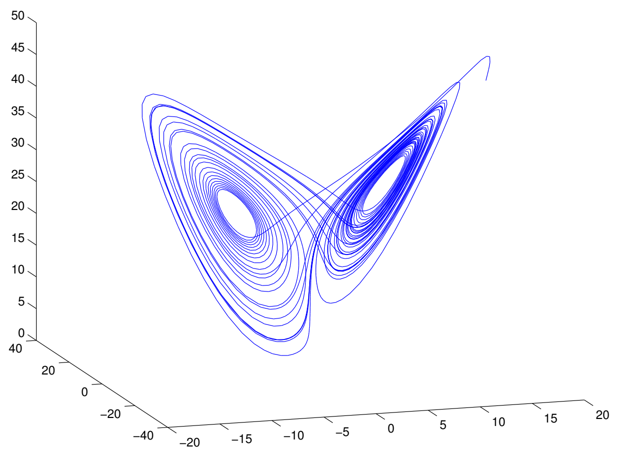
The meteorologist Edward N. Lorenz formulated in 1963 a system of three coupled, nonlinear ordinary differential equations to model the Earth's atmosphere for long-term forecasting purposes. The position of an air particle in three-dimensional space as a function of time is determined by the coordinates $x(t), y(t), z(t)$. In Lorenz's model, they obey the equations of motion.

$$\begin{aligned} dx/dt &= a(y-x) \\ dy/dt &= x(b-z)-y \\ dz/dt &= xy-cz \end{aligned}$$

The numerical solution of the system exhibits chaotic behavior for certain parameter values. The typical parameter setting with chaotic solution is:

$$a = 10, \quad b = 28, \quad c = \frac{8}{3}$$

Solve this system for $0 \leq t \leq 50$, initial conditions $x = 20, y = 20, z = 40$. Represent the solution as a curve in parametric form in an xyz coordinate system (using the `plot3` command).



The solution is known for its typical butterfly shape.

L 9 Ninth Lab Unit

Content of the ninth exercise unit:

- A higher-order differential equation: Transforming into 1st order system
- Stability: further examples
- Systems of higher-order differential equations: Transforming into 1st order systems

L 9.1 Higher-order Differential Equations

Higher-order differential equations can be written as systems of first-order differential equations.

Simple example: We are looking for a function $y(x)$. Its second derivative should satisfy

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

with initial conditions (here x_0, c_1, and $c_2 \in \mathbb{R}$ are given values)

$$\begin{aligned} y(x_0) &= c_1 \\ y'(x_0) &= c_2 \end{aligned}$$

It holds: From **one second-order** differential equation, a system of **two first-order** differential equations is formed.

Introduce two auxiliary functions $z_1(x)$ and $z_2(x)$, defined by

$$\begin{aligned} z_1 &= y \\ z_2 &= y' \end{aligned}$$

Then the corresponding system reads

$$\begin{aligned} z_1' &= z_2 \\ z_2' &= f(x, z_1, z_2) \end{aligned}$$

The initial conditions of the system are

$$\begin{aligned} z_1(x_0) &= c_1 \\ z_2(x_0) &= c_2 \end{aligned}$$

General Scheme: Higher-order DEs $\longrightarrow$ System of 1st Order DEs

From a d-th order differential equation, a system of d first-order differential equations is formed.

Given a d-th order differential equation. Make the d-th derivative explicit in the form $y^{(d)} = f(x, y, y', \dots, y^{(d-1)})$

Introduce d auxiliary functions $z_1(x), z_2(x), \dots, z_d(x)$. Set $z_1 = y,\ z_2 = y', \dots,\ z_d = y^{(d-1)}$.

Attention: For the d-th derivative, no auxiliary function is needed anymore!

Write a system of d first-order differential equations for the $z_i = z_i(x)$:

$$\begin{aligned} z_1' &= z_2 \\ z_2' &= z_3 \\ &\vdots \\ z_{d-1}' &= z_d \\ z_d' &= f(x, z_1, \dots, z_d) \end{aligned}$$

The d initial conditions of the original problem determine the values for $y(x_0), y'(x_0), \dots, y^{(d-1)}(x_0)$. From these, d initial conditions for $z_1, z_2, \dots, z_d$ are obtained:

$$\begin{aligned} z_1(x_0) &= y(x_0) \\ z_2(x_0) &= y'(x_0) \\ &\vdots \\ z_d(x_0) &= y^{(d-1)}(x_0) \end{aligned}$$

Task 75: Mathematical Pendulum

The differential equation of the mathematical pendulum (displacement $\phi = \phi(t)$, acceleration due to gravity g, pendulum length ℓ) is

$$\ddot{\phi} + \frac{g}{\ell} \sin \phi = 0$$

Transform it into a system of two first-order differential equations and solve it for $\ell = 1\,\mathrm{m}$, $g = 9.8\,\mathrm{m/s^2}$, initial displacement $\phi(0) = 0.1\,\mathrm{rad}$, initial angular velocity $\dot{\phi}(0) = 0\,\mathrm{rad/s}$. Solve it for a time span of 4 seconds and plot the oscillation (displacement ϕ as a function of t). Change the initial displacement to $\phi(0) = 1.4\,\mathrm{rad}$ and compare the oscillation shape and period.

Task 76: Example: Blasius Equation

The Blasius equation for the function $u = u(x)$ describes laminar flow in a near-wall boundary layer.

$$uu'' + 2u''' = 0$$

Initial conditions are

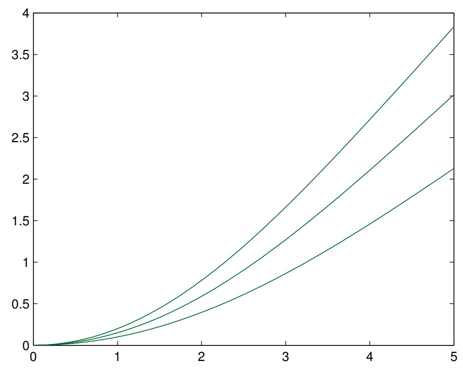
$$\begin{aligned} u(0) &= 0 \\ u'(0) &= 0 \\ u''(0) &= a \text{ , where } a \text{ is still freely choosable.} \end{aligned}$$

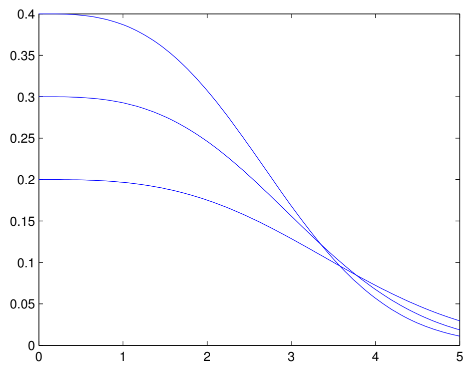
Transform the differential equation into a system of three first-order differential equations.

Solve the differential equation for $a = 0.2$, $a = 0.3$, and $a = 0.4$ each up to $x = 5$.

Graphically represent the three solutions $u(x)$ in one plot. In a second plot, represent the three derivatives $u''(x)$.

Find (e.g., through systematic trial and error) the a value as accurately as possible for which $u''(5) = 0.02$.





L 9.2 Stability: Further Explanations, More General Test Problems, Stiff Systems

Section L 8.1.1 has already briefly addressed the concept of "stability": A method is called *stable* with step size h if, for the model problem – the differential equation of the decaying exponential function $y = e^{-x}$ –

$$y' = -y \qquad y(0) = 1$$

the numerical solution converges to zero as x increases.

L 9.2.1 More General Test Problems, Stability Region

The concept of "stability" is about this: if the exact solution decays exponentially, then the numerical solution at the chosen step size should also somehow provide decaying approximation values.

The differential equation $y' = -y$ is a *very* simple test problem. It is useful to investigate the stability of methods for somewhat more general equations:

For the function $y(x)$, consider the initial value problem

$$\begin{aligned} y' &= \lambda y \\ y(0) &= 1 \end{aligned}$$

for $\lambda \in \mathbb{R}$ and (only then does the problem become truly more general) also $\lambda \in \mathbb{C}$. But to keep it from becoming too overwhelming, we will now keep the step size fixed[42] at $h = 1$.

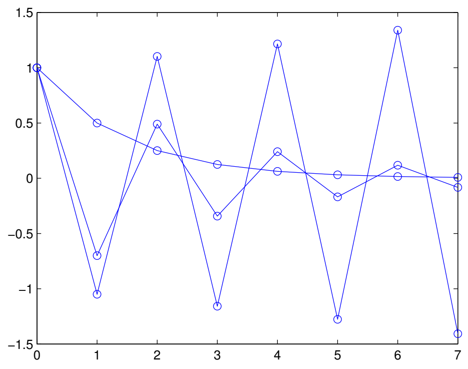
[42]It's pointless to vary both λ and h. All results obtained for step size 1 and parameter λ hold for general h with parameter λ/h.

