Lecture Notes

Fundamentals of Computing

Davoud Mirzaei

Uppsala University

June 10, 2024

Contents

This lecture addresses some general ideas behind numerical computations ranging from representation of numbers in computers to stability and accuracy of standard algorithms for some simple mathematical problems. Some parts of the lecture follow [**Dahlquist-Bjork:2008**], [**Higham:2002**], and [**Gautschi:2012**].

1 What is scientific computing?

Scientific computing is concerned with the design and analysis of algorithms for solving mathematical problems that arise in science and engineering. It is distinguished from most other parts of computer science in that it deals with quantities that are continuous, as opposed to discrete. Continuous quantities are functions and equations whose underlying variables (time, distance, velocity, temperature, density, pressure, stress, ...) are continuous in nature. This subject is also called *numerical analysis* or *computational mathematics* but nowadays it is mostly refereed to as *scientific computing*. Figure 1 shows that scientific computing could be viewed as the intersection of computer science, applied mathematics and science and engineering.

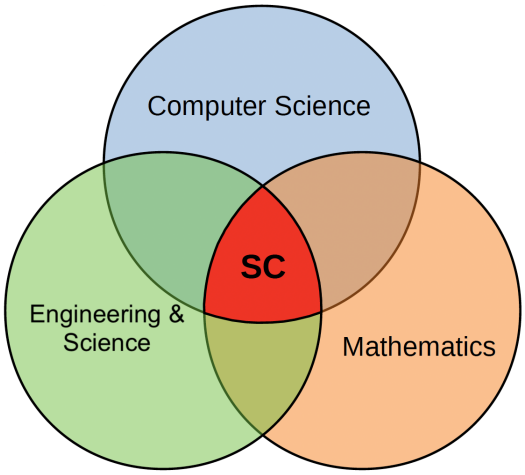


Figure 1: The land of scientific computing (SC)

Most problems in continuous mathematics, such as those involving derivatives, integrals, or nonlinearities, cannot be solved exactly (analytically) and must be addressed using approximate processes that eventually converge to numerical solutions. A key aspect of scientific computing is the development of convergent algorithms and the analysis of the accuracy of these approximations. Therefore, another critical factor in scientific computing is its focus on the effect of *approximations*. We summarize that the key distinguishing features of scientific computing include

- dealing with continuous quantities (e.g., time, distance, velocity, temperature, density, pressure) typically measured by real numbers, and
- considering the effects of approximations.

1.1 Computational simulation

The classic pair of opposed but mutually supporting scientific paradigms are *theory* and *experimentation*. A third paradigm, *computational simulation*, emerged through the work of John von Neumann and others in the mid-20th century[1]. Computational simulation involves representing and emulating physical systems or processes using computers.

Today, computation has become an equal and indispensable partner alongside theory and experiment in the pursuit of knowledge and technological advancement. Numerical simulation allows the study of complex systems and natural phenomena that would be too expensive, dangerous, or even impossible to study theoretically or through direct experimentation. For instance, in astrophysics, the detailed behavior of two colliding black holes is too complex to determine theoretically and impossible to observe directly or replicate in a laboratory. To simulate this scenario computationally, one needs an appropriate mathematical representation (such as Einstein's equations of general relativity), an algorithm to solve these equations numerically, and a sufficiently powerful computer to implement the algorithm. Similarly, in improving automobile safety, crash testing on a computer is far less expensive and dangerous than real-life testing. This allows for a more thorough exploration of potential design parameters, leading to the development of optimal designs.

The overall problem-solving process in computational simulation typically includes the following steps:

1. **Develop a mathematical model:** This involves formulating a mathematical representation, usually in the form of equations, of the physical phenomenon or system of interest.
2. **Develop algorithms:** Create numerical algorithms to solve these equations.
3. **Implement and execute the algorithms:** Convert the algorithms into computer software and run the simulations.
4. **Interpret and validate the results:** Analyze the computed results and verify their accuracy, repeating any or all of the preceding steps as necessary.

Step 1, known as mathematical modeling, requires specialized knowledge of the relevant scientific or engineering disciplines, as well as applied mathematics. Steps 2 and 3, which involve designing, analyzing, implementing, and using numerical algorithms and software, constitute the core focus of scientific computing.

As a simple and funny example, consider a bird family (mother bird and chicks) sitting on an elastic wire of length ℓ meters, fixed at both ends. Under some mild simplifying assumptions, the displacement of the wire from the horizontal line (denoted as y) satisfies a simple second-

[1]Jim Gray, a 1998 Turing Award winner and a leading computer scientist, proposed a "fourth paradigm" in scientific research in one of his last talks in 2007. This paradigm, data-intensive science, is a methodological approach to discovery based on data analysis, extending beyond theoretical and experimental research and computational simulations.

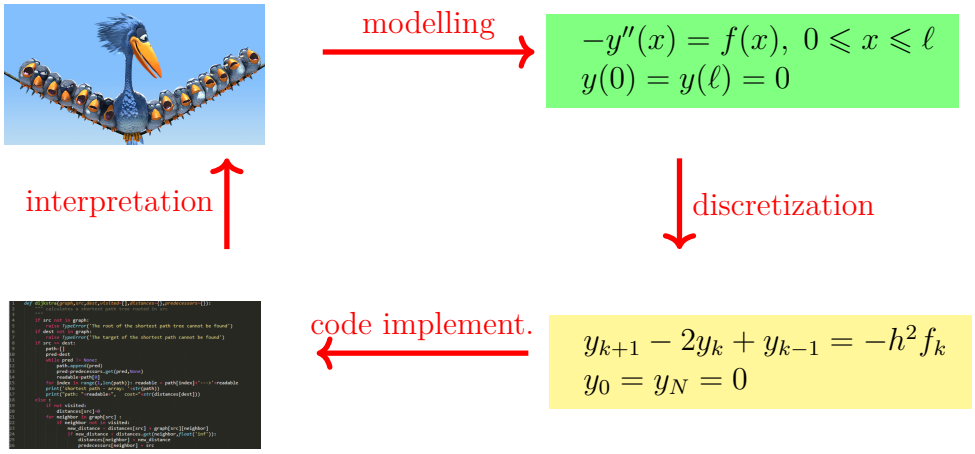


Figure 2: Problem solving steps: birds on the wire!

order ordinary differential equation (ODE) with boundary conditions (see Figure 2). To solve this, we can use the finite difference method (FDM) to discretize the underlying ODE and boundary conditions. In the third step, we implement the discretized equations in a computer program to obtain a numerical (approximate) solution, $y = (y_0, y_1, \ldots, y_N)$, at discrete points $x_0, x_1, \ldots, x_N$ in interval $[0, \ell]$. This numerical solution approximates the exact continuous solution $y(x)$ for $x \in [0, \ell]$. Finally, we interpret and validate our numerical solution by comparing it to the real phenomenon, i.e., the birds on the wire.

As a real-world problem, consider the simulation of the aerodynamics of a car to enhance its efficiency and reduce fuel consumption. This problem involves the mathematical modeling of two fundamental physical phenomena, fluid dynamics and solid mechanics. Specifically, it requires solving the Navier-Stokes equations for fluid flow (air) and the elasticity equations for the solid structure (car body). These equations are partial differential equations (PDEs) that describe the conservation of physical quantities such as momentum and energy. In practical scenarios, finding exact solutions to these PDEs using analytical techniques is often impossible due to their complexity. This is where computational methods come into play. For the fluid domain, the Finite Volume Method (FVM) is commonly employed, while the Finite Element Method (FEM) is typically used for the solid domain. To ensure the reliability of these numerical solutions, it is essential to analyze the algorithms in terms of stability, convergence, and computational complexity. Stability analysis ensures that the numerical solution behaves correctly as it progresses, while convergence analysis guarantees that the solution approximates the true solution as the computational parameters are refined. Computational complexity assesses the efficiency of the algorithm in terms of time and resources required.

Addressing such problems requires a deep understanding of mathematical theory, proficiency in computational techniques, and expertise in high-performance computing. These skills enable us to develop, implement, and optimize algorithms that deliver accurate and efficient solutions

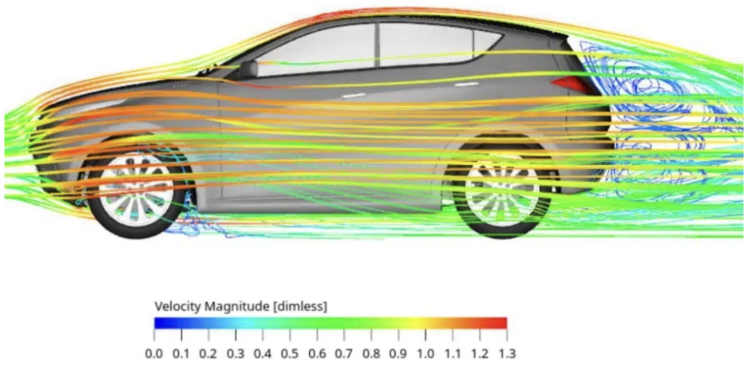


Figure 3: Simulation of the velocity filed and pressure of the air surrounding a car (image from `www.vias3d.com`)

to complex scientific and engineering problems. Figure 3 illustrates an example of such a simulation, showing the velocity field and its magnitude (represented by colors) for a sample car. This example not only demonstrates the practical application of numerical methods but also highlights the importance of interdisciplinary knowledge in scientific computing.

2 Sources of approximation

In the course of a numerical algorithm, computational results are influenced by various types of approximation errors. These errors can propagate from their sources to subsequently computed quantities, sometimes with significant amplification or damping. Some approximations may occur before a computation begins:

a. **Simplifications in the mathematical model.** When developing a mathematical model for a natural phenomenon, certain simplifying assumptions are often made to facilitate the modeling process. These assumptions help reduce complexity but may introduce minor deviations from real-world behavior. For example, in modeling a pendulum, we might assume the string is massless and ignore air resistance. In the heat conduction problem of a rod, we often assume the rod is composed of a homogeneous material. In economic calculations, we might assume that the interest rate remains constant over a given period. These assumptions are essential for making complex models manageable and solvable, but they do so at the cost of introducing small deviations from the actual physical solutions.

b. **Errors in input data.** Input data can often be the result of measurements that have been contaminated by various types of errors. The finite precision of laboratory instruments introduces systematic errors, while variations in the experimental environment lead to random errors in the input data. Additionally, input data may have been generated by a previous computational step, the results of which were only approximate.

These types of approximation errors are generally considered uncontrollable in numerical anal-

ysis. However, providing feedback to the developer of the mathematical model can sometimes be beneficial. Such communication can lead to refinements in the model and reduce the impact of these errors on the final results. In scientific computing, however, the focus is mostly on two other types of errors:

c. **Discretization errors** (sometimes calles **truncation error**),

d. **Rounding errors** (also called **roundoff errors**).

The main distinction between rounding and discretization errors is that rounding errors arise from *arithmetic* calculations, i.e., manipulation of *digits* of numbers (so they mainly have a computer arithmetic nature) while discretization errors stem from *algorithmic* calculations (so they mainly have a *mathematical* nature). Although the most parts of this lecture are devoted to understanding machine arithmetic and rounding errors, here we categorize some sources of discretization errors:

(a) replacing an infinite process by a finite approximation, for example,

- (discrete case) replacing an infinite series by a summation of a finite number of terms
- (continuous case) replacing an integral of a function by a finite summation of values of the function as in the trapezoidal or Simpson's rules.

(b) replacing an infinitesimal process by a finite approximation, e.g., replacing the limit in differentiation by a finite approximation

(c) truncation of an iteration that, in theory, should continue forever after a finite number of iterations in practice. This happens e.g., in iterative methods for solving

- nonlinear equations (rootfinding) like the Newton-Raphson method
- linear *systems* of equations like the Jacobi or Gauss-Seidel, SOR, conjugate gradient, etc.

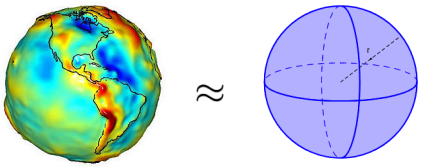
Doing these, the contribution of the remaining terms or iterations are not taken into account. This is why this type of error is also called the 'truncation error'.

Example 2.1. The surface area of the Earth cab be computed using the formula

$$A = 4\pi r^2$$

for the surface area of a sphere of radius r. The use of this formula for the computation involves a number of approximations:

- The Earth is modeled as a sphere, which is an idealization of its true shape.



- The value for the radius, $r \approx 6370$ km, is based on a combination of empirical mea-

surements and previous computations.

- The value for π is given by an infinite limiting process, which must be truncated at some point, e.g. $\pi \approx 3.1415$.
- The numerical values for the input data, as well as the results of the arithmetic operations performed on them, are rounded in a computer or calculator.

The accuracy of the computed result $A \approx 2.5146 \times 10^5 \text{km}^2$ depends on all of these approximations.

In the above example we observed the effects of modeling, input data and rounding errors. To give an example for discretization error, let us recall the Taylor's expansion theorem which is one of the most fundamental results throughout mathematics whose importance is well beyond simple understanding of the discretization errors[2].

Theorem 2.1. Let f be continuously differentiable up to order $n+1$ on the interval $[a,b]$ and $x_0 \in [a,b]$. Then, for every $x \in [a,b]$, there exists a point ξ between x_0 and x such that

$$f(x) = p_n(x) + r_n(x),$$

where

$$p_n(x) = f(x_0) + f'(x_0)(x-x_0) + \frac{f''(x_0)}{2!}(x-x_0)^2 + \cdots + \frac{f^{(n)}(x_0)}{n!}(x-x_0)^n,$$

and

$$r_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x-x_0)^{n+1}.$$

Here, p_n is referred to as the Taylor polynomial of degree n of function f around x_0, and r_n is the remainder associated with the polynomial p_n. Since $p_n(x)$ retains only the first $n+1$ terms of the infinite series, it is appropriate to view the remainder $r_n(x)$ as the discretization or truncation error corresponding to $p_n(x)$.

Example 2.2. Assume that $f(x) = \exp(x)$ and $x_0 = 0$. Since all derivatives of exponential function at zero are 1, the Taylor series expansion reads as

$$\exp(x) = \underbrace{1 + x + \frac{x^2}{2!} \cdots + \frac{x^n}{n!}}_{p_n(x)} + \underbrace{\frac{\exp(\xi)}{(n+1)!}x^{n+1}}_{r_n(x)}, \tag{2.1}$$

for a value ξ between x and 0.

Now, we use $n = 3$ and compute an approximation for $\sqrt{e}$ with an upper bound on the (truncation) error. We set $x = \frac{1}{2}$ and $n = 3$ in (2.1) to get

$$p_3(1/2) = 1 + \frac{1}{2} + \frac{1}{8} + \frac{1}{48} = \frac{79}{48}$$

as an approximation for $\sqrt{e}$. The corresponding discretization error is

$$r_3(1/2) = \frac{\exp(\xi)}{4!}(\frac{1}{2})^4 = \frac{\exp(\xi)}{384},$$

[2]Even in this course we will use Taylor's theorem later e.g., in the analysis of the discretization errors in derivation of numerical differentiation formulas and in the numerical solution of ODEs.

where $\xi \in (0, \frac{1}{2})$. Since exponential is an increasing function we have $\exp(\xi) < \exp(1/2) < 2$. Therefore, *an* upper bound for absolute value of discretization error is

$$|r_3(1/2)| < \frac{2}{384} \approx 0.0052.$$

Let us review what we did in the above example. The problem was to find an approximation for $\sqrt{e}$. The algorithm we used to solve the problem was the degree-3 Taylor polynomial of the exponential function. The discretization error was the result of replacing an infinite number of terms in that series with a polynomial of degree three only. In addition, note that we have *not* committed any rounding errors as long as we keep our approximation in the rational form $\frac{79}{48}$; the only error till now is the discretization error. However, rounding errors occur the moment we represent $\frac{79}{48}$ approximately e.g., as 1.6458. It can be shown that the first few digits of $\sqrt{e}$ are 1.648721270700128. So, we observe that the first three digits 1.64 of our approximation were indeed correct and that the amount of discretization error was indeed about 0.0029 which is, of course, smaller than the upper bound we computed.

2.1 Absolute and relative errors

Suppose that $\widehat{x}$ is an approximation for a real number x. Two ways to measure the amount of error in $\widehat{x}$ are the absolute error

$$|x - \widehat{x}|$$

and the relative error defined as

$$\frac{|x - \widehat{x}|}{|x|} \tag{2.2}$$

provided that $x \neq 0$. If we have both small and large quantities in a computation, it is the relative error that is more useful.

