Chapter 1 Basics of Numerical Analysis and Scientific Computing

1.1 Motivations

> **Notation:** "*Numerical analysis is the study of algorithms for the problems of continuous mathematics.*" — Lloyd N. Trefethen

In the realm of mathematics, we frequently encounter continuous problems. However, computers are limited to discrete representations. For example, computers can only approximate irrational numbers like π or $\sqrt{2}$. Additionally, they use approximations for basic mathematical functions such as sine, cosine, etc. Numerical analysis bridges this gap, offering a rigorous framework for translating continuous mathematical problems into discrete problems that computer can handle. It is at the heart of many scientific and technological advances. Numerical analysis and computer science are inextricably linked. Advances in computing have empowered numerical methods to tackle increasingly complex problems. Computer scientists design efficient algorithms to implement these methods, while mathematicians provide the theoretical underpinnings. This synergistic relationship is indispensable in fields like artificial intelligence and data science, where numerical techniques are ubiquitous.

1.2 Floating-Point Arithmetic and Rounding Errors

Floating-point arithmetic is a numerical representation used in computers to approximate real numbers. Given that computers operate with finite memory, they cannot represent most real numbers exactly. Instead, they represent them using a finite number of bits, which leads to the following key concepts:: Representation of Floating-Point Numbers and Rounding Errors

1.2.1 Representation of Numbers in Machine

1.2.1.1 Introduction

In this section, we will introduce the concepts of mantissa, exponent, and how numbers are represented on a calculator or computer.

Base 10 is the natural base we work with and the one found in calculators. A decimal number, or decimal, has several different representations by simply changing the position of the decimal point and adding a power of 10 at the end of the number's representation. The part to the left of the decimal point is the integer part, and the part to the right before the exponent is called the mantissa. For example, the number x = 1234.5678 has several representations:

$$x = 1234{,}5678 = \ 1234{,}5678\,.\,10^{0} = \ 1{,}2345678.\,10^{3} = \ 0{,}0012345678.\,10^{6} \quad (1)$$

In each representation in Eq. (1):

- The *"mantissa"* is the significant part of the number, located to the left of the power of 10. In our example, it can be 1234,5678, 1,2345678, or 0,0012345678.
- The *"exponent"* is the power of 10 that shifts the decimal point: 0, 3, or 6.

Why this representation?

It offers great flexibility to represent very large or very small numbers, while using a fixed number of digits.

1.2.1.2 Representation of a Number in Machine: Floating-Point Numbers

The binary system is the foundation of computer arithmetic. In this system, numbers are represented using only two digits: 0 and 1.

Let's take the decimal number 39 and 3,625. In binary, it is represented as 100111 and 11.101, respectively in Eq. (2) and Eq. (3).

$$39 = 32 + 4 + 2 + 1 = 2^5 + 2^2 + 2^1 + 2^0 = (100111)_2 \quad (2)$$

$$3{,}625 = 2^1 + 2^0 + 2^{-1} + 2^{-3} = (11.101)_2 = (1.1101)_2 \quad (3)$$

In general, any real number *x* can be represented in a base *b* (*b = 10 for a calculator, b = 2 for a computer*) by: its sign "+ or –", the "mantissa *m"* (*also called significand*), the "base *b"*, and an "exponent *e"*. The mantissa is usually normalized to have a leading non-zero digit. By varying e, the decimal point is made to '*float*'. However, due to the finite nature of computer memory, only a finite subset of real numbers can be exactly represented. Consequently, a machine-level real number or floating-point number is subject to rounding errors, which can lead to loss of precision in certain calculations.

$$\check{X} = \pm m.b^e$$
$$m = D, D \ldots . D \quad and\ e = D \ldots D, \ where\ D \in \{0,1, \ldots, b-1\} \quad (4)$$

Approximate representations of π are: (0.031,2), (3.142,0), (0.003,3). It can be observed that these representations do not yield the same level of precision. To ensure uniqueness and optimal precision, a normalized mantissa is employed, where the leading digit before the radix point is non-zero. Normalized machine numbers adhere to this convention. In base 2, the initial bit of the mantissa is invariably 1, and thus, it is omitted to save a bit. The exponent is constrained to a finite range, L <= e <= U (typically L < 0 and U > 0). Consequently, the system is defined by four integral parameters: the base b (usually 2), the precision t (number of digits in the mantissa), and the minimum and maximum exponents, L and U, respectively.

