1. Motivation and Introduction

Numerical Algorithms in Computer Science – Basics and Applications

What is Numerics?

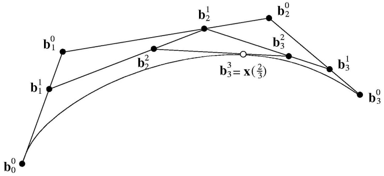
- **Numerical Mathematics**:
  - Part of (applied) mathematics.
  - Designing computational methods for continuous problems mainly from linear algebra (solving linear equation systems, finding eigenvalues etc.) and calculus (finding roots or extrema etc.).
  - Often related to approximations (solving differential equations, computing integrals) and therefore somewhat atypical for mathematics.
  - Analysis of numerical algorithms: memory requirements, computing time, if approximations: accuracy of approximation.
- **Numerical Programming**:
  - Branch of computer science.
  - Efficient implementation of numerical algorithms (memory efficient, fast, considering hardware settings (e.g. cache), parallel).

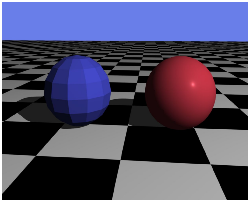
- **Numerical Simulation**:
  - Main field of application of numerical methods.
  - Present in nearly every discipline of science and engineering
  - It provides the third possibility of knowledge acquisition, the other two “classics” being theoretical examination and experiment
  - At all times it has been the main occupation of high performance computers (supercomputers or number crunchers).

1.1. Fields of Application – Numerical Methods in Computer Science

Geometric Modeling

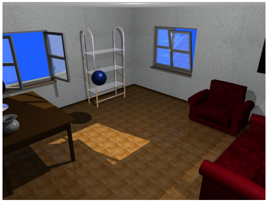
- **Geometric modeling** or **CAGD (Computer-Aided Geometric Design)** deals with the modeling of geometric objects on a computer (car bodies, dinosaurs for Jurassic Park, . . .).
- Especially for **nonlinear curves and surfaces** there are a number of numerical methods including efficient algorithms for their generation and modification:
  - **Bézier curves and surfaces**
  - **B-spline curves and surfaces**
  - **NURBS** (Non-Uniform Rational B-Splines)
- Such methods are based on the methods of interpolation from Chapter 2.





Computer Graphics

- **Computer graphics** is a very computationally intensive branch of computer science:
  - In the context of **ray tracing**, to compute highlight and reflection effects, a huge number of intersection points of rays with objects of the scenery have to be computed – which leads to the problem of solving a system of linear or nonlinear equations (see Chapters 4 and 6).
  - In the context of the **radiosity method** for computing diffuse illumination, a large linear system of equations is constructed which usually has to be solved iteratively – this is covered in Chapter 6.
  - All computer games or flight simulations require very powerful numerical algorithms due to their real time characteristics.



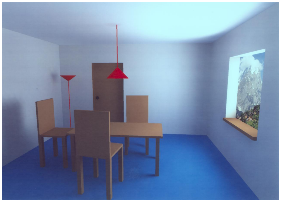
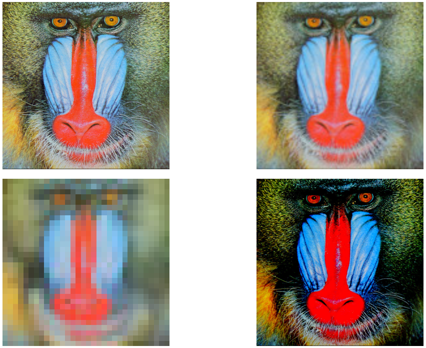


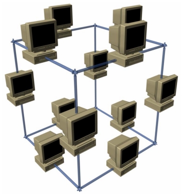
Image Processing

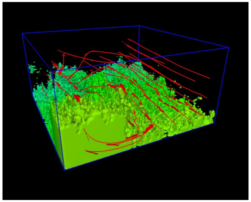
- **Image processing** without numerical methods is also unthinkable. Almost all filters and transformations are numerical algorithms, most times related to the **fast Fourier transformation (FFT)**.
- In addition, most methods for **image compression** (such as JPEG) rely on numerical transformations (discrete cosine transformation, wavelet transformation)
- We will have a quick look at those transformations in Chapter 2.



Numerical Simulation & High Performance Computing

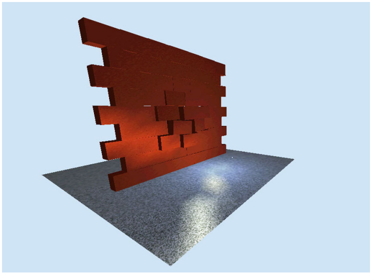
- The links to numerics are nowhere as strong as in **High-Performance Scientific Computing**, i.e. the numerical simulation on high-performance computers.
- Supercomputers spend a major part of their lives with numerical calculations, that's why they are trimmed especially on floating point performance
- Here, efficient methods to solve differential equations numerically are needed – a first foretaste of this will be given in Chapter 8.