Since in most times the correct value x is not known, we cannot compute the exact amount of (absolute or relative) error. Instead, we either approximate the error or try to find a bound for it. For instance, in Example 1.5 we obtained an upper bound for the amount of (absolute) discretization error in approximating $\sqrt{e}$ by $\frac{79}{48}$. In a similar spirit, we will see later in the fundamental theorem of rounding errors that we can find an upper bound for the amount of (relative) error resulting from rounding a real number to a machine number.

If x is a vector then the absolute and the relative error are defined as

$$\|x - \widehat{x}\| \quad \text{and} \quad \frac{\|x - \widehat{x}\|}{\|x\|}$$

respectively, where $\|\cdot\|$ denotes some vector norm. For functions, norms defined on function spaces can be replaced.

3 Computer representation of numbers

To analyse the algorithms, it is essential to understand how machines store numbers, represent them, and perform computations. In everyday life, we use base-10 numbers, likely because

humans historically began counting with their fingers. Computers, however, use base-2 representation and binary arithmetic, as electrical devices typically distinguish between two states, e.g. lamps being off or on, or magnetic fields being clockwise or counterclockwise. However, here we discuss the representation of numbers in an arbitrary integer base $\beta \geq 2$.

3.1 Fixed-point representation of numbers

Given an integer base $\beta \geq 2$, for any real number x there exists $n \in \mathbb{N}$ such that x can be represented as a possibly infinite string like

$$\begin{aligned} x &= (-1)^{\sigma}(d_n\beta^n + d_{n-1}\beta^{n-1} + \cdots + d_0 + d_{-1}\beta^{-1} + d_{-2}\beta^{-2} + \cdots) \\ &=: \pm(d_n d_{n-1} \cdots d_0.d_{-1}d_{-2}\cdots)_\beta \end{aligned}$$

where $\sigma \in \{0, 1\}$ characterizes the sign of x and the *digits* d_k are integers in $\{0, 1, \ldots, \beta - 1\}$. This is the **position system** in which one can give simple and general rules for the arithmetic operations. In computers, real or integer numbers are typically stored using 32 or 64 bits, which are referred to as *word lengths*. In the first generation of computers, calculations were performed using a **fixed-point** number system. In this system, numbers have a fixed number of p digits in the fractional part, i.e.,

$$\widehat{x} = \pm(d_n d_{n-1} \cdots d_0.d_{-1}d_{-2}\cdots d_{-p})_\beta.$$

Let us denote the set of all representable numbers in this system by $F(\beta, n, p)$. The smaller the base is, the simpler these rules become. This is just one reason why most computers operate in base 2, the *binary* number system, where $d_k \in \{0, 1\}$. The addition and multiplication then take the following simple form:

$$0 + 0 = 0, \quad 0 + 1 = 1 + 0 = 1, \quad 1 + 1 = 10, \quad 0 \cdot 0 = 0, \quad 0 \cdot 1 = 1 \cdot 0 = 0, \quad 1 \cdot 1 = 1.$$

The digits in the binary system are called **bits** (**b**inary dig**its**). If the computer's word length is $s + 1$ bits (including the sign bit), then for $\beta = 2$, the range of representable numbers is $F(2, n, p) \subset [-2^{s-p}, 2^{s-p}]$. For example, if $s+1 = 64$ and $n = p = 31$, then $2^{s-p} \approx 4.2950 \times 10^9$. This interval is not large enough for many real-world applications. Scientists work with numbers of vastly different scales, from biologists dealing with microscopic quantities to astrophysicists measuring immense distances between celestial bodies. Thus, it is crucial to have a number system capable of handling both very small and very large numbers.

The fixed-point number system provides the same level of sensitivity to both small and large numbers. This sacrifices the efficiency of this system. For instance, the distance between 0 and its next representable number in $F(\beta, n, p)$ is identical with the distance between the largest and second-largest numbers in this system. This wastes many bits for representing unnecessary numbers that could be instead used to accommodate a wider range of numbers.

Workout 3.1. What is the number of numbers in $F(\beta, n, p)$. Determine the largest and smallest positive numbers in this system. Show the distance between two consecutive numbers in $F(\beta, n, p)$ is β^{-p} (the distance is independent of the magnitude of numbers).

3.2 Floating-point representation of numbers

Every real number x can be represented as

$$\begin{aligned} x &= (-1)^\sigma \ (d_0.d_1d_2d_3\cdots)_\beta \times \beta^e \\ &= (-1)^\sigma (d_0 + d_1\beta^{-1} + d_2\beta^{-2} + \cdots) \times \beta^e \end{aligned}$$

where $\sigma \in \{0, 1\}$, *digits* $d_k \in \{0, 1, \ldots, \beta - 1\}$, and *exponent* e is an integer. In computers we truncate the fraction and approximate x by

$$\widehat{x} = (-1)^\sigma \ (d_0.d_1d_2\cdots d_{p-1})_\beta \times \beta^e \tag{3.1}$$

with $d_0 \neq 0$ (for normalization) and exponent e limited by

$$L \leqslant e \leqslant U.$$

The string $d_0d_1\cdots d_{p-1}$ is called *mantissa* or *significand* and the portion $d_1\cdots d_{p-1}$ of the mantissa is called the *fraction*. Also, p is called the *precision* of the floating-point system. The key fact here is that in the floating point format the place of the base-β point can be changed by adjusting the exponent. The condition $d_0 \neq 0$ makes the representation unique. For example, among all of the following representations of the decimal number 123.4, e.g.

$$(1.234)_{10} \times 10^2 = (0.1234)_{10} \times 10^3 = (0.01234)_{10} \times 10^4,$$

the representation $(1.234)_{10}\times 10^2$ is permitted. The set of all real numbers that can be expressed in the form (3.1), with $d_0 \neq 0$, is denoted by $\mathbb{F}(\beta, p, L, U)$ and is referred to as the set of **normalized** floating-point (or machine) numbers. When the parameters are either fixed, known, or irrelevant to the context, this set is simply denoted by $\mathbb{F}$. Numbers in $\mathbb{F}$ are symmetric with respect to zero. The limited range of the exponent implies that $\widehat{x}$ is limited in magnitude to an interval which is called the *range* of the floating-point system. If $\widehat{x}$ is larger in magnitude than the largest number in the set $\mathbb{F}$, then $\widehat{x}$ cannot be represented at all. The same is true, in a sense, of numbers smaller than the smallest nonzero number in $\mathbb{F}$.

Example 3.1. We generate the full list of all normalized numbers in the set $\mathbb{F}(2, 3, -2, 1)$ and express each number in base 10. Such numbers are represented as $\pm(d_0.d_1d_2)_2 \cdot 2^e$ for $e = -2, -1, 0, 1$. Starting from the smallest exponent value $e = -2$, we obtain the following members:

$$\pm(1.00)_2 \times 2^{-2} = \pm(1 \times 2^0 + 0 \times 2^{-1} + 0 \times 2^{-2}) \times 2^{-1} = \pm\frac{4}{16} = \pm 0.25,$$

$$\pm(1.01)_2 \times 2^{-2} = \pm(1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2}) \times 2^{-1} = \pm\frac{5}{16} = \pm 0.3125,$$

$$\pm(1.10)_2 \times 2^{-2} = \pm(1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2}) \times 2^{-1} = \pm\frac{6}{16} = \pm 0.375,$$

$$\pm(1.11)_2 \times 2^{-2} = \pm(1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2}) \times 2^{-1} = \pm\frac{7}{16} = \pm 0.4375.$$

Moving to cases $e = -1$, $e = 0$ and $e = 1$ we obtain numbers $\{\pm\frac{8}{16}, \pm\frac{10}{16}, \pm\frac{12}{16}, \pm\frac{14}{16}\}$, $\{\pm\frac{16}{16}, \pm\frac{20}{16}, \pm\frac{24}{16}, \pm\frac{28}{16}\}$ and $\{\pm\frac{32}{16}, \pm\frac{40}{16}, \pm\frac{48}{16}, \pm\frac{56}{16}\}$, respectively. The positive side of this set is illustrated in Figure 4. As we can see, the smallest positive normalized number is $(1.00)_2\times$

$2^{-2} = 4/16 = 0.25$ and the largest number is $(1.11)_2 \times 2^{+1} = 56/16 = 3.5$.

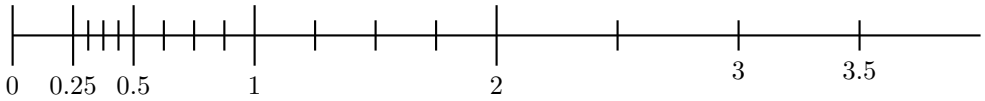


Figure 4: Positive normalized floating-point numbers in $\mathbb{F}(2, 3, -2, 1)$

Floating-point numbers are not uniformly distributed throughout their range, but are equally spaced only between successive powers of β. This means that the distance between consecutive points is the same for a fixed exponent e but increases as e is increased. The smallest and the largest positive numbers are denoted by x_{min} and x_{max}, respectively.

Workout 3.2. Count the number of normalized machine numbers in $\mathbb{F}(\beta, p, L, U)$. Determine x_{min} and x_{max} in this set.

Remark 3.1. For two arbitrary exponents e and e' with $L \leq e, e' \leq U$, the the cardinality of both sets $\mathbb{F}(\beta, p, L, U) \cap [\beta^e, \beta^{e+1}]$ and $\mathbb{F}(\beta, p, L, U) \cap [\beta^{e'}, \beta^{e'+1}]$ is the same. You can check this property in the toy example!

The floating-point representation of number 1 in base β is $1 = (1.00\cdots00)_\beta \times \beta^0$. The smallest machine number larger than 1 is equal to

$$(1.00\cdots01)_\beta \times \beta^0 = 1 \times \beta^0 + 0 \times \beta^{-1} + \cdots + 0 \times \beta^{-(p-2)} + 1 \times \beta^{-(p-1)} = 1 + \beta^{-(p-1)}.$$

The distance between these two successive floating-point numbers determines the precision of the machine and reflects the relative error of approximation in the system.

Definition 3.3. The distance between 1 and the smallest machine number larger than 1 is called **machine epsilon** and is denoted with ε_M. The value of machine epsilon is $\varepsilon_M = \beta^{-(p-1)}$.

The machine epsilon plays an important role in the analysis of rounding errors in numerical algorithms. The distance between *any* machine number $x \in \mathbb{F}(\beta, p, L, U)$ and its consecutive machine number is called **unit in the last place** and is denoted by $\text{ulp}(x)$. It states the weight of the last digit in the mantissa of the normalized number x. Without loss of generality, we only consider positive numbers; an analogous result holds for negative numbers. If x has representation $x = (d_0.d_1d_2\cdots d_{p-1})_\beta \times \beta^e$ and if all its digits are not $\beta - 1$, then the next number, say x^+, is

$$x^+ = (d_0.d_1d_2\cdots d_{p-1} + 0.00\cdots01)_\beta \times \beta^e = x + \beta^{-(p-1)} \times \beta^e.$$

This shows that

$$\text{ulp}(x) = \beta^{e-p+1} = \varepsilon_M \beta^e.$$

If all digits of x are $\beta - 1$ then $x^+ = (1.00\cdots0)_\beta \times \beta^{e+1}$. Now, x^+ is the first number in the next interval (with a new exponent; for example floating-point numbers 0.5, 1 and 2 in the toy example). In this case, we again can show that $\text{ulp}(x) = \beta^{e-p+1}$.

Note that if $x > 0$, then $\text{ulp}(x)$ is equal to the distance between x and the smallest machine number larger than x while if $x < 0$, then $\text{ulp}(x)$ is the distance between x and the largest machine number smaller than x. The concept of ulp is just a generalization of machine epsilon as $\varepsilon_M = \text{ulp}(1)$.

Example 3.2. In the toy example since the machine number 2.5 belongs to $[2^1, 2^2]$ which corresponds to exponent $e = 1$ we have $\text{ulp}(2.5) = 2^{1-3+1} = 0.5$. Similarly, the exponent of the machine number $x = 1.75$ is 0 thus $\text{ulp}(1.75) = 2^{0-3+1} = 0.25$. Similarly, $\text{ulp}(1) = \varepsilon_M = 0.25$.

Remark 3.2. Since computers use base 2 with digits $\{0, 1\}$, the only choice for the leading bit d_0 in the normalized binary floating-point numbers is $d_0 = 1$. This creates an opportunity to save some memory if we avoid occupying the bit d_0 with the fixed value of 1 by imposing it implicitly. This implicit bit whose value is always 1 for any normalized binary number is called the **hidden bit**.

3.3 Rounding modes

In computers, a real number x must be mapped in to (approximated by) a 'nearby' machine number before we can start any computation with it. More precisely, we need a map

$$\text{fl} : \mathbb{R} \to \mathbb{F}(\beta, p, L, U)$$

from the uncountable set of real numbers to the set of machine numbers. Such mappings are called *rounding modes*. The process of choosing a nearby floating-point number $\text{fl}(x)$ to approximate a given real number x is called *rounding*, and the error introduced by such an approximation is called *rounding error*, or *roundoff error*. Two commonly used rounding rules are **chopping** and **rounding to nearest**[3]. Assume that $x = \pm(d_0.d_1\cdots d_{p-1}d_p\cdots)_\beta \times \beta^e$. In chopping rule, the mantissa of x is truncated after the $(p-1)-$st digit to get

$$\text{fl}(x) = \pm(d_0.d_1\cdots d_{p-1})_\beta \times \beta^e.$$

Since $\text{fl}(x)$ is indeed the next floating-point number towards zero from x, this rule is sometimes called *round toward zero*. In rounding to nearest we have

$$\text{fl}(x) = \pm(d_0.d_1\cdots d_{p-2}\tilde{d}_{p-1})_\beta \times \beta^e,$$

[3]Two additional rounding modes are *rounding to up* (or round toward ∞) and *rounding to down* (or round toward $-\infty$) which are used in interval arithmetic. In interval arithmetic, numbers are represented by intervals (an upper and lower bound) rather than a single value. For example, π may be approximated by [3.141, 3.142].

where $\tilde{d}_{p-1}$ is either d_{p-1} or $d_{p-1}+1$ depending on to which one of the consecutive floating-point numbers x is closer. If x falls exactly midway then we have a *tie*. In this case x is rounded to the nearest floating-point number with an even last significant digit. This rule is called *round to nearest ties to even*.

Clearly, $\mathrm{fl}(x) = x$ if $x \in \mathbb{F}$. Moreover, $\mathrm{fl}(x) \leqslant \mathrm{fl}(y)$ if $x \leqslant y$ for all $x, y \in \mathbb{R}$ (monotonicity property). Since the distance between two consecutive machine numbers is $\varepsilon_M \beta^e$, the relative error of chopping can be bounded as

$$\frac{|\mathrm{fl}(x) - x|}{|x|} \leqslant \frac{\varepsilon_M \beta^e}{(d_0.d_1 d_2 \cdots)_\beta \beta^e} \leqslant \frac{\varepsilon_M}{(1.0)_\beta} = \varepsilon_M.$$

Similarly, for rounding to nearest we have

$$\frac{|\mathrm{fl}(x) - x|}{|x|} \leqslant \frac{\varepsilon_M}{2}.$$

Note that, in a floating-point system both large and small numbers are represented with nearly the same relative precision. The relative roundoff error for rounding to nearest is half of that for chopping, the main reason rounding to nearest is the default rounding rule in all standard systems although it is more expensive to implement. From here on, by 'rounding' we mean 'rounding to nearest'.

Theorem 3.4. In the floating-point number system $\mathbb{F}(\beta, p, L, U)$ every real number in the floating-point range can be represented with a relative error, which does not exceed the **unit roundoff** u, which is defined by

$$u = \begin{cases} \varepsilon_M/2, & \text{if rounding is used} \\ \varepsilon_M, & \text{if chopping is used} \end{cases}$$

The quantity u is a natural unit for relative changes and relative errors. For example, termination criteria in iterative methods usually depend on the unit roundoff. The following theorem is the fundamental theorem of rounding errors.

Theorem 3.5. If $x \in \mathbb{R}$ is such that $|x| \in [x_{min}, x_{max}]$, then

$$\mathrm{fl}(x) = x(1+\delta), \quad |\delta| \leqslant u. \tag{3.2}$$

The proof of this theorem follows the error bounds on chopping and rounding to nearest mappings. The fundamental rule (3.2) will be used to analyze the effect of initial and intermediate rounding errors in the final result of an algorithm.