Example: Let's take the number π (Pi) and represent it in base 2 with a 5-bit mantissa and an exponent ranging from -2 to 2.
Choice of an approximation for π: We will use the approximation 3.1416.
Normalization of the mantissa: To normalize, we shift the decimal point so that there is a 1 before the decimal. Thus, 3.1416 becomes 1.1001×2^1.
Binary representation:
The mantissa (without the leading 1) is: 1001
The exponent is: 1 (which corresponds to 2^1 in our example)
Therefore, the floating-point representation of π in our system is $(1.1001{,}1)$.
To summarize, the number π is represented by:
Sign: + (positive)
Mantissa: 1.1001 (normalized)
Exponent: 1
Base: 2

Mathematical computations involve real numbers z drawn from a continuous interval $\in]-\infty,+\infty[$. However, the finite precision of computers necessitates approximations for most real numbers. For instance, $\frac{1}{3}, \sqrt{2}, \; and \; \pi$, with its infinite decimal expansion, cannot be exactly represented in a machine. Even the simplest calculations then become approximate. Practical experience shows that this limited set of representable numbers is largely sufficient for calculations on a computer; the numbers used in computations are machine numbers $\check{x} \in]\check{x}_{min},..,\check{x}_{max}[$. Hence, any real number x must be mapped to a machine number $\check{x}$ before it can be processed by a computer.

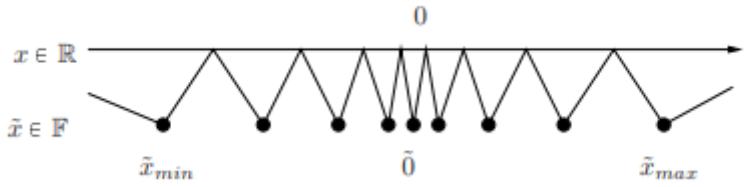

Figure 1. *Representation of real number $x \in R$ by machine numbers $\check{x} \in F$*

Definitions

Definition1: Machine precision, it is described by the machine number ε, ε is the smallest machine number such that $1 + \varepsilon > 1$ on the machine. It is the distance between the integer 1 and the closest number $\check{x} \in F$, which is greater than 1.

Definition2: The IEEE 754 standard is a widely adopted framework for representing floating-point numbers in computer systems. It defines formats for both single-precision and double-precision numbers, ensuring consistency and accuracy in numerical computations across different computing platforms.

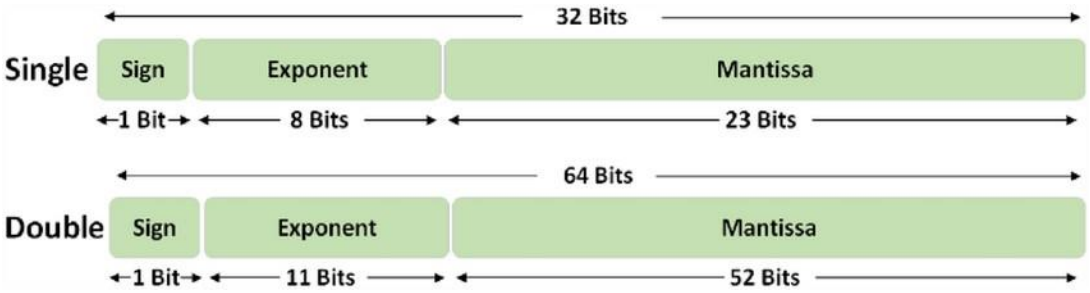

Figure 2. *Representation of The IEEE 754 standard for both single-precision and double-precision numbers*

The IEEE 754 standard defines a framework for representing floating-point numbers in computing, ensuring consistency and accuracy across platforms. It includes two primary formats: single precision (32 bits) and double precision (64 bits). In both formats, the representation is divided into three main components: the sign bit, which indicates whether the number is positive or negative; the exponent, which is biased to allow for both positive and negative values; and the mantissa, which represents the significant digits of the number in a normalized form. The bias for single precision is 127, while for double precision, it is 1023. For single precision, the format consists of 1 sign bit, 8 exponent bits, and 23 mantissa bits, while double precision uses 1 sign bit, 11 exponent bits, and 52 mantissa bits. This

structure allows for efficient computation and standardized handling of special values, such as zero, infinity, and NaN (Not a Number), facilitating reliable numerical operations in various applications.

Example: Here's the representation of −6.75 in both single-precision and double-precision IEEE 754 formats.

1. Single Precision (32 bits)

Step 1: Convert to Binary
- *Absolute value*: 6.75 in binary is 110.11.

Step 2: Normalize
- *Normalize to:* 1011×2^2

Step 3: Sign Bit
- *Sign (S):* 1 (negative)

Step 4: Exponent
- Actual exponent: 2
- Biased exponent: $2 + 127 = 129$
- In binary: 10000001

Step 5: Mantissa
- Mantissa (without leading 1): 10110000000000000000000 (23 bits)

Final Representation

Putting it all together:
| 1 | 10000001 | 10110000000000000000000 |

2. Double Precision (64 bits)

Step 1: Convert to Binary
- *Absolute value*: 6.75 in binary is 110.11.