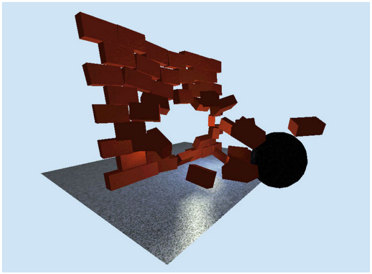




Games Engineering

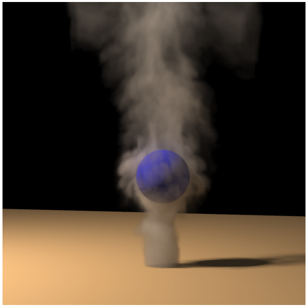
- A **Physics Engine** for realistic simulations (collisions, water flows, flying bullets).
- Many physical phenomena can be described by ordinary differential equations whose numerical solution will be discussed in Chapter 5.





Games Engineering

- **Special effects**: Curtains in the wind, flames, fog and smoke, biomechanics, etc.
- Handling those effects often results in systems of linear equations.
  How to solve those systems is discussed in Chapters 4 and 6.



from Ronald Fedkiw et al.

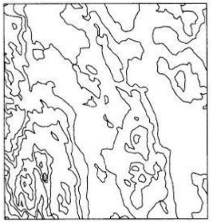
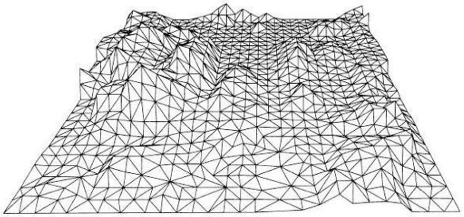
```
[illegible] fps Vsync on (512x512), R8G8B8A8_UNORM (MS1, Q0)
HARDWARE: NVIDIA GeForce 8800 GTX
Press F1 for help
```

from NVidia

The Principle of Discretization

- In numerics, we have to deal with *continuous* problems, but computers can in principle only handle *discrete* items:
  - Computers do not know real numbers, particularly no $\sqrt{2}$, no π, and no $1/3$, but only approximations of discretely (separately, thus not densely) lying numbers.
  - Computers do not know functions such as the sine, but only know approximations consisting of simple components (e.g. polynomials).
  - Computers do not know complicated regions such as circles but only approximations, e.g. by a set of pixels.
  - Computers don't know operations such as differentiation but only approximations, e.g. by the difference quotient.

- The magic word for the successful transition “continuous $\rightarrow$ discrete” is called **discretization**. We discretize
  - real numbers by introduction of **floating point numbers**, see section 1.2;
  - regions (e.g. time intervals when solving ordinary differential equations numerically (see Chapter 6) or spatial regions when solving partial differential equations numerically) by introducing a **grid** of discrete **grid points**;
  - operators such as d/dx by forming difference quotients from function values in adjacent grid points.

Discrete terrain model (right) including contour lines (left)

1.2. Floating Point Numbers and Rounding

Discrete and Finite Sets of Numbers

- The set $\mathbb{R}$ of real numbers is *unbounded* and *continuous* (between two distinct real numbers always lies another real number), there are infinitely, even *uncountably* many real numbers.
- The set $\mathbb{Z}$ of integers is discrete with constant distance 1 between two neighboring numbers, but it is also unbounded.
- The set of **numbers that can be exactly represented** by a computer is inevitably finite, and hence discrete and bounded.

- The probably easiest realization of such a set of numbers and of the arithmetic using it, is **integer arithmetic**:
  - only using integers, typically in a range $[-N, N]$ or $[-N+1, N]$
  - apparent disadvantage: big problems with all continuous concepts (derivatives, convergence, ...)
- The so called **fixed point arithmetic** also allows non-integers:
  - working with decimal numbers with a constant number of digits left and right of the decimal point, typically in a range such as [-999.9999, 999.9999] with (as in $\mathbb{Z}$) a fixed distance between neighboring numbers
  - obvious disadvantage: fixed range of numbers, frequent overflow
  - observation: between 0 and 0.001 additional numbers are often wished for, whereas between 998 and 999 a rougher partition would be sufficient
- A **floating point arithmetic** also works with decimal numbers, but allows a varying position of the decimal point and therefore a variable size and a variable location of the representable range of numbers.

Floating Point Numbers – Definition

- Definition of **normalized t-digit floating point numbers to basis B** ($B \in \mathbb{N} \setminus \{1\}, t \in \mathbb{N}$):

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

  – M is called **mantissa**, E **exponent**.
  – The normalization (no leading zero) assures uniqueness of the representation: $1.0 \cdot 10^2 = 0.1 \cdot 10^3$.
  – discrete set of numbers, infinite range of numbers
  – We assume a varying distance between neighboring numbers (constant number of subdivisions, independent of the exponent).