3.4 Subnormal numbers

Recall the toy example once more. Due to the normalization requirement $d_0 \neq 0$, there is a relatively large gap between zero and the smallest positive number. Additionally, zero

cannot be represented by the normalized floating-point representation (3.1). These have an unfortunate impact on the validity of some of the most important algebraic properties when performing arithmetic with machine numbers in $\mathbb{F}$. For instance, in the current scenario, there could be two different normalized machine numbers, x and y, such that $x - y$ falls within this gap and is consequently approximated by zero! A remedy for this situation is to allow the leading digit d_0 to be zero but only when the exponent is at its minimum value L. Then the gap around zero can be filled in by additional floating-point numbers which are called **subnormal** or denormalized numbers. A subnormal floating-point number is of the form

$$\widehat{x} = \pm(0.d_1 d_2 \cdots d_{p-1})_\beta \beta^L, \quad d_k \in \{0, 1, \ldots, \beta - 1\}.$$

The fixed exponent $e = L$ makes this representation unique.

Example 3.3. For the toy example subnormal numbers are

$$\pm(0.00)_2 2^{-2} = \pm 0, \quad \pm(0.01)_2 2^{-2} = \pm\frac{1}{16}, \quad \pm(0.10)_2 2^{-2} = \pm\frac{2}{16}, \quad \pm(0.11)_2 2^{-2} = \pm\frac{3}{16}.$$

The non-negative normalized and denormalized numbers are depicted in Figure 5.

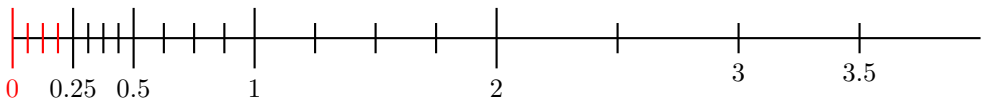


Figure 5: Non-negative normal and subnormal floating-point numbers in $\mathbb{F}(2, 3, -2, 1)$

Workout 3.6. Count the number of subnormal floating-point numbers in $\mathbb{F}(\beta, p, L, U)$. Determine the distance between two successive subnormal numbers.

The introduction of subnormal numbers guarantees that the subtraction of (nearby) floating-point numbers (with the same sign or the addition of floating-point numbers with opposite signs) never gives zero, i.e., the essential relation

$$x = y \Leftrightarrow x - y = 0$$

is now valid for $x, y \in \mathbb{F}$. This is caused by the fact that subnormal numbers can represent the non-zero distance between two nearby floating-point numbers. Several examples of how subnormal numbers make writing reliable floating-point code easier are analyzed in [**Demmel:1984**].

Remark 3.3. Even with subnormal numbers, the validity of the formula

$$x = y \Leftrightarrow x - y = 0$$

is not guaranteed if $x, y \notin \mathbb{F}$.

Note that, subnormal numbers have inherently lower precision than normalized numbers because they have fewer significant digits in their fractional parts.

Remark 3.4. If x is the result of an operation on numbers of $\mathbb{F}$ and $x \in (-\infty, -x_{max}) \cup (x_{max}, \infty)$ then $\mathrm{fl}(x)$ can not be defined. On the other side, if $x \in (-x_{min}, x_{min})$, the operation of rounding is defined anyway (even in absence of subnormal numbers). The first case is referred to **overflow** and the second case to **underflow**. The values of x_{min} and x_{max} are called the underflow level (UFL) and the overflow level (OFL), respectively.
In the presence of subnormal numbers, we can think of **gradual underflow** instead of underflow itself. Because if x is approximated by a subnormal number with representation $\mathrm{fl}(x) = \pm(0.0\cdots 0 d_{p-k}\cdots d_{p-1})_\beta \beta^L$ with $d_{p-k} \neq 0$ for an integer k with $1 \leqslant k < p$, then

$$\frac{|\mathrm{fl}(x) - x|}{|x|} \leqslant \frac{1}{2}\frac{\beta^{L-(p-1)}}{(0.0\cdots 010\cdots 0)_\beta \beta^L} = \frac{1}{2}\frac{\beta^{-(p-1)}}{\beta^{-(p-k)}} = \frac{1}{2}\beta^{-k+1}.$$

This shows that the relative roundoff error increases *gradually* as k (the number of significant digits of x) decreases, or equivalently as $\mathrm{fl}(x)$ approaches zero. Remember that in the absence of subnormal numbers we have underflowing to zero as soon as x falls into the underflow region.

3.5 IEEE standard for floating point arithmetic

In 1941, Konrad Zuse built the first computer in Berlin, which was called Z3. It was an electro-mechanical computer with a binary ($\beta = 2$) system for machine numbers occupying a total of 22 bits. Z3 was destroyed in a bombardment of Berlin during the World War II. It is often said that the first fully electronic computer was ENIAC[4], built between 1943 and 1945 at the University of Pennsylvania[5]. Subsequently, more efficient and smaller computers were developed. During 1960-1980, floating point computation was used as a basis of scientific computation. However, each computer manufacturer developed its own floating point system. See Table 1 for a few examples.

Table 1: Different formats on different computers

computer	β	p	$U = -L$
IBM 7090	2	27	2^7
Borroughs 5000 Series	8	13	2^6
IBM 360/370	16	6	2^6
DEC 11/780 VAX	2	24	2^7
Hewlett Packard 67	10	10	99

Due to differences between various implementations of floating-point systems, computer programs were hardly portable and often yielded different results when running the same code on different machines. A program that produced accurate results on one computer might not even

[4]Electronic Numerical Integrator and Computer
[5]January 15 is known as ENIAC Day and is celebrated annually in the United States.

run on another. These discrepancies were caused not only by variations in parameters such as β, p, L, and U, but also by differences in the implementation details of floating-point arithmetic. To address these issues, in a great cooperation between academic computer scientists and hardware designers, a standard for binary floating point representation and arithmetic was developed in 1985 which was supported by the Institute of Electrical and Electronics Engineers (shortly, IEEE). The standard was called *IEEE 754*. In 1985, a second standard was established by IEEE (called IEEE 854) for both decimal and binary floating-point systems[6].

The IEEE standard specifies two basic representation formats, *single* or fp32, and *double* or fp64. The general structure of the two data types single and double are the same and the only difference is in the number of bits used to represent the mantissa and the exponent. See Figure 6.

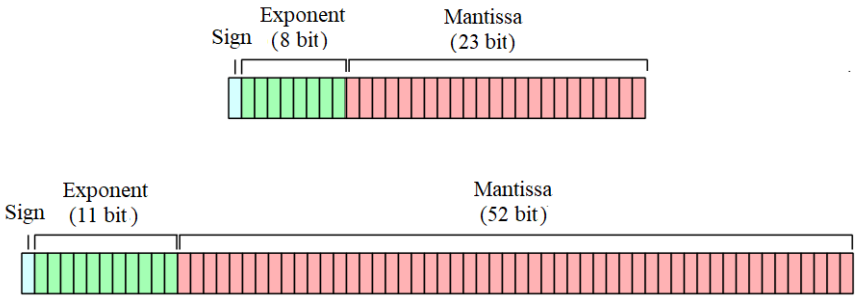


Figure 6: Number of bits in single (top) and double (down) precisions in the IEEE standard.

In the single data type (fp32), there are a total of 32 bits, the first one of which is called the sign bit and represents the sign of the number, the next 8 bits correspond with the exponent of the number, and the remaining 23 bits are used to represent the mantissa. Following the idea of the *hidden bit*, the precision of the single format is therefore equal to $p = 23 + 1$. These numbers are increased to 11 bits for exponent, 52 bits for mantissa with precision $p = 52 + 1$ for the double format. A *biased exponent* is stored and no sign bit used for the exponent which prevent wasting one bit to represent the sign of the exponent. In single precision, $L = -126$ and $U = 127$, and $e + 127$ is stored in eight bits. This range includes 254 integers, which is two fewer than the total of $2^8 = 256$ numbers that can be represented with eight bits. The other two exponents, -127 and $+128$, are reserved for special numbers. In double precision, $L = -1022$, $U = 1023$ and the biased exponent $e + 1023$ is stored in 11 bits. Again two extra representable exponents -1023 and 1024 are reserved for storing special numbers.

The IEEE standard includes *extended single* and *extended double* formats that offer extra precision and exponent range. The characteristics of all formats are summarized in Table 2.

As the double precision satisfies the requirements for the extended single format, so three precisions single, double and extended double suffice. Extended double format can for instance

[6]The leader of the academic computer scientists who developed the standard was William Kahan from the University of California, Berkeley. He received the ACM (Association for Computing Machinery) Turing Award in 1989 for his contributions to this standard.

Table 2: Single and double and extended precision IEEE formats

Format	Bits No.	p	e	L	U
Single	32 bits	24	8 bits	-126	127
Extended single	43 bits	32	11 bits	-1022	1023
Double	64 bits	53	11 bits	-1022	1023
Extended double	79 bits	64	15 bits	$-16,382$	$16,383$

be used in intermediate calculations for computing of elementary functions accurately in the double precision.

In IEEE standard, an exponent $e = L - 1$ (for example $e = -127$ in single precision) and a nonzero mantissa corresponds to a subnormal number $\pm(0.d_1 \cdots d_{p-1})_2 2^L$. There are distinct representations for $+0$ and -0 with exponent $e = L - 1$, and a zero mantissa[7]. This means that the representation of $+0$ in the single precision is

$$0|00000000|00000000000000000000000$$

Note that the biased exponent $e + 127 = -127 + 127 = 0$ is stored in eight exponent bits.

Infinity (overflow) is also signed and $\pm\infty$ is represented by the exponent $e = U + 1$ and a zero mantissa. It is also obtained from operation $x/0$ for $x \neq 0$. The IEEE infinity obeys the mathematical rules $\infty + \infty = \infty$, $(-1) \times \infty = -\infty$ and $x/\infty = 0$.

The standard also introduces special numbers called `NaN` (Not a Number) for undefined floating point operations like $0/0$, $\infty - \infty$ and $0 \times \infty$. A `NaN` is stored with exponent $U + 1$ and a nonzero mantissa. When a `NaN` and an ordinary floating-point number are combined the result is another `NaN`. A `NaN` is also often used for uninitialized or missing data. Table 3 summarizes all possible representations in IEEE 754 standard.

Table 3: IEEE 754 representation. Here $m = d_1 \cdots d_{p-1}$ is mantissa

Type of Number	Exponent	Mantissa	Represents
Zero	$e = L - 1$	$m = 0$	± 0
Subnormal	$e = L - 1$	$m \neq 0$	$\pm(0.m)_2 2^L$
Normal	$L < e < U$	$m \neq 0$ or $m = 0$	$\pm(1.m)_2 2^e$
Infinity	$e = U + 1$	$m = 0$	$\pm\infty$
Not a Number	$e = U + 1$	$m \neq 0$	`NaN`

Example 3.4. To represent the decimal number 89.75 in IEEE standard single format, we start by converting it to the binary system. We have $89 = (1011001)_2$ and $0.75 = (0.11)_2$. This gives

$$89.75 = (1011001.11)_2 = (1.01100111)_2 \times 2^6.$$

This number is a positive normalized floating-point number (with the hidden bit $d_0 = 1$), an

[7]The signed zero distinguishes between positive and negative underflowed numbers. Another use of singed zero is in the computation of complex elementary functions.

exponent $e = 6$, and a mantissa $m = 01100111$. We store the biased exponent $e+127 = 133$, which in binary is $(10000101)_2$. Thus, the final IEEE 754 representation is

$$0|10000101|01100111000000000000000$$

Note that the 14 remaining bits in the mantissa are filled with zeros.

Workout 3.7. Which decimal numbers do these two floating-point single precision arrays represent?

(a) 1|01011001|01110100000000000000000

(b) 1|00000000|01110100000000000000000

The machine epsilon, the smallest positive normalized number and the largest normalized number in the single precision are

$$\varepsilon_M = 2^{-(p-1)} = 2^{-23} \approx 1.19 \times 10^{-7}$$
$$x_{min} = (1.00\cdots0)_2 2^{-126} \approx 1.18 \times 10^{-38}$$
$$x_{max} = (1.11\cdots1)_2 2^{+127} \approx 3.40 \times 10^{+38}.$$

These numbers are refined to

$$\varepsilon_M = 2^{-52} \approx 2.22 \times 10^{-16}$$
$$x_{min} = (1.00\cdots0)_2 2^{-1022} \approx 2.23 \times 10^{-308}$$
$$x_{max} = (1.11\cdots1)_2 2^{+1023} \approx 1.80 \times 10^{+308}$$

in the double precision format. The double data type (fp64) is the default format in several software for scientific computing.

In Python the following commands print all machine parameters in double precision.

```
import numpy as np
print(np.finfo(float))
```

The output is:

```
Machine parameters for float64
---------------------------------------------------------------
precision =  15   resolution = 1.0000000000000001e-15
machep =    -52   eps =        2.2204460492503131e-16
negep =     -53   epsneg =     1.1102230246251565e-16
minexp =  -1022   tiny =       2.2250738585072014e-308
maxexp =   1024   max =        1.7976931348623157e+308
nexp =       11   min =        -max
---------------------------------------------------------------
```

Here, `eps` stands for machine epsilon, `epsneq` for unit roundoff, `tiny` for the smallest positive normalized numbers, `max` for the largest normalized number, `nexp` for number of allocated bits for the exponent, `minexp` for L and `maxexp` for $U + 1$. The decimal `precision` 15 is reported here instead of the binary precision 53. To access the variables, for example to access the unit roundoff, we can write

```
u = np.finfo(float).epsneg
```

To observe the corresponding parameters for the single precision format write

```
print(np.finfo(np.float32))
```

Note that the comments above are provided in `numpy` Python library as we imported this library with abbreviation `np` in the first input line. The ulp of a number can be obtained by command `spacing` from `numpy` module or the command `ulp` from `math` module.

```
In [1]: import numpy as np
In [2]: import math
In [3]: math.ulp(5)
Out[3]: 8.881784197001252e-16
In [4]: np.spacing(5)
Out[4]: 8.881784197001252e-16
In [5]: math.ulp(1)
Out[5]: 2.220446049250313e-16
In [6]: math.ulp(0)
Out[6]: 5e-324
```

As we observe, `math.ulp(1)` is another command to access the machine epsilon. Also, `math.ulp(0)` gives the smallest positive subnormal representable floating-point number[8].

3.6 Disasters caused by inappropriate use of floating point arithmetic

Even though the rounding errors in individual operations are typically small, their accumulation in complicated algorithms can lead to significant errors with potentially disastrous consequences. Here are a few historical events illustrating such cases.

Patriot missile failure in 1991 due to rounding errors[9]: On February 25, 1991, during the Iraq and Kuwait war, an American Patriot missile battery in Dharan, Saudi Arabia, failed to intercept an incoming Iraqi Scud missile. The Scud struck an American Army camp and killed 28 soldiers. The Patriot missile system was designed to track incoming missiles, predict

[8] See also `sys` module for these and other system-specific parameters and functions.

[9] Source: Robert D. Skeel. Roundoff error and the patriot missile. SIAM News, 25(4): 11, Jul. 1992.

their paths, and intercept them. However, it failed due to an inaccurate calculation of the time when the Patriot should have been launched. The time was tracked by the system's internal clock in tenths of a second, which was then multiplied by 1/10 to convert it to seconds. This calculation was done using a 24-bit register. The value 1/10, which has a non-terminating binary expansion, was rounded to 24 bits after the radix point. Although the rounding error seemed negligible, it accumulated over time and led to a significant error. By the time the Patriot battery had been operational for about 100 hours, and an easy calculation shows that the resulting time error due to the rounding error was about 0.34 seconds. The binary expansion of 1/10 is

$$\begin{aligned}(0.1)_{10} &= 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + 2^{-12} + 2^{-13} + \cdots \\ &= (1.10011001100\cdots)_2 \times 2^{-4} = (1.100\overline{1100})_2 \times 2^{-4}.\end{aligned}$$

The 24-bit register stored this value as 0.00011001100110011001100, introducing an error of approximately 0.000000095 in decimal. When multiplied by the number of tenths of a second in 100 hours, this error became $0.000000095 \times 100 \times 60 \times 60 \times 10 = 0.34$. A Scud missile travels at about 1676 meters per second, so travels more than half a kilometer in 0.34 seconds. This error was sufficient to place the incoming Scud outside the "range gate" that the Patriot missile was tracking.

Figure 7: Ariane 5 rocket (left), explosion after lift off (right). Photos from `https://www.esa.int`.

Explosion of the Ariane 5 rocket in 1996 due to an overflow error[10]: On June 4, 1996, an Ariane 5 rocket launched by the European Space Agency exploded just 40 seconds after lift-off. This rocket was on its maiden voyage following a decade of development costing $7 billion. The destroyed rocket and its cargo were valued at $500 million. It turned out that the cause of the failure was an overflow error in the inertial reference system. Specifically a 64 bit floating point number relating to the horizontal velocity of the rocket with respect to the platform was converted to a 16 bit signed integer. The number was larger than 32,768, the

[10]Source: M. Dowson, The Ariane 5 Software Failure, ACM SIGSOFT Software Engineering Notes. 22 (1997): 84.

largest integer storeable in a 16 bit signed integer, and thus the conversion failed.

