Scientific Computing Fundamentals

Computational Journey of Quantum Mechanics

Overview

An important part of scientific computing is to understand that our goal is to **understand**, not to **generate numbers**! But to understand, one requires to know the potentials, pit-falls, and systematic mistakes that could happen while working on a simulation, approxi-mation or a visualization. In this notebook we investigate some bare concepts of how computers work, how they give out errors and how to avoid these routes.

We would start by understanding machine numbers, and rounding errors and we move on to talks about different algorithm designs, data structures, and some visualization tips. These are easy, yet fundamental aspects of our goals in this series. But don't fear, you can

always check this content again and gain more insight about the topics discussed here.

I would also leave a list of resources for the enthusiasts wanting to learn it to core.

Numbers and Errors

Human vs Computer

We're all familiar with numbers from an early age in school. We count them, add and subtract them, and as we dive deeper into our exploration of the universe—or even the complexities of social and individual phenomena—we develop more sophisticated ways of using them. But the numbers we use daily are a product of human perception and convenience. Our mathematical notation, specifically the decimal system (base 10), is deeply tied to the way we evolved to understand the world, likely influenced by the fact that we have ten fingers. However, this way of representing numbers is just one possibility among many, and it isn't necessarily the most efficient for every application. When we step into the realm of computers, the way numbers are handled shifts dramatically, driven not by human convenience but by the physical constraints and architecture of digital systems.

Computers don't "think" in base 10; instead, they operate using a binary system (base 2), where numbers are represented using only two symbols: 0 and 1. This fundamental shift is due to the nature of electronic circuits, which rely on switches that are either on or off, corresponding naturally to the binary states of 1 and 0. Every number we see on a screen, every calculation performed by a processor, and every piece of digital information is encoded in this binary language. But binary isn't the only alternative—other bases like octal (base 8) and hexadecimal (base 16) are frequently used in computing because they allow for more compact representations of binary data. Understanding how numbers are stored and manipulated in different bases is crucial for fields ranging from low-level programming and computer engineering to cryptography and data science. The way numbers are represented in computers is a perfect example of how mathematics adapts to fit the constraints and capabilities of the medium in which it operates.

Representing Numbers

Binary to Decimal

You should be quite familiar with base-10 arithmetic; digital computers, on the other hand, use base-2 or binary representation for arithmetic . Like base-10, binary is also written using positional notation, however restricted to only allowing the digits 0 and 1. For example, the binary number $10\,110.1_2$ is converted to base-10 as follows :

$$10\,110.1_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} = 22.5$$

Let’s implement this algorithm in Mathematica. It’s simple but we should be careful about some cases that the user might give us wrong numbers or notation:

In[1]:=
```
ToDecimal[binary_String] := Module[
 {digits, values, lengthOfIntegers, lengthOfFractions, decimalValue},

 (* Split the string at the decimal point *)
 digits = StringSplit[binary, "."];

 (* Ensure there are at most one or zero decimal points *)
 If[Length[digits] > 2, Return[$Failed]];

 (* Convert characters to numbers and validate *)
 values = ToExpression /@ (Characters /@ digits);
 If[! AllTrue[Flatten[values], (# === 0 || # === 1) &], Return[$Failed]];

 (* Compute decimal value for integer part *)
 lengthOfIntegers = Length[values[[1]]];
 decimalValue = Sum[values[[1]][[i]] * 2^(lengthOfIntegers - i), {i, 1, lengthOfIntegers}];

 (* Compute fractional part if it exists *)
 If[Length[digits] == 2,
  lengthOfFractions = Length[values[[2]]];
  decimalValue += Sum[values[[2]][[i]] * 2^-i, {i, 1, lengthOfFractions}]
 ];

 (* Return the final computed decimal value *)
 decimalValue
]
```

In[2]:=
```
ToDecimal["10110.1"] // N
```

Out[2]=
```
22.5
```

Decimal to Binary

Converting from base-10 to binary is a little more involved, but still quite elementary. For example, let's convert 23.625 to binary, we start with the integer component, and continuously divide by 2, storing each remainder until we get a result of 0

$23 / 2 = 11 \text{ remainder } 1$

$\Rightarrow 11 / 2 = 5 \text{ remainder } 1$

$\Rightarrow 5 / 2 = 2 \text{ remainder } 1$

$\Rightarrow 2 / 2 = 1 \text{ remainder } 0$

$\Rightarrow 1 / 2 = 0 \text{ remainder } 1$

The remainder of the first division gives us the **least significant digit**, and the remainder of the final division gives us the **most significant digit**. Therefore we have:

$23 - 10\,111_2$.