- The adoption of a feasible range for the exponent leads to the **machine numbers**:

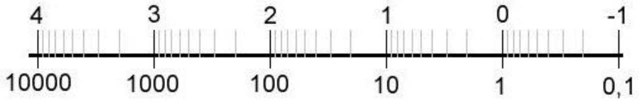
$$\mathbb{F}_{B,t,\alpha,\beta} \ := \ \left\{ f \in \mathbb{F}_{B,t} : \alpha \leq E \leq \beta \right\} .$$

  – The quadruple (B, t, α, β) completely characterizes the system of those machine numbers, in computers such a system is used most times.
  – Of a concrete number therefore M and E have to be stored.

- Often the terms *floating point number* and *machine number* are used interchangeably; normally B and t are clear in context, which is why we will only write $\mathbb{F}$ in the following.

- Example (note $54410_{10} = 31102022_4$):

$$\begin{aligned} 126880 \cdot 10^{-34}: &\qquad B = 10, t = 6, M = 126880 \in [10^5, 10^6[, E = -34\,, \\ 40001 \cdot 2^3: &\qquad B = 2, t = 16, M = 40001 \in [2^{15}, 2^{16}[, E = 3\,, \\ -54110 \cdot 4^0: &\qquad B = 4, t = 8, |M| = 54110 \in [4^7, 4^8[, E = 0\,. \end{aligned}$$



Floating Point Numbers – Range of Representable Numbers

- The **absolute distance** between two neighboring floating point numbers is not constant:
  - Consider for instance the pairs of neighbors $9998 \cdot 10^0$ and $9999 \cdot 10^0$ (distance 1) as well as $1000 \cdot 10^{-7}$ and $1001 \cdot 10^{-7}$ (distance 10^{-7}) in case $B = 10$ and $t = 4$.
  - If the absolute values of the numbers become bigger, the “mesh width” of the discrete grid of floating point numbers also increases – we get a logarithmic scale.
  - That’s reasonable: a million doesn’t make a big difference to national debt but considering one’s own wage a 100 euros difference is quite important for most people.
  - Overall, the usage of floating point numbers increases the range of representable numbers compared to fixed point numbers.

- The maximal possible **relative distance** between two neighboring floating point numbers is called **resolution** ϱ. It holds:

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

- For the boundaries of the representable region, we get:
  - **smallest positive machine number**: $\sigma := B^{t-1}\cdot B^{\alpha}$
  - **biggest machine number**: $\lambda := (B^t - 1)\cdot B^{\beta}$

Floating Point Numbers – Examples

- Famous and most important example is the floating point number format set by the IEEE (Institute of Electrical and Electronics Engineers), which is defined in the US norm ANSI/IEEE-Std-754-1985 and traces back to a patent of Konrad Zuse from the year 1936 (!):

level	B	t	α	β	ϱ	σ	λ
single precision	2	24	-149	104	2^{-23}	2^{-126}	$\doteq 2^{128}$
double precision	2	53	-1074	971	2^{-52}	2^{-1022}	$\doteq 2^{1024}$
extended precision	2	64	-16445	16320	2^{-63}	2^{-16382}	$\doteq 2^{16384}$

- Single precision hence corresponds to approx. 6 to 7 decimal digits, at double precision approx. 15 decimal digits are stored.
- Exactly those definitions are behind the nomenclature used in standard programming languages (e.g `FLOAT` or `DOUBLE` in C).

Floating Point Numbers – Exceptions

Exceptions in which one has to rely on correct troubleshooting by the system's arithmetic:

- **NaN (Not-a-Number)**: undefined value, implemented as **quiet** (inherits quietly) or **signalizing** (activates alert)
- **Exponent overflow**: absolute value of the number is bigger than λ
- **Exponent underflow**: absolute value of the number is smaller than σ (threatens to happen e.g. if comparing $a < b$ is realized by comparing their difference with 0).

The Principle of Rounding

- As floating point numbers are also discrete, certain real numbers can slip. Each of those has to be sensibly assigned to a suitable floating point number – we **round off**, the according transformation is called **rounding**.
- For every arbitrary $x \in \mathbb{R}$, there exists exactly one left and one right neighbor in $\mathbb{F}$:

  $$f_l(x) := \max\{f \in \mathbb{F} : f \leq x\}, \qquad f_r(x) := \min\{f \in \mathbb{F} : f \geq x\}.$$

  In the special case $x \in \mathbb{F}$, of course $f_l(x) = f_r(x) = x$ holds.
- An explicit formula for the floating point numbers of $x > 0$, $x = (M + \delta) \cdot B^E$, $0 \leq \delta < 1$ is given by

  $$f_l(x) = M \cdot B^E, \qquad f_r(x) = \begin{cases} f_l(x) & \text{if } \delta = 0, \\ (M+1) \cdot B^E & \text{otherwise}. \end{cases}$$