The *stability region* of a method is the set of all $\lambda \in \mathbb{C}$ for which, in the above model problem with step size $h = 1$, the sequence of computed approximation solutions converges to zero.

The exact solution of the initial value problem is known to be $y = e^{\lambda x}$. For $\lambda = a + bi \in \mathbb{C}$, it reads $y(x) = e^{ax}(\cos(bx) + i\sin(bx))$. The real and imaginary parts of the solution are therefore cosine and sine oscillations, multiplied by the factor e^{ax}.

Of interest in this context are the exponentially decaying oscillations, i.e., $a < 0$ or, equivalently, $\mathrm{Re}(\lambda) < 0$. We wish for a stable solution method to qualitatively compute such decaying oscillations correctly. Hence, the stability region of a solution method should ideally encompass the entire left half-plane of the complex plane.

However, simple test calculations show: for the explicit Euler method, only certain λ values lie in the stable range. Here are approximation solutions of the explicit Euler method for various λ: For $\lambda = -0.5$, monotonically decreasing; for $\lambda = -1.7$, oscillating but still convergent; and for $\lambda = -2.05$, oscillating and divergent.



Task 77: Example: Stability of One-Step Methods for $\lambda \in \mathbb{C}$

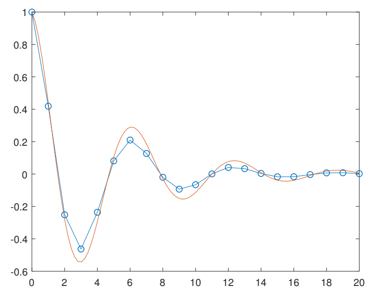
It gets really exciting when you try out values of $\lambda \in \mathbb{C}$ in the test problem for $y(x)$:

$$\begin{aligned} y' &= \lambda y \ , \\ y(0) &= 1 \ , \end{aligned}$$

Choose, for example, $\lambda = -0.2 + \mathrm{i}$. Investigate the explicit and modified Euler methods, as well as the three-step method from Exercise 68. You don't need to change anything else in the sample program `GDGdemo.m`; it happily computes with complex numbers too. Plot the real part of the numerical and exact solution.

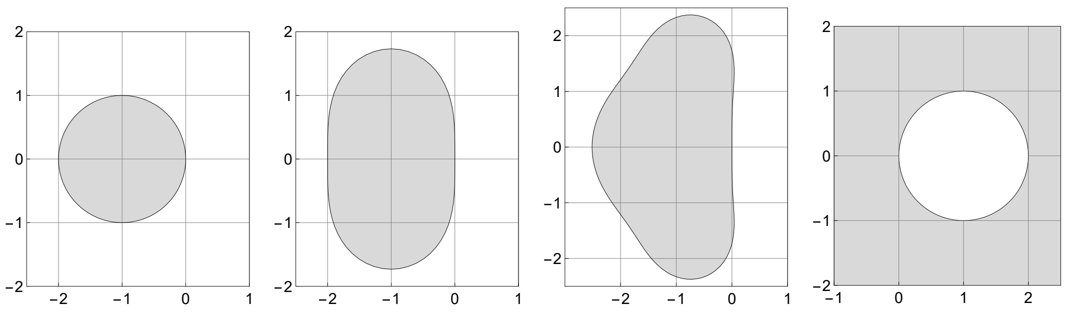
The real part of the exact solution is a damped oscillation. For $\lambda = a + bi$, it reads $y(x) = e^{ax}\cos(bx)$.

On the right, you see the numerical and exact solution of the model problem for $\lambda = -0.2 + \mathrm{i}$. Calculations were done with the three-step method from Exercise 68. The numerical solution decays, so λ lies within the stability region of the method. How about the other methods (explicit, modified, and implicit Euler methods, Heun method)? What happens for $\lambda = -0.2 + 2\mathrm{i}$?



The figures below show as gray areas in the complex plane the stability regions of the methods (from left to right:) Explicit Euler method; Modified Euler method; Three-step explicit method; Implicit Euler method. You can try it out: choose a λ from the gray area, then the method computes a sequence of decaying values. White area – exponentially growing values.

Your Task: Choose for each method (Explicit Euler Method, Modified Euler Method, Three-Step Explicit Method, Implicit Euler Method) one λ in the stable region and one in the unstable region and plot the corresponding approximation solutions.



L 9.2.2 Stiff Systems

If there were only ordinary first-order differential equations, then a general-purpose method like MATLAB's `ODE45` would be completely sufficient for numerical solution in most cases; there would be no need to worry about stability or stability regions.

However, in practice, systems of differential equations often occur, and of a special type: *stiff systems*: See lecture slides 8th lecture.

Brief summary:

> If a method is stable, it can solve stiff systems with a reasonable, problem-specific step size. Unstable methods only provide usable approximate solutions for stiff systems with unrealistically small step sizes.

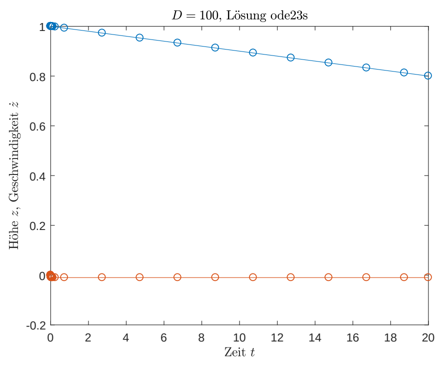
Task 78: Slow Motion

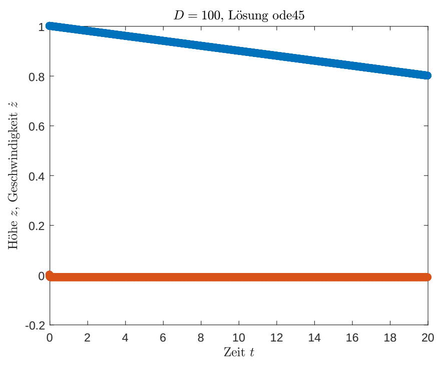
The lecture slides present an example of free fall in a viscous medium: a glass marble sinks in honey. The equation of motion for the height $z(t)$ reads, with suitable scaling of units:

$$\ddot{z}(t) + D\dot{z}(t) + 1 = 0 \text{ with } D \gg 1$$

Choose as initial conditions $z(0) = 1$, $\dot{z}(0) = 0$. Try to reproduce the two figures for $D = 100$ (see also the lecture slides).

What happens for $D = 10$, what for $D = 1000$? Compare also the MATLAB solvers `ode23`, `ode23t`, `ode113`, and `ode15s`: which ones are suitable for which D, which are not?





About the physical background: The value for D can be derived from Stokes' law for the frictional force on spherical bodies as

$$D = \frac{6\pi\eta}{m}\sqrt{\frac{r^3}{g}}$$

with η: dynamic viscosity, for honey you can assume $\eta \approx 10^4$Pa s. For a 10 g heavy ball with 1 cm diameter, you get D values in the thousands. The function $z(t)$ is measured in dimensionless units. To convert to height in meters and time in seconds, you need to multiply z by the radius r and t by $\sqrt{r/g}$.

Task 79: A simple stiff system

Given: Initial value problem for $\mathbf{y} = \mathbf{y}(x)$, $\quad 0 \le x \le 3$

$$\begin{array}{rcrr} y_1'(x) & = & -1999y_1(x) & -1998y_2(x) \\ y_2'(x) & = & 999y_1(x) & +998y_2(x) \end{array}, \qquad \text{Initial conditions} \qquad \begin{array}{rl} y_1(0) & = +1 \\ y_2(0) & = -1 \end{array}.$$

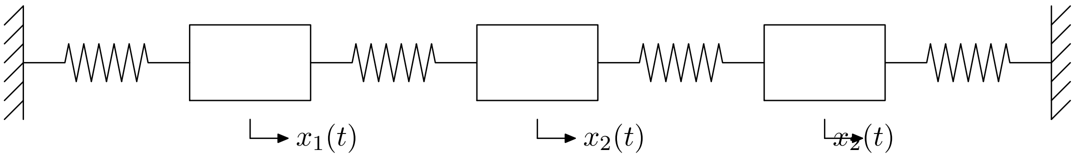
This system is also discussed in the slides of the 8th lecture. Test, similar to the previous exercise, the MATLAB solvers `ode45`, `ode15s`, `ode23`, `ode23t`, `ode113`, and `ode23s`: which ones are suitable, which are not? Compare with the two figures on the lecture slides and present the solutions similarly.

L 9.3 Systems of Higher Order Differential Equations

Section L 9.1 has already demonstrated the transformation of a higher-order differential equation into a system of first-order differential equations. Here comes the general case:

From n differential equations of dth order, a system of $n\,d$ first-order differential equations is obtained.

As an example, consider the equations of motion for three masses coupled by springs.



