# Scientific Computing II

Michael Hanke
School for Computer Science and Communication

November 5, 2010

## Contents

This paper gives a short introduction into error analysis and construction principles for numerical algorithms. Besides very general considerations examples from numerical linear algebra are treated in more detail.

This manuscript is a fast and dirty translation of the first chapter of German lecture notes used at Humboldt University of Berlin.

The basic question to be discussed is the difference between theory (formulation of an algorithm) and practice (implementation on a computer) for numerical algorithms. Numerical algorithms are usually formulated using the field of real numbers. On a computer, we have only a small finite subset of these numbers available, namely, floating point numbers. Given a single arithmetic operation there is often only a small difference between the exact result in real numbers and the machine result obtained after rounding. So, does it matter? Unfortunately, the answer is yes. Therefore we must have a closer look at the behavior of an algorithm with respect to rounding errors ("stability"). Note that there is no problem with respect to integer arithmetic since it is always exact on a computer. Rounding error analysis is a typical problem of numerical analysis.

**Example 0.1.** Consider the following identity in real numbers:

$$\frac{1}{\sqrt{a+b}-\sqrt{a}} - c \equiv \frac{\sqrt{a+b}+\sqrt{a}}{b} - c.$$

Sei $a = 1000$, $b = 0.001$, $c = 62500$. Evaluating both expressions on a handheld calculator with a precision of 8 decimal digits provides us with 0.0 for the left expression while the right one yields 745.568. Which of these results is "more accurate"?

*Problem:* When implementing mathematically equivalent formula in finite precision arithmetic the results may be subject to large errors.

**Example 0.2.** The following expression shall be evaluated:

$$333.75 \cdot b^6 + a^2 \cdot (11 \cdot a^2 \cdot b^2 - b^6 - 121 \cdot b^4 - 2) + 5.5 \cdot b^8 + a/(2 \cdot b).$$

Let the data be $a = 77617.0$, $b = 33096.0$. On a computer IBM 4381 (agreed, this is an ancient machine) using the operating system VM and the programming system VS FORTRAN the following results have been calculated (rounded to 7 decimals)

| | | |
|---|---|---|
| single precision | (6 Hexadecimal digits) | 1.172604 |
| double precision | (14 Hexadecimal digits) | 1.172604 |
| extended precision | (28 Hexadecimal digits) | 1.172604 |

The exact results is

$$-\frac{54767}{66192} = -0.827396059946821\ldots.$$

*Problem:* How can one decide if a result is reliable?

As a simple exercise you can implement this problem on a computer of your choice using a programming environment of your choice. What is the result you obtain?

**Example 0.3.** Let $A = (a_{ij})_{i,j=1}^{n}$ be a square matrix. We are looking for $D = \det A$. For the determinant $D$, we have the explicit expression,

$$D = \sum_{i=1}^{n} a_{1i} A_{1i},$$

where $A_{1i}$ denotes the cofactor of the element $a_{1i}$. Let $z_n$ the number of arithmetic operations necessary for calculating $\det A$. Then it holds

$$\begin{aligned} z_n &= (n-1) + n + n z_{n-1} \\ &= n(z_{n-1} + 2) - 1. \end{aligned}$$

Hence it holds $z_n > n!$. For a serial computer with a floating point performance of 1 Gflops this translates to running times of

$$\begin{aligned} &n = 11, \quad n! = 39961800, && t \approx 0.04\text{s} \\ &n = 21, \quad n! = 51090942171709440000, && t \approx 1620\text{a}. \end{aligned}$$

Obviously, this is impossible with today's technology. Will it be possible sometimes in the future?

(Bakhvalov, 1973) Consider a hypothetic parallel computer with volume $V$. Let each arithmetic unit be a cube with side length $\Delta$. It is reasonable to assume that one arithmetic operation on that unit takes at a time of $\Delta/c$ with the speed of light $c$. Then, the number of operations per second is bounded by $Vc/\Delta^3$. To give an example: Let $V = 1\text{km}^3$, $\Delta = 10^{-8}$cm (approximately radius of an atom). Then the computing speed is limited by $3 \cdot 10^{57}$ operations per second. At the same time, $100! = 10^{159.9\ldots}$.

*Problem:* Mathematical formulae which are very usable in theoretic considerations are completely unusable in practical computations.

The task of Numerical Analysis consists of the development of efficient, implementable algorithms for the solution of computational problems together with the provision of accuracy estimates. Nowadays, we expect even a robust implementation together with reliable *a-posteriori* error estimations. So it requires a deep understanding of tools and methods from both mathematics and computer science.

When numerically solving a problem a number of unavoidable sources of error are present:

- Model errors: A process happening in reality must be described by mathematical expressions. This includes a decision on which properties are essential and which are negligible. A mathematical model is always an approximation of reality.
- Errors in data: Parameters of the real process are only known approximately, say, up to a certain masurement accuracy.
- Errors in the numerical computation:

- Diskretisation errors: Mapping of continuous processes onto discrete values.
- Representation errors: The representation of numbers is usually connected with a lost of accuracy, e.g., $\pi$ cannot be represented exactly on a computer.
- Truncation errors: Every computation must be finished in finite time. This means that every algorithm must be finished after finitly many steps. Problems for which there does not exist a finite algorithm are usually tackled by constructing an infinite sequence converging towards the solution sought. In that case, only finitely many term can be computed which amounts to truncating the sequence.
- Rounding erros: Arithmetic operations with real numbers can only be done with a finite precision on a computer.

The theoretical understanding of the problems as well as of the numerical methods and their implementation is crucial for a critical estimate of computational results.

The following bon mot is attributed to Karl Nickel:

- The (naive) beginner believes in every single digit in a computational result.
- The (experienced) programmer has confidence in half the number of digits.
- The (knowing) pessimist suspects even the sign.

The aim of this notes is to provide you with tools for a rigorous investigation of the results of computations on a machine. I can only emphasize how important this is: A (slow) laptop is capable of carrying out more than $10^9$ arithmetic operations per second. In consequence this means that the computer makes $10^9$ (rounding) errors every second. Why can you expect that the result is "correct"? More precisely, how can you be sure that the result has an accuracy which fulfills your requirements?

In order to be more definite we will consider the solution of linear systems of equations in more detail. On one hand, the underlying analysis and algebra is well-known. On the other hand, methods of numerical linear algebra are very well investigated. Nevertheless, a good deal of mathematical machinery will be necessary.

**Literature:** To be added!

# 1 Basics of Error Analysis

## 1.1 Norms of Vectors and Matrices

Obviously, vectors and matrices are the elementary components of linear systems of equations. In order to be able to quantify the errors appearing in the computations with them (what is a "large" error, what is a "small" error) we need notions for the distance of vectors (and matrices) from each other. Our measure of distances will be *norms*.

**Definition.** A mapping $\|\cdot\| : \mathbb{R}^n \longrightarrow \mathbb{R}$ is called a *norm* in $\mathbb{R}^n$ if it holds

1. $\|x\| \geq 0$ for all $x \in \mathbb{R}^n$, and $\|x\| = 0$ if and only if $x = 0$.
2. $\|\alpha x\| = |\alpha|\|x\|$ for all $x \in \mathbb{R}^n$, $\alpha \in \mathbb{R}$.
3. $\|x+y\| \leq \|x\| + \|y\|$ for all $x, y \in \mathbb{R}^n$ (triangle inequality).

**Conclusion.** *Let $\|\cdot\|$ be a norm in $\mathbb{R}^n$. Then it holds:*

*(i)* *$d : \mathbb{R}^n \times \mathbb{R}^n \longrightarrow \mathbb{R}$ with $d(x,y) := \|x-y\|$ is a metric in $\mathbb{R}^n$ (a measure for a distance).*

*(ii)* *$|\|x\| - \|y\|| \leq \|x-y\| \leq \|x\| + \|y\|$ for all $x, y \in \mathbb{R}^n$.*

**Example 1.1.** (i) Let $p \in \mathbb{R}$, $p \geq 1$.

$$\|x\|_p := \left( \sum_{i=1}^{n} |x_i|^p \right)^{1/p}.$$

$\|\cdot\|_p$ is a norm in $\mathbb{R}^n$.

(ii) The expression

$$(x,y) := x^T y = \sum_{i=1}^{n} x_i y_i$$

defines a so-called *scalar product* in $\mathbb{R}^n$. Obviously, it holds $(x,x)^{1/2} = \|x\|_2$. Moreover, the *Cauchy-Schwarz inequality*

$$|(x,y)| \leq \|x\|_2 \|y\|_2 \text{ for all } x, y \in \mathbb{R}^n$$

holds true.

(iii) $\|x\|_\infty := \max\limits_{i=1,\ldots,n} |x_i|$ defines a norm in $\mathbb{R}^n$.