- Reasonable postulations for a rounding method $\text{rd} : \mathbb{R} \to \mathbb{F}$ are:
  - surjectivity: $\forall f \in \mathbb{F} \ \exists x \in \mathbb{R}$ with $\text{rd}(x) = f$
  - idempotence: $\text{rd}(\text{rd}(x)) = \text{rd}(x) \forall x \in \mathbb{R}$
  - monotony: $x \leq y \Rightarrow \text{rd}(x) \leq \text{rd}(y) \qquad \forall x, y \in \mathbb{R}$
- There are different ways to round sensibly (i.e. following the above postulations).

Types of Rounding

- We distinguish between three important types of **directed rounding**:
  - At **rounding down** the number is mapped onto the left neighbor:

$$\mathrm{rd}_-(x) \;:=\; f_l(x)\,.$$

  - At **rounding up** it is accordingly mapped to the right neighbor:

$$\mathrm{rd}_+(x) \;:=\; f_r(x)\,.$$

  - **Chopping off** maps the number onto the neighbor closer to zero:

$$\mathrm{rd}_0(x) \;:=\; \begin{cases} f_l(x) & \text{if } x \geq 0\,, \\ f_r(x) & \text{if } x \leq 0\,. \end{cases}$$

    The idea that underlies this notation is to neglect (chop off) every digit from a certain decimal place onward.

- In practice, the most important form of rounding is **correct rounding**, which doesn't know a preferred direction:

$$\mathrm{rd}_*(x) := \begin{cases} f_l(x) & \text{if } x \leq \frac{f_l(x)+f_r(x)}{2}\,, \\ f_r(x) & \text{if } x \geq \frac{f_l(x)+f_r(x)}{2}\,, \end{cases}$$

plus a rule for the procedure in the case of $x = \ldots$, i.e. if x lies exactly in the middle of its two neighbors (e.g. rounding such that the resulting mantissa is even).

- You can easily check that all four ways of rounding introduced here are surjective, idempotent, and monotonous.

The Relative Rounding Error

- Due to rounding, errors are inevitable in numerical computations. We distinguish:
  - **absolute rounding error**: $\text{rd}(x) - x$
  - **relative rounding error**: $\frac{\text{rd}(x)-x}{x}$, if $x \neq 0$
- As the whole construct of floating point numbers is aiming at a high relative precision, the relative rounding error will play the decisive role for every analysis. This error has to be estimated to judge the possible effect of rounding errors in a numerical algorithm.
- If identifying the relative rounding error with ε, from the formula above directly follows

$$\text{rd}(x) \;=\; x \cdot (1+\varepsilon) \qquad \forall x \in \mathbb{R}\,.$$

- For the relative rounding error the following bounds apply:
  - directed rounding:

$$|\,\varepsilon\,| \leq \;\varrho$$

  - correct rounding:

$$|\,\varepsilon\,| \leq \;\frac{1}{2}\varrho$$

- Therefore, the relative rounding error is directly linked to the resolution.

Rounding Errors – a Dramatic Example

- In the second gulf war an American patriot missile missed an approaching Iraqi Scud missile on February 25, 1991 in Saudi Arabia. The Scud missile hit barracks killing 28 US soldiers.
- The cause was a rounding error:
  - The interior clock of the patriot missile saved the time elapsed since booting up the system in tenths of seconds (24-Bit-register).
  - As the tenth of a second is not exactly representable in a binary system, only the first 24 digits were used for calculations, and a resulting rounding error occurred:

$$0.1\,s \;=\; (0.000\overline{1100})_2\,s \;\approx\; 0.000110011001100110011001100\,s\,,$$

error $\approx 9.5 \cdot 10^{-8}\,s\,.$

  - Since booting up the last time, the system hadn't been shut down.
  - After 100 hours of operation, the rounding error had accumulated to

$$100 \cdot 60 \cdot\; 60 \cdot 10 \cdot 9.5 \cdot 10^{-8}\,\text{tenths of a second} \;\approx\; 0.34\,\text{seconds}\,.$$

  - During this time the Scud missile covered a distance of about 570 metres and couldn't be detected by the Patriot missile's sensors anymore.