3.7 Algebraic properties of floating point arithmetic

The set of real numbers is a *field* which makes the exact manipulation and analysis in $\mathbb{R}$ perfectly easy. However, floating-point addition and multiplication are commutative but not associative, and the distributive law also fails for them. The same holds true for subtraction and division. This makes the analysis of floating-point computations quite difficult. Here we give some examples to illustrate the situation.

Example 3.5. Let $x = 0.5, y = 2.5$ and $z = 0.75$ in the toy example $\mathbb{F}(2, 3, -2, 1)$ with rounding to nearest ties to even. We then have

$$\mathrm{fl}(0.5 + \mathrm{fl}(1.5 + 0.75)) = \mathrm{fl}(0.5 + \mathrm{fl}(2.25)) = \mathrm{fl}(0.5 + 2) = \mathrm{fl}(2.5) = 2.5,$$
$$\mathrm{fl}(\mathrm{fl}(0.5 + 1.5) + 0.75) = \mathrm{fl}(\mathrm{fl}(2) + 0.75) = \mathrm{fl}(2 + 0.75) = \mathrm{fl}(2.75) = 3,$$

which shows that the floating-point addition is not associative.

Example 3.6. Let $x = 2, y = \varepsilon_M$, and $z = -2$ in IEEE double precision. Let us compare $(x + y) + z$ and $x + (y + z)$ by doing the operations in Python:

```
In [1]: import math
In [2]: eps = math.ulp(1)
In [3]: (2 + eps) - 2
Out[3]: 0.0
In [4]: 2 + (eps - 2)
Out[4]: 2.220446049250313e-16
```

Following the definition of the machine epsilon and because the distance between any two consecutive machine numbers in interval $[1, 2]$ is the same, we conclude that the distance between any two consecutive machine numbers in $[1, 2]$ is `eps`. In addition, the distance between consecutive machine numbers $[2, 4]$ is 2`eps`, i.e., the smallest machine number larger than 2 is `2+2eps`. Since `2+eps` is not a machine number, we have to round it to the nearest machine number. `2+eps` is right in the middle of 2 and `2+2eps` and since 2 is an even number (the last bit of its mantissa in base two is zero) and so `2+2eps` is an odd number, we understand why we should get `(2+eps) = 2`. That is basically why we got zero as the result of `(2+eps) - 2`. Furthermore, because the set of machine numbers is symmetric with respect to zero, the distance between consecutive machine numbers from interval $[-2, -1]$ is `eps`. Therefore, the exact value of `eps-2` is somewhere in the interval $[-2, -1]$ where the gap between consecutive machine numbers is `eps`. This means that `eps-2` is actually a machine number! So, there is no rounding errors in computing $y + z$. That is why we got `eps` when we add $x = 2$ to $(y + z)$.

Workout 3.8. Show with some examples that (a) floating point multiplication is not necessarily associative, (b) floating point multiplication is not necessarily distributive over floating point addition, i.e. there exist $x, y, z \in \mathbb{F}$ such that $\mathrm{fl}(x\mathrm{fl}(y+z)) \neq \mathrm{fl}(\mathrm{fl}(xy)+\mathrm{fl}(xz))$. Also, (c) there exist $x, y, z \in \mathbb{F}$ such that $\mathrm{fl}(x+y) = \mathrm{fl}(x+z)$ while $y \neq z$, and (d) there exist $x, y, z \in \mathbb{F}$ such that $\mathrm{fl}(xy) = \mathrm{fl}(xz)$ while $y \neq z$. Property (c) for $y = 0$ and property (d) for $y = 1$ are valid showing that addition and multiplication neutral elements are not unique in floating-point arithmetic.

These all show that the order of floating-point operations could affect the accuracy of the result. We will see some examples in analysis of algorithms in section 5.

3.8 Floating-point arithmetic models

The set of real numbers is closed under the four basic arithmetic operations while the set of machine numbers is not. More precisely, if $x, y \in \mathbb{F}$ and $* \in \{+, -, \times, /\}$, then it may happen that $x * y \notin \mathbb{F}$. For example, consider the operation of dividing 1 by 3 on a machine whose representation base β is either 2 or 10. We know that the exact value of $1/3$ has infinitely many digits in both binary and decimal representations. Thus the results need to be rounded to a floating-point number. The rounding rule (3.2) suggests the model

$$\mathrm{fl}(x * y) = (x * y)(1+\delta), \quad |\delta| \leqslant u, \quad * \in \{+, -, \times, /\} \tag{3.3}$$

for normalized numbers $x, y \in \mathbb{F}$ provided that $x * y$ is in the normalized range. Sometimes the floating-point computation is more precise than what the model (3.3) assumes. An obvious example is that when $x * y \in \mathbb{F}$, there is no rounding error at all. Among elementary functions, the roundoff error introduced in computing $\sqrt{x}$ in IEEE standard obeys the same model

$$\mathrm{fl}(\sqrt{x}) = \sqrt{x}(1+\delta), \quad |\delta| \leqslant u, \quad x \in \mathbb{F}.$$

It is shown in [**Higham:2002**] that the model (3.3) is valid for addition and subtraction operators if the arithmetic is supported by the **guard digit** for subtraction. Otherwise a weaker model holds true. The role of guard digit can be easily explained by the following simple example from the mentioned book. Consider a floating-point arithmetic system with base $\beta = 2$ and precision $p = 3$. Let us subtract from 1 the next smaller floating-point number:

$$\begin{array}{rcr} 1.00 \times 2^0 & & 1.00 \times 2^0 \\ -1.11 \times 2^{-1} & \Longrightarrow & -0.111 \times 2^0 \\ \hline & & 0.001 \times 2^0 \end{array} = 1.00 \times 2^{-3}$$

Since for subtraction we have to scale both numbers to the same exponent (the larger one), a third digit is introduced in the mantissa of the second number. This digit, if not dropped, is known as *guard digit*. As the same as some old machines, let us do the subtraction without

the guard digit:

$$\begin{array}{rcrl} 1.00 \times 2^0 & & 1.00 \times 2^0 & \\ -1.11 \times 2^{-1} & \Longrightarrow & -0.11 \times 2^0 & \text{(last digit dropped)} \\ \hline & & 0.01 \times 2^0 & = 1.00 \times 2^{-2} \end{array}$$

The computed solution is twice, and so has relative error 100%. The lack of a guard digit is a serious drawback. Almost all modern processors use the guard digit.

Workout 3.9. Assume that x and y are floating-point numbers in base 2 with $y/2 \leqslant x \leqslant 2y$. Show that $\mathrm{fl}(x-y) = x - y$ provided that the guard digit is supported and $x - y$ does not underflow.

Some computers perform a **fused multiply-add** (FMA) operation that enables expressions like $(x * y) \pm z$ (a floating-point operation followed by an addition/subtraction) to commit just one rounding error[11]

$$\mathrm{fl}(x * y \pm z) = (x * y \pm z)(1+\delta), \quad |\delta| \leqslant u$$

for $x, y, z \in \mathbb{F}$. This capability enables the number of rounding errors in many algorithms to be approximately halved. For example the inner product $s = x^T y$ between two vectors $x, y \in \mathbb{F}^n$ can be computed with just n rounding errors instead of the usual $2n-1$. See the code below.

```
def InnerProd(x,y):
    s = 0
    for k in range(len(x)):
        s = x[k]*y[k] + s
    return s
```

Workout 3.10. Consider the Newton's method for finding the root of $f(x) = a - 1/x$ for a given real number a. Show that the computation of x_{k+1} from x_k (successive Newton's iterations) can be expressed as two multiply-adds, thus the roundoff errors is reduced by a factor of $1/4$ if FMA operation is available.

We close this subsection by noting that one can use the modified model

$$\mathrm{fl}(x * y) = \frac{x * y}{1+\delta}, \quad |\delta| \leqslant u, \quad * \in \{+, -, \times, /\}, \quad x, y \in \mathbb{F} \tag{3.4}$$

instead of model (3.3). The proof is straightforward and is left as an exercise for the reader. For common numbers x any y, the value δ may differ in (3.3) and (3.4) but $|\delta|$ is bounded by the same u in both models.

Workout 3.11. Prove (3.4).

[11] For example Intel Itanium, IBM RISC System/6000 and IBM Power PC.

4 Conditioning of problems and stability of algorithms

Suppose we are given a mathematical problem (mathematical model) to solve. In practice, the problem we actually solve involves inputs which are perturbed versions of the actual input data; for example, the data may be contaminated by measurement errors or roundoff errors in computer. Thus, it is important to have some knowledge about the sensitivity of the model to such perturbations in its inputs. The fundamental question we face is as follows:

> *How much the solution of the perturbed problem is close to the solution of the original problem?*

This concept is usually referred to as the **conditioning** of the problem. A problem is called **well-conditioned** if it has few sensitivity to the perturbations, i.e., the solution of the perturbed problem remains close to the solution of the original problem. Otherwise the problem is called **ill-conditioned**.

We should distinguish between the conditioning of a mathematical problem and the conditioning of a computational algorithm we design to solve it. The latter is usually referred to as the **stability** of the algorithm. However, sometimes the term *stability* is used for both mathematical model and algorithm. We discuss these concepts in the following two sections.

4.1 Conditioning of a mathematical problem

First, we try to involve you into the concept through some simple examples. Then we will investigate the theory behind our observations.

Example 4.1. Many practical problems raise the need for solving a linear system of equations of the form

$$Ax = b$$

as a mathematical model, where $A \in \mathbb{R}^{n\times n}$ (i.e., an $n \times n$ matrix with real entries) and $b \in \mathbb{R}^n$ (i.e., an n vector with real entries) are given inputs and $x \in \mathbb{R}^n$ is the output (solution) we should compute. In some other applications, A may be a rectangular matrix with different number of rows and columns. However, in this example we assume that A is a square and nonsingular matrix. Assume further that our modeling results in a matrix A of the *Hilbert* form

$$A = \begin{bmatrix} 1 & \frac{1}{2} & \cdots & \frac{1}{n} \\ \frac{1}{2} & \frac{1}{3} & \cdots & \frac{1}{n+1} \\ \vdots & \vdots & & \vdots \\ \frac{1}{n} & \frac{1}{n+1} & \cdots & \frac{1}{2n-1} \end{bmatrix} =: H_n. \tag{4.1}$$

This matrix is symmetric and positive definite and thus nonsingular. Assume that the vector b is defined as the sum of all columns of A. Then the exact solution x of $Ax = b$ is indeed $x = [1, 1, \ldots, 1]^T$. Let us ignore the exact solution and solve the system using a stable

algorithm such as *Gauss elimination with pivoting*. In Python such algorithm is available via the `solve` command in the `numpy` library in the submodule `linalg`. This submodule contains many other linear algebra solvers as well. We need to define a subroutine for creating a Hilbert matrix of size $n \times n$. In the script below we assume $n = 11$, form a linear Hilbert system of equations and solve it using the `solve` command from the mentioned library. Outputs are the computed solution $\widehat{x}$ (= `xh`) and the relative error in the infinity norm.

```
import numpy as np
def hilbert(n):
    A = np.zeros([n,n]);
    for i in range(n):
        for j in range (n):
            A[i,j]=1/(i+j+1)
    return A
n = 11
A = hilbert(n)
b, x = np.sum(A, axis = 0), np.ones(n)
xh = np.linalg.solve(A,b)
print('xh = ', np.round(xh,8))
e = np.linalg.norm(x-xh,np.inf)/np.linalg.norm(x,np.inf)
print('RelErr = ', np.round(e,8))
```

The output is

```
xh =  [0.99999999 1.00000065 0.99998308 1.0001891  0.99887333 1.00396191
       0.99137162 1.0117659  0.99022415 1.00452416 0.99910608]
RelErr =  0.011765895169415952
```

We observe an unexpected result. We would typically expect a relative error close to machine precision (approximately $u \approx 10^{-16}$ in double precision) for this small system size. However, we observe an error of the order 10^{-2} in the output. This indicates a loss of around 14 significant decimal digits in the computation; our computed result is 10^{14} times worse than what we might expect from an ideal computation. What happened? One might initially blame the Python solver `solve`, but this function is based on the stable Gaussian elimination algorithm with pivoting. The true source of this serious issue lies in the original model $Ax = b$, as we will soon see.

Example 4.2 (James H. Wilkinson's Example)**.** Consider the problem of finding the roots of a polynomial of degree n of the form

$$p(x) = x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0. \tag{4.2}$$

for known real coefficients $a_0, a_1, \ldots, a_{n-1}$. If $\xi \in \mathbb{C}$ is a root, then ξ is a function of coefficients, say $\xi : \mathbb{R}^n \to \mathbb{C}$. In this example, we are going to test the sensitivity of ξ with respect to perturbation in input values $a_0, \ldots, a_{n-1}$. We start with the following polynomial of degree eight[a]:

$$\begin{aligned} p(x) = x^8 - 36x^7 + 546x^6 - 4536x^5 + 22449x^4 \\ - 67284x^3 + 118124x^2 - 109584x + 40320. \end{aligned} \tag{4.3}$$

This nasty polynomial has indeed the beautiful representation $p(x) = (x-8)(x-7)\cdots(x-1)$. The exact roots are real and non-repeated numbers $\xi = 8, 7, \ldots, 1$. However, in practice, a polynomial is usually represented in terms of its coefficients as shown in (4.2), and one often needs to solve $p(x) = 0$ to obtain some or all of its roots.

Several approaches exist for polynomial rootfinding. A dominant algorithm involves forming the *companion matrix*

$$A = \begin{bmatrix} -a_{n-1} & -a_{n-2} & \cdots & -a_1 & a_0 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 1 & 0 \end{bmatrix}$$

where the polynomial (4.2) is its characteristic polynomial (prove this!). An iterative algorithm based on QR factorization is then applied to compute the eigenvalues of. Then an iterative algorithm based on QR factorization is applied to compute the eigenvalues of A (roots of p). This algorithm has implemented in `numpy` library in Python. The algorithm works well for low-degree polynomials[b].

In the following Python code, we use the `roots` function to find the roots of (4.3). To observe the sensitivity of the roots to perturbations in the coefficients, we keep all coefficients of (4.3) unchanged except for $a_7 = -36$, which we perturb by 0.001. The perturbed roots are then computed using the same function.

```
import numpy as np
coeffs = [1, -36, 546, -4536, 22449, -67284, 118124, -109584, 40320]
original_roots = np.roots(coeffs)
perturbed_coeffs = coeffs.copy()
perturbed_coeffs[1] -= 0.001
perturbed_roots = np.roots(perturbed_coeffs)
print('OriginalRoots = ', original_roots)
print('PerturbedRoots = ', perturbed_roots)
```

The output is:

```
OriginalRoots = [8. 7. 6. 5. 4. 3. 2. 1.]
PerturbedRoots = [8.27260278  6.49985871+0.7292706j  6.49985871-0.7292706j
                  4.57483609  4.16253083  2.99113515  2.00017793  0.9999998]
```

As we observe, a small perturbation in one of the coefficients results in significant changes in the solution, to the extent that some roots (the second and third roots) become complex. In this example, the rootfinding algorithm performed its task correctly, and `PerturbedRoots` is indeed the vector of roots of the perturbed polynomial. This example illustrates that the roots of a polynomial, when considered *as functions of its coefficients*, are inherently ill-conditioned. This sensitivity to coefficient perturbations is independent of the algorithm used to calculate the roots. We will analyze this phenomenon in more detail shortly.

[a] The original Wilkinson's polynomial is of degree 20.

[b] A more efficient algorithm which does not consider the roots as a function of coefficients (but as a function of polynomial values) exists, yet not implemented in Python. Search polynomial rootfinding in *Chebfun*. Chebfun is an open-source software system written in MATLAB for numerical computation.