The positions $x_1(t)$, $x_2(t)$, $x_3(t)$ of the three masses as a function of time t are determined by the following system of second-order differential equations (For simplicity, all masses and spring constants are assumed to be 1).

$$\begin{aligned} \ddot{x}_1 &= -2x_1 + x_2 \\ \ddot{x}_2 &= x_1 - 2x_2 + x_3 \\ \ddot{x}_3 &= x_2 - 2x_3 \end{aligned}$$

Initial values are

$$x_1(0) = 1,\ x_2(0) = x_3(0) = \dot{x}_1(0) = \dot{x}_2(0) = \dot{x}_3(0) = 0\ .$$

This means initially only the first mass is displaced, the other two are in their equilibrium positions; all initial velocities are 0.

In physics, the function determined by differential equations is often denoted not by $y(x)$ but by $x(t)$. Here, x is a spatial coordinate and t is time. The expression $\dot{x}$ means the first derivative and $\ddot{x}$ means the second derivative of x with respect to t.

We transform to a system of six first-order differential equations by introducing the following auxiliary functions:

$$\begin{gathered} z_1(t) = x_1(t),\ z_2(t) = x_2(t),\ z_3(t) = x_3(t), \\ z_4(t) = \dot{x}_1(t),\ z_5(t) = \dot{x}_2(t),\ z_6(t) = \dot{x}_3(t)\ . \end{gathered}$$

The equivalent first-order system then reads

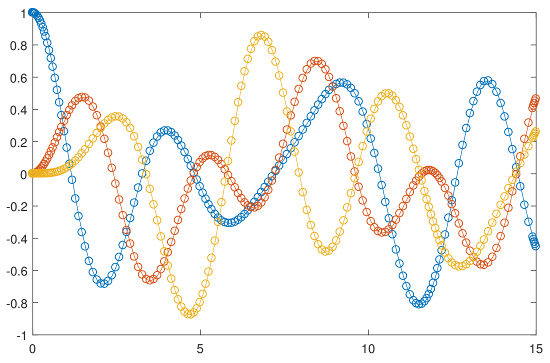
$$\begin{aligned} \dot{z}_1 &= z_4 \\ \dot{z}_2 &= z_5 \\ \dot{z}_3 &= z_6 \\ \dot{z}_4 &= -2z_1 + z_2 \\ \dot{z}_5 &= z_1 - 2z_2 + z_3 \\ \dot{z}_6 &= z_2 - 2z_3 \end{aligned}$$

The standard solver `ode45` requires a function file `threemasses.m`, which computes the six derivatives as a vector `zdot` for an input vector `z`.

```
function zdot=threemasses(t,z)
zdot=[z(4);
      z(5);
      z(6);
      -2*z(1)+z(2);
      z(1)-2*z(2)+z(3);
      z(2)-2*z(3)
      ];
```

Thus, we only need one script file `solution1.m` in which the initial conditions for the sought vector-valued function `y` and its derivative are specified, and the interval of interest is defined. Then, we can already call the solver `ode45`, which is started here with the option to output only the first three components of the system. Our plot then only shows the curves for the displacements $z_1(t) = x_1(t), z_2(t) = x_2(t), z_3(t) = x_3(t)$.

```
z0=[1;0;0;0;0;0];
tspan=[0 15];
options=odeset('OutputSel',[1,2,3])
ode45(@threemasses,tspan,z0,options
```



Try replacing the command `ode45(@threemasses,tspan,z0,options)` with `[t,z]=ode45(@threemasses,tspan,z0,options)`. What changes in the output?

Task 80: Kepler Ellipses

Planets follow elliptical paths; the sun is located at one of the foci of each ellipse (1st Kepler's Law). The coordinates of a planet in the $x-y$ plane are sought as a function of time: $x = x(t), y = y(t)$. The equations of motion are:

$$\begin{aligned} r &= \sqrt{x^2 + y^2} \\ \ddot{x} &= -\frac{x}{r^3} \\ \ddot{y} &= -\frac{y}{r^3} \end{aligned}$$

(For simplicity, gravitational constant and solar mass are set to 1). Initial conditions:

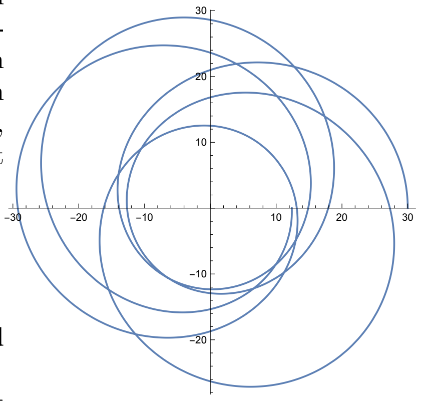
$$x(0) = 10,\ \dot{x}(0) = 0,\ y(0) = 0,\ \dot{y}(0) = 0.2$$

With these initial values, one revolution takes $T = 31.25\,\pi \approx 98.174\,770\,4$. With exact calculation, the orbit would close at this time T, meaning the values at this time would be equal to the initial values. What is the error of the standard solver?

Show the elliptical orbit in the (x, y) plane. **Note:** We are **not** asking for a diagram where $x(t)$ and $y(t)$ are plotted against the t-axis!

Task 81: Kepler was around 1600. Update Interstellar 2014

In the science fiction film *Interstellar* (2014), Prof. Brand (Michael Caine) attempts to solve gravitational field equations. His daughter Amalia Brand (Anne Hathaway) ventures close to a black hole in her spacecraft. In real life, Prof. Brand (Clemens Brand) presents you with the following task regarding the orbit of a spacecraft, 30 km away from a black hole with solar mass. The orbit equation in polar coordinates is (for r in km):



$$r''(\phi) = -\frac{r(\phi)^2}{24} + \frac{2r'(\phi)^2}{r(\phi)} + r(\phi) - \frac{9}{2}$$

Solve for the initial conditions $r(0) = 30,\ r'(0) = 0$, and $0 \le \phi \le 10\pi$.
Plot the orbit in a Cartesian coordinate system $x = r\cos(\phi),\ y = r\sin(\phi)$.

How close does the spacecraft come to the black hole, at what angle? MATLAB Tip: Extracting the value and position of the minimum in a data array; accessing the corresponding element in another data array;

```
[rMin,indMin]=min(r)
phiMin=phi(indMin)
```

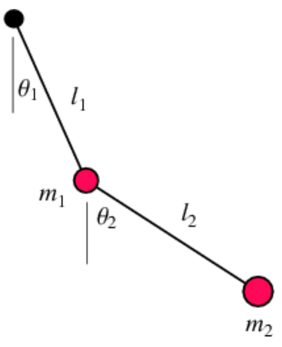
Also, read off the minimum r value from the graph for verification.

The relativistic effects are caused only by the term $-\frac{9}{2}$ (which is theoretically $-\frac{3}{2}$ times the Schwarzschild radius; for the Sun, it's roughly 3 km). If you omit this term, you get a classical Kepler orbit. Plot it in the graph for comparison.

Task 82: Double pendulum

A double pendulum consists of a pendulum to which another pendulum is attached.

It is a simple example of a system with complex dynamic behavior. At small displacements, it oscillates approximately periodically; above a certain system energy, it exhibits chaotic behavior. The equations of motion for the displacements θ_1 and θ_2 as functions of time t are, for small amplitudes (in this approximation, no chaotic behavior occurs yet):



$$\ddot{\theta}_1 = -\frac{g}{\ell_1}\left((1+\mu)\theta_1 - \mu\theta_2\right)$$
$$\ddot{\theta}_2 = -\frac{g(1+\mu)}{\ell_2}\left(\theta_2 - \theta_1\right)$$

(a) Choose $\ell_1 = \ell_2 = 1,\ \mu = 1/10$ ($\mu = m_2/m_1$, mass ratio), $g = 10$. Initially, only deflect the lower of the two pendulums from its rest position, so

$$\theta_1 = 0, \quad \theta_2 = 1, \quad \dot{\theta}_1 = 0, \quad \dot{\theta}_2 = 0 \text{ for } t = 0,$$

Solve for $0 \le t \le 50$.

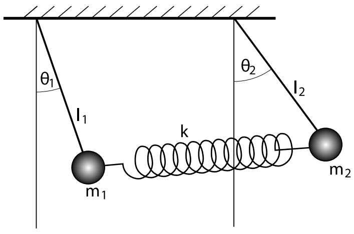
(b) Plot the amplitudes θ_1, θ_2 as functions of t.

(c) Determine from the numerical data and check in the graph: What is the maximum value reached by θ_1, and at what time does this occur?

Task 83: Coupled pendulums

Two pendulums, between which an energy exchange can occur (for example, through a coil spring), are referred to as *coupled pendulums*.