1.3. Floating Point Arithmetic

Calculating with Floating Point Numbers

- At mere rounding of numbers, the exact value is known. That changes already with the simplest calculations:
  - From the first arithmetic operation on only approximations are operated with.
  - The exact execution of basic arithmetic operations $* \in \{+, -, \cdot, /\}$ in the system $\mathbb{F}$ of floating point numbers is usually impossible – even when using arguments from $\mathbb{F}$: How can the sum of 1234 and 0.1234 be exactly represented with four digits?
- Therefore, we need a “clean” floating point arithmetic that avoids building up accumulated errors.
- Notation:
  - $a * b \in \mathbb{R}$ and usually $a * b \notin \mathbb{F}$ for the *exact* result of the arithmetic operation $*$
  - $a \dot{*} b \in \mathbb{F}$ for the *actually computed* result of the arithmetic operation $*$
- Again, the relative error is the interesting quantity:

$$\varepsilon(a,b) \ := \ \frac{a \dot{*} b - a * b}{a * b}\,, \quad \text{for } a * b \neq 0.$$

- The different postulations for a “clean” floating point arithmetic now differ in the postulations for $a \dot{*} b$.

The Ideal Floating Point Number Arithmetic

- What is ideal in the sense of floating point numbers?
  - Ideally, the computed result should match the rounded exact result:

$$a \dot{*} b \;=\; \mathrm{rd}(a * b) \qquad \forall a, b \in \mathbb{F}, \quad \forall * \in \{+, -, \cdot, /\}\,.$$

  - Reason: This error is inevitable – even when the exact result is known, after all the exact result also has to be forced into the corset of $\mathbb{F}$, i.e. it has to be rounded.
  - Such an **ideal arithmetic** is not utopia but possible. The IEEE standard requests it for the basic operations in binary floating point arithmetic and even for square roots, namely for all three introduced accuracy levels and for all four introduced types of rounding.
- With this, we get bounds for the rounding error of our arithmetic operations in the ideal arithmetic:

$$a \dot{*} b \;=\; \mathrm{rd}(a * b) \;=\; (a * b)(1 + \varepsilon(a, b)) \qquad \forall a, b \in \mathbb{F}$$

  with

$$|\,\varepsilon(a, b)\,| \leq\; \bar{\varepsilon} \;=\; \frac{1}{2}\varrho \text{ bzw. } \varrho \qquad \text{(depending on the type of rounding)}\,.$$

- The variable $\bar{\varepsilon}$ is called **machine accuracy** or **computational accuracy** and depends only on the parameters B and t of the floating point arithmetic.

Relaxations

- Although an ideal arithmetic technically is feasible, in some computers only an alleviated version is realized.
- **strong hypothesis**:
  - There exists an $\tilde{\varepsilon} = O(\varrho)$ that bounds the relative error in every case:

$$a \dot{*} b \;=\; (a * b)(1 + \varepsilon(a,b))$$

    with

$$\mid \varepsilon(a,b) \mid \leq \; \tilde{\varepsilon} \, .$$

  - The strong hypothesis applies for most computers.
- **weak hypothesis**:
  - With $\tilde{\varepsilon}$ from above only holds

$$a \dot{*} b \;=\; (a(1+\varepsilon_1)) * (b(1+\varepsilon_2))$$

    with

$$\mid \varepsilon_1 \mid, \mid \varepsilon_2 \mid \leq \; \tilde{\varepsilon} \, .$$

  - That means, there is no direct functional dependency of the calculated result on the exact result any more.
  - At least this weak postulation applies for nearly every computer.

Surprising Properties of Floating Point Arithmetic

- The floating point operators $\dot{*}$ do not have the same properties as their “authentic” pendants.
- We will study this at the example of floating point addition $\dot{+}$:
  - The floating point addition is not associative.
  - Depending on the order of execution of calculation, different final results can occur.
- For demonstration, the numbers 2^{20}, 2^4, 2^7, -2^3 as well as -2^{20} shall be added. The exact result is 136. Depending on the bracketing of the summands we get different results when calculating with 8 binary digits:

$$\begin{aligned}
(((2^{20} \dot{+} -2^{20}) \dot{+} 2^4) \dot{+} -2^3) \dot{+} 2^7 &\doteq 136 \\
2^{20} \dot{+} (-2^{20} \dot{+} (2^4 \dot{+} (-2^3 \dot{+} 2^7))) &\doteq 0 \\
(2^{20} \dot{+} (-2^{20} \dot{+} 2^4)) \dot{+} (-2^3 \dot{+} 2^7) &\doteq 120 \\
(2^{20} \dot{+} ((-2^{20} \dot{+} 2^4) \dot{+} -2^3)) \dot{+} 2^7 &\doteq 128
\end{aligned}$$

1.4. Rounding Error Analysis

A-priori Rounding Error Analysis

- A numerical algorithm is a finite sequence of basic arithmetic operations with a clearly defined order.
- The floating point arithmetic presents an essential source of error in numerical algorithms.
- Therefore, the most important goals in this regard for a numerical algorithm are:
  - **small discretization error**: as little influence of discretization as possible
  - **efficiency**: minimal runtime
  - **small rounding error**: as little influence of (accumulated) rounding errors as possible
- The latter goal requires an **a priori rounding error analysis**:
  - Which bounds can be determined for the total error assuming a certain quality of the basic operations?