In the above examples, we encountered several ill-conditioned problems. But how can we measure well- or ill-conditioning quantitatively? To have a general definition, assume that $x \in D \subset \mathbb{R}^m$ is the input vector, $y \in \mathbb{R}^m$ is the output vector, and F is a map (problem model) that relates x and y via

$$F(x, y) = 0. \tag{4.4}$$

Sometimes, the output y can be explicitly represented in terms of the input x. In this case, there exists a function $f : D \to \mathbb{R}^n$ such that

$$y = f(x). \tag{4.5}$$

Definition 4.1. Let $x \in D$ be the input vector, and let δx be a perturbation in x such that $x + \delta x \in D$. Similarly, let y be the output vector and δy be a perturbation in y such that

$$F(x + \delta x, y + \delta y) = 0,$$

or in explicit form,

$$y + \delta y = f(x + \delta x).$$

The problem given by $F(x, y) = 0$ or equivalently $y = f(x)$ is said to be well-conditioned if it possesses a *unique solution* and the perturbations in y are bounded in a controlled manner by the perturbations in x. Specifically, the condition for well-conditioning is

$$\frac{\|\delta y\|}{\|y\|} \leq C \frac{\|\delta x\|}{\|x\|}$$

for $x \neq 0$ and $y \neq 0$, where C is a relatively small constant. In special cases when $x = 0$ and $y \neq 0$ the above bound is replaced by

$$\frac{\|\delta y\|}{\|y\|} \leq C\|\delta x\|.$$

If $x \neq 0$ and $y = 0$ we may write

$$\|\delta y\| \leq C\frac{\|\delta x\|}{\|x\|},$$

and finally if $x = 0$ and $y = 0$ then we must replace it by

$$\|\delta y\| \leq C\|\delta x\|.$$

In each of these cases, the constant C reflects how sensitively the solution y responds to changes or perturbations in the input x. A small C indicates that the problem is well-conditioned, meaning small changes in the input result in small changes in the output. Conversely, if C is large, the problem is ill-conditioned, implying that small changes in the input can cause large changes in the output, which can be problematic for numerical computations.

For many problems, the constant C in the above definition can be estimated. This estimate of C is known as the condition number of the problem. The condition number provides a measure of how sensitive the solution of a problem is to changes or errors in the input. According to Definition (4.1), the condition number of a problem can be obtained as the following ratio:

$$\frac{\text{amount of perturbation in the output (solution) of the problem}}{\text{amount of perturbation in the input of the problem}}$$

Depending on the criterion used to measure the amount of perturbation in the input and in the output of a given problem, *relative* or *absolute* condition numbers are defined.

Consider the simplest case $m = n = 1$ with the explicit representation (4.5) for the problem. Assume that f is twice continuously differentiable. The Taylor expansion then gives

$$y + \delta y = f(x + \delta x) = f(x) + \delta x f'(x) + \mathcal{O}(|\delta x|^2).$$

Ignoring the error term for small input perturbation δx, we can write

$$\frac{\delta y}{y} \approx \frac{x f'(x)}{f(x)} \cdot \frac{\delta x}{x}.$$

provided that $x \neq 0$ and $y \neq 0$. Consequently, we can define the relative *condition number of* f *at point* x by

$$(\text{cond } f)(x) := \frac{|x||f'(x)|}{|f(x)|} \tag{4.6}$$

to have

$$\frac{|\delta y|}{|y|} \approx (\text{cond} f)(x)\frac{|\delta x|}{|x|}.$$

The condition number shows, approximately, how much the perturbation in the input data

amplifies in the solution. A large condition number indicates ill-conditioning, while a small condition number signifies well-conditioning. The threshold for what constitutes "large" or "small" depends on the specific problem and the desired accuracy. If $x = 0$ but $y \neq 0$, then δx should be measured absolutely and δy relatively. In this case we define

$$(\text{cond}\, f)(x) := \frac{|f'(x)|}{|f(x)|}, \quad \frac{|\delta y|}{|y|} \approx (\text{cond} f)(x)|\delta x|.$$

On the other hand, if $x \neq 0$ and $y = 0$, then

$$(\text{cond}\, f)(x) := |x||f'(x)|, \quad |\delta y| \approx (\text{cond} f)(x)\frac{|\delta x|}{|x|}.$$

Finally, if $x = y = 0$ the condition number of f can be defined as

$$(\text{cond}\, f)(x) := |f'(x)|, \quad |\delta y| \approx (\text{cond} f)(x)|\delta x|.$$

Example 4.3. Consider the solution y of the quadratic equation $y^2 - 2ay + 1 = 0$ for input value $a > 1$. In the implicit form we may write $F(a, y) = y^2 - 2ay + 1 = 0$, but solving y in terms of x gives the explicit form

$$y_\pm = a \pm \sqrt{a^2 - 1} =: f_\pm(a).$$

This problem indeed has two solutions. We can treat y_+ and y_- either individually or together. Since $f'_\pm(a) = 1 \pm \frac{a}{\sqrt{a^2-1}}$, from (4.6) we have

$$(\text{cond}\, f_\pm)(a) := \frac{|f'_\pm(a)||a|}{|f_\pm(a)|} = \frac{|a|}{\sqrt{a^2 - 1}}, \quad a > 1.$$

This shows that for values of a far from 1, the problem is well-conditioned, while for values of a close to 1 (i.e. when the roots tend to become of multiplicity 2) the problem becomes ill-conditioned.

The ill-conditioning can be bypassed by using a simple change of variable to obtain an equivalent but well-conditioned problem for values of a near 1, and even for $a = 1$. If we use $b = a + \sqrt{a^2 - 1}$ then the quadratic equation is reformulated as

$$y^2 - \frac{1 + b^2}{b} y + 1 = 0.$$

In this case we have

$$y_+ = f_+(b) = \frac{1}{b}, \quad y_- = f_-(b) = b.$$

It is left to you to show that both f_+ and f_- are well-conditioned functions for values of b (or a) close to 1.

Example 4.4. In this example, we determine the conditioning of a problem that cannot be easily represented in an explicit form. Consider the nonlinear equation

$$x^n - ae^{-x} = 0, \quad a > 0, \quad n \geqslant 1.$$

Here, we assume that n is a fixed positive integer, a is the input data, and x is the output (noting that this is contrary to our usual notation where x typically represents the input).

This equation has exactly one positive root, denoted by ξ (prove this!). We aim to measure the sensitivity of this root with respect to a perturbation in a. Therefore, we consider ξ as a function of a, say $\xi(a)$. By estimating the condition number $(\operatorname{cond}\xi)(a)$, we will demonstrate that $\xi(a)$ is a well-conditioned function of a. Since an explicit form for ξ is not available, implicit differentiation can be used to compute $\xi'(a)$. We have

$$[\xi(a)]^n - ae^{-\xi(a)} = 0.$$

Implicit differentiation with respect to a yields

$$n\xi'(a)[\xi(a)]^{n-1} - e^{-\xi(a)} + a\xi'(a)e^{-\xi(a)} = 0,$$

or

$$\xi'(a) = \frac{e^{-\xi(a)}}{n[\xi(a)]^{n-1} + ae^{-\xi(a)}}.$$

Using this in the definition of condition number gives

$$(\operatorname{cond}\xi)(a) = \frac{\xi'(a)a}{\xi(a)} = \frac{ae^{-\xi}}{n\xi^n + a\xi e^{-\xi}} = \frac{ae^{-\xi}}{nae^{-\xi} + a\xi e^{-\xi}} = \frac{1}{n+\xi} \leqslant \frac{1}{n}.$$

Since all involved quantities are positive, absolute values were not needed in the definition of the condition number. The derived bound for the condition number indicates that the root ξ as a function of a is well-conditioned.

For the case of arbitrary m and n, assume that

$$x = (x_1, \ldots, x_m)^T \in \mathbb{R}^m, \quad y = (y_1, \ldots, y_n)^T \in \mathbb{R}^n,$$

and $f : \mathbb{R}^m \to \mathbb{R}^n$ maps data x into solution y. In an element-wise form we can write

$$y_k = f_k(x_1, \ldots, x_m), \quad k = 1, 2, \ldots, n.$$

Assume further that each function $f_k : \mathbb{R}^m \to \mathbb{R}$ has partial derivatives with respect to all m variables at point x.

One way to measure the sensitivity of solution y with respect to small changes in x is to subject only one variable, x_j, to a perturbation and observe the resulting change in just one component y_k. Then we can apply the univariate definition (4.6) and obtain

$$\kappa_{kj}(x) := \frac{\left|\frac{\partial f_k}{\partial x_j}(x)\right| |x_j|}{|f_k(x)|}. \tag{4.7}$$

If a component of x, or of y, vanishes, one should modify (4.7) as discussed earlier. This gives us a whole matrix

$$K(x) = [\kappa_{kj}(x)] \in \mathbb{R}^{n\times m}$$

of condition numbers. If a single condition number is sought, we can use a matrix norm to obtain

$$(\operatorname{cond} f)(x) := \|K(x)\|. \tag{4.8}$$

The condition number can be estimated in another way, often simpler but sometimes misleading. Let the relative perturbation in input $x \in \mathbb{R}^m$ and output $y \in \mathbb{R}^m$ be measured by

$$\frac{\|\delta x\|_\infty}{\|x\|_\infty} \quad \text{and} \quad \frac{\|\delta y\|_\infty}{\|y\|_\infty},$$

respectively, where $\delta x = (\delta x_1, \ldots, \delta x_m)^T$ and $\delta y = (\delta y_1, \ldots, \delta y_n)^T$. Now, in analogy to the scalar case and using the linear multivariate Taylor expansion, we have

$$y_k + \delta y_k = f_k(x + \delta x) = f_k(x) + \sum_{j=1}^{m} \frac{\partial f_k}{\partial x_j}(x)\delta x_j + \mathcal{O}(\|\delta x\|_\infty^2).$$

Ignoring the error term for a small input perturbation δx, we have, at least approximately,

$$|\delta y_k| \leqslant \sum_{j=1}^{m} \left|\frac{\partial f_k}{\partial x_j}(x)\right| |\delta x_j| \leqslant \|\delta x\|_\infty \max_k \sum_{j=1}^{m} \left|\frac{\partial f_k}{\partial x_j}(x)\right|$$

for all $k = 1, 2, \ldots, n$. This simply gives[12]

$$\|\delta y\|_\infty \leqslant \|\delta x\|_\infty \|J_f(x)\|_\infty, \tag{4.9}$$

where $J_f(x)$ is the *Jacobian matrix* defined by

$$J_f(x) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_m} \\ \vdots & \vdots & & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_m} \end{bmatrix} \in \mathbb{R}^{n \times m}.$$

For $x \neq 0$ and $y \neq 0$ from (4.9) we have

$$\frac{\|\delta y\|_\infty}{\|y\|_\infty} \leqslant \frac{\|J_f(x)\|_\infty \|x\|_\infty}{\|f(x)\|_\infty} \cdot \frac{\|\delta x\|_\infty}{\|x\|_\infty}$$

which suggests us to define

$$(\operatorname{cond} f)(x) := \frac{\|J_f(x)\|_\infty \|x\|_\infty}{\|f(x)\|_\infty}. \tag{4.10}$$

In situations where either $x = 0$ or $y = 0$, a modification in definition is carried out similar to the univariate case. For $m = n = 1$, this definition of condition number reduces to (4.6). However, for $m, n > 1$ the condition number in (4.10) may mislead us from the actual conditioning of the problem. The reason is that the norms sometimes tend to destroy the details. For example, if x has components of vastly different magnitudes, then $\|x\|_\infty$ is simply equal to the largest of these components, and all the others are ignored. See the workout below for an example.

Workout 4.2. Let $x = [x_1, x_2]^T$ and

$$f(x) = \begin{bmatrix} \frac{1}{x_1} + \frac{1}{x_2} \\ \frac{1}{x_1} - \frac{1}{x_2} \end{bmatrix}.$$

Compute the condition number of f using both formulas (4.8) and (4.10). Show that the formula (4.8) exhibits the potential for ill-conditioning for certain values of x while the formula (4.10) indicates that f is a well-conditioned function for all values of x.

Example 4.5. Coming back to Example (4.2), here we analyze the ill-conditioning of the

[12] Remind that the norm infinity of a $n \times m$ matrix A is defined as $\|A\|_\infty = \max\limits_{1 \leqslant k \leqslant n} \sum\limits_{j=1}^{m} |a_{kj}|$.

roots of a polynomial as functions of its coefficients. Let ξ be a fixed root of polynomial

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

where $a_0 \neq 0$, i.e., $\xi \neq 0$. Besides, assume that ξ is a simple root, i.e., $p'(\xi) \neq 0$. The root ξ is a complex function of coefficients $a = (a_0, \ldots, a_{n-1})$;

$$\xi : \mathbb{R}^n \to \mathbb{C}, \quad \xi = \xi(a_0, \ldots, a_{n-1}).$$

Recalling (4.7), we have

$$\kappa_k(a) = \frac{\left|\frac{\partial \xi}{\partial a_k}(a)\right| |a_k|}{|\xi|}. \tag{4.11}$$

The vector of condition numbers then is $K(a) = (\kappa_0, \ldots, \kappa_{n-1})$. To have an explicit representation for κ_k we must compute the partial derivatives of ξ with respect to coefficients a_k. Since ξ is a root, we have

$$\xi^n + a_{n-1}\xi^{n-1} + \cdots + a_k\xi^k + \cdots + a_1\xi + a_0 = 0.$$

An implicit differentiation with respect to a_k yields

$$n\xi^{n-1}\frac{\partial \xi}{\partial a_k} + (n-1)\xi^{n-2}\frac{\partial \xi}{\partial a_k} + \cdots + \xi^k + a_k\xi^{k-1}\frac{\partial \xi}{\partial a_k} + \cdots + a_1\frac{\partial \xi}{\partial a_k} + 0 \equiv 0.$$

This equation is equivalent to

$$p'(\xi)\frac{\partial \xi}{\partial a_k} + \xi^k = 0.$$

Since $p'(\xi) \neq 0$, we can write

$$\frac{\partial \xi}{\partial a_k} = -\frac{\xi^k}{p'(\xi)}.$$

Inserting into (4.11) and using the $\|\cdot\|_1$ for condition number, we obtain

$$(\operatorname{cond}\xi)(a) = \|K(x)\|_1 = \frac{1}{|\xi p'(\xi)|}\sum_{k=0}^{n-1}|a_k||\xi|^k = \frac{\sum_{k=0}^{n-1}|a_k||\xi|^k}{\left|\sum_{k=0}^{n-1}ka_k\xi^k\right|}. \tag{4.12}$$

The formula (4.12) shows that the problem has a potential to become ill-conditioned as (depending on signs of a_k and ξ) the denominator could be much smaller than the numerator. As an example, the condition numbers for roots of polynomial

$$\begin{aligned} p(x) &= x^8 - 36x^7 + 546x^6 - 4536x^5 + 22449x^4 - 67284x^3 + 118124x^2 - 109584x + 40320 \\ &= (x-8)(x-7)(x-6)(x-5)(x-4)(x-3)(x-2)(x-1), \end{aligned}$$

considered in Example 4.2, are computed using formula (4.12) and given in Table 4.

Table 4: Condition numbers of the roots of polynomial (4.3)

ξ_j	8	7	6	5
$(\operatorname{cond}\xi_j)(a)$	0.14×10^9	0.47×10^9	0.44×10^9	0.18×10^9
ξ_j	4	3	2	1
$(\operatorname{cond}\xi_j)(a)$	0.35×10^8	0.25×10^7	0.48×10^5	0.72×10^2

Once more, look at the output `PerturbedRoots` in the Python code provided in Example

4.2 to observe that the larger the condition number of a root, the more significant the change in the root becomes. The worst condition numbers are associated with roots $\xi = 6, 7$ which result in a pair of complex perturbed roots.

Example 4.6 (Conditioning of a Linear System of Equations)**.** Here we analyze the disaster observed in Example 4.1 using the concept of condition numbers. Consider again the linear system

$$Ax = b, \tag{4.13}$$

for a given nonsingular $n \times n$ matrix A and a nonzero vector $b \in \mathbb{R}^n$. Here, $x \in \mathbb{R}^n$ is served as the solution (output) and we aim to investigate the conditioning of the problem when the input data A and b are subjected to small perturbations. For simplicity, we only perturb the vector b and keep the matrix A unchanged. So, we consider x as a function of b only;

$$x = A^{-1}b := f(b).$$

Since $J_f(b) = A^{-1}$, from (4.10) we have

$$(\text{cond } f)(b) = \frac{\|b\|\|A\|}{\|A^{-1}b\|} = \frac{\|Ax\|\|A^{-1}\|}{\|x\|},$$

and since there is a one-to-one correspondence between x and b, we find for the worst condition number

$$\max_{b\neq 0} (\text{cond } f)(b) = \max_{x\neq 0} \frac{\|Ax\|}{\|x\|} \cdot \|A^{-1}\| = \|A\| \cdot \|A^{-1}\|$$

using the definition of a natural norm of A. The number on the far right no longer depends on b and is called the **condition number** of the matrix A. We denote it by

$$\text{cond}(A) := \|A\| \cdot \|A^{-1}\|. \tag{4.14}$$

To get a deeper insight into cond(A), consider the perturbed system

$$A(x + \delta x) = b + \delta b$$

which together with (4.13) gives the explicit representation $\delta x = A^{-1}\delta b$ for the output perturbation δx. Taking norm from both sides of this relation gives

$$\|\delta x\| \leqslant \|A^{-1}\|\|\delta b\|.$$

On the other side, from original system (4.13) we have $\|b\| \leqslant \|A\|\|x\|$, or

$$\frac{1}{\|x\|} \leqslant \frac{\|A\|}{\|b\|}.$$

Multiplying both sides of recent equations, we get

$$\frac{\|\delta x\|}{\|x\|} \leqslant \|A\| \cdot \|A^{-1}\| \frac{\|\delta b\|}{\|b\|} = \text{cond}(A) \frac{\|\delta b\|}{\|b\|}. \tag{4.15}$$

The error bound (4.15) simply describes what happened in Example 4.1 for the Hilbert system. In fact, rounding errors imply a small perturbation in the input data of machine epsilon order, i.e.,

$$\frac{\|\delta b\|_\infty}{\|b\|_\infty} = u \approx 10^{-16}$$

in the double precision floating-point format. The condition number of a Hilbert matrix of size 11×11 is approximately 10^{+15}. The error bound (4.15) then gives

$$\frac{\|x - \widehat{x}\|_\infty}{\|x\|_\infty} \lesssim 10^{+15} \cdot 10^{-16} = 0.1$$

which indicates that approximately 15 decimal digits will be lost when solving such a Hilbert system with any standard algorithm in double precision format. Our observation in Example 4.1 confirms this conclusion.