**Definition.** Let $A$ be a $m \times n$-matrix, $\|\cdot\|_X$ a norm in $\mathbb{R}^n$, $\|\cdot\|_Y$ a norm in $\mathbb{R}^m$. The value

$$\|A\| := \sup_{x \neq 0} \frac{\|Ax\|_Y}{\|x\|_X} = \sup_{\|x\|_X = 1} \|Ax\|_Y$$

denotes the *induced (by $\|\cdot\|_X$, $\|\cdot\|_Y$) matrix norm.*

**Lemma 1.1.** *(i) For given $\|\cdot\|_X$, $\|\cdot\|_Y$ is the induced matrix norm $\|\cdot\| : \mathbb{R}^{m\times n} \to \mathbb{R}$ a norm.*

*(ii) For all $x \in \mathbb{R}^n$ it holds: $\|Ax\|_Y \le \|A\|\|x\|_X$.*

*(iii) Let additionally $\mathbb{R}^k$ be equipped with the norm $\|\cdot\|_Z$, and $B$ a $k \times m$-matrix. Then,*

$$\|BA\| \le \|B\|\|A\|.$$

*Proof.* (i) Obviously, it holds $\|A\| \ge 0$ for all $A \in \mathbb{R}^{m\times n}$. Moreover:

$$\begin{aligned} \|A\| = 0 \quad &\text{iff} \quad \sup_{x\neq 0} \frac{\|Ax\|_Y}{\|x\|_X} = 0 \\ &\text{iff} \quad Ax = 0 \quad \text{for all } x \\ &\text{iff} \quad A = 0. \end{aligned}$$

Let $\alpha \in \mathbb{R}$. Then:

$$\|\alpha A\| = \sup_{\|x\|_X=1} \|\alpha Ax\|_Y = |\alpha| \sup_{\|x\|_X=1} \|Ax\|_Y = |\alpha|\|A\|.$$

Assume additionally $B \in \mathbb{R}^{m\times n}$. Then it holds,

$$\begin{aligned} \|A+B\| &= \sup_{\|x\|_X=1} \|(A+B)x\|_Y \\ &\le \sup_{\|x\|_X=1} (\|Ax\|_Y + \|Bx\|_Y) \\ &\le \sup_{\|x\|_X=1} \|Ax\|_Y + \sup_{\|x\|_X=1} \|Bx\|_Y \\ &= \|A\| + \|B\|. \end{aligned}$$

(ii) Obvious.

(iii) We may estimate:

$$\begin{aligned} \|BA\| &= \sup_{x\neq 0} \frac{\|BAx\|_Z}{\|x\|_X} \\ &= \sup_{Ax\neq 0} \frac{\|BAx\|_Z}{\|Ax\|_Y} \times \frac{\|Ax\|_Y}{\|x\|_X} \\ &\le \sup_{Ax\neq 0} \frac{\|BAx\|_Z}{\|Ax\|_Y} \times \sup_{Ax\neq 0} \frac{\|Ax\|_Y}{\|x\|_X}. \end{aligned}$$

□

**Example 1.2.** (i) For $\|\cdot\|_X = \|\cdot\|_1$, $\|\cdot\|_Y = \|\cdot\|_1$ it holds

$$\|A\|_1 = \max_{j=1,\dots,n} \sum_{i=1}^{m} |a_{ij}| \qquad \text{(column sum norm)}.$$

(ii) For $\|\cdot\|_X = \|\cdot\|_\infty$, $\|\cdot\|_Y = \|\cdot\|_\infty$ it holds

$$\|A\|_\infty = \max_{i=1,\dots,m} \sum_{j=1}^{n} |a_{ij}| \qquad \text{(row sum norm)}.$$

(iii) For $\|\cdot\|_X = \|\cdot\|_2$, $\|\cdot\|_Y = \|\cdot\|_2$ it holds

$$\|A\|_2 = \lambda_{\max}^{1/2} \qquad \text{(spectral norm)},$$

where $\lambda_{\max}$ denotes the maximal eigenvalue of $A^T A$.

*Remark* 1.1. (i) The spectral norm is very expensive to compute. Instead of the spectral norm often the following expressions are used:

$$\|A\|_F := \left( \sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}^2 \right)^{1/2} \qquad \text{(Frobenius norm)},$$
$$\|A\|_{\max} := n \max_{\substack{i=1,\dots,m \\ j=1,\dots,n}} |a_{ij}|$$

For both expressions, Lemma 1.1 holds if $\|\cdot\|_X$ and $\|\cdot\|_Y$ are chosen to be $\|\cdot\|_2$.

(ii) If, for all $x \in \mathbb{R}^n$, $\|Ax\|_Y \le \alpha \|x\|_X$, then $\|A\| \le \alpha$.

A property which is very usable in applications is given in the following definition.

**Definition.** Two norms $\|\cdot\|_X, \|\cdot\|_Y$ in $\mathbb{R}^n$ are called *equivalent* if there exist constants $m, M > 0$ such that

$$m\|x\|_X \le \|x\|_Y \le M\|x\|_X$$

for all $x \in \mathbb{R}^n$.

**Theorem 1.2.** *All norms in $\mathbb{R}^n$ are equivalent.*

*Proof.* We show that all norms are equivalent to $\|\cdot\|_\infty$. Since equivalence of norms is a transitive property, the theorem will be proved. Let $\|\cdot\|$ a norm and $e^i$ the $i$-th unit vector. Then,

$$\|x\| = \|\sum_{i=1}^{n} x_i e^i\| \le \sum_{i=1}^{n} |x_i| \|e^i\| \le \|x\|_\infty \sum_{i=1}^{n} \|e^i\| =: \|x\|_\infty M,$$

where $M = \sum_{i=1}^{n} \|e^i\|$. Moreover,

$$|\|x\| - \|y\|| \leq \|x-y\| \leq M\|x-y\|_\infty,$$

which implies that $\|\cdot\| : (\mathbb{R}^n, \|\cdot\|_\infty) \longrightarrow \mathbb{R}$ is (Lipschitz-) continuous. On the other hand, $S_1 := \{x \in \mathbb{R}^n | \|x\|_\infty = 1\}$ is closed and bounded. Consequently, $\|\cdot\|$ reaches a minimum $m > 0$ on it: $m \leq \|\tilde{x}\|$ for all $\tilde{x} \in S_1$. If $x \neq 0$ is any vector, then $\left\| \dfrac{x}{\|x\|_\infty} \right\|_\infty = 1$, i.e., $m \leq \left\| \dfrac{x}{\|x\|_\infty} \right\|_\infty$. Consequently, $m\|x\|_\infty \leq \|x\|$. □

*Remark* 1.2. (i) Using the result above we can use any norm in error estimates. Practically, one chooses a norm which is most convenient to use. The estimates are then valid in any norm. The penalty to pay is slightly larger constants due to the constants $m$, $M$ in the equivalence estimates.

(ii) From an application point of view, norms should be chosen in such a way that they describe exactly what we mean by large or small errors. Usually, there are a number of "natural" norms available which are dictated by the problem (the application).

In the following we will usually omit the indices in the denotation of the norms if there is no fear of ambiguity. It is convenient to use the so-called Landau symbols $O$/$o$ for describing the asymptotic behavior of complex functions. They will be defined below.

**Definition.** Let $f : D \subseteq \mathbb{R}^n \to \mathbb{R}^m$ and $g : D \subseteq \mathbb{R}^n \to \mathbb{R}^k$ be two mappings.

(i) We write $f(x) = O(g(x))$ for $x \to x_0$ ($x \in D$) if

$$\limsup_{x \to x_0} \frac{\|f(x)\|}{\|g(x)\|} < \infty.$$

(ii) We write $f(x) = o(g(x))$ for $x \to x_0$ ($x \in D$) if

$$\lim_{x \to x_0} \frac{\|f(x)\|}{\|g(x)\|} = 0.$$

(iii) For $m = k$, we write $f(x) \doteq g(x)$ für $x \to x_0$ ($x \in D$) if $f(x) - g(x) = o(g(x))$.

(iv) For $m = k = 1$, we write $f(x) \dot{\leq} g(x)$ für $x \to x_0$ ($x \in D$) if $f(x) \leq h(x) \doteq g(x)$.

Because of Theorem 1.2 these notions are independent of the choice of the norm.

## 1.2 Conditioning of Problems

The numerical solution of a problem consists of computing results (solutions) from given values (input data) according to well-defined rules.

**Example 1.3.** 1. Computation of the function value of a scalar function $y = f(x)$;

2. Solution of a linear system of equations $Ax = b$.

Let the input data be denoted by $d$ , the results by $a$, the compuatational rules by $P$, then the problem reads in short form

$$a = P(d).$$

In Example 1.3, this translates to:

1. $d = x$, $a = y$, $P = f$
2. $d = (A,b)$, $a = x$, $P(d) = A^{-1}b$

In general, the input data for numerical computations are known approximately, only. *By deciding which quantities are considered to be input data we implicitely decide about which data are considered to be exactly known, and which data must be considered to be subject to errors.*

The accuracy of the data can be described by the absolute and relative errors, respectively,

$$\begin{aligned} \delta_x &:= \tilde{x} - x, & \|\delta_x\| &\le \Delta_x, \\ \|\tilde{x} - x\| &= \varepsilon_x \|x\|, & \varepsilon_x &\le \mathrm{E}_x. \end{aligned}$$

For scalar quantities, the relative error can be alternatively defined by $\tilde{x} = (1+\varepsilon_x)x$, $|\varepsilon_x| \le \mathrm{E}_x$.