The equations of motion for the angular displacements $\theta_1 = \theta_1(t)$ and $\theta_2 = \theta_2(t)$ are:



$$\ddot{\theta}_1 = -\frac{g}{\ell}\theta_1 + \frac{k}{m}(\theta_2 - \theta_1)$$
$$\ddot{\theta}_2 = -\frac{g}{\ell}\theta_2 - \frac{k}{m}(\theta_2 - \theta_1)$$

(a) Choose $\ell = 1, m = 1, g = 10, k = \frac{1}{2}$ and solve for $0 \le t \le 50$ with the initial conditions

$$\theta_1 = 1,\ \theta_2 = 0,\ \dot{\theta}_1 = 0,\ \dot{\theta}_2 = 0 \text{ for } t = 0.$$

(b) Plot the displacements $\theta_1 = \theta_1(t), \theta_2 = \theta_2(t)$ as function graphs.

(c) Read from the graph: When approximately ($\approx$ second accuracy) is the first pendulum almost at rest while the second pendulum is swinging with maximum amplitude?

(d) Use MATLAB's `max` command and find in the result vector for the second pendulum the maximum displacement and the corresponding time.

Task 84: Three-body problem

A spacecraft moves in the gravitational field of Earth and the Moon. The equations of motion are given here on the right.
μ is the mass ratio in the Earth-Moon system. The other quantities are defined as shown next to it.
The solution $[x(t); y(t)]$ gives the position of the spacecraft in the orbital plane. Coordinate origin at the center of the Earth, the x-axis always points towards the Moon. Length unit is the Earth-Moon distance (380 000 km). Time unit is one lunar orbit (27 days). (So, the Moon always has position [1; 0], the coordinate system rotates compared to an inertial system with one rotation per unit time. The equations of motion in the inertial system would be even more complicated!)

$$\ddot{x} = 2\dot{y} + x - \frac{\mu^*(x+\mu)}{r_1^3} - \frac{\mu(x-\mu^*)}{r_2^3}$$
$$\ddot{y} = -2\dot{x} + y - \frac{\mu^* y}{r_1^3} - \frac{\mu y}{r_2^3}$$

$$\mu = 1/82.45$$
$$\mu^* = 1 - \mu$$
$$r_1 = \sqrt{(x+\mu)^2 + y^2}$$
$$r_2 = \sqrt{(x-\mu^*)^2 + y^2}.$$

(a) Transform the equations of motion into a system of first-order differential equations and solve for $0 \le t \le 10$ with initial conditions

$$x = 1.2; \quad y = 0; \quad \dot{x} = 0; \quad \dot{y} = -1 \qquad \text{for } t = 0\ .$$

(b) Illustrate the trajectory of the spacecraft in the xy-plane.

(c) The calculation starts with the spacecraft just behind the Moon. At what time T does the spacecraft first reach negative x-values? (Then it is near the Earth, and it would be time for a braking maneuver if the astronauts want to return home.)

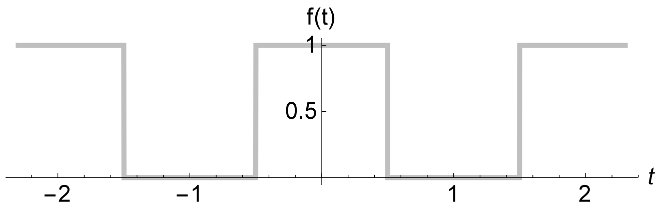
(d) Change the y-component of the initial velocity in the percentage range and by trial and error find that value (to $\pm 0.5\%$ accuracy) for which the orbit closes as closely as possible.

L 10 Tenth Lab Unit: Fourier Series, Fourier Analysis, Fast Fourier Transform (FFT)

There is no separate chapter on this topic in the main part of the script. The material is explained and worked on based on the lecture slides (click!) and exercise problems.

L 10.1 Fourier Series Approximate Periodic Functions

Consider a periodic signal $f(t)$ in the form of a square wave with period $T = 2$. For $-\frac{1}{2} < t < \frac{1}{2}$, $f(t) = 1$, and for $\frac{1}{2} < t < \frac{3}{2}$, $f(t) = 0$.



Approximate this function using Fourier series. First, use terms up to order 1, then 5, and finally 55.

Instructions:

Fourier Series of a periodic function $f(t)$ with period T:

$$f(t) = \frac{a_0}{2} + a_1 \cos(\frac{2\pi}{T}t) + a_2 \cos(\frac{2\pi}{T}2t) + \cdots + a_n \cos(\frac{2\pi n}{T}t) + \cdots$$
$$+ b_1 \sin(\frac{2\pi}{T}t) + b_2 \sin(\frac{2\pi}{T}2t) + \cdots + b_n \sin(\frac{2\pi n}{T}t) + \cdots$$

where

$$a_n = \frac{2}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}} f(t)\cdot\cos(\frac{2\pi n}{T}t)\,\mathrm{d}t \quad b_n = \frac{2}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}} f(t)\cdot\sin(\frac{2\pi n}{T}t)\,\mathrm{d}t$$

The summation symbol Σ and angular frequency $\omega = 2\pi/T$ simplify the notation. Also, for the integrals, it only matters to integrate over a full period; whether from $-T/2$ to $T/2$, from 0 to T, or over any other range covering a full period, is irrelevant. Hence, the limits starting from any t_0 to $t_0 + T$ are given in the following formulation.

Fourier Series for $f(t)$ with period T and angular frequency $\omega = \frac{2\pi}{T}$:

$$f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cos(n\,\omega t) + b_n \sin(n\,\omega t)$$

where

$$a_n = \frac{2}{T}\int_{t_0}^{t_0+T} f(t)\cdot\cos(n\,\omega t)\,\mathrm{d}t \quad b_n = \frac{2}{T}\int_{t_0}^{t_0+T} f(t)\cdot\sin(n\,\omega t)\,\mathrm{d}t$$

In this example, $T = 2$. However, the function f is nonzero only in the range $-\frac{1}{2} < t < \frac{1}{2}$ and is constant with $f(t) = 1$ there. This greatly simplifies the calculation of the integrals:

- The integration does not run from $-T/2$ to $T/2$, but only where $f \neq 0$, i.e., in the interval $-\frac{1}{2} < t < \frac{1}{2}$.
- Note: this signal is symmetric with respect to the y-axis! Since $\sin(a) = -\sin(-a)$, the contributions of positive and negative t values cancel out when integrating (signal times sine-term); thus, all b_n coefficients are 0 here.

So, here's the translation:

It remains to compute:

$$a_n = \int_{-1/2}^{1/2} 1 \cdot \cos(n\pi t)\, dt$$

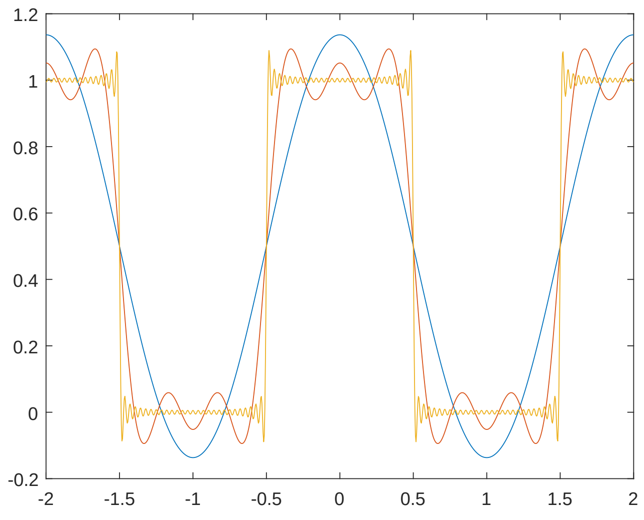
Result:

$$\begin{aligned} a_0 &= 1 \\ a_n &= (-1)^{(n-1)/2} \cdot \frac{2}{n\pi} && \text{for } n > 0 \text{ odd} \\ a_n &= 0 && \text{for } n > 0 \text{ even} \end{aligned}$$

Thus, the Fourier series is:

$$f(t) = \frac{1}{2} + \frac{2}{\pi} \left(\cos(\pi t) - \frac{1}{3}\cos(3\pi t) + \frac{1}{5}\cos(5\pi t) - \frac{1}{7}\cos(7\pi t) \pm \cdots \right)$$

And here are the approximations up to $n = 1,\ 5,\ 55$:



You can try summing even more terms to make the approximation more accurate. But no matter how many terms you sum:

Gibbs Phenomenon: When approximating discontinuous functions with Fourier series, overshoots occur in the vicinity of jump discontinuities. These overshooting spikes become narrower with increasing n, but the maximum deviation remains constant at about 10% of the jump height.

Task 85: Square Wave Function and Gibbs Phenomenon

Given is a periodic square wave signal as above, but with narrower rectangles: period $T = 2$ as before, but $f(t) = 1$ only for $-\frac{1}{4} < t < \frac{1}{4}$, otherwise $f(t) = 0$.