To convert the fractional part 0.625, we instead multiply the fractional part by 2 each time, sorting each integer until we have a fractional part of zero:

$0.625 \times 2 = 1.25$

$\Rightarrow 0.25 \times 0.5$

$\Rightarrow 0.5 \times 2 = 1.0$

The first multiplication gives us the **most significant digit**, and the final multiplication gives us the **least significant digit**. So, reading from top to bottom:

$0.625 = 0.101_2$

Putting them together:

$23.625 = 10\,111.101_2$

Here's the implementation of this process in Mathematica:

```
ToBinary[number_Real] :=
 Module[{integerPart, fractionalPart, intBinary, fracBinary, frac, count},
  (*Extract integer and fractional parts*)
  integerPart = Floor[number];
  fractionalPart = number - integerPart;
  (*Convert integer part to binary*)
  intBinary = "";
  If[integerPart == 0,
   intBinary = "0",
   While[integerPart > 0,
     intBinary = ToString[Mod[integerPart, 2]] <> intBinary;
     integerPart = Quotient[integerPart, 2];];];
  (*Convert fractional part to binary*)
  fracBinary = "";
  frac = fractionalPart;
  count = 0;
  While[frac > 0 && count < 10,
   (*Limit to 10 decimal places to prevent infinite loops*)
   frac *= 2;
   fracBinary = fracBinary <> ToString[Floor[frac]];
   frac -= Floor[frac];
   count++;];
  (*Return final binary representation*)
  If[fractionalPart == 0, intBinary, intBinary <> "." <> fracBinary]
 ]

ToBinary[number_Integer] := ToBinary[N[number]]
(*Overload to handle pure integers*)
```

In[5]:=
```
ToBinary[23.625]
```

Out[5]=
```
10111.101
```

Memory Limitation and Fixed-Point Representation

Computers have limited amount of memory to store numbers. Therefore, we must work intelligently and choose a finite set of bits to represent our real-valued numbers, split between the integer parts and the fractional parts of the number. This is referred to as **fixed-point representation**.

This means that we for example choose a 16-bit fixed-point representation. And then

choose to have 4 of the total bits for the integer parts and the rest for fractional parts. This strategy helps us manage the amount of memory, but lacks something important.

Errors

Suppose I want to work with the strategy above. I want to have a representation of the number 0.625:

```
In[6]:= ToBinary[0.625]
Out[6]= 0.101
```

Everything seems fine now 0.6250000001:

```
In[7]:= ToBinary[0.6250000001]
Out[7]= 0.1010000000
```

See the difference? Exactly! There's none! My representation holds these to numbers as the same representation. It's not one-to-one. Something even worse can happen when I would try to have 23. Remember I only have 4 bits on my integer part:

```
In[8]:= ToBinary[23]
Out[8]= 10111
```

This is even worse! I cannot represent numbers greater than 15! There must be a more accurate and free way.

Floating-Point Representation

Commonly, computers don't use fixed-point representation (thanks to intelligent people who saw this coming). Not being able to control where the decimal point goes is inconvenient and stupid (in most cases).

The solution is to use **floating-point representation**, which works in a similar manner to scientific notation. We ensure that all numbers, regardless of their size, are represented using the same number of significant figures. As an example we write the following numbers in scientific notation:

$(a)\ 10 = 1010_2 = 1.0100000_2 \times 2^3$
$(b)\ -0.8125 = -0.1101_2 = -1.1010000_2 \times 2^{-1}$
$(c)\ 9.2 = 1001.0011001100110011\ldots_2 \approx 1.001001_2 \times 2^3$

Unlike the usual base-10, we use exponents of 2 in our base-2 scientific notation. Even

more importantly, as we are restricting the number of significant figures of our base-2 scientific notation, there is still truncation error in our representation of 9.2.