The following observations are crucial for the understanding of accuracy estimates and the behaviour of numerical algorithms. In general, only the error bounds $\Delta_x, \mathrm{E}_x$ are known while the errors are (obviously) unknown. As a consequence, *the actual problem $a = P(d)$ is not distinguishable from any other problem*

$$\tilde{a} = P(\tilde{d})$$

where the input data $\tilde{d}$ fulfill $\|\delta_d\| \le \Delta_d$ and $|\varepsilon_d| \le E_d$, respectively. All these problems are equally valid given the information provided, that is the input data *and the error bounds*. Consequently, we must understand as the *solution of the given problem* the sets

$$\begin{aligned} \mathscr{A}_a(d, \Delta_d) &:= \{\tilde{a} = P(\tilde{d}) \,|\, \|\delta_d\| \le \Delta_d\}, \\ \mathscr{A}_r(d, E_d) &:= \{\tilde{a} = P(\tilde{d}) \,|\, \|\varepsilon_d\| \le E_d\}. \end{aligned}$$

In practice, this is impossible since these sets can be very complex. From a practical point of view, we are interested in simple characteristics of this set. A simple measure is the diameter of these sets because it mesures the maximal uncertainty (or, error) in the results. In order to assess the accuracy of $\tilde{a}$ the quantities $\Delta_d$ (resp. $E_d$) and $\operatorname{diam} \mathscr{A}_a(d,\Delta_d)$ (resp. $\operatorname{diam} \mathscr{A}_r(d,E_d)$) must be related to each other. However, even this task is practically too hard to do. As a replacement, one ususally tries to derive estimates of the kind

$$\|P(\tilde{d}) - P(d)\| \leq L(d,\Delta_d)\|\tilde{d} - d\| \tag{1.1}$$

$$\frac{\|P(\tilde{d}) - P(d)\|}{\|P(d)\|} \leq K(d,E_d)\frac{\|\tilde{d} - d\|}{\|d\|} \tag{1.2}$$

where $L$, $K$ shall be *realistic* constants. Obviously, the expression $L(d,\Delta_d)\Delta_d$ is an overestimation of $\operatorname{diam} \mathscr{A}_a(d,\Delta_d)$. The larger the constants $L$ or $K$ are, the larger will be the uncertainty (error) in the result of the problem $P$ for a given error bound of the input data. If such an estimate does not exist, then the errors in the result may become arbitrarily large! The latter situation makes any calculation meaningless.

Stop here shortly. It's a good point to contemplate about the setting. Proceed only when you have understod the ideas.

**Definition.** (i) The constants $L$ ($K$) are called an *absolute (relative) condition number* of $P$.

(ii) The problem $P$ is called *well conditioned* if $L$ or $K$ are not too large.

(iii) If there exist estimates of the type (1.1), (1.2), then $P$ is called *correctly posed*. Otherwise, $P$ is called an *ill-posed* problem.

*Remark* 1.3. (i) If $L$ is an absolute condition number, and $d \neq 0$, $P(d) \neq 0$, then $K := L\frac{\|d\|}{\|P(d)\|}$ is a relative condition number.

(ii) Correctness of the problem $P$ is equivalent to the (Lipschitz-) continuous dependence of the solution from the data.

*Remark* 1.4. The computation of condition numbers is often a very hard problem and may require very deep mathematical tools. This is especially true for problems like partial differential equations, optimization problems etc. Sometimes it is possible to obtain an asymptotic estimate of the condition number by using differential calculus.

Let $D \subseteq \mathbb{R}^n$ open, $P : D \longrightarrow \mathbb{R}^m$ and $d \in D$ be given. Moreover, let $\Delta_d$ be a bound for the absolute error. In order to distinguish condition numbers for different problems we will use an additional index. As an example, (1.1) will be written like

$$\|\tilde{a} - a\| \leq L_P(d,\Delta_d)\|\tilde{d} - d\|, \quad \|\tilde{d} - d\| \leq \Delta_d, \quad \tilde{d} \in D,$$

with $a = P(d)$, $\tilde{a} = P(\tilde{d})$. In general, $L_P$ is very hard to compute since $P$ can be very complex. Since we are essentially only interested in the order of magnitude of $L_P$, it

is sufficient to know approximations of it. This aim will be reached by comparing $P$ with easier accessible functions $h$.

Let $h : D \longrightarrow \mathbb{R}^m$ be a function with

$$P(\tilde{d}) = h(\tilde{d}) + o(\tilde{d} - d) \qquad \text{für } \tilde{d} \longrightarrow d, \quad \tilde{d} \in D. \tag{1.3}$$

This is a quantification of the requirement that $h$ is a close approximation of $P$ in a (possibly small) neighborhood of $d$. Obviously, taking the limit $\tilde{d} \longrightarrow d$ leads to

$$P(d) = h(d).$$

Let now $L_h(d, \Delta_d)$ be an absolute condition number of $h$:

$$\|h(\tilde{d}) - h(d)\| \leq L_h(d, \Delta_d)\|\tilde{d} - d\|.$$

Hence, it holds

$$\begin{aligned} \|P(\tilde{d}) - P(d)\| &\leq \|P(\tilde{d}) - h(\tilde{d})\| + \|h(\tilde{d}) - h(d)\| \\ &\leq \left( \underbrace{\frac{o(\tilde{d} - d)}{\|\tilde{d} - d\|}}_{\tilde{d} \longrightarrow d \longrightarrow 0} + L_h(d, \Delta_d) \right) \|\tilde{d} - d\|. \end{aligned}$$

This means that, for sufficiently small $\Delta_d$, it holds

$$L_P(d, \Delta_d) \approx L_h(d, \Delta_d). \tag{1.4}$$

Consider the scalar case $n = m = 1$ first. Easy structured functions are, for example, linear functions

$$h(\tilde{d}) = \alpha(\tilde{d} - d) + \beta.$$

For such a function, we have obviously $L_h(d, \Delta_d) = |\alpha|$. Since $P(d) = h(d)$, it holds $\beta = P(d)$. We must determine $\alpha$ in such a way that (1.3) holds. Substitution yields

$$\begin{aligned} P(\tilde{d}) &= \alpha(\tilde{d} - d) + P(d) + o(\tilde{d} - d) \\ \frac{P(\tilde{d}) - P(d)}{\tilde{d} - d} &= \alpha + \frac{o(\tilde{d} - d)}{\tilde{d} - d}. \end{aligned} \tag{1.5}$$

Because of $\lim\limits_{\tilde{d} \longrightarrow d} \dfrac{o(\tilde{d} - d)}{\tilde{d} - d} = 0$ such an $\alpha$ exists if and only if $\lim\limits_{\tilde{d} \longrightarrow d} \dfrac{P(\tilde{d}) - P(d)}{\tilde{d} - d}$ exists. In that case it holds

$$\alpha = \lim_{\tilde{d} \longrightarrow d} \frac{P(\tilde{d}) - P(d)}{\tilde{d} - d}.$$

$\alpha$ is called the *differential quotient* or *derivative* of $P$ at $d$. One writes $\alpha = P'(d)$. Finally, we obtain

$$L_P(d, \Delta_d) \approx P'(d) \text{ for } \Delta_d \text{ sufficiently small.}$$

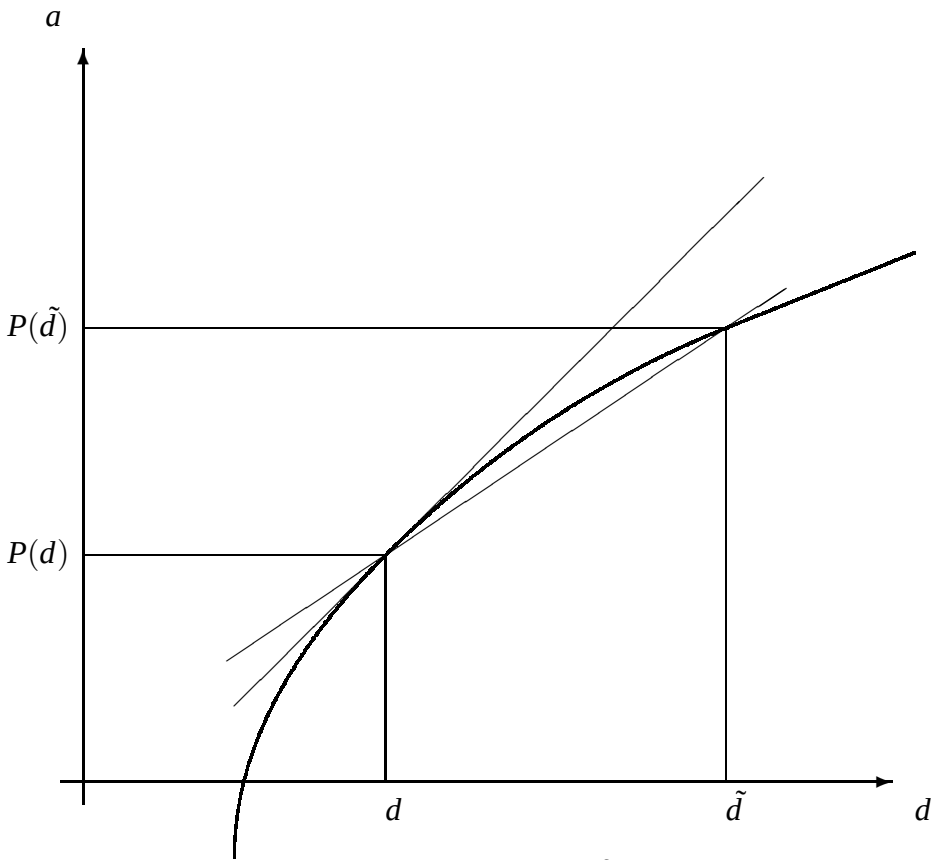


Figure 1: Geometric interpretation of the derivative

$\alpha$ has a nice geometric interpretation (see Figure. 1). While the difference quotient represents the slope of the secant, $\alpha$ represents the slope of the tangent of the graph of $P$ at $d$.