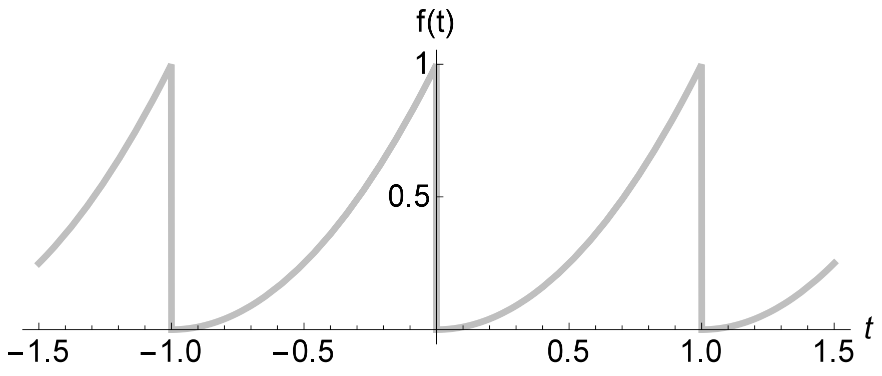
Compute the terms of the Fourier series (Use MATLAB's symbolic integration if you cannot evaluate the integrals with pen, paper, and brain alone!)

Draw the approximations up to $n = 1,\ 5,\ 55$ and compare with the above figure.

The approximations show the Gibbs phenomenon. Read off the maximum height from the graph. Does this value improve if you increase n further? (Choose n as high as you want or MATLAB can reasonably calculate.)

Task 86: Slanted Sawtooth Function

This is an example of the general case where both sine and cosine terms occur. Given is the function $y = t^2$ for $0 \leq t \leq 1$, periodically extended outside this t-interval.



Compute the coefficients of the Fourier series up to $n = 5$. Plot y and the Fourier approximation.

You can use MATLAB's symbolic integration, for example:

```
>> int((1-t^2)*cos(3*pi*t),t,0,1)
 ans =
 2/(9*pi^2)
```

Also, draw approximations with higher n and explain the term "Gibbs phenomenon" based on the representations.

Task 87: Power, Root Mean Square

Given again is the square wave signal $f(t)$ with period $T = 2$. For $-\frac{1}{2} < t < \frac{1}{2}$, $f(t) = 1$, and for $\frac{1}{2} < t < \frac{3}{2}$, $f(t) = 0$.

In many applications, the *root mean square* is used to represent power or energy:

$$\frac{1}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}} f(t)^2 \,\mathrm{d}t$$

Compute this value for the square wave signal.

The Fourier series of this function is given above. Sum the squares of all Fourier coefficients up to $n = 1,\ 5,\ 55$ according to the following formula (Be careful, the coefficient a_0 is always a special case in Fourier formulas and often causes confusion!):

$$P = \frac{a_0^2}{4} + \frac{1}{2}\sum_{n>1} a_n^2 + b_n^2$$

What do you notice in comparison to the previously calculated integral?

In physics and engineering, this relationship is formulated as follows: The energy of a signal in the time domain (integral over f^2) is (up to scaling) equal to its energy in the frequency domain. Each frequency > 0 contributes $\frac{1}{2}(a_n^2 + b_n^2)$ to the total energy. In complex notation, this relationship would be simpler because the coefficient for $n = 0$ would not be a special case.

How large is the contribution of the lowest three frequencies (the first three non-zero terms in the Fourier series) to the total energy for the square wave function?

L 10.2 Fourier Analysis Finds Frequencies, FFT Finds'em Faster

Task 88: Frequency Analysis of an Audio Signal

Fourier analysis decomposes a signal into its frequency components. This guide shows how to calculate all frequency components from which an audio signal is composed.

Download the file `klang1.wav` from the exercise homepage (click!) and read the data into MATLAB. There is a special command for this,

```
[X,fs] = audioread('klang1.wav');
```

The vector `X` contains the signal measurements, recorded with the sampling frequency `fs` (in Hertz, $1\,\mathrm{Hz} = \mathrm{s}^{-1}$). That means, there are `fs` data points per second; the total signal duration is therefore

$$\frac{\text{number of data points} - 1}{\text{sampling frequency}}$$

Create the time axis corresponding to the data points and plot the signal.

Note: The original texts contain LaTeX commands and references which are not translated. Please let me know if you need translations for those parts as well. You see a decaying oscillation. If you are curious and want to know how it sounds: the command `sound(X,fs)` plays the signal. Zooming in (Here, the time interval is approximately $0.194 \le t \le 0.208$) reveals an approximately sinusoidal shape, overlaid with other effects.

You can estimate the frequency f of this clearly visible frequency component directly from the graph: Count the wave peaks in the interval [0.194; 0.208], divide by the time duration. Thus, you have already extracted essential information from the data: A major part consists of sinusoidal oscillations with $\approx 1400\,\mathrm{Hz}$, overlaid with additional oscillations.

However, this is only a very rough approximation; the signal is much more complex.

The fundamental idea of Fourier analysis is:

> A signal, given by n data points, can be represented as a sum of n oscillatory terms (sine, cosine, complex exponential functions)

Now, we have found *one* oscillation. There are still about 60 000 more to go (that's how many data points the signal contains). A direct approach to compute these n components would require $O(n^2)$ computational operations—far too expensive for large datasets! In principle, Karl Friedrich Gauss already designed a faster computational method for this purpose in 1805. Since the 1960s, algorithms for computers known as "FFT" (fast Fourier transform) have been available and are now indispensable in signal processing. The FFT reduces the computational complexity from $O(n^2)$ to $O(n \log n)$. Only because of this is FFT usable in real-time applications.

In MATLAB, the command is simply

```
Y = fft(X);
```

The n signal measurements in the data vector `X` correspond to the same number of Fourier coefficients in the discrete Fourier transformed `Y`.

The formula behind the `fft` command, which MATLAB evaluates very cleverly and computationally efficiently, is

$$Y_k = \sum_{j=1}^{n} X_j \exp\left[\frac{-2\pi i}{n}(j-1)(k-1)\right]$$

A small drawback: this standard FFT command does not directly provide sine and cosine coefficients for a classical Fourier series (the formula with the a_n and b_n terms). However, these coefficients can be easily determined.

The representation of the signal vector `X` as a sum of complex exponential functions is given by

$$X_j = \frac{1}{n}\sum_{k=1}^{n} Y_k \exp\left[\frac{2\pi i}{n}(j-1)(k-1)\right]$$

This looks more complicated than it is. The important thing is the structure of this formula:

> Data vector X = Scaling factor times Sum of ...
> ...(Amplitude Y_k times oscillation with frequency ω_k)
>
> FFT:
> Data vector $X \mapsto$ Frequency vector Y, in MATLAB: `Y = fft(X)`
>
> Inverse FFT:
> Frequency vector $Y \mapsto$ Data vector X , in MATLAB: `X = ifft(Y)`

Another detail: The FFT formula works only with the vectors `X` and `Y`. It indexes the `X` components with $j = 1, 2, \ldots, n$. It knows nothing about the associated time axis with time points $t_1, t_2, \ldots, t_n$ and the angular frequencies ω_k associated with Y_k. The conversion or mapping from index j to time point t_j and Fourier component k to angular frequency ω_k is given by

$$t_j = \frac{j-1}{f_s}\ , \qquad \omega_k = \frac{2\pi(k-1)}{n} f_s$$

For real data vectors with even length n: The real and imaginary parts of the Fourier terms `Y(2)` to `Y(n/2)`, multiplied by $2/n$, respectively provide the cosine and sine amplitudes at frequencies

$$\frac{f_s}{n},\quad 2\frac{f_s}{n},\quad 3\frac{f_s}{n},\ \ldots,\ \frac{n-1}{2}\frac{f_s}{n}$$

(Multiplied by 2π, these are the angular frequencies ω_k.) A special case is the term `Y(1)`: it is the sum of all signal values; divided by n, it gives the mean value of the signal. Another special case is (for even n) the term `Y(n/2+1)`: it only needs to be multiplied by $1/n$ and provides the cosine amplitude at the frequency $f_s/2$. (For odd n, there is no information about $f_s/2$.)

Another important point for real data vectors: only information up to the middle of the Y vector actually contains useful data; the second half contains the same values, mirrored and complex conjugated.

FFT provides frequency information up to half the sampling frequency f_s.
$f_s/2$ is called the **Nyquist frequency**.

To accurately capture frequency components of a signal up to a frequency $f_{\max}$, the sampling frequency f_s must be greater than $2f_{\max}$. Human hearing reaches (at best in young ears) up to about 22 000 Hz. That's why on audio CDs (and in our example file `klang1.wav`) the sampling frequency is $f_s = 44\,100$ Hz.