Although in Example (4.6) we have considered only perturbations in the right-hand vector b, it turns out that the error bounds still depend on the condition number in (4.14) when we also account for perturbations in the matrix A. See Workout 4.3 below.

Workout 4.3. Consider the linear system of equations $Ax = b$ for nonsingular matrix $A \in \mathbb{R}^{n\times n}$. Let δx be the perturbation raised in x caused by changing A to $A + \delta A$ but keeping b unchanged. Prove that

$$\frac{\|\delta x\|}{\|x\|} \leqslant \frac{\text{cond}(A)\frac{\|\delta A\|}{\|A\|}}{1 - \text{cond}(A)\frac{\|\delta A\|}{\|A\|}},$$

provided that δA is so small such that $\|\delta A\|\|A^{-1}\| \leqslant 1$. Moreover, prove that under perturbation in both A and b we have

$$\frac{\|\delta x\|}{\|x\|} \leqslant \frac{\text{cond}(A)}{1 - \text{cond}(A)\frac{\|\delta A\|}{\|A\|}} \left(\frac{\|\delta b\|}{\|b\|} + \frac{\|\delta A\|}{\|A\|}\right).$$

It is interpreted from Example 4.6 and Workout 4.3 that the condition number of a matrix plays a crucial role in numerical matrix computations.

As an example, the Hilbert matrix (4.1) is an ill-conditioned matrix. Table 5 shows the condition number of this matrix for different values of n in both the infinity norm and the 2-norm. We observe a rapid increase in the condition numbers as n grows. A Hilbert matrix with small size 13×13, which has a condition number of order 10^{17}, destroys all 16 decimal significant digits in the double precision floating-point format.

Table 5: Condition numbers of the Hilbert matrices of different sizes

n	3	5	7	9	11	13
$\text{cond}_2(H_n)$	5.24×10^2	4.77×10^5	4.75×10^8	4.93×10^{11}	5.22×10^{14}	4.79×10^{17}
$\text{cond}_\infty(H_n)$	7.48×10^2	9.44×10^5	9.85×10^8	1.10×10^{12}	1.23×10^{15}	8.53×10^{17}

Another well-known ill-conditioned matrix is the *Vandermonde matrix*. This matrix is of

the form

$$V_n = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ t_1 & t_2 & \cdots & t_n \\ \vdots & \vdots & & \vdots \\ t_1^{n-1} & t_2^{n-1} & \cdots & t_n^{n-1} \end{bmatrix} \in \mathbb{R}^{n\times n}$$

where $t_1, t_2, \ldots, t_n$ are some distinct real numbers. For example, for equality spaced numbers

$$t_k = -1 + \frac{2(k-1)}{n-1}, \quad k = 1, 2, \ldots, n,$$

in interval $[-1, 1]$, the condition numbers of V_n are given in Table 6 for some values of n. Although they do not grow as quickly as those for the Hilbert matrix, they still increase exponentially fast. Worse than exponential growth is observed if one takes harmonic numbers

$$t_k = \frac{1}{k}, \quad k = 1, 2, \ldots n.$$

In this case, we can show that the condition number of V_n grows as n^{n+1}, which is significantly worse than the condition number of the Hilbert matrix H_n.

Table 6: Condition numbers of the Vandermonde matrices of different sizes

n	10	15	20	25	30	35
$\text{cond}_2(V_n)$	2.63×10^3	1.10×10^6	2.72×10^8	7.05×10^{10}	1.84×10^{13}	4.61×10^{15}
$\text{cond}_\infty(V_n)$	2.06×10^4	5.58×10^6	1.75×10^9	4.91×10^{11}	1.46×10^{14}	4.16×10^{16}

In scientific computing, it is important to avoid mathematical and computational models that result in ill-conditioned matrices whenever possible. There are often alternative models and simulations that lead to well-conditioned matrices. For example, when solving polynomial interpolation using the monomial basis $\{1, x, \ldots, x^n\}$, the algorithm involves the Vandermonde matrix, which can be ill-conditioned. This can be circumvented using other algorithms such as Newton's method[13] or the *barycentric Lagrange interpolation* method. Similarly, solving the best polynomial approximation problem in the 2-norm leads to a Hilbert system if the monomial basis $\{1, x, \ldots, x^n\}$ is used. This issue can be avoided by employing an orthogonal basis instead.

When there is no flexibility to choose alternative models and methods, matrix *preconditioners* can be helpful. If A is an ill-conditioned matrix, a preconditioner P is a nonsingular matrix such that PA is well-conditioned. Then we can solve

$$PAx = Pb$$

instead of original system $Ax = b$. However, constructing a preconditioner P for a matrix A is not straightforward in many circumstances. There is no universal approach for building preconditioners that work effectively for all matrices.

[13]Newton's method may introduce other instability issues.

Remark 4.1. The condition number of a square, nonsingular matrix A is defined by (4.14). In a forthcoming lecture, we will explore an extension of this concept to non-square and even rank-deficient matrices using singular value decomposition (SVD).

Remark 4.2. In Python, an algorithm for estimating the condition numbers of a matrix is implemented in the `numpy.linalg` library. It works for 1, 2, infinity (`np.inf`) and Frobenius (`'fro'`) norms. For instance, to get the condition number in norm infinity we write

```
np.linalg.cond(A,np.inf)
```

The default command `np.linalg.cond(A)` gives the condition number in the Frobenius norm.

4.2 Stability of an algorithm

There might be different numerical algorithms for solving a given mathematical problem. Some algorithms are more sensitive than others to small errors in the input, meaning that small perturbations might cause large perturbations in the output of the algorithm.

The precise definition of the concept of stability varies depending on the subject area. While we do not intend to provide a rigorous definition, we aim to offer a general insight into the concept of stability through a definition and some concrete examples.

Mathematical models are typically approximated via a *numerical method* by converting to a new **discretized** version with possibly new approximated inputs. The solution of the discretized problem is expected to well approximate the solution of the original problem. Analogous to the primary problem (4.4) or (4.5), the discretized problem can be expressed as

$$F_N(x_N, y_N) = 0, \quad \text{or} \quad y_N = f_N(x_N) \tag{4.16}$$

where F_N represents the discretized model, x_N denotes approximate inputs, and y_N represents the solution of the discretized problem. The subscript N is a discretization parameter. This is an intermediate (and crucial) step of *problem-solving* in scientific computing. All procedures for solving continuous problems such as ordinary and partial differential equations (ODEs and PDEs) necessarily involve this step, and a significant amount of research in numerical analysis and scientific computing is dedicated to designing and developing efficient discretization techniques for various types of mathematical problems.

The concept of **stability** can be extended here in a manner similar to that described for the conditioning of a mathematical problem in the preceding section, by replacing f, x, and y with f_N, x_N, and y_N, respectively. However, it is important to note that, rather than stability, the discretized problem (4.16) is subject to another issue that requires the discretized model F_N to accurately approximate the exact model F as the discretization becomes finer. This property is known as **consistency**. In many cases, consistency, along with some form of stability, implies the **convergence** of the approximate solution y_N to the exact solution y.

In a forthcoming lecture, we will learn these concepts (consistency and stability) in detail for the numerical solution of ODEs. However, in this section (and the next section), we will assume that we are given a discretized problem to program into the computer, and our problem is only subject to roundoff errors in its inputs. Our goal is to investigate the sensitivity of the algorithm with respect to input perturbations. Roughly speaking, we refer to an algorithm as **numerically stable** if rounding errors occurring during the course of the algorithm are not amplified rapidly; in other words, if small perturbations in the inputs of the algorithm result in only small perturbations in its output.

In the stability analysis of an algorithm, different strategies can be employed, including **forward analysis** and **backward analysis**. To simplify notation, we drop the subscript N from the discretization problem (4.16) and assume that

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

is our numerical algorithm with exact input x and exact output y. The algorithm F accepts an input data x contaminated with roundoff error δ, i.e., $x+\delta$, and produces a numerical solution $\widehat{y}$, which we hope to be close to the exact solution y. Thus, we may write

$$F(x + \delta, \widehat{y}) = 0. \tag{4.17}$$

In a forward analysis, we assume that the relative error in δ is proportional to the machine's precision, say $\|\delta\|/\|x\| \leqslant C_{in}u$ where C_{in} is a small constant, and provide a bound on the error in the solution. That is, we look for a constant C_{out} such that

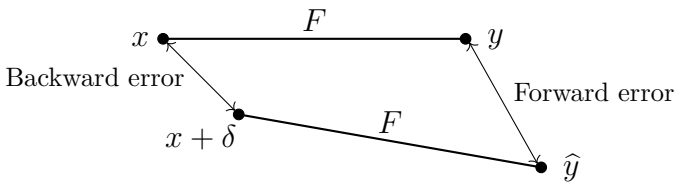
$$\frac{\|y - \widehat{y}\|}{\|y\|} \leqslant C_{out}u.$$

If, depending on the problem and the expected accuracy, C_{out} is a rather small constant, we say that the algorithm is *forward stable*.

In backward analysis, given a certain computed solution $\widehat{y}$, we look for a perturbation δ on the data such that (4.17) is satisfied. We say the algorithm is *backward stable* if the bound on perturbation δ is small, i.e.,

$$\frac{\|\delta\|}{\|x\|} \leqslant C_{bkw}u$$

where C_{bkw} is a small constant. In other words, an algorithm is backward stable if *the computed solution $\widehat{y}$ is the exact solution of a nearby problem.*



Forward and backward analyses are two different instances of the so called *a priori analysis*, and can be applied to investigate not only the stability of an algorithm but also the convergence of the solution of a discretized problem (numerical method) to the solution of a mathematical

model. In this case it is referred to as *a priori error analysis*, which can again be performed using either a forward or a backward technique. In *a posteriori analysis*, we aim to provide an estimate on the error of the solution in terms of the reminder. More precisely, we bound the error $y - \widehat{y}$ as a function of the residual

$$r = F(x, \widehat{y}).$$

Example 4.7. Let's revisit the linear system of equations $Ax = b$ for a nonsingular matrix $A \in \mathbb{R}^{n\times n}$ with a fixed positive integer n. Here, A and b are inputs, and x is the solution. Suppose this system is solved using a numerical algorithm for linear systems of equations, such as the Gauss elimination algorithm with pivoting. In almost all cases, A and b are subject to roundoff errors of magnitudes, at least at the level of the machine's precision u. Let the computed solution be denoted by $\widehat{x}$.

In the forward analysis for the Gauss elimination algorithm, we estimate the error $x - \widehat{x}$ in terms of bounds on relative perturbations in A and b, here u. We seek a constant C_{out} (which naturally depends on n) such that

$$\frac{\|x - \widehat{x}\|}{\|x\|} \leqslant C_{out} u.$$

In the backward analysis, on the other hand, we estimate the perturbations E and e that need to be introduced to A and b, respectively, in order to obtain

$$(A + E)\widehat{x} = b + e$$

for the computed solution $\widehat{x}$. The bounds on E and e can be expressed in terms of u:

$$\frac{\|E\|}{\|A\|} \leqslant C_{bkw} u, \quad \frac{\|e\|}{\|b\|} \leqslant C_{bkw} u.$$

The constants C_{bkw} depend on n, of course. Experimental evidence suggests that for most matrices appeared in practical problems, the constant C_{out} in the forward analysis and the constant C_{bkw} in the backward analysis are both of the order n, the size of the system. This would constitute an acceptable bound, making the Gauss algorithm known as a stable algorithm for a majority of practical problems.

Finally, in *a posteriori* error analysis, we seek an estimate for the error $x - \widehat{x}$ as a function of the residual $r = b - A\widehat{x}$. Since $x - \widehat{x} = A^{-1}r$, we can write $\|x - \widehat{x}\| \leqslant \|A^{-1}\|\|r\|$. On the other hand, from $Ax = b$, we have $1/\|x\| \leqslant \|A\|/\|b\|$. These two bounds together yield

$$\frac{\|x - \widehat{x}\|}{\|x\|} \leqslant \frac{\mathrm{cond}(A)}{\|b\|}\|r\|,$$

which provides a bound on the forward error in terms of the residual r. This error bound shows that a small residual does not necessarily correspond to a small error in the solution, especially if the matrix A is ill-conditioned.

At the end of this section let us address a key lesson concerning the question of *when can we expect a numerical solution to be accurate?* The answer is remarked below.

Remark 4.3. In general, we can have high confidence in the accuracy of a numerical solution if the problem is well-conditioned *and* a numerically stable algorithm has been employed to solve it. However, if the problem is ill-conditioned or if a numerically unstable algorithm has been used, the computed solution may be inaccurate.

5 Roundoff error analysis for some simple algorithms

According to (3.3), a single floating-point operation introduces an amplification factor $(1+\delta)$ (or $(1+\delta)^{-1}$ in (3.4)) in the computed result. For an arithmetic with multiple floating-point operations (for example summing or multiplying n numbers), the expression of the result may contain a sort of products of $(1+\delta_k)^{\pm 1}$ terms. The following Lemma proposes an elegant way to simplify such expressions [**Higham:2002**].

Lemma 5.1. If $|\delta_k| \leqslant u$ and $\rho_k = \pm 1$ for $k = 1, \ldots, n$, and $nu \leqslant 1$ then

$$\prod_{k=1}^{n}(1+\delta_k)^{\rho_k} = (1+\theta_n), \quad |\theta_n| \leqslant \frac{nu}{1-nu} =: \gamma_n.$$

Proof. We use an induction on n. Assume that the result is true for n. We have for case $\rho_{n+1} = 1$,

$$\prod_{k=1}^{n+1}(1+\delta_k)^{\rho_k} = (1+\theta_n)(1+\delta_{n+1}) =: (1+\theta_{n+1})$$

where $\theta_{n+1} = \theta_n + \delta_{n+1} + \theta_n\delta_{n+1}$. Thus, we can write

$$\begin{aligned}
|\theta_{n+1}| &\leqslant |\theta_n| + |\delta_{n+1}| + |\theta_n\delta_{n+1}| \\
&\leqslant \gamma_n + u + u\gamma_n = \frac{nu}{1-nu} + u + \frac{nu^2}{1-nu} \\
&= \frac{nu + u(1-nu) + nu^2}{1-nu} = \frac{(n+1)u}{1-nu} \leqslant \frac{(n+1)u}{1-(n+1)u} = \gamma_{n+1}.
\end{aligned}$$

The case $\rho_{n+1} = -1$ can be proved, similarly. ■

5.1 Multiplication

Assume that $x_1, x_2, \ldots, x_n \in \mathbb{F}$ are nonzero numbers and let $s_n = x_1x_2\cdots x_n$ be their product. We assume that the multiplication is carried out from left to right.The following simple Python function demonstrates how this operation is performed.

```
def Multiplication(x):
    s = x[0]
    for k in range (1,len(x)):
        s  = s * x[k]
    return s
```

In this function, `x` is a list of numbers, and the product is computed iteratively by multiplying

each element of the list from left to right. The variable s holds the intermediate product at each step, and the final product is returned at the end.