In the multi-dimensional case, the difference quotient does no longer exist. In order to obtain a quantity generalising the notion of the derivative from the one-dimensional case, one argues slighly differently. As a starting point, as ansatz functions for $h$ (affine) linear functions are chosen,

$$h(\tilde{d}) = A(\tilde{d} - d) + b,$$

where $b \in \mathbb{R}^m$ and $A$ an $m \times n$-matrix is. It holds

$$\|h(\tilde{d}) - h(d)\| = \|A(\tilde{d} - d) + b - b\| \leq \|A\| \|\tilde{d} - d\|,$$

such that $L_h(d, \Delta_d) = \|A\|$.

We need to find $A$, $b$. From (1.3) it follows as before $b = P(d)$. Unfortunately, we cannot use (1.5) in order to determine $A$. Therefore, one takes the property (1.3) as a definition: $P$ is called *differentiable* (sometimes *Frèchet differentiable*) in $d \in D$ if there exist $\Delta > 0$ and $A \in \mathbb{R}^{m \times n}$ such that, for all $\tilde{d}$ with $\|\tilde{d} - d\| \leq \Delta_d$ it holds

$$P(\tilde{d}) = P(d) + A(\tilde{d} - d) + o(\tilde{d} - d).$$

$A$ is called the *derivative* of $P$ at $d$, and one writes $A = P'(d)$.

This definition does not provide any practical method for actually computing the derivative. However, the following considerations may help in practical computations. Let in the following $P$ be differentiable at $d$ and $A = P'(d)$.

1. Let $z \in \mathbb{R}^n$ be fixed and $\tilde{d} = d + tz$. Then it holds

$$\begin{aligned} P(d + tz) - P(d) &= tAz + o(tz) \quad \text{für } t \longrightarrow 0 \\ \frac{1}{t}(P(d + tz) - P(d)) &= Az + o(z) \quad \text{für } t \longrightarrow 0 \\ Az &= \lim_{t \longrightarrow 0} \frac{1}{t}(P(d + tz) - P(d)). \end{aligned}$$

   This relation holds true if $P$ is differentiable at $d$. One may ask the opposite question: If the limit exists for all $z \in \mathbb{R}^n$, is $P$ then differentiable at this point? The answer is no. Therefore, one introduces a new definition. If this limit exists for all $z \in \mathbb{R}^n$ and has the representation $Az$, then $P$ is called *Gâteaux differentiable* at $d$, and $A$ is the *Gâteaux derivative*.

2. Let us simplify this limit further. Let, in particular, $z = e^j$ be the $j$-th unit vector. Then $Az = Ae^j$ is the $j$-th column of $A$. Let $P = (P_1, \ldots, P_m)^T$ and $d = (d_1, \ldots, d_n)^T$. Then

$$\frac{1}{t}(P_i(d + tz) - P_i(d)) = \frac{1}{t}(P_i(d_1, \ldots, d_j + t, \ldots, d_n) - P_i(d_1, \ldots, d_j, \ldots, d_n)).$$

Hence, $a_{ij}$ is the derivative of the (scalar) function $P_i$ with respect to $d_j$, where all other variables are considered to be parameters. $a_{ij}$ is called the *partial derivative* $a_{ij} := \frac{\partial P_i}{\partial d_j}$. $A$ is called the *Jacobi matrix* or simply *Jacobian*.

The derivation yields

$$P \text{ differentiable} \implies P \text{ Gâteaux differentiable} \implies \text{Jacobi matrix exists.}$$

The corresponding reverse implications are not true. However, for practical purposes, the Jacobian can easily be computed and used in estimations.

**Example 1.4** (Wilkinson)**.** The problem consists of determining the roots of the polynomial

$$p(x) = (x-1)(x-2)\dots(x-20) = x^{20} - 210x^{19} + - \cdots + 20!.$$

Obviously, the root are $1, 2, \dots, 20$. All coefficients of this polynomial are integers such that they can be represented exactly (i.e. without rounding errors) on a computer. But let us assume that we made a tiny error in the coefficient infront of $x^{19}$. The coefficient becomes $210 + \varepsilon$. We assume that this coefficient is wrong in the least significant bit in a 32-bits representation such that $\varepsilon = -2^{-23} \approx -1.2 \times 10^{-7}$ (relative error $\approx 0.5 \times 10^{-9}$). The roots of the perturbed polynomial are

$$\begin{aligned}
\tilde{x}_1 &= 1.000000000,\\
\tilde{x}_4 &= 4.000000000,\\
\tilde{x}_{10,11} &= 10.095266145 \pm 0.643500904i,\\
\tilde{x}_{16,17} &= 16.730737466 \pm 2.812624894i,\\
\tilde{x}_{20} &= 20.846908101.
\end{aligned}$$

These error are quite large compared to the errors in the data. Even some of the simple and well separated roots have become complex conjugate pairs.

Let us determine the conditioning of our problem. Let $\tilde{x}_i$ be the $i$-th root of the perturbed polynomial $\tilde{p}(x)$. Define $\tilde{p} = p + \varepsilon g$, $g(x) = 210x^{19}$. Then the perturbed roots can be considered as functions of the perturbation $\varepsilon$, $\tilde{x}_i = \tilde{x}_i(\varepsilon)$ while $\tilde{x}_i(0) = i$ is the unperturbed root.

The absolute condition number of our problem can be estimated by the size of the derivative $d\tilde{x}_i(0)/d\varepsilon$. This derivative can be determined as follows. By definition, $\tilde{x}_i$ fulfills the identity $\tilde{p}(\tilde{x}_i(\varepsilon)) \equiv 0$ for all $\varepsilon$. Differentiation with respect to $\varepsilon$ yields

$$p'(\tilde{x}_i(\varepsilon))\frac{d\tilde{x}_i(\varepsilon)}{d\varepsilon} + \varepsilon g'(\tilde{x}_i(\varepsilon))\frac{d\tilde{x}_i(\varepsilon)}{d\varepsilon} + g(\tilde{x}_i(\varepsilon)).$$

For $\varepsilon = 0$ we obtain

$$\frac{d\tilde{x}_i(0)}{d\varepsilon} = -\frac{g(\tilde{x}_i(0))}{p'(\tilde{x}_i(0))}.$$

Inserting the data of our problem, we obtain, for example,

$$L_{\tilde{x}_{20}} \approx 0.9 \times 10^{10}$$
$$L_{\tilde{x}_{16}} \approx 3.7 \times 10^{14}.$$

These are really large condition numbers! In conclusion, the problem of determining the roots of a polynomial can be severly ill-conditioned. Note that the present problem is nevertheless well-posed.

**Example 1.5.** A very simple example of an ill-posed problem is the evaluation of the following function $f$ close to zero.

$$f(x) = \begin{cases} 1/x, & x \neq 0 \\ 0, & x = 0 \end{cases}.$$

Since $f$ is not continuous in 0, this problem cannot be well-posed. Even if we take as the domain of definition the set $D = \mathbb{R} \setminus \{0\}$ (on which $f$ is continuous), we obtain

$$\left| \frac{1}{x} - \frac{1}{y} \right| = \frac{1}{|xy|} |x-y| \geq L|x-y|$$

for $L \leq 1/|xy|$. Since this value is unbounded, the problem is ill-posed for $x, y \to 0$.

**Example 1.6.** Very often the ill-posedness of a problem is not that obvious. Consider the simple linear optimization problem

$$f(x_1, x_2) := x_1 \longrightarrow \max!$$

with respect to the constraints

$$x_1 \geq 0, \quad x_2 \geq 0$$
$$x_1 \leq 1, \quad x_2 \leq 1$$
$$\varepsilon x_1 + x_2 = 0.$$

The data of the problem shall be $\varepsilon$. The solutions are

$\varepsilon \neq 0$**:** $x_1 = x_2 = 0$ and $f(x_1, x_2) = 0$.

$\varepsilon = 0$**:** $x_2 = 0$, $x_1 = 1$ and $f(x_1, x_2) = 1$.