To display the frequency spectrum:

```
f = linspace(0, fs/2, n/2+1);
plot(f, abs(Y(1:n/2+1)))
```

The plot shows some distinct, sharply defined frequency peaks in the range up to about 4000 Hz. On the other hand, there are no significant contributions above about 8000 Hz. It's pointless to plot up to the Nyquist frequency, here 22 050 Hz. Zoom in on the interesting frequency range (interactively or with the command: `axis([0 8000 0 max(abs(Y))])`).

Another hint: If your signal is not given at "aquidistant measuring points, but on another given time range `t`, the corresponding commands are

```
f=linspace(0,fs/2,n/2+1);
Y = nufft(X,t,f);
```

Task 89: Sound of a Guitar String

The file `klang2.wav` (see the exercise homepage (click!)) contains a short audio recording of a plucked string. The sound spectrum consists of the fundamental frequency and a series of overtones. Analyze the frequency spectrum, display the frequency spectrum in the range up to 5000 Hz. A common choice is a semi-logarithmic representation,

```
f = linspace(0, fs/2, n/2+1);
index=find(f<5000);
semilogy(f(index), abs(Y(index)))
```

Answer the following questions:

- Which frequency provides the strongest contribution?
- What is the frequency of the fundamental vibration? (This is the pitch that a musician hears.)
- Up to which frequency can overtones be clearly identified?

Task 90: Tides

This example aims to show: Fourier analysis can detect periodic effects in a dataset and filter them out from background noise.

The sea level fluctuates due to tides (ebb and flow). Various astronomical and geographical effects with different frequencies (or corresponding periods) overlap. Wind and weather cause additional, unpredictable fluctuations. The theory predicts (among other things) the following components:

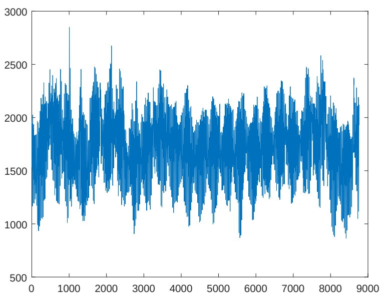
Designation	Name	Period (in hours)
M2	Semi-diurnal principal lunar constituent	12.4206012
S2	Semi-diurnal principal solar constituent	12
N2	Principal lunar elliptic semidiurnal tide 1st order to M2	12.65834751
K1	Lunisolar diurnal constituent	23.93447213
O1	Principal lunar diurnal constituent	25.81933871
M4	Shallow-water overtide of M2	6.210300601
M6	Shallow-water overtide of M2	4.140200401

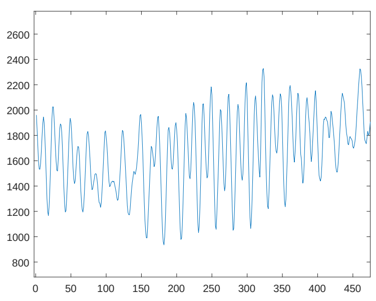
The dataset `Triest2014.dat` on the exercise homepage contains hourly measured sea level values for the year 2014 at the gauge in the port of Trieste.
(Source: `http://uhslc.soest.hawaii.edu/data/download/fd#uh829`).
Fourier analysis can filter out the various periodic components.

Read the data.

```
X=load('Triest2014.dat');
n=length(X);             % Number of data points
fs = 1;                  % Sampling frequency 1/hour
```

A plot of the data vector gives you a first overview. If you zoom in appropriately, you can clearly see 12- and 24-hour periodic fluctuations. (Time unit on the x-axis is 1 hour!) However, you can also see that various short- and long-term fluctuations seem to overlap. Fourier analysis detects the individual periodic oscillations that superimpose to create the data vector.





An important quantity is the sampling frequency f_s (*sampling rate, sampling frequency*, hence the s in the subscript). With time unit hours and one measurement per hour, this is simply

$$f_s = 1, \qquad \text{(Unit 1/h)}$$

Perform the Fourier transformation and calculate the corresponding frequencies:

```
Y = fft(X);
f = linspace(0, fs/2, n/2+1);
```

The term `Y(1)/N` is the average of all data, i.e., the mean sea level. The terms `abs(Y(2))` to `abs(Y(N/2+1))` correspond to the amplitudes of harmonic oscillations with frequencies (from f_s/N to half the sampling frequency $f_s/2$ in steps of f_s/N).

For this dataset, it is not the frequency but the period that is more meaningful. Oscillations with periods up to about 30 hours are of interest. Long-period processes describe fluctuations over days or months; we do not want to investigate them here.

Calculate the corresponding period vector `T` for the frequency vector `f`. Filter out the relevant range from the T vector and plot the corresponding amplitudes.

```
T = 1./f;
index = find(T<30);
plot(T(index), abs(Y(index)))
```

From the graph, you can read, for example:

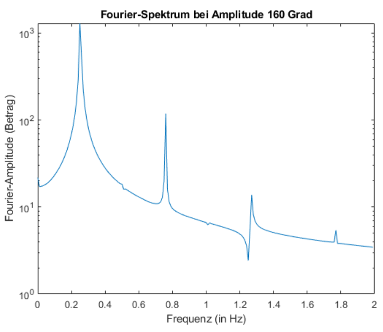
- With which period do the oscillations provide the strongest, second-strongest, and third-strongest contribution?
- What is the relationship between the amplitudes of the principal lunar tide M2 and the principal solar tide S2?
- Can the overtides M4 and M6 be detected in the data?

Task 91: Pendulum with Large Amplitude

The differential equation of the mathematical pendulum is given by

$$\ddot{\phi} + \frac{g}{\ell}\sin\phi = 0$$

with $\phi : t \mapsto \phi(t)$ representing the angular displacement (in radians) as a function of time t (in seconds); gravitational acceleration g in m/s^2; pendu



Choose values for ℓ and g and compute the solution $\phi(t)$ at 1000 equidistant data points in the range $0 < t \leq 100$ for amplitudes of 45°, 90°, 170°. Draw a frequency spectrum of the solution for each case. Your drawings could look similar to the one shown on the right here. You should be able to read from the graphs:

- The first frequency peak (fundamental frequency) is near

$$f_0 = \frac{1}{2\pi}\sqrt{\frac{g}{\ell}}$$

  (This is the classical formula for the oscillation frequency of the pendulum at small amplitudes.)
- With larger initial displacement, the oscillation frequency decreases.
- The greater the initial displacement, the more frequency peaks also appear at higher frequencies (harmonics).

- The frequencies of the harmonics are integer multiples of the fundamental frequency.
- Ideally, the frequency peaks would be sharp spikes at specific frequency values. In reality, the frequencies in the signal are spread over some adjacent data points in the Fourier spectrum (leakage effect).

Provide information for an amplitude of 170°:

1. By what percentage is the oscillation period longer compared to the linear approximation for small amplitudes?
2. What is the relationship between the amplitude of the first harmonic and the amplitude of the fundamental frequency?

(These results are independent of your specific choice for ℓ and g.)

L 10.3 Aliasing and Leakage

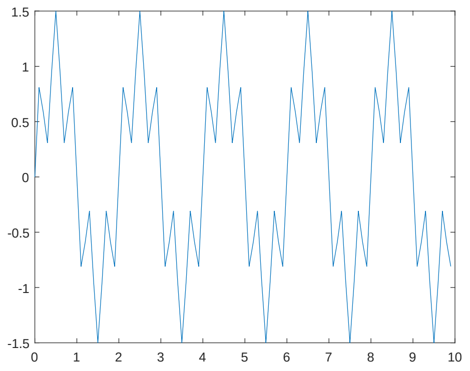
The Fourier analysis of a signal for which data are available only at finite discrete time points cannot accurately detect all frequencies present in the continuous, indefinitely ongoing signal. Two essential limitations are aliasing and leakage. This section presents examples of these phenomena.

The following commands generate a data vector representing the signal

$$X(t) = \sin(\pi t) + \frac{1}{2}\sin(5\pi t)$$

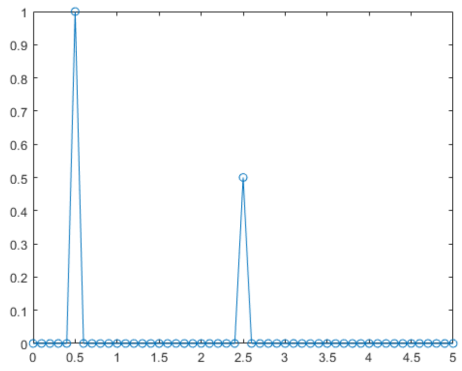
with $n = 100$ data points and sampling frequency $f_s = 10\,\text{Hz}$.

```
fs=10;
n = 100;
t = linspace(0,(n-1)/fs,n);
X = sin(pi*t) + 1/2*sin(5*pi*t);
```



Fourier analysis correctly displays the two frequencies 0.5 Hz and 2.5 Hz. The corresponding amplitudes are also correct because the plot was scaled by $2/n$.

```
%% Fourier Transformation
Y = fft(X);
%% Frequency axis, draw frequency spect
f = linspace(0, fs/2, n/2+1);
index=find(f<10000);
plot(f(index), abs(Y(index))*2/n,'-o')
```



However, this representation (sharp frequency peaks at the correct locations) is not the norm. The next task illustrates: frequencies can be displayed at the wrong location (aliasing) or blurred (leakage).