Step 2: Normalize
- *Normalize to:* 1011×2^2

Step 3: Sign Bit
- *Sign (S):* 1 (negative)

Step 4: Exponent
- Actual exponent: 2
- Biased exponent: $2 + 1023 = 1025$
- In binary: 10000001001 (11 bits)

Step 5: Mantissa
- Mantissa (without leading 1): 1011000000000000000000000000000000000000000000000000 (52 bits)

Final Representation

Putting it all together:
| 1 | 10000001001 | 1011000000000000000000000000000000000000000000000000 |

1.2.2 Rounding Errors

Rounding errors occur when numbers are approximated to fit within the finite precision of a computer's numerical representation. These errors arise because many real numbers cannot be represented exactly in a binary format, leading to discrepancies between the true value and its computed representation. When performing arithmetic operations, results may require more precision than can be accommodated, necessitating rounding to fit the format. As a result, rounding errors can accumulate in iterative calculations, potentially leading to significant inaccuracies in the final output.

1.2.2.1 Absolute Error

The absolute error measures the difference between the exact value and the approximate value obtained through computation. It is defined as:

$$Absolute\ Error = \left|\text{Exact Value} - \text{Approximate Value}\right| \quad (5)$$

For example, if the exact value of a number is 3.14159 and the computed value is 3.14, the absolute error is: $|3.14159-3.14|=0.00159$

1.2.2.2 Relative Error

Relative error provides a measure of the absolute error in relation to the size of the exact value, allowing for a better understanding of the error's significance in context. It is calculated as:

$$\text{Relative Error} = \frac{\left|\text{Exact Value}-\text{Approximate Value}\right|}{\left|\text{Approximate Value}\right|} \quad (5)$$

Using the previous example, the relative error for the values 3.14159 (exact) and 3.14 (approximate) would be:

$$\text{Relative Error} = \frac{|0.00159|}{|3.14159|} \approx 0.000506 \quad (6)$$

Rounding errors can significantly impact the accuracy of numerical computations due to limited precision in representing real numbers. Absolute error quantifies the direct difference between exact and approximate values, while relative error contextualizes this difference by comparing it to the size of the exact value. Understanding both types of errors is crucial for assessing the reliability of numerical results in various applications.

1.3 Stability and Error Analysis of Numerical Methods and Problem Conditioning

To develop reliable and accurate numerical algorithms, it is essential to integrate error analysis, stability analysis, and considerations of problem conditioning:

1.3.1 Algorithm Selection:

Algorithm selection in numerical analysis is a multi-faceted process that involves understanding the problem, analyzing errors, evaluating stability and complexity, and considering problem conditioning. By systematically evaluating these factors, one can choose the most appropriate numerical method that balances accuracy, efficiency, and reliability, ensuring that the chosen approach yields trustworthy results in practice. This careful selection process is essential for successfully solving complex mathematical problems across various scientific and engineering disciplines. Here's a detailed breakdown of the factors to consider when selecting an algorithm.

1.3.1.1 Understanding the Problem

Before selecting an appropriate algorithm, it is crucial to thoroughly understand the nature of the problem at hand. First, identify the "***type of problem"*** you are dealing with, whether it involves linear systems, nonlinear equations, optimization tasks, differential equations, or integration. Each type requires specific algorithms that leverage their unique characteristics and properties. Additionally, consider the

"***dimensionality"*** of the problem, as the number of variables can significantly impact algorithm performance. Some algorithms may excel in lower-dimensional spaces but encounter challenges as dimensionality increases—a phenomenon often referred to as the "curse of dimensionality." This understanding lays the groundwork for making informed choices about which numerical methods will yield the most accurate and efficient results for your specific context.

1.3.1.2 Error Analysis

Understanding the error characteristics of different algorithms is vital for selecting the most suitable method for a given problem. "*Truncation error*" refers to the error introduced when an algorithm approximates a solution, which can be assessed based on the method's step size *h*. For example, numerical differentiation methods exhibit truncation errors that depend on the step size, typically expressed as Eq. (7):

$$E_{\text{trunc}} \sim O(h^p) \qquad (7)$$

Where p denotes the order of the method.

In addition to truncation errors, it is essential to analyze "*round-off error"*, which results from the limitations of numerical precision during calculations. Algorithms requiring numerous arithmetic operations, particularly iterative methods, can accumulate significant round-off errors.