## 1.3 Floating Point Numbers

Numerical problems will be solved by using *floating point arithmetic*. Depending on the programming language, the corresponding data is called `REAL`, `DOUBLE PRECISION`, `real`, `float`, `double` or similar. In order to be able to investigate the properties of a

numerical algorithm with respect to rounding errors we will need a sufficiently precise (meaning even: slightly simplified) model of the corresponding machine arithmetic.

Let $\beta, t, E_1, E_2$ be positive integers with $\beta \geq 2$. The set of floating point numbers consists of all real numbers having the representation

$$x = \pm 0.m_1 \dots m_t \cdot \beta^e$$

where: $\beta$ - base of the number system, $e \in [-E_1, E_2]$,
$m$ - mantissa, $m_i \in \{0, \dots, \beta - 1\}$,
where either $m_1 > 0$ or $m_1 = \dots = m_t = 0$, $e = -E_1$.

**Example 1.7. ANSI/IEEE-Standard 754-1985; IEC-60559:1989; revised 2008**

| | | |
|---|---|---|
| $\beta = 2,$ | $t = 24,$ | (single precision) |
| | $t = 53,$ | (double precision) |
| $E_1 = 127,$ | $E_2 = 126,$ | (single precision) |
| $E_1 = 1023,$ | $E_2 = 1022$ | (double precision) |

Besides these precisions, an additional representation "extended" is required. For this one, it is only required that it has a higher accuracy than double precision.[1] Example realisations are:

| | | |
|---|---|---|
| Intel architecture | $t = 64$ | 15 Bit Exponent |
| HP-PA RISC | $t = 112$ | 15 Bit Exponenet |

**IBM 360 and similar** This floating point system is ancient but may serve as another example.

| | | |
|---|---|---|
| $\beta = 16,$ | $t = 6,$ | (single precision) |
| | $t = 14,$ | (double precision) |
| $E_1 = 64,$ | $E_2 = 63$ | |

With increasing length of the mantissa the accuracy of the computations is increasing, but the computational expense is also increasing, for example memory consumption and execution time for the arithmetic operations.[2] The length of the mantissa should be a well-chosen compromise between accuracy requirements and computational costs. Floating point arithmetic is computation with a fixed number of digits that is usually much larger than required by the accuracy constraints. On the other hand, the often huge number of floating point operations may lead to a large accumulated error. Gehard Wanner noted that someone who is carrying out millions of operation will make millions of errors.

[1] In fact, there is also an extended single precision required. However, the role of extended single precision is usually played by double precision.

[2] The execution time is heavily harware dependent. There may be a penalty or not.

The assessment of the accuracy of numbers which are the result of computations on a machine requires an in-depth knowledge of the properties of the underlying machine arithmetic and a careful error analysis of the algorithm.

Let the set of floating point numbers be denoted by **C**. Moreover, let MAX be the largest representable floating point number and MIN be the smallest positive representable floating point number. Then we have the following properties:

1. The closed interval [−MIN,MIN] contains only three floating point numbers.

2. For representing real numbers we need a mapping (rounding) $\mathrm{rd} : \mathbb{R} \longrightarrow \mathbf{C}$. This mapping has the properties:

$$\begin{aligned} |x| > \mathrm{MAX} &\Longrightarrow \mathrm{rd}(x) \quad \text{is not defined (overflow).} \\ |x| < \mathrm{MIN} &\Longrightarrow \mathrm{rd}(x) = 0 \quad \text{(underflow).} \end{aligned}$$

3. When representing real numbers $|x| < \mathrm{MAX}$ it holds

$$\mathrm{rd}(x) = x(1+\varepsilon(x))$$

with

$$\begin{aligned} \varepsilon(x) &= 0, && \text{if } x = 0 \\ \varepsilon(x) &= -1, && \text{if } 0 < |x| < \mathrm{MIN} \\ \varepsilon(x) &\le \nu, && \text{if } \mathrm{MIN} < |x| < \mathrm{MAX} \end{aligned}$$

and the *relative rounding error level*

$$\nu = \begin{cases} \frac{1}{2}\beta^{1-t}, & \text{(rounding to next floating point number)} \\ \beta^{1-t}, & \text{(rounding by chopping or similar)} \end{cases}$$

In the examples given above it holds with $\nu = \beta^{1-t}$:

$$\text{IEEE:} \qquad \nu = \begin{cases} 2^{-23} & \approx 1.2 \cdot 10^{-7} \\ 2^{-52} & \approx 1.1 \cdot 10^{-16} \\ 2^{-63} & \approx 1.1 \cdot 10^{-19} \\ 2^{-111} & \approx 3.8 \cdot 10^{-34} \end{cases}$$

$$\text{IBM:} \qquad \nu = \begin{cases} 16^{-5} & \approx 9.5 \cdot 10^{-7} \\ 16^{-13} & \approx 2.2 \cdot 10^{-16} \end{cases}$$

4. Floating point arithmetic. Let fl denote the result of a floating point operation. Then we assume the following property to hold:

$$\boxed{\text{Postulate: } \mathrm{fl}(x \,\mathrm{op}\, y) = \mathrm{rd}(x \,\mathrm{op}\, y)}$$

with op $= +, -, *, /$. In this context $\nu$ is called the *machine accuracy* ("machine epsilon"). Then it holds for all operations $x, y \in \mathbf{C}$ and $x \operatorname{op} y = 0$ or $|x \operatorname{op} y| \in$ [MIN,MAX]:

$$\mathrm{fl}(x \operatorname{op} y) = (x \operatorname{op} y)(1+\varepsilon) \text{ mit } |\varepsilon| \leq \nu.$$

One says that the operations have maximal accuracy.

Let $x' = x(1+\varepsilon)$, $y' = y(1+\varepsilon)$. Then this is equivalent to

$$\begin{aligned} \mathrm{fl}(x \pm y) &= x' \pm y' \\ \mathrm{fl}(x * y) &= x' * y = x * y' \\ \mathrm{fl}(x/y) &= x'/y, \end{aligned}$$

i.e., the result of the floating point operation is the exact result of the same operation in real numbers with slightly perturbed operands.

5. Problems of floating point arithmetic.

   **Overflow.** If the absolute value of a result of the operation is larger than MAX, it cannot be represented as a floating point number. This arithmetic exception is called *overflow*. Depending on the hardware or user requirements either the computation will be cancelled or the result will become a fictious value of $\pm$ INF.

   **Underflow.** If the absolute value of a result of the operation is smaller than MIN, the result is often (but not always!) rounded to zero. The relative error of the result becomes 100% in this case!

   **Cancellation.** The addition of two numbers having almost the same absolute value but opposite signs leads to a decrease of the valid number of digits. The result will have a large relative error.

   **Arithmetic rules.** Even in case that there is neither overflow nor underflow, the usual arithmetic rules do no longer hold. By our postulate, both addition and multiplication are commutative. However, these operations are neither associative nor distributive. This lack of properties becomes especially important when a compiler tries to reorder expressions in order to optimize for speed.[3]

*Remark* 1.5. The implementation of the already cited ANSI/IEEE standard for floating point arithmetic is nowadays almost a must for hardware vendors. Therefore, we will consider it here in more detail. The main goals of the standardizing committee can be summarized as follows:

1. A consistent representation of floating point numbers on all machines;

[3] A good compiler should warn you about such possibilities.

2. correctly rounded arithmetic;

3. consistent and reasonable handling of exceptional situations (for example overflow, underflow, division by zero).

The standard requires the presence of three number formats, `single`, `double`, and `extended`. The first two formats are defined in details while vendor has some freedom on how to implement the third one. The only requirement is that `extended` is more accurate than `double`.

The `single` representation uses 32 bits. The number base is 2. In contrast to our model given above, nonzero floating point numbers have a mantissa of the kind $1.m_2\ldots$. Since the first digit is alway 1, there is no need to save it. This omission is called "hidden bit" representation. The exponent contains 8 bits. The exponent does not carry a sign by adding 127 to its binary presentation. The exponents 0 and 255 have a special meaning:

**0:** As we have seen before, the normalized representation of floating point numbers leads to a "hole" in the representable numbers around 0. In order to try to avoid this hole, close to 0 (which corresponds to the smallest exponent 0) unnormalized number representations are allowed. These so-called subnormal numbers have a smaller accuracy than the normalized numbers, but the interval between $-$MIN and MIN is filled with equidistant numbers. This way, one has also a unique representation of the real number 0 as well as a unique interpretation of the bit pattern consisting of all 0.[4]

**255:** The highest possible exponent is reserved for the representation of the "numbers" $+\infty$ and $-\infty$. The mantissa will be 0 in this case. If the mantissa is not equal to 0, the value is interpreted as "Not a Number" (NaN). This value is necessary for a consistent behavior in exceptional situations and arithmetic operations including $+\infty$ and $-\infty$.

Summarizing we obtain the interpretation for all possible 32-bits patterns as given in Figure 2. The double representation has a completely analogous structure where 64 bits are used. Out of these 64 bits, 11 bits are used for the exponent and 52 bits for the mantissa. The exponent offset is 1023. Details are provided in Figure 3.