Forward and Backward Error Analysis

- For the a priori rounding error analysis there are two obvious strategies:
  - **Forward analysis**: Interpret the computed result as perturbed exact result (practical, because that leads directly to the relative error, however in general, it is very difficult to calculate due to error correlation).
  - **Backward analysis**: Interpret the computed result as the exact result of perturbed input data (the easier and more popular method)
  - Interpretation of the backward analysis: If the input perturbations due to rounding error analysis are of the same order as their blurring (quite frequently anyway due to measurement errors, previous calculations), then everything is alright with the algorithm in this regard.
- Note: The weak hypothesis only allows for a backward analysis, the strong hypothesis and the ideal arithmetic allow for a backward and a forward interpretation, whereby the relative error of the computed result is bounded by ε in every case:

$$
\begin{array}{rcccl}
a \dot{+} b & = & (a+b)(1+\varepsilon) & = & a(1+\varepsilon) + b(1+\varepsilon) \\
a \dot{\cdot} b & = & ab(1+\varepsilon) & = & a\sqrt{1+\varepsilon} \cdot b\sqrt{1+\varepsilon} \\
 & & \text{forward} & & \text{backward}
\end{array}
$$

An Example: The Horner Scheme

- Task: Find the value $y := p(x)$ of the polynomial $p(x) := \sum_{i=0}^{n} a_i x^i$ for a given x.
- Algorithm: Horner scheme

$$y := (\ldots(((a_n x + a_{n-1})x + a_{n-2})x + a_{n-3})\ldots + a_1)x + a_0$$

or

```
y := a[n];
for i:=n-1 downto 0 do y:=y*x+a[i] od;
```

- For every step according to the strong hypothesis, we have

$$\tilde{y} := (\tilde{y} \cdot x \cdot (1+\mu_i) + a_i) \cdot (1+\alpha_i)$$

with α_i and μ_i bounded by $\tilde{\varepsilon}$.
- Transformations provide

$$\tilde{y} = \sum_{i=0}^{n} \tilde{a}_i x^i$$

with

$$\tilde{a}_i := a_i \cdot (1+\alpha_i) \cdot (1+\mu_{i-1}) \cdot \ldots \cdot (1+\alpha_0), \qquad \alpha_n := 0.$$
- That means: The *computed* value $\tilde{y}$ can be interpreted as *exact* value of a polynomial with slightly differed coefficients.

1.5. The Concept of Condition

Definition and Examples

- **Condition** is a very crucial, but most times only qualitatively defined concept of numerics:
  - How sensitive is the result of a problem concerning changes in the input?
  - At high sensitivity we speak of **bad condition** or an **ill-conditioned problem**, if the sensitivity is low we speak accordingly of **good condition** and a **well-conditioned problem**.
- Very important: The condition number is a property of the examined *problem*, not of the used algorithm.
- Examples:
  - Solving a linear system of equations $Ax = b$:
    Input data is $A \in \mathbb{R}^{n,n}$ and $b \in \mathbb{R}^n$, result is $x \in \mathbb{R}^n$.
  - Compute the roots of a polynomial of order n with real coefficients:
    Input data are the polynomial coefficients $a_0, ..., a_n$, result are the n (complex) roots of p.
  - Compute the eigenvalues of a matrix A:
    Input data is the matrix $A \in \mathbb{R}^{n,n}$, results are all complex λ with $Ax = \lambda x$ for an eigenvector x different from zero.

Well- and Ill-Conditioned Problems

- Perturbations δx in the input data have to be studied because the input is often imprecise (obtained by measuring or from former calculations) and, thus, such perturbations occur frequently, even at exact computing.
- **well-conditioned problems**:
  - A rule to keep in mind: Small δx lead to little perturbations δy of the results.
  - Perturbations of the input, thus, are relatively uncritical.
  - Here it pays to invest in a good algorithm.
- **ill-conditioned problems**:
  - Rule to keep in mind: Even smallest δx can lead to big δy.
  - The solution reacts in an extremely sensitive way to perturbations of the input.
  - Here, even excellent algorithms generally have difficulties.
- From the perspective of a numerical programmer:
  - Ill-conditioned problems are very difficult (in extreme cases even impossible) to deal with numerically.
  - Every error in the input data, every inaccuracy in the run-up by rounding can distort the computed result completely when the problem is ill-conditioned.