To achieve a comprehensive understanding of overall accuracy, one must consider the "*total error"*, which can be expressed as Eq. (8):

$$E_{\text{total}} = E_{\text{trunc}} + E_{\boldsymbol{round-off}} \qquad (8)$$

Recognizing how these error components interact and combine is crucial for selecting an algorithm that maintains a manageable total error, ensuring reliable and accurate numerical results.

1.3.1.3 Stability Analysis

The stability of an algorithm is a critical factor that indicates how errors propagate during computations. Stable algorithms are designed to avoid significant amplification of errors throughout the calculation process.

For instance, direct methods like Gaussian elimination are typically stable; however, they can encounter large truncation errors, particularly when applied to ill-conditioned problems. In contrast, iterative methods, such as the Jacobi method, may be unstable, especially if their convergence heavily relies on the initial guess or the conditioning of the problem. The "***condition number"*** of a matrix A, denoted as κ(A), serves as a measure of sensitivity to input variations and numerical errors. It is defined as Eq. (9)

$$\kappa(\mathrm{A}) = \|A\|\|A^{-1}\| \qquad (9)$$

A high condition number indicates potential instability, suggesting that small changes in the input can lead to large variations in the output. Therefore, selecting algorithms that effectively mitigate the effects of high condition numbers is essential for ensuring reliable numerical results, particularly in sensitive computational scenarios.

1.3.1.4 Complexity and Efficiency

When selecting an algorithm, it's important to evaluate both its computational complexity and efficiency. "***Time complexity"*** measures how the execution time increases with the size of the input. For instance, solving a system of linear equations using Gaussian elimination has a time complexity of $O(n^3)$, which can be computationally expensive for large systems.

In contrast, iterative methods like the Conjugate Gradient algorithm often offer improved performance, particularly for large sparse systems, as they can converge faster and require fewer operations. Additionally, "***space complexity"*** should be assessed to understand the memory requirements of the algorithm. Some methods necessitate extra space for storing intermediate results, which can pose challenges in resource-constrained environments. Balancing time and space complexity is crucial for selecting efficient algorithms that meet the demands of specific problems while optimizing resource usage.

1.3.1.5 Specific Characteristics of Algorithms

When selecting an algorithm, it is crucial to consider its "***convergence behavior***, as different methods converge at different rates. For example, Newton's method converges quadratically near the solution, while the Bisection method exhibits linear convergence, making it slower but more robust for certain problems. Additionally, the "***robustness"*** of an algorithm is essential; methods sensitive to minor input changes may not be suitable for real-world applications. "***Adaptivity"*** is another key feature; some algorithms can dynamically adjust their parameters based on problem characteristics, such as adaptive step-size methods in numerical integration, which modify the step size according to estimated errors. Evaluating these characteristics helps ensure the selection of algorithms that are efficient, reliable, and resilient under various conditions.

1.3.1.6 Problem Conditioning

Understanding the conditioning of a problem is crucial for algorithm selection, particularly in differentiating between *"**well-conditioned"*** and *"**ill-conditioned"*** problems. Ill-conditioned problems are sensitive to small input perturbations, which can lead to significant output variations, complicating the numerical solution. In such cases, it is often beneficial to use regularization techniques or select more robust algorithms that minimize error propagation. These strategies help stabilize the solution, ensuring that minor changes in input do not cause disproportionately large errors in the results. Recognizing the problem's conditioning enables informed decisions about which algorithms will provide reliable and accurate solutions.

1.3.1.7 Availability of Libraries and Tools

When choosing an algorithm, it's essential to consider the availability of implementation resources, such as libraries and tools that facilitate application. Libraries like NumPy and SciPy, along with specialized numerical solvers, can save significant time and effort by offering pre-implemented functions and optimized algorithms. Additionally, strong community support and comprehensive documentation are invaluable; they help users troubleshoot issues and provide practical insights into algorithm usage. This support enhances the implementation of numerical methods, ensuring effective application while minimizing potential challenges.

1.3.2 Refinement and Adaptation

Numerical methods can often be refined based on error and stability assessments. For instance, adaptive algorithms can adjust parameters like step size dynamically based on estimated errors.

1.3.3 Validation and Testing

Rigorous testing and validation against known solutions or benchmarks can help verify that the combined considerations of error, stability, and conditioning lead to reliable numerical results.

Error analysis and stability are vital for ensuring the reliability and accuracy of numerical algorithms. A comprehensive understanding of these concepts, combined with an awareness of problem conditioning, allows developers to create robust numerical methods suitable for a wide range of applications. By systematically addressing these aspects, we can enhance the performance and trustworthiness of numerical computations in science and engineering.