Task 92: Missing Frequencies, Falsely Hidden or Blurred

Generate data vectors containing higher signal frequencies, for example,

$$X(t) = \sin(2\pi t) + \frac{1}{2}\sin(12\pi t)$$

or

$$X(t) = \cos(4\pi t) + \cos(9\pi t)$$

or

$$X(t) = \cos(11\pi t) + \sin(15\pi t)$$

What signal frequencies does the Fourier analysis provide? Where do the results agree with the signal frequencies, and where do they not?

Aliasing is the term for errors that occur when the signal $X(t)$ contains frequency components greater than or equal to the Nyquist frequency, which is half the sampling frequency.

$$\text{Nyquist Frequency } f_{\text{Ny}} = \frac{f_s}{2}$$

The Fourier analysis erroneously interprets higher frequency components as frequencies below the Nyquist frequency. (These high frequencies appear under "alias names" as low frequencies.)

Refer to Wikipedia under "Aliasing Effect" for very illustrative representations!

The data points in the spectrogram of a discrete Fourier transformation have a spacing of f_s (sampling frequency) on the frequency axis. In our example: 0.1 Hz. If the signal frequency does not exactly match such a discrete frequency value, the spectrogram does not show sharp peaks. The signal frequency spreads over several adjacent data points. It "leaks" to neighboring data points.

An example is a signal with frequency $f = \frac{8}{3}$, for example,

$$X(t) = \cos(\frac{16\pi}{3}t);$$

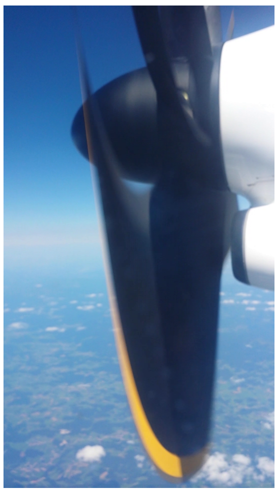
As before, plot the frequency spectrum. What frequency does the peak in the diagram correspond to? What frequencies appear as the two next strongest contributions?

> **Leakage Effect**: The Fourier analysis of a discrete data vector provides sharp frequency data points only in special cases – sampling frequency being an integer multiple of the signal frequency. In general, a sinusoidal signal spreads over several adjacent frequency data points.

Leakage can also be seen as a form of aliasing: The correct frequency partially appears in the spectrogram as neighboring frequencies.

The figure in Task 91 also reveals a leakage phenomenon: the first peak is not as sharp as it should theoretically be. The third peak shows a downward outlier. These disturbances are mainly due to the finite signal duration and sampling frequency.

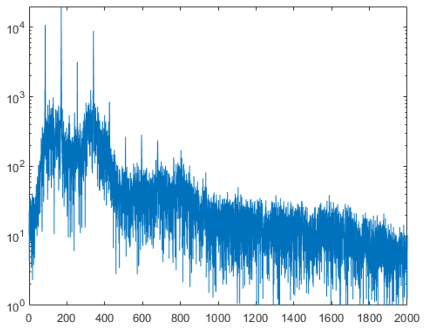
Task 93: Above the Clouds



When you book a cheap window seat airline ticket, this is what the view outside might look like. It's also quite noisy inside the cabin. The short video clip `propeller.mp4` (downloaded from the exercise homepage) gives an impression of that.
In the video, the propeller seems to turn quite slowly. If you look closely and pay attention to the pitch of the propeller blades, it even appears to be rotating backward! Don't panic, those are aliasing effects. You can determine the actual propeller speed from the operating noise through Fourier analysis.
The command

```
[y,fs] = audioread('propeller.mp4')
```

reads the audio signal.



Plot the frequency spectrum in the range 0–2000 Hz. In semi-logarithmic representation, you can see several sharp peaks above a broadband background noise.
The peaks correspond to the Fourier components of the signal from the rotating propeller blades: fundamental frequency and its harmonics. The engine is a Pratt & Whitney Canada Corp PW 150 with a maximum propeller speed of 1200 rpm. The propeller has six blades, so the signal's fundamental frequency is six times the rotational speed.

What speed (in rpm, revolutions per minute) can you determine from the signal?

The video shows 29.9 frames per second. By what angle does one propeller blade rotate from one frame to the next? By what angle does the six-blade propeller *seem* to rotate due to the aliasing effect?

L 10.4 Filtering Signals in the Fourier Domain

Examples of filtering data vectors and data fields.

Task 94: Editing Audio Signals in the Fourier Domain

Read the file `klang2.wav` and play it:

```
[X,fs] = audioread('klang2.wav');
n = length(X);
sound(X,fs)
```

Apply FFT to the signal `X`.

```
Y = fft(X);
```

Now you are going to edit the signal in the frequency domain. You can transform the edited signal back and listen to the result.

Mapping index k in vector $Y \mapsto$ frequency f_k in the frequency domain:

$$f_k = \frac{k-1}{n} f_s \quad \text{for } k \leq \frac{n}{2}$$

You can amplify or attenuate components in Y and thus influence the corresponding frequencies in the audio signal.

A filter vector F multiplies a signal Y in the Fourier domain:

$$Y_{\text{filtered}} = Y \cdot F$$

For a real signal vector X and its Fourier transform Y, the components $F(k)$ of the filter vector F for $k > 1$ must be symmetric with respect to index $n/2 + 1$. In MATLAB notation:

```
F(n/2+2:n) = F(n/2:-1:2);
```

Define different filter functions.

- Low-pass filter: only allows frequencies up to a certain cutoff frequency f_{cut} to pass through; higher frequencies are suppressed. Here, for example, a low-pass filter vector with cutoff frequency 200 Hz:

  ```
  Filt = ones(size(Y));
  fcut=200;
  ncut=round(fcut/fs*n)+1; % Index in Y vector
  Filt(ncut:n/2) = 0;
  % Filter must be symmetric around n/2+1
  Filt(n/2+2:n) = Filt(n/2:-1:2);
  ```

  Apply this filter, transform the signal back from the frequency to the time domain, and listen to the difference:

  ```
  Y1 = Y.*Filt;
  X1 = ifft(Y1);
  sound(X1,fs)
  ```

The original signal was the sound of a low A-string on the guitar, fundamental frequency 110 Hz. A low-pass filter with 200 Hz cuts off all overtones. Therefore, the filtered signal sounds somewhat dull and empty.

Try it out: if you choose a cutoff frequency of 100 Hz, you won't hear any sound anymore because now the fundamental frequency is also filtered out. If you choose higher cutoff frequencies, the filtered signal sounds more and more similar to the original signal.

- High-pass filter: only allows frequencies above a certain cutoff frequency f_{cut} to pass through; lower frequencies are suppressed.

  Try it out: The higher you choose the cutoff frequencies, the sharper the filtered signal sounds.

- Bandpass filter: only allows frequencies within a range $f_{\min} < f < f_{\max}$ to pass through.

- You can also choose wilder filter functions. However, make sure that the filter vector is symmetric around `n/2+1`: `Filt(n/2+2:n) = Filt(n/2:-1:2);` (Otherwise, the transformed signal may contain imaginary parts!) Also, the filter and data vectors must have the same shape (both row or column vectors); otherwise, applying the filter function `Y.*Filt` will result in errors.

- If creating a symmetric filter vector seems too complicated, you can simply truncate the imaginary part of the transformed signal for simplicity. Instead of `X1 = ifft(Y1);`, you write `X1 = real(ifft(Y1));`. This is a rough approach, but it works.

  However, the signal from file `klang2.wav` is not very complex. You will better hear the influence of different filters, for example, if you process the recording of a human voice.