Let us investigate the arithmetic operations now. First, consider only the normalized and subnormal numbers. There are four rounding modes defined. Let, for $x \in \mathbb{R}$, $x_-$ denote the largest floating point number being $\leq x$ and $x_+$ denote the smallest floating point number with $\geq x$.

- Rounding towards $-\infty$: $\mathrm{rd}(x) = x_-$;
- Rounding towards $+\infty$: $\mathrm{rd}(x) = x_+$;

[4]The hidden bit representation would not allow for a representation of 0.

$$\boxed{\pm|a_1\dots a_8|b_1\dots b_{23}}$$

| Exponent $a_1\dots a_8$ | numerical value |
|---|---|
| $(00000000)_2 = \quad (0)_{10}$ | $\pm(0.b_1\dots b_{23})_2 \times 2^{-126}$ |
| $(00000001)_2 = \quad (1)_{10}$ | $\pm(1.b_1\dots b_{23})_2 \times 2^{-126}$ |
| $(00000010)_2 = \quad (2)_{10}$ | $\pm(1.b_1\dots b_{23})_2 \times 2^{-125}$ |
| $\vdots$ | $\vdots$ |
| $(11111110)_2 = (254)_{10}$ | $\pm(1.b_1\dots b_{23})_2 \times 2^{127}$ |
| $(11111111)_2 = (255)_{10}$ | $\begin{cases}\pm\infty, & \text{if } b_1 = \dots = b_{23} = 0\\ \text{NaN}, & \text{else}\end{cases}$ |

Figure 2: IEEE: single precision

$$\boxed{\pm|a_1\dots a_{11}|b_1\dots b_{52}}$$

| Exponent $a_1\dots a_{11}$ | numerical value |
|---|---|
| $(00000000000)_2 = \quad (0)_{10}$ | $\pm(0.b_1\dots b_{52})_2 \times 2^{-1022}$ |
| $(00000000001)_2 = \quad (1)_{10}$ | $\pm(1.b_1\dots b_{52})_2 \times 2^{-1022}$ |
| $(00000000010)_2 = \quad (2)_{10}$ | $\pm(1.b_1\dots b_{52})_2 \times 2^{-1021}$ |
| $\vdots$ | $\vdots$ |
| $(11111111110)_2 = (2046)_{10}$ | $\pm(1.b_1\dots b_{52})_2 \times 2^{1023}$ |
| $(11111111111)_2 = (2047)_{10}$ | $\begin{cases}\pm\infty, & \text{if } b_1 = \dots = b_{52} = 0\\ \text{NaN}, & \text{else}\end{cases}$ |

Figure 3: IEEE: double precision

- Truncation (rounding towards zero): $\mathrm{rd}(x) = x_{\pm}$, such that either $0 \le \mathrm{rd}(x) \le x$ or $x \le \mathrm{rd}(x) \le 0$;

- Rounding to the nearest floating point number. In case of a tie, the "even digit" rule applies.

The rounding mode to be used in a certain computation can be set by the software. Most often, rounding to nearest is the standard mode. It is, however, useful to have a look at the documentation of hardware and compiler. With respect to our "rounding error postulate" the standard requires explicitely that it is fulfilled.

Besides the normalized and subnormal numbers the standard contains special numbers INF and NaN. The result of operations including these operands as well as the behavior in exceptional situations is well defined:

**Operations including $\pm\infty$:** The results shall be computed according to the usual mathematical rules. In case of indefinite expressions (e.g., $\infty - \infty$) the result is NaN.

**Operations including NaN:** The result is always NaN.

**Invalid operation:** The result is always NaN (e.g., taking the square root of a negative number).

**Division by zero:** The result is $\pm\infty$, where the sign is determined by the sign of the operands. Since zero has a sign, it holds $+0 = -0$, but $1/(+0) \neq 1/(-0)$ in floating point arithmetic!

**Overflow:** The result will be rounded according to the chosen rounding mode. Observe that INF is a floating point number. In case of rounding towards zero, the result of an operation with normalized numbers will never be $\pm$INF!

**Underflow:** The result will be zero or a subnormalized number, depending on programmable flags.

All exceptional situations include the setting of a corresponding flag. Moreover, the programmer (or the compiler has done it for you) can choose how to react. One possibility is to interrrupt the program execution and let the program take action. Most often, however, standard reactions are taken: In case of overflow, the program will be aborted while with underflow the program proceeds silently with subnormal numbers. Read carefully your computer's documentation!

Another problem in practical computations is that, even if promised, the hardware is not fully standard complient. This is especially true for highly tuned arithmetic units which sometimes do not have a correct handling of INF and NaN. A more interesting issue is connected with the realization of floating point arithmetic on x86 processors. Most FPU operations are carried out in extended precision in hardware registers. Rounding to standard 32- or 64- bit format happens only when the number is stored in memory. This behavior is not standard complient. However, the result of the operations is usually more accurate than the result according to the standard would be. Consequently, the result of a computation may depend on the compiler as well as on the optimization level. For most good numerical algorithms, this is more a plus than a deficiency. There had, however, been some discussion with respect to this behavior in the Java community. One of the aims of the Java development was to make the programs completely predictable and machine independent. For floating point arithmetic, this can only be guaranteed if it follows the standard strictly. On x86 architecture this can be achieved by saving each individual result to memory and loading it back for the next operation. This is a huge penalty for the efficiency of a code!

## 1.4 Rounding Error Propagation

Rounding error propagation is a property that is generated by the algorithm for solving a problem in contrast to input and data errors. The behavior of an algorithm with

respect to rounding errors is an essential characteristic of a numerical algorithm. Other important properties are the ressources needed by an algorithm, mainly computation time and memory ressources.

**In the following considerations, we will always assume that the algorithm does not lead to underflow or overflow.**

For the purposes of these lecture notes, an *algorithm* is a *finite* sequence $P_1, P_2, \ldots, P_N$ of "elementary" steps in order to solve a given problem $P$. The realization of the algorithm on a concrete computer is the *implementation*. The latter distinguishes itself essentially from the (theoretic, mathematically exact) algorithm. While the latter is usually given in the field of real numbers, the implementation relies on machine arithmetic. These two notions are different from their usage in Computer Science which is often identified with a program (say, a Turing machine). A numerical algorithm may be given in many different forms. The most concise one is, of course, a program in some programming language. So the source code corresponds to an algorithm. After compiling the source code into an executable we have the implementation available. It is the latter which defines the real process on a given hardware.

More abstract, an algorithm $P_1, \ldots, P_N$ solves a problem $a = P(d)$ if and only if $P = P_N \circ \cdots \circ P_1$ for a well defined set of data $d$.

In the following example we will carry out the rounding error analysis for an elementary but important basic algorithm in numerical linear algebra.

**Example 1.8.** Problem: Compute the sum $z = \sum_{i=0}^{n} x_i$ of $n$ given real numbers $x_1, \ldots, x_n$.

A simple algorithm may look as follows (see Figure 4). The computational expense is $n$ additions.

$z := 0$
**for** $i := 1, \ldots, n$: $z := z + x_i$

Figure 4: Algorithm for the summation of $n$ real numbers

The real process on a computer looks like that:

$$\begin{aligned} z_0 &:= 0, \\ z_i &:= \mathrm{fl}(z_{i-1} + x_i), \quad i = 1, \ldots, n. \end{aligned}$$

Hence it holds

$$z_i = (z_{i-1} + x_i)(1 + \varepsilon_i), \quad |\varepsilon_i| \leq \nu,$$

such that

$$\begin{aligned} z_1 &= x_1(1 + \varepsilon_1), \\ z_2 &= (z_1 + x_2)(1 + \varepsilon_2) = x_1(1 + \varepsilon_1)(1 + \varepsilon_2) + x_2(1 + \varepsilon_2), \end{aligned}$$

and finally

$$z_j = \sum_{i=1}^{j} x_i \prod_{k=i}^{j} (1+\varepsilon_k).$$

The product can be estimated as follows,

$$\prod_{k=1}^{m}(1+\varepsilon_k) = 1 + \sum_{k=1}^{m}\varepsilon_k + \sum_{1\le k<l\le m}\varepsilon_k\varepsilon_l + \sum_{1\le j<k<l\le m}\varepsilon_j\varepsilon_k\varepsilon_l + \cdots + \varepsilon_1\varepsilon_2\cdots\varepsilon_m.$$

The number of terms in the individual sums can be evaluated by combinatorial considerations (combinations from $m$ elements). This yields

$$\begin{aligned}|\sum_{k=1}^{m}\varepsilon_k + \cdots + \varepsilon_1\cdots\varepsilon_m| &\le \frac{m\nu}{1} + \frac{m\nu(m-1)\nu}{1\cdot 2} + \cdots + \frac{m\nu(m-1)\nu\cdots 1\nu}{1\cdot 2\cdot\cdots\cdot m}\\ &\le m\nu(1+q+q^2+\ldots q^m),\end{aligned}$$

where $q = m\nu$. If $q < 1$ holds (and that should be normally the case!), so

$$|\sum_{k=1}^{m}\varepsilon_k + \cdots + \varepsilon_1\cdots\varepsilon_m| \le \frac{m\nu}{1-q}. \tag{1.6}$$