Relative error of the Basic Arithmetic Operations

- We examine the arithmetic basic operations. Those obviously represent a numerical problem and, thus, have a condition number.
- For this purpose, we investigate the relative output error and introduce the absolute output error as the difference between exact result with exact input data and exact result with perturbed input data,

  $$f(x+\delta x) - f(x)$$

  and as the relative output error the quotient

  $$\frac{f(x+\delta x) - f(x)}{f(x)}$$

  The absolute and relative condition number is the quotient of the respective output error and the input error:

  $$\frac{f(x+\delta x) - f(x)}{\delta x} \qquad (absolute)$$

  $$\frac{\frac{f(x+\delta x)-f(x)}{f(x)}}{\frac{\delta x}{x}} \qquad (relative)$$

- The following shows the resulting condition numbers or the leading order term in each case (higher, for example square terms in the perturbations δa and δb are neglected):

operation	absolute output error	relative output error	condition
addition/subtraction	$\delta a \pm \delta b$	$(\delta a \pm \delta b)/(a \pm b)$	?
multiplication	$\approx b \cdot \delta a + a \cdot \delta b$	$\delta a/a + \delta b/b$	≈ 1
division	$\approx \delta a/b - a \cdot \delta b/b^2$	$\delta a/a - \delta b/b$	≈ 1

- At multiplication, division, and square root the relative condition number stays within the relative input perturbation $\frac{\delta a}{a}$ and $\frac{\delta b}{b}$.
- Unlike at the real subtraction $(ab > 0)$: If the exact result is close to zero, the relative condition can become arbitrarily large.

The Condition of Compound Problems

- Usually the condition of a problem $p(x)$ to the input x is not defined as above by a simple difference (thus by the relative error) but by the derivative of the result w.r.t. the input:
$$\text{cond}(p(x)) \;:=\; \frac{\partial p(x)}{\partial x}\,.$$
- When decomposing the problem p into two or more subproblems, the result is (due to the chain rule)
$$\text{cond}(p(x)) \;=\; \text{cond}(r(q(x))) \;=\; \frac{\partial r(z)}{\partial z}\,|_{z=q(x)} \cdot \frac{\partial q(x)}{\partial x}\,.$$
- Of course, the total condition of $p(x)$ is independent of the decomposition, but the partial conditions do depend on it. This might lead to problems:
  - Let p be well-conditioned with an excellently conditioned first part q and lousily conditioned second part r.
  - If now errors occur in the first part, those could lead to a catastrophe in the second part.
  - Consider: $\frac{\partial p}{\partial x} = O(10^{-10})\,, \qquad \frac{\partial r}{\partial z} = O(10^{10})\,, \qquad \frac{\partial q}{\partial x} = O(10^{-20})\,.$

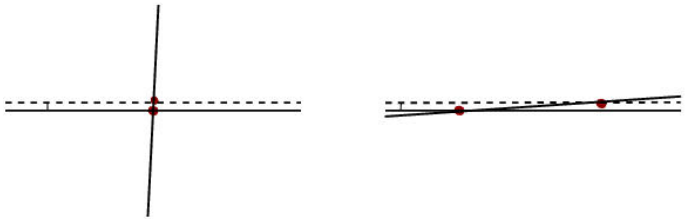
    Rounding errors in the first step of order $O(10^{-14})$ will be inflated to the dimension of $O(10^{-4})$ by r during the second step – and, hey presto, there are only 4 significant digits left, although p was that well-conditioned!

Example 1: The Symmetric Problem of Eigenvalues

- As an example, we examine the problem of finding the n real eigenvalues of a symmetric matrix $A = A^T \in \mathbb{R}^{n,n}$.
- The complete problem is very well-conditioned: Small perturbations of the input (the elements of the matrix) only lead to small perturbations of the eigenvalues.
- A solution strategy well-known from linear algebra provides a decomposition into the two subproblems “setting up the characteristic polynomial” and ”finding its roots”:
  - The first subproblem is perfectly conditioned, the second one lousily: Even errors in the last significant digit of the polynomial’s coefficients lead to a completely different graph, therefore, to a different position of the roots.
  - Therefore, the total result is completely useless (cf. Chapter *Interpolation*) – finding roots of polynomials should always be avoided as a subproblem.
  - The consequence: The eigenvalues of A certainly must not be determined in this way.
  - For comfort: There are other ways, i.e. without ill-conditioned subproblems.

Example 2: The Intersection Point of Two Non-Parallel Straight Lines

- In the plane, the point of intersection of two straight non-parallel lines $ax + by = c$ and $dx + ey = f$ shall be determined:
  - Input data: the coefficients a, b, c, d, e, f of the linear equations
  - Result: the coordinates $\bar{x}$ and $\bar{y}$ of the intersection point
- Geometrically, it is clear:
  - If the lines run almost orthogonally, the problem of determining the intersection point is very well-conditioned.
  - On the contrary, if the lines run almost parallel, the problem of determining the intersection point is very ill-conditioned.