Task 95: Editing Image Files in the Fourier Domain

The following commands read an image file and display a grayscale version of it:

```
% Read an image file
photoarray = imread('katzekatze.jpg');

% A standard formula converts RGB values to grayscale
redpix = single(photoarray(:,:,1))/255;
greenpix = single(photoarray(:,:,2))/255;
bluepix = single(photoarray(:,:,3))/255;
graypix = 0.299*redpix + 0.587*greenpix + 0.114*bluepix;
graypix = rot90(graypix,2); % this way the photo is displayed correctly...

figure(1)
imagesc(graypix)
colormap(gray), axis off image

% Fourier transformed
Y = fft2(graypix);
```

The command `fft2` performs a 2-dimensional Fourier transformation of the data field. Now you can selectively amplify or attenuate frequencies in the Fourier domain. If you only keep the lowest frequency components (low-pass filter), the image becomes blurred and softened:

```
% Low-pass filter
% Number of Fourier modes to keep
% n=8;  % hardly recognizable
% n=16; % recognizable, very blurry
 n=30; % blurry
% n=50; % blurry
% n=100 % moderately good
% n=200 % quite good!

 Y(n:end-n,:) = 0;
 Y(:,n:end-n) = 0;

y = real(ifft2(Y));
figure(3)
imagesc(y)
colormap(gray), axis off image
```

When you modify Fourier components, the transformed signal contains imaginary parts. Only if the filter function has certain symmetry properties, the transformed signal is also real. In the one-dimensional FFT of the audio signal, we considered this. In the two-dimensional case, the symmetry condition is more complicated. Therefore, for simplicity, we simply truncate the imaginary part of the transformed signal. (This is a brute-force method; it works, but the total signal energy is reduced by the amount of the truncated imaginary parts.)

You can also try wilder filter functions and see how changes in the Fourier domain affect the image.

L 11 Eleventh Lab Unit

Content of this lab unit:

- Second Knowledge Assessment: Topics
- Further examples of eigenvalue problems and single-step methods for ordinary differential equations.

L 11.1 Second Knowledge Assessment: Topics

The second knowledge assessment presents you with two subtasks from the following topics:

- Eigenvectors and vector iteration. Typical examples: Problem 64 and the following problems 96 and 97.
- Systems of higher-order differential equations, transforming into 1st order systems. Typical: Problems 80–84.

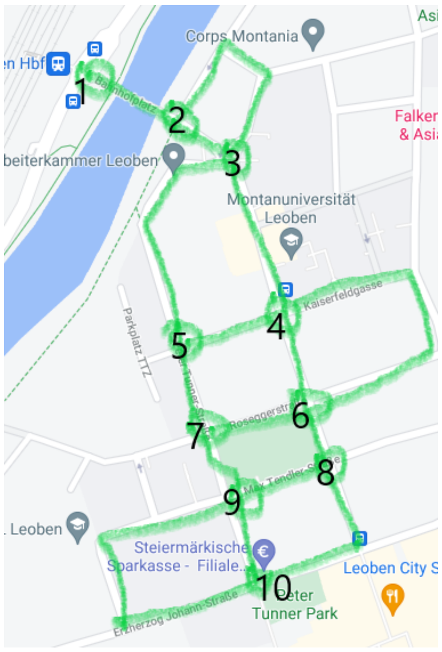
  Explicit and implicit single-step methods implemented by yourself: Problem 68 with sample program `GDGdemo.m` (click!), additional problems 98 and 99.
- Fourier series, Fourier analysis, filtering: see the problems in the previous unit.

L 11.2 Vector Iteration, Eigenvectors: Examples for Review

Task 96: Random Walk

This is problem 64 in a slightly different context. Here, we model a *Random Walk* or a *Markov Process*. It doesn't matter if you're not familiar with these terms, which are important for many applications; here, they are presented to you in an illustrative example.

The sketch on the right shows a simplified road network between the main square and the train station in Leoben. A group of students has celebrated the approaching end of the semester a bit too enthusiastically at the main square and is now staggering aimlessly through the university district. At each of the points 1–10, each person randomly and with equal probability chooses one of the possible directions. As a result, everyone disperses throughout the road network after some time.



This cheerful evening can be modeled as follows:
A vector $\mathbf{x} \in \mathbb{R}^{10}$ contains in component i the number of people currently at point i or on their way there. A matrix $A = [a_{ij}]$ indicates in column j with what probability someone leaves node j in the direction of i.
Then, somewhat simplistically speaking, the matrix-vector product $A \cdot \mathbf{x}$ gives the distribution of people one unit of time later. Continued matrix-vector multiplication leads to a stable state: the distribution of people in the network no longer changes.

Start with the initial vector $\mathbf{x}^{(0)} = [0; 0; \ldots 0; 100]$. This means: 100% probability of being at node 10, the entire group starts at Peter Tunner Park.

Set the transition probability matrix as

$$A = \begin{bmatrix} 0 & 1/3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1/2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2/3 & 0 & 1/4 & 1/3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/4 & 0 & 1/3 & 1/2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/4 & 1/4 & 0 & 0 & 1/3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1/2 & 0 & 0 & 1/3 & 1/3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1/3 & 1/4 & 0 & 0 & 1/4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1/4 & 0 & 0 & 1/4 & 1/3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1/3 & 1/3 & 0 & 2/3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1/3 & 1/2 & 0 \end{bmatrix}$$

(Tip: common PDF viewers allow copying the matrix elements; with little effort, they can be transferred into MATLAB files as matrices.)

- Calculate, starting from $\mathbf{x}^{(0)}$, the state vectors

$$\begin{aligned} \mathbf{x}^{(1)} &= A\mathbf{x}^{(0)} \\ \mathbf{x}^{(2)} &= A\mathbf{x}^{(1)} \\ &\vdots \\ \mathbf{x}^{(k)} &= A\mathbf{x}^{(k-1)} \end{aligned}$$

  until the change in the 2-norm $\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\|_2 < 10^{-2}$.
- How large is component 3 of $\mathbf{x}^{(k)}$ (the probability of a person being at node 3, at MUL)?
- Also, calculate with MATLAB commands directly the eigenvector corresponding to the largest absolute eigenvalue of A. What is the eigenvalue?
- Scale the associated eigenvector so that the sum of the components equals 100 and compare it with the result of vector iteration. You should be able to find agreement within the accuracy of the computation.

Task 97: Creditworthiness

Major banks categorize their customers' creditworthiness according to a tier system: from 1="'highest creditworthiness"' to 5="'no credit possible"'. Monthly data is collected describing the customers' transitions between the tiers. Consider the following model, given by the matrix A below, which represents these transitions:

$$A = \begin{pmatrix} 0.92 & 0.06 & 0.03 & 0.003 & 0.001 \\ 0.07 & 0.8 & 0.14 & 0.007 & 0.0007 \\ 0.005 & 0.1 & 0.62 & 0.15 & 0.012 \\ 0.003 & 0.03 & 0.11 & 0.44 & 0.46 \\ 0.002 & 0.01 & 0.1 & 0.4 & 0.52 \end{pmatrix}$$

Here, the entry in row i and column j indicates what percentage of customers in tier j will move to tier i in the next month. If a distribution v_0 of customers into tiers 1–5 is given proportionally, for example:

$$v_0 = \begin{pmatrix} 0.1 \\ 0.2 \\ 0.2 \\ 0.3 \\ 0.2 \end{pmatrix},$$

the distribution for the next month v_1 is obtained by $v_1 = Av_0$.

Calculate the distribution after three months with the given initial distribution v_0.

Which distribution v_s does not change with respect to the monthly tier transitions? Write this question as an eigenvalue problem (as a comment in your .m file). What is the proportion of customers in tier 5?

What other eigenvalues does this matrix have?

L 11.3 Ordinary Differential Equations: Further Explicit and Implicit Methods

Task 98: Implicit Trapezoidal Method

A step of the *implicit trapezoidal method* for the numerical solution of a differential equation $y' = f(x, y)$ reads

$$y(x+h) = y(x) + \frac{h}{2}\left[f\Big(x, y(x)\Big) + f\Big(x+h, y(x+h)\Big)\right] .$$

(a) Compute, for the differential equation $y' = (1-x)y$ with step size $h = 1/2$ and starting condition $y(0) = 1$, an approximation for $y(3)$.

(b) Repeat the calculation with $h = 1/4$ and provide the end value $y(3)$.

(c) The 12-digit accurate value is $y(3) = 0.223\,130\,160\,148$. Compare the discretization errors from the results (2a) and (2b). In what ratio ϵ_1/ϵ_2 are the errors? What order of error p do you suspect?

Task 99: Rocket Science from the Last Millennium

In a NASA report from 1969, Erwin Fehlberg describes the following single-step method

$$F(x, y, h) = \frac{1}{512}(k_1 + 510k_2 + k_3) \quad \text{with}$$

$$k_1 = f(x, y)\ , \quad k_2 = f\Big(x + \frac{h}{2},\ y + \frac{h}{2}k_1\Big)\ , \quad k_3 = f\Big(x + h,\ y + \frac{h}{256}(k_1 + 255k_2)\Big)$$

(a) Implement the method for the differential equation $y' = \sin(y) + x$ and compute with step size $h = 1/2$ starting from the initial condition $y(1) = -1$ an approximation for $y(4)$.

(b) Repeat the calculation with $h = 1/4$ and provide the end value $y(4)$.

(c) The (to 12 decimal places) accurate value is $y(4) = 5.787\,447\,802\,140$. Compare the discretization errors from the results (2a) and (2b). In what ratio ϵ_1/ϵ_2 are the errors? What order of error p do you suspect?