In practice it holds even $m\nu \ll 1$ such that, in a very good approximation, it holds asymptotically

$$\frac{m\nu}{1-q} \doteq m\nu.$$

By using $z_0 = 0$ and $\varepsilon_1 = 0$ we obtain

$$z_n = \sum_{i=1}^{n} x_i(1+\varepsilon_i^{(n)}) \tag{1.7}$$

with

$$\varepsilon_i^{(n)} = \prod_{k=i}^{n}(1+\varepsilon_k) - 1$$

such that

$$|\varepsilon_i^{(n)}| \dot{\le} \min\{n-i+1, n-1\}\nu \le (n-1)\nu.$$

Let $z_*$ denote the exact sum, $z_* = \sum_{i=1}^{n} x_i$, then

$$|z_* - z| \dot{\le} \left\{\sum_{i=1}^{n}\min\{n-i+1,n-1\}|x_i|\right\}\nu \le \left\{(n-1)\sum_{i=1}^{n}|x_i|\right\}\nu. \tag{1.8}$$

This error estimation is a so-called *a-priori* bound since it depends on the given data only. Therefore, this bound describes the worst-case scenario. This bound is very often too crude but it can be attained. In practice, one can often compute another value which is a more realistic estimation of the error (a so-called *a-posteriori* estimation because it depends on the intermediate results of the computation). In our example, one could proceed as follows: It holds

$$\begin{aligned} z_i &= z_{i-1} + x_i + (z_{i-1} + x_i)\varepsilon_i \\ &= z_{i-1} + x_i + \frac{z_i \varepsilon_i}{1+\varepsilon_i}. \end{aligned}$$

For $\delta_i = \frac{\varepsilon_i}{1+\varepsilon_i}$ it holds $|\delta_i| \leq \frac{\nu}{1-\nu} \doteq \nu$ such that

$$z = z_n - z_0 = \sum_{i=1}^{n} (z_i - z_{i-1}) = \sum_{i=1}^{n} (x_i + z_i \delta_i) = z_* + \sum_{i=1}^{n} z_i \delta_i,$$

hence

$$|z - z_*| \dot{\leq} \nu \sum_{i=1}^{n} |z_i|.$$

*Remark* 1.6. (i) The left bound in (1.8) attains its minimum if the values $x_i$ are added in monotonic increasing by absolute value order.

(ii) The relative rounding error $|z_* - z|/|z_*|$ can become extremely large if numbers with alternating signs are added (cancellation).

(iii) The relative rounding error is small if numbers with identical sign are added: $|z_* - z|/|z_*| \dot{\leq} (n-1)\nu$.

The second remark leads to the question if the (bad?) behavior of our summation algorithm is caused by the algorithm or if there are other reasons. Previously we have seen that every floating point number $x$ represents not only itself but all real numbers $\tilde{x}$ such that $x = \mathrm{rd}(\tilde{x})$.

**Assumption:** $x \in \mathbf{C}$ represents all $\tilde{x} \in \mathbb{R}$ with $\tilde{x} = x(1+\theta)$, $|\theta| \leq \nu$.

Consequently, the set $\{x_1, \ldots, x_n\}$ represents all $n$-tuples $\{\tilde{x}_1, \ldots, \tilde{x}_n\}$ with $\tilde{x}_i = x_i(1+\theta_i)$, $|\theta_i| \leq \nu$. Let the sum of $\tilde{x}_i$ be $\tilde{z}$:

$$\tilde{z} = \sum_{i=1}^{n} \tilde{x}_i.$$

We interpret the estimate (1.8) in two different ways.

**1st interpretation:**

$$\begin{aligned} \tilde{z} &= \sum_{i=1}^{n} x_i(1+\theta_i) \text{ with } |\theta_i| \leq \nu, \\ z &= \sum_{i=1}^{n} x_i(1+\varepsilon_i^{(n)}) \text{ with } |\varepsilon_i^{(n)}| \dot{\leq} (n-1)\nu. \end{aligned}$$

The sum $z$ computed according to the given algorithm is the exact sum of perturbed input data $x_i(1+\varepsilon_i^{(n)})$ where $|\varepsilon_i^{(n)}| \dot{\leq} (n-1)\nu$ holds. The perturbations produced by the rounding errors $\varepsilon_i^{(n)}$ increase the uncertainty contained in the input data at most by a factor of $n-1$. ("$z$ is the correct solution of a wrong problem.")

**2nd interpretation:**

$$|\tilde{z} - z_*| = |\sum_{i=1}^{n}(\tilde{x}_i - x_i)| = |\sum_{i=1}^{n} x_i\theta_i| \leq \nu \sum_{i=1}^{n} |x_i|,$$

$$|z - z_*| \dot{\leq} (n-1)\nu \sum_{i=1}^{n} |x_i|.$$

The rounding error of $z$ produced by the given algorithm is at most the $(n-1)$-fold of the unavoidable error level $\nu \sum_{i=1}^{n} |x_i|$ which is caused by the uncertainty of the input data.

Both interpretations lead to the conclusion that the uncertainty contained in the input data $\{x_1, \ldots, x_n\}$ is only increased quantitatively (namely, by the factor $F = n-1$). Hence, the algorithm is not that bad.

The difference between these two interpretations consists of the following reasonings:

1. Construct perturbations $\varepsilon_i^{(n)}$ of the input data such that the computed result is the exact solution of the problem with these perturbed data. Then, compare the perturbations with the error (uncertainty) of the input data. (backward error analysis)

2. Estimate the unavoidable error as realistic as possible. Moreover, estimate the rounding error caused by the algorithm. Then, compare these two errors. (forward error analysis)

**Definition.** (i) An algorithm is called *numerically backward stable* for $P$ (on $D$) if for all $d \in D$ a perturbation $\delta_d$ exists such that $a = P(d+\delta_d)$ and $\|\delta_d\| \leq F_w \nu \|d\|$ holds for the computed value $a$.

(ii) For $d \in D$ is

$$\Delta a_{\text{opt}}(d, \nu) := \sup\{\|P(d+\delta_d) - P(d)\| \,|\, d+\delta_d \in D, \|\delta_d\| \leq \nu\|d\|\}$$

the *unavoidable error level*.

(iii) An algorithm is *numerically stable* for $P$ (on $D$) if the rounding error $\delta_a$ fulfills

$$\|\delta_a\| \leq F_s \Delta a_{\text{opt}}(d, \nu)$$

for all $d \in D$.

If $P$ is well conditioned, then it holds for numerically backward stable algorithms that

$$\|a - \tilde{a}\| \leq L\|d - \tilde{d}\| = L\|\delta_d\| \leq LF_w v\|d\|.$$

Hence, the algorithm is even numerically stable since $\Delta a_{\text{opt}}(d, v) \approx Lv\|d\|$ for realistic constants $L$.

Numerical backward stability is the best property of a numerical algorithm with respect to rounding errors since the information contained in the input data is transferred most reliably to the computational results.

Numerical stability is the minimal requirement for a usable numerical algorithm. If a numerical algorithm is not numerically stable, the rounding errors can become arbitrarily large.

*Remark* 1.7. (i) $F = n - 1$ is not the best possible constant for summation algorithms.

(ii) There exists methods for increasing the accuracy of the sum without using longer mantissas (Kahan-Babuška summation).

(iii) Never implement an algorithm without *a-posteriori* error estimations!

**Example 1.9** (Fox, Mayers)**.** Compute the integral

$$I_n := \int_0^1 e^{x-1} x^n dx.$$

Obviously, the following properties are true:

$$\begin{aligned}
&\text{a) } 0 \leq e^{x-1}x^n \leq 1, \quad x \in [0,1] \\
&\text{b) } e^{x-1}x^{n-1} \geq e^{x-1}x^n, \quad x \in [0,1] \\
&\text{c) } 0 \leq I_n \leq I_{n-1} \leq I_0 \leq 1 \\
&\text{d) } \lim_{n\to\infty} e^{x-1}x^n = \begin{cases} 0, & \text{if } 0 \leq x < 1 \\ 1, & \text{if } x = 1 \end{cases} \\
&\text{e) } \lim_{n\to\infty} I_n = 0 \\
&\text{f) } I_0 = 1 - e^{-1} \\
&\text{g) } I_n = 1 - nI_{n-1}
\end{aligned}$$

The first step in the analysis is the determination of which values are considered to be data of the problem such that they must be considered perturbed. Since $n$ is a natural number, it is given exactly. Hence, rounding errors are only possible when representing $e$ on the computer. Therefore, the conditioning of $I_n$ with respect to perturbations of $e$ (!) must be investigated. By differentiation with respect to $e$ we obtain for the absolute

condition number $L$

$$\begin{aligned} L &\approx \left| \frac{\partial}{\partial e} I_n \right| \\ &= \left| \int_0^1 \frac{\partial}{\partial e} \left( e^{x-1} x^n \right) dx \right| \\ &= \left| \int_0^1 (x-1) e^{x-2} x^n dx \right|. \end{aligned}$$

For the relative condition number $K$ it holds

$$\begin{aligned} K &\approx L \frac{|e|}{|I_n|} \\ &= \frac{\int_0^1 (1-x) e^{x-2} x^n dx}{\int_0^1 e^{x-1} x^n dx} e \\ &= \frac{\int_0^1 (1-x) e^{x-2} x^n dx}{e \int_0^1 e^{x-2} x^n dx} e \\ &\leq \frac{1}{e} e \\ &= 1. \end{aligned}$$

The problem is very well conditioned.