To analyse this algorithm, let $s_k = x_1 x_2 \cdots x_k$ denote the k-th partial product. In the following analysis, and throughout the lecture, a letter with hat (e.g. $\widehat{x}$) denotes a computed (floating-point) quantity. Using the standard model (3.3) we have

$$\begin{aligned}
\widehat{s}_1 &= \mathrm{fl}(x_1) = x_1 \quad \text{(indeed } \delta_1 = 0\text{)} \\
\widehat{s}_2 &= \mathrm{fl}(\widehat{s}_1 x_2) = \widehat{s}_1 x_2(1+\delta_2) = x_1 x_2(1+\delta_2) \\
\widehat{s}_3 &= \mathrm{fl}(\widehat{s}_2 x_3) = \widehat{s}_2 x_3(1+\delta_3) = x_1 x_2(1+\delta_2) x_3(1+\delta_3).
\end{aligned}$$

for $|\delta_k| \leqslant u$, $k = 2, 3$. Continuing in the same way, we obtain

$$\widehat{s}_n = x_1 x_2(1+\delta_2) x_3(1+\delta_3) \cdots x_n(1+\delta_n), \quad |\delta_k| \leqslant u, \tag{5.1}$$

and using Lemma 5.1 we can write

$$\begin{aligned}
\widehat{s}_n &= x_1 x_2 \cdots x_n (1+\delta_2)(1+\delta_3) \cdots (1+\delta_n) \\
&= s_n(1+\theta_{n-1}), \quad |\theta_{n-1}| \leqslant \gamma_{n-1}.
\end{aligned} \tag{5.2}$$

The expression (5.1) is a *backward error* analysis and can be interpreted as follows: the computed product

$$\widehat{s}_n = \widehat{x}_1 \widehat{x}_2 \cdots \widehat{x}_n$$

is indeed the exact product of perturbed data

$$\widehat{x}_1 = x_1, \quad \widehat{x}_2 = x_2(1+\delta_2), \ \ldots, \ \widehat{x}_n = x_n(1+\delta_n), \quad |\delta_k| \leqslant u.$$

Each relative perturbation is bounded by u, so the perturbations are tiny. This shows that the computed solution $\widehat{s}_n$ is the product of *nearby* data $\widehat{x}_k$.

On the other hand, (5.2) gives the *forward error* bound

$$\frac{|\widehat{s}_n - s_n|}{|s_n|} \leqslant \gamma_{n-1},$$

which estimates the difference between the computed solution $\widehat{s}_n$ and the exact solution s_n.

The above analysis exhibits that multiplication algorithm is *backward stable* independent of n (the number of operands), and *forward stable* if n (number of operands) is not a large number. Note than γ_n is increasing in n.

5.2 Summation

Assume $x_1, \ldots, x_n \in \mathbb{F}$ and let $s_n = x_1 + \cdots + x_n$. If the sum is computed using the usual recursive summation, as the code below,

```
def Summation(x):
    s = x[0]
    for k in range (1,len(x)):
        s  = s + x[k]
    return s
```

then after some detailed analysis similar to what we did in the previous example for multiplication, we obtain

$$\widehat{s}_n = x_1(1+\theta'_{n-1}) + x_2(1+\theta_{n-1}) + x_3(1+\theta_{n-2}) + \cdots + x_n(1+\theta_1), \quad |\theta_k| \leqslant \gamma_k, \tag{5.3}$$

and

$$|\widehat{s}_n - s_n| \leqslant |x_1|\gamma_{n-1} + |x_2|\gamma_{n-1} + |x_3|\gamma_{n-2} + \cdots + |x_n|\gamma_1. \tag{5.4}$$

From (5.3) we observe that the computed solution $\widehat{s}_n$ is indeed the exact sum of nearby data

$$\widehat{x}_1 = x_1(1+\theta'_{n-1}), \quad \widehat{x}_k = x_k(1+\theta_{n-k+1}), \text{ with } |\theta'_{n-1}| \leqslant \gamma_{n-1},\ |\theta_{n-k+1}| \leqslant \gamma_{n-k+1}.$$

This is a backward error analysis. Although at first glance summation might appear to result in smaller roundoff errors than multiplication, a comparison with the multiplication algorithm reveals that the recursive summation algorithm has larger backward error bounds. This is because the backward errors u for multiplication are replaced by values γ_{n-1} and γ_{n-k+1} in the summation algorithm. From (5.4), we have the forward error bound

$$|\widehat{s}_n - s_n| \leqslant \gamma_{n-1} \sum_{k=1}^{n} |x_k|. \tag{5.5}$$

This upper bound holds independently of the summation order. However, the upper bound (5.4) can be minimized if the terms x_k are added to the sum in increasing order of magnitude. By summing smaller terms first, the impact of roundoff errors is reduced, leading to a more accurate result. This strategy leverages the associative property of addition to mitigate the accumulation of roundoff errors.

In the above analysis, we observed that summation in floating-point arithmetic can introduce significant errors in the final computed solution. To address this issue, various techniques have been developed for more accurate summation. One such clever algorithm is *compensated summation*, a method that incorporates a correction term to reduce rounding errors. This technique captures the rounding errors and feeds them back into the summation process, thereby improving accuracy [**Kahan:1965**]. For more details on compensated summation and other advanced techniques, see [**Higham:2002**].

5.3 Inner product and matrix multiplications

Consider the inner product $s_n = x^T y$, where $x, y \in \mathbb{F}^n$. We assume that the evaluation of $s_n = x_1y_1 + \cdots + x_ny_n$ is performed from left to right. Let $s_k = x_1y_1 + \cdots + x_ky_k$ denote the k-th partial sum. Using the standard floating-point model (3.3) and without fused multiply-add (FMA) operations, we have, after some calculations

$$\widehat{s}_n = x_1y_1(1+\theta'_n) + x_2y_2(1+\theta_n) + x_3y_3(1+\theta_{n-1}) + \cdots + x_ny_n(1+\theta_2)$$

for $|\theta'_n| \leqslant \gamma_n$ and $|\theta_k| \leqslant \gamma_k$ for $k = 2, \ldots, n-1$. This is a backward error analysis, showing that

$$\widehat{s}_n = \widehat{x}_1\widehat{y}_1 + \cdots + \widehat{x}_n\widehat{y}_n$$

where

$$\widehat{x}_k = x_k,\ k = 1, \ldots, n, \quad \widehat{y}_1 = y_1(1+\theta'_n), \quad \widehat{y}_k = y_k(1+\theta_{n-k+2}),\ k = 2, \ldots, n.$$

Alternatively, we could perturb x_k and leave y_k alone. Using a vector form, we can write

$$\widehat{s}_n = \mathrm{fl}(x^T y) = x^T(y + \delta y) = (x + \delta x)^T y, \quad |\delta x| \leqslant \gamma_n |x|, \quad |\delta y| \leqslant \gamma_n |y| \tag{5.6}$$

with $|x|$ denoting the vector with elements $|x_k|$. Inequalities between vectors (and, later, matrices) are understood componentwise. A forward error bound follows from (5.6):

$$|x^T y - \mathrm{fl}(x^T y)| \leqslant \gamma_n \sum_{k=1}^{n} |x_k y_k| = \gamma_n |x|^T |y|. \tag{5.7}$$

If $y = x$, a high relative accuracy is obtained for computing $x^T x$. However, in general, high relative accuracy is not guaranteed if $|x^T y| \ll |x|^T |y|$.

Workout 5.2. To compute the inner product $s_n = x^T y$ of vectors $x, y \in \mathbb{F}^n$ for an even n, assume $m = n/2$, $s_1 = x(1:m)^T y(1:m)$, $s_2 = x(m+1:n)^T y(m+1:n)$, and $s_n = s_1 + s_2$. Prove the forward error bound

$$|\widehat{s}_n - s_n| \leqslant \gamma_{n/2+1} |x|^T |y|$$

which shows the error bound is almost halved by separating the inner product in two pieces. Generalized the idea by breaking the inner product into k pieces and find the optimal k.

With the analysis of the inner product in hand, it becomes straightforward to analyze matrix-vector and matrix-matrix multiplications. Consider $A \in \mathbb{F}^{m \times n}$, $x \in \mathbb{F}^n$ and $y = Ax$. The vector y can be formed by computing m inner products $y_k = a_{k\cdot}^T x$, $k = 1, \ldots, m$, where $a_{k\cdot}^T$ is the k-th row of A. From (5.7) we have

$$\widehat{y}_k = (a_{k\cdot} + \delta a_{k\cdot})^T x, \quad |\delta a_{k\cdot}| \leqslant \gamma_n |a_{k\cdot}|, \quad k = 1, \ldots, m,$$

which gives the backward error

$$\widehat{y} = (A + \delta A)x, \quad |\delta A| \leqslant \gamma_n |A|. \tag{5.8}$$

From (5.8), we obtain a forward error bound as

$$|\widehat{y} - y| \leqslant \gamma_n |A||x|. \tag{5.9}$$

Note that comparison operators (equalities and inequalities) between matrices and vectors are understood component-wise. Normwise bounds readily follow. For example, it is not difficult to show that if $|x| \leqslant |y|$, then $\|x\|_p \leqslant \|y\|_p$ for $p = 1, 2, \infty$, and indeed for any p. The same holds true for matrices: if $|A| \leqslant |B|$, then $\|A\|_p \leqslant \|B\|_p$ but just for $p = 1, \infty$. For $p = 2$ we have $\|A\|_2 \leqslant \sqrt{\min\{m,n\}}\|B\|_2$. Thus, the component-wise forward bound (5.9) results in

$$\|\widehat{y} - y\|_p \leqslant \gamma_n \|A\|_p \|x\|_p, \quad p = 1, \infty.$$

and

$$\|\widehat{y} - y\|_2 \leqslant \sqrt{\min\{m,n\}}\gamma_n \|A\|_2 \|x\|_2.$$

Workout 5.3. Assume that $A \in \mathbb{F}^{m \times p}$ and $B \in \mathbb{F}^{p \times n}$ and $C = AB$. Derive forward and backward component-wise error bounds for this matrix-matrix multiplication.

5.4 Cancellation

In this section, we discuss cancellation, a particularly dangerous phenomenon in numerical computing that can cause significant accuracy loss during simple operations like subtracting machine numbers with the same sign that have been previously rounded and share several leading digits. The same issue occurs in floating-point addition of machine numbers with opposite signs that have similar magnitudes and have been previously rounded.

To be precise, consider a (potentially long) chain of numerical computations where $\widehat{x}$ and $\widehat{y}$ are two machine numbers that are our existing p-digit approximations to some real quantities x and y, respectively. These quantities x and y may have infinitely many digits. Assume that among the significant digits of $\widehat{x}$ and $\widehat{y}$, k of the leading digits are the same. Now, consider the task of computing $\mathrm{fl}(\widehat{x} - \widehat{y})$. Ideally, we would want to compute the exact value $x - y$, but in practice, we can only compute $\mathrm{fl}(\widehat{x} - \widehat{y})$ with p-digit precision. We have

$$\begin{aligned} & (d_0.d_1d_2\cdots d_{k-1}\, d_k\, d_{k+1}\cdots d_{p-1})_\beta \times \beta^e \\ -\ & (d_0.d_1d_2\cdots d_{k-1}\, c_k\, c_{k+1}\cdots c_{p-1})_\beta \times \beta^e \\ \hline = \ & (0\ .\ 0\ 0\ \cdots\ 0\ \ f_k\ f_{k+1}\cdots f_{p-1})_\beta \times \beta^e \end{aligned}$$

When normalized, this result is

$$(f_k.f_{k+1}\cdots f_{p-1}\underbrace{00\cdots0}_{k \text{ times}})_\beta \times \beta^{e-k}.$$

Here, k last significant digits in the mantissa become zero, and their truly correct values are not known. This loss of significant digits in the subtraction of two close machine numbers is known as **cancellation error**.

To understand why cancellation in the operation $\mathrm{fl}(\widehat{x} - \widehat{y})$ is dangerous, consider that the resulting digits $(f_k.f_{k+1}\cdots f_{p-1})_\beta$ have been computed by subtracting the $(d_k\, d_{k+1}\cdots d_{p-1})_\beta$ part of $\widehat{x}$ and the $(c_k\, c_{k+1}\cdots c_{p-1})_\beta$ part of $\widehat{y}$. These tail digits of $\widehat{x}$ and $\widehat{y}$ are the most affected by rounding errors, and thus, have a higher chance of being different from the corresponding digits of x and y. As a result, cancellation can cause a significant loss of accuracy because the number of correct digits in the output $\mathrm{fl}(\widehat{x} - \widehat{y})$ might be much less than the number of correct digits in the inputs $\widehat{x}$ and $\widehat{y}$. This is why the cancellation phenomenon is sometimes referred to as *catastrophic cancellation*.

Example 5.1. We assume that

$$x = 1.23456702645$$
$$y = 1.2345664932563685$$

and compute $x - y$ in a floating point system with $\beta = 10$ and $p = 7$. Notice that in a decimal system with $p = 7$ we basically have seven significant decimal digits which is similar

to using single precision in the IEEE standard. With rounding to nearest we have

$$\widehat{x} = \text{fl}(x) = 1.234567$$
$$\widehat{y} = \text{fl}(y) = 1.234566.$$

Therefore, we can write

$$\text{fl}(\widehat{x} - \widehat{y}) = \text{fl}(\text{fl}(x) - \text{fl}(y)) = \text{fl}(0.000001) = 1.000000 \times 10^{-6}.$$

On the other hand, the correct result $x - y$, which in general can be computed only in theory, is as follows:

$$\begin{aligned} x - y &= 1.2345670264500000 - 1.2345664932563685 \\ &= 0.0000005331936315 = 5.331936315 \times 10^{-6}. \end{aligned}$$

The relative error then can be obtained as

$$\frac{|\text{fl}(\widehat{x} - \widehat{y}) - (x - y)|}{|x - y|} = \frac{|1.000000 \times 10^{-6} - 5.331936315 \times 10^{-6}|}{|5.331936315 \times 10^{-6}|} \approx 0.87$$

which is huge in comparison with the corresponding unit roundoff ($\approx 10^{-7}$). Our inputs $\widehat{x}$ and $\widehat{y}$ had seven correct significant digits while our output $\text{fl}(\widehat{x} - \widehat{y})$ does not have even one significant correct digit!

Example 5.2. Let us design an algorithm to approximate the irrational number π by considering it as the circumference of a semi-circle with radius 1. To this aim, we divide the semi-circle to n equi-length arcs to obtain an internal regular semi-polygons; see Figure 8.

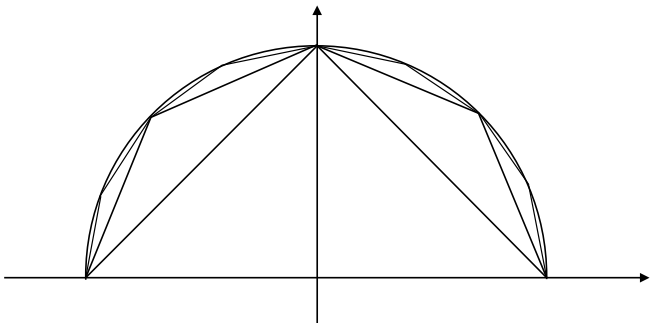


Figure 8: Polygons approximation of a circle.

The circumference of the semi-circle can be approximated by the sum of lengths of its line segments (circumference of the semi-polygons). Since the length of each side of the polygons is $\sin(\pi/n)$, the circumference of the semi-polygons is equal to

$$f(n) = n \sin \frac{\pi}{n}.$$

Obviously, $\lim_{n\to\infty} f(n) = \pi$. Let us increase n by powers of 2, i.e., $n = 1, 2, 4, \ldots, 2^k, \ldots$, and define

$$p_{k-1} = f(2^k) = 2^k \sin 2^{-k}\pi, \quad k = 1, 2, \ldots$$

From identity

$$\sin \frac{\alpha}{2} = \sqrt{\tfrac{1}{2}(1 - \cos \alpha)}, \quad 0 \leqslant \alpha \leqslant 2\pi,$$

we can write for $\alpha = 2^{-k}\pi$,

$$p_k = 2^{k+1}\sin\frac{\alpha}{2} = 2^{k+1}\sqrt{\tfrac{1}{2}(1-\cos\alpha)} = 2^{k+1}\sqrt{\frac{1}{2}\left(1-\sqrt{1-\sin^2\alpha}\right)}.$$

Using the fact that $\sin\alpha = 2^{-k}p_{k-1}$, we have the following recursive formula for the $\{p_k\}$ sequence:

$$p_k = 2^{k+1}\sqrt{\frac{1}{2}\left(1-\sqrt{1-[2^{-k}p_{k-1}]^2}\right)}, \quad k = 1, 2, \ldots, \quad p_0 = 2. \tag{5.10}$$

Theoretically, the sequence $\{p_k\}$ tends to π as k increases. Let's see what happens if we compute it numerically with a Python code.

```
import numpy as np
import matplotlib.pyplot as plt
K = 30
p = np.zeros(K)
p[0] = 2
for k in range (1,K):
    p[k] = 2**(k+1)*np.sqrt(0.5*(1-np.sqrt(1-2**(-2*k)*p[k-1]**2)))
plt.figure()
plt.semilogy(np.arange(K), abs(np.pi-p), marker = 's', color ='red')
plt.title("Approximation of $\pi$")
plt.xlabel("$k$"), plt.ylabel("$|\pi-p_k|$"),
```

The error $|\pi - p_k|$ is shown on the left-hand side of Figure 9. Initially, the error decreases to about 10^{-9} until $k = 14$, but unexpectedly begins to increase for larger k values, up to $k = 29$. We observe a V-shape error plot that reveals an instability in numerical computation.