1.6. The Concept of Stability

Numerically Acceptable Results

- With the concept of condition we are now able to characterize problems. Now we will have a look at the characterization of numerical algorithms.
- As we have already seen, input data can be perturbed. From the mathematical point of view, that means that they are only fixed within a certain tolerance, meaning they lie e.g. in a neighborhood

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

  of the exact input x. Hence, any such $\tilde{x}$ has to be considered as virtually equal to x. With this, the following definition suggests itself:
- An approximation $\tilde{y}$ for $y = p(x)$ is called **acceptable**, if $\tilde{y}$ is the exact solution to one of the above $\tilde{x}$, thus

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

- In literature varying weaker definitions can be found.
- The proof of acceptability can be – similar to backward calculation at rounding error analysis – carried out by a *validation computation*.
- The occurring error $\tilde{y} - y$ has different sources:
  - rounding errors
  - method or discretization errors: Series and integrals are approximated by sums, derivatives by difference quotients, iterations will stop after a few iteration steps, ...

Numerically Stable Algorithms

- **Stability** is another vital concept in numerics. A numerical algorithm is called **(numerically) stable**, if for all permitted input data perturbed in the size of computational accuracy $O(\tilde{\varepsilon})$ acceptable results are produced under the influence of rounding and method errors.
- A stable algorithm can definitely produce large errors – for example when the problem to solve is ill-conditioned. In this case, acceptable results can be positioned far away from the exact results.
- What is stable, what is unstable?
  - The basic arithmetic operations are numerically stable under the precondition of the weak hypothesis.
  - Compositions of stable methods are not necessarily stable – otherwise the statement above would indicate everything being numerically stable.
  - For methods to numerically solve ordinary and partial differential equations, stability is a very vital topic. For the former, see Chapter *Ordinary Differential Equations*.

Example of an Unstable Algorithm

- A simple example shall clarify the phenomenon of stability. The bigger of the two roots of the quadratic equation

$$x^2 + 2px - q = 0$$

is to be found, namely for the concrete input data

$$p = 500, \qquad q = 1\,.$$

- The formula

$$x := \sqrt{p^2 + q} - p$$

well-known from school delivers

$$\sqrt{250001} - 500 = 0.00099999900\ldots\,.$$

- When computing with 5 digits, the computed result however is zero. Zero can only be root for the input $q = 0$ – which however is no modification within the computing accuracy $O(10^{-5})$.
- Therefore, the computed result is not acceptable and the algorithm is unstable.

- Note that for $p = q = 1$ no problems occur – not even when computing with 5 digits.
- The remedy: Transform the above formula into

$$x := \frac{q}{\sqrt{p^2 + q} + p} .$$

  This formula represents a stable instruction for calculation.
- What does unstable behavior look like?
  - For example it takes the form of **oscillations**: The computed approximate solution for an ordinary differential equation oscillates around the exact solution and, therefore, shows a totally different behavior compared with the exact solution – so it can't be an acceptable result.

1.7. Cancellation

The Phenomenon of Cancellation

- **Cancellation** describes the effect occurring at the subtraction of two numbers with same signs that leading identical digits cancel each other, that means leading non-zeros disappear. The number of relevant digits can be reduced dramatically.
- Loss of significance impends particularly when both numbers are of the same order of magnitude and sign.
- Examples:
  - Subtract 4444.4444 from 4444.5555. Both numbers have eight significant digits, the result only has four!
  - Subtract 999999 from a million. We assume a perturbation of ± 1 for both numbers and, beside the exact result 1, get the exactly calculated result of the perturbed numbers:

    $$(1000000 + 1) - (999999 - 1) = 3 .$$

    Hence, the relative output error is

    $$\frac{\delta(a-b)}{a-b} = \frac{3-1}{1} = 2 ,$$

    although the relative perturbation of the input data was only of the order $O(10^{-6})$. The condition number is therefore in $O(10^{6})$

- This gets even more alarming for complete cancellation, i.e. when the exact result would be zero – in this case the relative error becomes infinitely large.
- A nice example: Compute e^{-20} via the known expansion of the exponential function. Keep adding up until the value does not change any more.
  Observation: Instead of the correct value of approximately $2.061 \cdot 10^{-9}$ a completely incorrect result may be delivered due to cancellation!
  When computing with 7 digits, the Maple program

```
x := -20;
n := 100;
y := 1.0;
s := 1.0;
Digits := 7;
for i from 1 to n do
    y := y*x/i;
    s := s+y;
od;
s;
```

  delivers the result 7.014115, for 14 digits (`Digits := 14`) $9.253 \cdot 10^{-7}$, for 21 digits indeed the correct $2.061 \cdot 10^{-9}$.

$$e^{-20} \approx S(n)$$

n	$S(n)$
1	-19.0
2	181.0
3	-1152.333
...	...
7	-186231.6
8	448688.6
...	...
30	$1.599699 \cdot 10^6$
31	$1.011906 \cdot 10^6$
32	620347
...	...
41	-2124.511
42	1005.751
43	-450.185
...	...
58	7.014426
59	7.014010
60	7.014149
61	7.014104
62	7.014119
63	7.014114
64	7.014115
65	7.014115