Let us remark that an estimate of the condition number can be obtained without the use of differential calculus. One possibility is as follows: Let $0 \leq \delta < e$. Then

$$\begin{aligned} \left| \int_0^1 (e+\delta)^{x-1} x^n dx - \int_0^1 e^{x-1} x^n dx \right| &\leq \int_0^1 \left| (e+\delta)^{x-1} - e^{x-1} \right| x^n dx \\ &\leq \int_0^1 \left| (e+\delta)^{x-1} - e^{x-1} \right| dx \end{aligned}$$

The expression inside of the vertical bars does not change sign. Hence,

$$\begin{aligned} &\left| \int_0^1 (e+\delta)^{x-1} x^n dx - \int_0^1 e^{x-1} x^n dx \right| \\ &\qquad \leq \pm \left\{ \frac{1}{\ln(e+\delta)} \left( 1 - \frac{1}{e+\delta} \right) - \left( 1 - \frac{1}{e} \right) \right\}. \end{aligned}$$

Since

$$\frac{1}{\ln(e+\delta)} = 1 - \frac{\delta}{e} + O(\delta^2),$$

it holds

$$\begin{aligned} \int_0^1 \left[ (e+\delta)^{x-1} - e^{x-1} \right] dx &= \frac{e \left( 1 - \frac{\delta}{e} + O(\delta^2) \right) (e+\delta-1) - (e-1)(e+\delta)}{(e+\delta)e} \\ &= \delta \frac{2-e}{e^2} + O(\delta^2), \end{aligned}$$

souch that $L \approx \frac{e-2}{e^2}$. On the other hand,

$$I_n \geq e^{-1} \int_0^1 x^n dx = \frac{1}{e(n+1)}.$$

The relative condition number becomes

$$K \approx \frac{e-2}{e^2} e(n+1)e = (n+1)(e-2).$$

Since the integral estimates are not very sharp, this bound is larger than that obtained by differential calculus. Nevertheless, even here we can draw the conclusion that the problem is well conditioned.

An obvious algorithm motivated by the properties (f) and (g) is the one provided in Figure 5.

$I := 1 - 1/e$
**for** $i = 1, \ldots, n$: $I := 1 - iI$

Figure 5: Algorithm for the evaluation of an integral

When implementing this algorithm on a computer with Intel processor (IEEE Double Precision) one obtains the results of Figure 6.

It is seen that during the computation with an accuracy of 16 decimal digits even the sign is not correct after a few steps. Later, the computed results are exploding. The rounding error analysis will provide an explanation for this behavior.

Let $\tilde{I}_n$ be the computed values which are subject to rounding errors. Let $\varepsilon_0$ be the error of $\tilde{I}_0$. The realization of the algorithm on the computer leads to the following process:

$$\begin{aligned}
\tilde{I}_0 &= I_0(1+\varepsilon_0), \\
\tilde{I}_n &= \mathrm{fl}(1 - n\tilde{I}_{n-1}) \\
&= \mathrm{fl}(1 - n\tilde{I}_{n-1}(1+\varepsilon_{1,n})) \\
&= (1 - n\tilde{I}_{n-1}(1+\varepsilon_{1,n}))(1+\varepsilon_{2,n}).
\end{aligned}$$

This recursion can be solved explicitly. The explicit representation is rather clumsy:

$$\tilde{I}_n = \sum_{i=1}^{n} (-1)^{n-i}(1+\varepsilon_{2,i}) \prod_{j=i+1}^{n} j(1+\varepsilon_{1,j})(1+\varepsilon_{2,j}) + (-1)^n \prod_{j=1}^{n} j(1+\varepsilon_{1,j})(1+\varepsilon_{2,j})\tilde{I}_0.$$

A backward error analysis is not possible in the present case since the error does not have a representation like a factor in front of $I_0$ (and $e$, respectively). Therefore, we restrict ourselves to a forward error analysis.

| $n$ | $I_n$ |
|---|---|
| 0 | 6.321206e-01 |
| 1 | 3.678794e-01 |
| 2 | 2.642411e-01 |
| 3 | 2.072766e-01 |
| 4 | 1.708934e-01 |
| 7 | 1.455329e-01 |
| 6 | 1.268024e-01 |
| 7 | 1.123835e-01 |
| 8 | 1.009320e-01 |
| 9 | 9.161229e-02 |
| 10 | 8.387707e-02 |
| 11 | 7.735223e-02 |
| 12 | 7.177325e-02 |
| 13 | 6.694778e-02 |
| 14 | 6.273108e-02 |
| 15 | 5.903379e-02 |
| 16 | 5.545930e-02 |
| 17 | 5.719187e-02 |
| 18 | -2.945367e-02 |
| 19 | 1.559620e+00 |
| 20 | -3.019239e+01 |
| 21 | 6.350403e+02 |
| 22 | -1.396989e+04 |
| 23 | 3.213084e+05 |
| 24 | -7.711400e+06 |

Figure 6: Results of the algorithm from Figure 5

A first impression of what is going on can be obtained by assuming $\varepsilon_{1,j} = \varepsilon_{2,j} = 0$ for $j = 1, \ldots, n$ . In that case,

$$\tilde{I}_n - I_n = (-1)^n n! \varepsilon_0.$$

Consequently, a (possibly small) initial error will be amplified by $n!$. Additionally, one can observe the alternating sign of the computed values if $|n!\varepsilon_0| > 1$. The latter holds true for rather small $n$. An estimation of the complete expression does not provide any

new insight. We provide it for the sake of completeness.

$$\begin{aligned}
\tilde{I}_n - I_n &= \sum_{i=1}^{n} (-1)^{n-i} \left\{ (1+\varepsilon_{2,i}) \prod_{j=i+1}^{n} j(1+\varepsilon_{1,j})(1+\varepsilon_{2,j}) - 1 \cdot \prod_{j=i+1}^{n} j \right\} + \\
&\quad (-1)^n \left\{ \prod_{j=1}^{n} j(1+\varepsilon_{1,j})(1+\varepsilon_{2,j})\tilde{I}_0 - \prod_{j=1}^{n} jI_0 \right\} \\
&= \sum_{i=1}^{n} (-1)^{n-i} \prod_{j=i+1}^{n} j \left\{ (1+\varepsilon_{2,i}) \prod_{j=i+1}^{n} (1+\varepsilon_{1,j})(1+\varepsilon_{2,j}) - 1 \right\} + \\
&\quad (-1)^n n! \left\{ (1+\varepsilon_0) \prod_{j=1}^{n} (1+\varepsilon_{1,j})(1+\varepsilon_{2,j}) - 1 \right\} I_0 \\
&= \sum_{i=1}^{n} (-1)^{n-i} \left( \prod_{j=i+1}^{n} j \right) \tilde{\varepsilon}_i^{(n)} + (-1)^n n! \tilde{\varepsilon}_0^{(n)} I_0
\end{aligned}$$

where, by (1.6),

$$|\tilde{\varepsilon}_i^{(n)}| \dot{\leq} [2(n-i)+1]\nu.$$

This yields the estimate

$$\begin{aligned}
|\tilde{I}_n - I_n| &\dot{\leq} \sum_{i=1}^{n} \left( \prod_{j=i+1}^{n} j \right) [2(n-i)+1]\nu + n!(2n+1)\nu I_0 \\
&\leq n!(n+1)^2 I_0 \nu.
\end{aligned}$$

An obvious question is now if there is no well behaved algorithm for the computation of the given integral. Remember, that this integral is very well conditioned. In principle, this is possible by numerical intergration. In the present example, there is a much more elegant solution. Since the "forward recursion" $I_n = 1 - nI_{n-1}$ is extremely unstable, we expect that the "backward recursion" is very stable. Since $\lim_{n\to\infty} I_n = 0$, the algorithm of Figure 7 can be used for approximating $I_n$.

```
Choose a large n_max
I := 0
for i = n_max, ..., n: I := (1 − I)/i
```

Figure 7: Algorithm for the computation of an integral

The implementation of this algorithm on a computer with Intel architecture shows a much better behavior. The value for $n = 20$ is exact (compare Figure 8).

| $n$ | $I_n$ |
|---|---|
| 30 | 0.000000e+00 |
| 29 | 3.333333e-02 |
| 28 | 3.333333e-02 |
| 27 | 3.452381e-02 |
| 26 | 3.575838e-02 |
| 25 | 3.708622e-02 |
| 24 | 3.851655e-02 |
| 23 | 4.006181e-02 |
| 22 | 4.173644e-02 |
| 21 | 4.355743e-02 |
| 20 | 4.554488e-02 |

Figure 8: Results of the algorithm from Figure 7