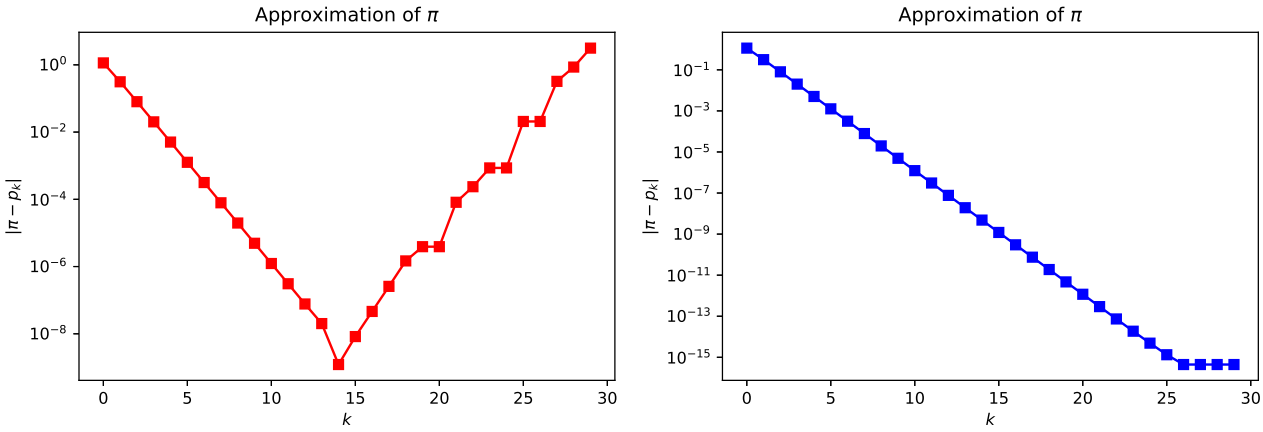


Figure 9: Approximation of π with two mathematically equivalent, but numerically different, formulas (5.10) (left) and (5.12) (right).

Let us reformulate (5.10) using the identity

$$1 - x = \frac{1 - x^2}{1 + x}, \quad x \neq -1 \tag{5.11}$$

for $x = \sqrt{1 - [2^{-k} p_{k-1}]^2}$. The new recursive sequence, which is theoretically equivalent to (5.10), is

$$p_k = p_{k-1} \sqrt{\frac{2}{1 + \sqrt{1 - [2^{-k} p_{k-1}]^2}}}, \quad k = 1, 2, \ldots, \quad p_0 = 2. \tag{5.12}$$

We update our Python code by replacing the line inside the loop with the new formula (5.12). The resulting error is plotted on the right-hand side of Figure 9. Notably, the error function now monotonically decreases to the level of machine precision. For a clearer comparison, both graphs are presented together in Figure 10.

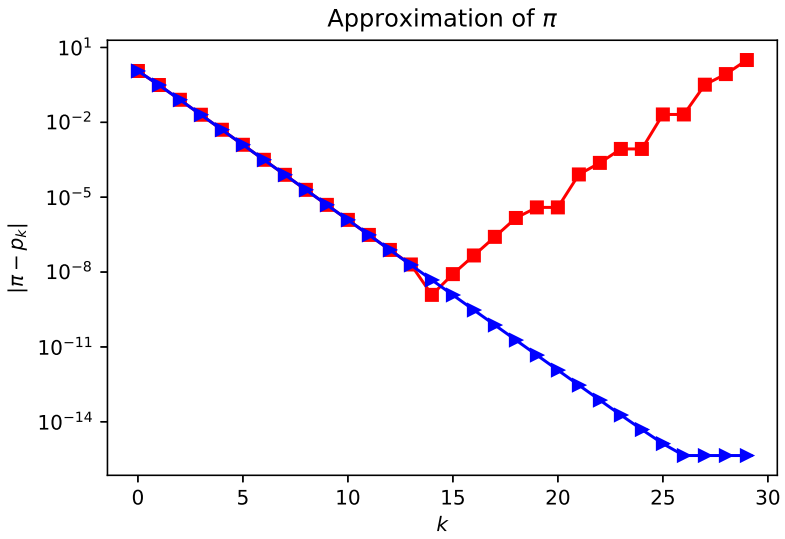


Figure 10: Approximation of π with two mathematically equivalent, but numerically different, formulas (5.10) (red line and square markers) and (5.12) (blue line and triangle markers).

Formulas (5.10) and (5.12) are mathematically equivalent, but there exists a source of cancellation error in formula (5.10) that makes it improper for floating-point arithmetic. Consider the term $1 - \sqrt{1 - [2^{-k} p_{k-1}]^2}$. As k increases, $2^{-k} p_{k-1}$ tends to zero, causing $\sqrt{1 - [2^{-k} p_{k-1}]^2}$ to approach 1. Consequently, there is a risk of cancellation error in the subtraction $1 - \sqrt{1 - [2^{-k} p_{k-1}]^2}$ for large values of k. In formula (5.12), we deliberately avoid this subtraction by using the simple identity (5.11). This minor adjustment eliminates the cancellation error and ensures the stability of the algorithm.

In numerical calculations, if possible one should try to avoid formulas with subtraction of close floating point numbers that give rise to cancellation error.

Remark 5.1. Beyond an illustration for cancelation error, the above examples reveals a fundamental insight in numerical computations: "mathematically equivalent" formulas or algorithms are not in general "numerically equivalent".

Workout 5.4. Reformulate the following expressions to avoid possible cancellations in computation.

$$1 - \cos x, \quad |x| \ll 1$$
$$\sin x - \cos x, \quad |x| \approx \frac{\pi}{4}$$
$$\ln\left(\sqrt{1+x^2} - x\right), \quad |x| \gg 1$$

We can apply our knowledge of the condition number to observe that subtraction of two close numbers (cancellation) is dangerous. This can be done in various ways but for our purposes it is enough to simply allow just one of the inputs x and y to vary and assume that the other one is constant. In particular, we consider the condition number of the problem of evaluating the function $f(x) = x - c$ for variable x and constant c. According to definition of condition number of univariate functions, i.e. (4.6), we have

$$(\mathrm{cond} f)(x) = \frac{|x| \cdot 1}{|x - c|} = \frac{|x|}{|x - c|}.$$

This means that when x and c are close to each other, the denominator of the condition number becomes small, leading to a large condition number. Therefore, it is not surprising that the cancellation phenomenon is dangerous. It is analogous to solving an ill-conditioned problem, where small changes in the input can result in significant changes in the output.

6 Complexity of an algorithm

The **complexity** (or **computational cost**) of an algorithm is its executing time and the amount of resources required to run it. Particular focus is given to *time* and *space* (memory) requirements. Calculating the complexity of an algorithm is therefore a part of the analysis of its efficiency.

Definition 6.1. *Time complexity* of an algorithm is generally expressed as the total number of required **fl**oating-point **op**eration**s** (**flops**) $\{+, -, \times, /\}$ on its input data to produce the final output.

If the input data is of size n, then the time complexity will be a function of n, say $f(n)$. Usually, we focus on the behavior of the complexity for large n, that is on its asymptotic behavior when n tends to the infinity. Therefore, the complexity is generally expressed by using big $\mathcal{O}$ notation.

Example 6.1. The inner product of two vectors $x, y \in \mathbb{R}^n$ that is $s = x_1y_1 + x_2y_2 + \cdots + x_ny_n$ requires n multiplications and $n - 1$ additions. The complexity for this computation is $f(n) = 2n - 1$ flops which is asymptotically expressed by $\mathcal{O}(n)$ flops.

Example 6.2. The time complexity for matrix-vector product Ax for $A \in \mathbb{R}^{n\times n}$ and $x \in \mathbb{R}^n$ is $f(n) = 2n^2 - n$ flops (why?) or asymptotically $\mathcal{O}(n^2)$ flops. The time complexity for matrix-matrix product AB for $A, B \in \mathbb{R}^{n\times n}$ is $f(n) = 2n^3 - n^2$ flops (why?) or $\mathcal{O}(n^3)$ flops.

Sometimes, the leading coefficient in $f(n)$ is also mentioned in the asymptotic expression of the time complexity. For example, the time complexity of the matrix-matrix product is reported as $2n^3$ flops. The reason should be clear; there is a huge difference between $1000n^3$ and $0.1n^3$, even for large values of n.

Example 6.3. The complexity of Gauss elimination algorithm for solving the linear system $Ax = b$ for $n \times n$ matrix A is $\frac{2}{3}n^3$ flops.

The usual units of time (seconds, minutes etc.) are not used for time complexity because they are dependent on the choice of a specific computer and on the evolution of technology.

Definition 6.2. *Space complexity* is generally expressed as the amount of memory required by an algorithm to load the inputs, execute, and produce the final solution (output).

The amount of memory needed depends on a variety of things such as the programming language, the compiler, or even the machine running the algorithm. Here, we just follow a simple rule: if we need to create a scaler this will require 1 **u**nit **o**f **s**pace (**uos**); if we create an array of size n, this will require n uos; and if we create a matrix (a two-dimensional array) of size $m \times n$, it will require mn uos.

Example 6.4. The space complexity of the standard algorithm for computing the inner product of two vectors $x, y \in \mathbb{R}^n$ is $2n$ uos to store x and y, and n auxiliary uos to store partial summations. If we let partial summations overwrite, only 1 auxiliary uos is needed. The output can also be stored in that auxiliary space. Consequently, the space complexity for this algorithm is $2n + 1$ uos.

Example 6.5. Consider the Gauss elimination algorithm for solving the system $Ax = b$ for square matrix A of size $n \times n$ and vector $b \in \mathbb{R}^n$. The input space is clearly $n^2 + n$ uos. The auxiliary space needed depends on how much thrifty the algorithm is implemented. Remember that the process of Gauss elimination consists of $n - 1$ steps to convert A to an upper triangular matrix. See the following illustration for $n = 4$.

$$\underset{A}{\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}} \longrightarrow \underset{A^{(1)}}{\begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \end{bmatrix}} \longrightarrow \underset{A^{(2)}}{\begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix}} \longrightarrow \underset{A^{(3)}}{\begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix}}$$

In each step k, the same operations are applied on vector $b^{(k-1)}$ to convert it to the new vector $b^{(k)}$. Finally, the equivalent upper triangular system

$$A^{(n-1)}x = b^{(n-1)}$$

should be solved using a backward substitution for solution x.
The process of elimination can be carried out without any auxiliary space if we overwrite A and b in each step, and store the auxiliary variables in the course of elimination in the lower diagonal part of A (zero positions). The backward substitution needs n auxiliary uos to store the output x. Consequently, the total space complexity is $n^2 + 2n$ uos.
This complexity analysis is valid for Gauss elimination algorithm without pivoting. For Gauss elimination with pivoting, a $2n + 1$ extra uos is required, $n + 1$ uos to interchange the rows of A and b in each step, and n uos to keep the permutation history in an array of size n.

Generally, when "complexity" or "computational cost" is used without being further specified, we mean the worst-case time complexity of the algorithm.

Additional workouts

Here are some additional exercises to help solidify your understanding of the concepts covered in the lecture. Some of these exercises are adapted from the references mentioned at the beginning of the lecture.

Workout 6.3. In the IEEE standard with double precision determine an interval for which the distance between the floating-point numbers is exactly 1.

Workout 6.4. How many double precision floating-point numbers do exist between two consecutive nonzero single precision floating-point numbers?

Workout 6.5. Show that

$$0.1 = \sum_{k=1}^{\infty} \left(2^{-4k} + 2^{-4k-1}\right)$$

and conclude $(0.1)_{10} = (0.000\overline{1100})_2$. The last four binary digits are repeated. In the IEEE standard with single precision show that if $\widehat{x} = \mathrm{fl}(0.1)$ then

$$\frac{x - \widehat{x}}{x} = \frac{1}{4}u.$$

Workout 6.6. Let x be a floating point number in IEEE double precision arithmetic satisfying $1 \leqslant x < 2$. Show that $\mathrm{fl}(x \times (1/x))$ is either 1 or $1 - \varepsilon_M/2$.

Workout 6.7. Show by an example that the inequalities $x \leqslant \mathrm{fl}((x+y)/2) \leqslant y$, where x and y are floating point numbers with $x \leqslant y$, can be violated in base 10 arithmetic. Show that $x \leqslant \mathrm{fl}(x + (y-x)/2) \leqslant y$ hold true in any base β arithmetic.

Workout 6.8. Show that with gradual underflow, if x and y are floating-point numbers and $\mathrm{fl}(x \pm y)$ underflows then $\mathrm{fl}(x \pm y) = x \pm y$. (no roundoff error for addition and subtraction when result falls in the underflow range).

Workout 6.9. (Kahan) Let $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$. Show that with the use of a fused multiply-add operation the algorithm

$$\begin{aligned} w &= bc \\ e &= w - bc \\ x &= (ad - w) + e \end{aligned}$$

computes $x = \det(A)$ with high relative accuracy.

Workout 6.10. A part of this course is devoted to numerical solution of ordinary differential equations (ODEs). The ODE is posed on a specific interval, $[a, b]$, and a numerical method works with a discretization of this interval with equidistance points $x_k := a + hk$, $k = 0, 1, \ldots, n$ with $h = (b-a)/n$. Compare the accuracy of the following formulas for computing x_k. Assume that a and b are floating-point numbers but h is not.

$$\begin{aligned} &\text{(I)} \quad x_k = x_{k-1} + h \\ &\text{(II)} \quad x_k = a + kh \\ &\text{(III)} \quad x_k = a(1 - k/n) + (k/n)b \end{aligned}$$

Usually, without any reason, (I) is used by users.

Workout 6.11. Compute the condition number of the following functions, and discuss any possible ill-conditioning.

$$\begin{aligned} f(x) &= \sin^{-1} x \\ f(x) &= x^{1/n}, \quad x > 0, \quad n \in \mathbb{N} \\ f(x) &= \cos(x), \quad |x| \leqslant \frac{\pi}{2} \end{aligned}$$

Workout 6.12. Show that

$$(\mathrm{cond}\, fg)(x) \leqslant (\mathrm{cond}\, f)(x) + (\mathrm{cond}\, g)(x).$$

What can be said about $(\mathrm{cond}\, f/g)(x)$?

Workout 6.13. Assume that $h(t) = g(f(t))$. Compute the condition number of h in terms of the condition of g and f. Apply it on function $h(t) = \frac{1+\sin t}{1-\sin t}$ to compute its condition number at $t = \pi/4$.

Workout 6.14. Consider the algebraic equation

$$x^n + ax - 1 = 0, \quad a > 0, \quad n \geqslant 2.$$

Show that the equation has exactly one positive root $\xi(a)$. Show that ξ is well-conditioned with respect to perturbations in a.

Workout 6.15. For a nonsingular matrix $A \in \mathbb{R}^{n\times n}$ show that

$$\operatorname{cond}_2(A) = \sqrt{\frac{\lambda_{\max}(A^TA)}{\lambda_{\min}(A^TA)}}.$$

If A is further symmetric then show

$$\operatorname{cond}_2(A) = \frac{\max|\lambda_k(A)|}{\min|\lambda_k(A)|}.$$

Workout 6.16. Assume that $A \in \mathbb{R}^{n\times n}$ and λ is an eigenvalue of A^TA. Show that

$$0 \leqslant \lambda \leqslant \|A^T\| \cdot \|A\|$$

where $\|\cdot\|$ is an operator norm. Using this show that

$$[\operatorname{cond}_2(A)]^2 \leqslant \operatorname{cond}_1(A) \cdot \operatorname{cond}_\infty(A).$$

provided that A is nonsingular.

References

[1] G. Dahlquist, Å. Björck, *Numerical Methods in Scientific Computing*, Volume 1, SIAM, Philadelphia, PA, 2008.

[2] J. W. Demmel, Underflow and the reliability of numerical software. SIAM J. Sci. Stat. Comput., 5(4):887–919, 1984.

[3] M. T. Heath, *Scientific Computing, an Introductory Survey*, revised 2nd edition, SIAM, Philadelphia, PA, 2018.

[4] W. Gautschi, *Numerical Analysis*, 2nd edition, Springer, 2012.

[5] Nicholas J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd edition, SIAM, Philadelphia, PA, 2002.

[6] W. Kahan, Further remarks on reducing truncation errors, Comm. ACM, 8(1) (1965) 40.