Representation

Floating-point representation as used by digital computers in written in the following normalised form:

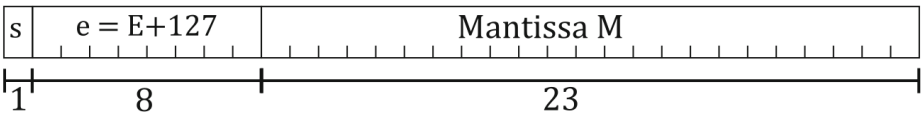
$$(-1)^S \times \left(1 + \sum_i M_i \, 2^{-i}\right) \times 2^E, \; E = e - d,$$

where:

- S is the sign bit: 0 for positive numbers and 1 for negative numbers
- M is the mantissa: bits that encode the fractional significant figures (the significant figures to the right of the decimal point -- the leading 1 is implied),
- E is the exponents of the number to be encoded,
- d is the offset of excess: The excess allows the representation of negative exponents, and is determined by the floating point standard used.
- e: bits that encode the exponent after being shifted by the excess.

The most common standard for floating-point representation is **IEEE754**; in fact, this standard is so universal that it will be very unlikely for you to encounter a computer that doesn't use it (unless you happen to find yourself using a mainframe or a very specialised supercomputer)

IEEE754 32-Bit Single Precision



The IEEE754 32-bit single precision floating-point number is so-called because of the 32-bits required to implement this standard. The bits are assigned as the figure above. 1 bit for the sign, 8 bits for the exponent, and 23 bits for the mantissa.

To jump into code right away. I have implemented a function that given a number transforms it into IEEE754 32-bit single precision representation:

In[9]:=
```
IEEE754SinglePrecision[number_Real] :=
 Module[{signBit, absNum, exponent, mantissa, exponentBits,
   mantissaBits, intPart, fracPart, normalizedMantissa, count},

  (*Step 1:Compute the sign bit*)
  signBit = If[number < 0, "1", "0"];
  absNum = Abs[number];

  (*Special Cases*)
  (*Zero case*)
  If[absNum == 0, Return["00000000000000000000000000000000"]];
  (*Infinity*)If[absNum == ∞, Return[signBit <> "11111111000000000000000000000000"]];
  (*NaN*)
  If[! NumericQ[absNum],
   Return[signBit <> "11111111111111111111111111111111"]];

  (*Step 2:Convert number to binary scientific notation*)exponent = Floor[Log2[absNum]];
  (*Get exponent*)
  normalizedMantissa = absNum / 2 ^ exponent;
  (*Normalize to 1.xxx form*)
  (*Step 3:Compute exponent bits (bias 127)*)
  exponentBits = IntegerString[exponent + 127, 2, 8];
  (*Step 4:Extract mantissa bits (23 bits)*)
  normalizedMantissa -= 1;
  (*Remove the leading 1 in 1.xxx*)
  mantissaBits = "";
  fracPart = normalizedMantissa;
  count = 0;
  While[fracPart > 0 && count < 23, fracPart *= 2;
   mantissaBits = mantissaBits <> ToString[Floor[fracPart]];
   fracPart -= Floor[fracPart];
   count++;];
  (*Ensure mantissa is exactly 23 bits long*)
  mantissaBits = StringPadRight[mantissaBits, 23, "0"];
  (*Step 5:Construct the final IEEE754 representation*)
  signBit <> exponentBits <> mantissaBits]
```

In[10]:=
```
IEEE754SinglePrecision[23.625]
```

Out[10]=
```
01000001101111010000000000000000
```

- **Exercise:** Convert 9.2 using the function, What is the percentage difference between the actual stored value and 9.2? This is the **truncation error.**

- **Exercise:** What is the possible range of numbers that we can encode using this standard? (Notice the special cases as well).
- **Exercise:** What are other standards, checkout IEEE754 64-bit Double, and Quadruple.

Floating-Point Arithmetic & Error

Arithmetic

When working through theoretical calculation using our standard mathematical toolset, we are used to applying the standard properties of arithmetic (such as multiplicative and additive commutativity). But do these properties still apply in the case of arithmetic performed on computers using floating-point representation? Consider the expression $-5+5+0.05$. Using the property of associativity we could calculate this using either $(-5+5)+0.05$ or $-5+(5+0.05)$. These are equivalent if we represent these numbers exactly, with no loss of precision.

This is what doesn't happen with our representation in computers. For the first expression if we try to represent 5,-5 we would have no problems but for 0.05:

In[11]:=
```
ToBinary[0.05]
```

Out[11]=
```
0.0000110011
```

In[12]:=
```
% // ToDecimal // N
```

Out[12]=
```
0.0498047
```

And for IEEE754 as well. The representation lacks precision.

In[13]:=
```
IEEE754SinglePrecision[0.05] == IEEE754SinglePrecision[0.050000000745058059692]
```

Out[13]=
```
True
```

The error we see is round-off/truncation error, due to limited precision.

- **Exercise:** Show that associativity doesn't work! By changing the order in which we add floating-point numbers together, we change the amount of truncation error that occurs. This can't be eliminated entirely; all we can do is to try and perform operations on number of similar magnitude first, in order to reduce the truncation error.

Like associativity, the property of distributivity also does not always hold for floating-point arithmetic, for example:

```
In[14]:= analyticAnswer = 0.1 + 0.1 + 0.1
         binaryAnswer = ToDecimal[ToBinary[0.1]] + ToDecimal[ToBinary[0.1]] + ToDecimal[ToBinary[0.1]] // N
Out[14]= 0.3
Out[15]= 0.298828
```

Machine Epsilon

In order to get a rough 'upper-bound' on the relative round-off error that occurs during floating-point arithmetic, it is common to see standards, programming languages, and software packages quote something called machine epsilon.

Machine Epsilon and Round-Off Error

Machine epsilon is defined by the difference between the floating-point representation of 1 and the next highest number. In Single precision, we know that $1 \equiv 1.00000000000000000000000_2$ and the next highest number is represented by changing the least significant bit in the mantissa to 1, giving $1.00000000000000000000001_2$. Therefore, in single precision IEEE754 floating-point representation:

$$\epsilon = 2^{-23}$$

Repeating this process for double precision floating-point numbers results in $\epsilon = 2^{-52}$. In general, if the mantissa is encoded using M bits resulting in $M + 1$ significant figures the machine epsilon is $\epsilon = 2^{-M}$

Relative Error

The important take-away here is that this value represents the minimum possible spacing between two floating point numbers, and as such provides an upper bound on the relative truncation error introduced by floating point numbers. To see how this works, lets consider a floating-point representation of a number x; we'll denote this $F(x)$. To calculate the relative error of the floating-point representation, we subtract the original exact value x, and divide by x:

$$\text{relative error} = \frac{F(x) - x}{x}$$

Now, we know that the relative error must be bound by the machine epsilon, so substituting this in to form an inequality:

$$-\epsilon \leq \frac{F(x) - x}{x} \leq \epsilon$$

by taking the absolute value and rearranging this equation, we also come across an expression for and upper bound on the absolute error of the floating-point representation:

$$\left| F(x) - x \right| \leq \epsilon \left| x \right|$$

Basic Algorithm

Developing Pseudo-Code for Scientific Problems

Writing Clear and Concise Pseudocode

Pseudocode serves as a crucial intermediate representation between abstract algorithmic concepts and concrete programming implementations. Its value lies in its ability to communicate computational thinking without getting bogged down in language-specific syntax.

Core Principles

Readability

Pseudocode that prioritizes readability uses vocabulary and structures that feel intuitive to both technical and non-technical audiences. This means:

- Using descriptive variable names that indicate their purpose (e.g., totalStudents rather than just t)
- Employing everyday language for operations (e.g., “add item to list” instead of “push item onto stack”)

- Including brief comments for any potentially confusing sections
- Breaking complex operations into named sub-procedures

Simplicity

Effective pseudocode distills algorithms to their essential logic by:

- Avoiding implementation details irrelevant to the core algorithm
- Omitting memory management considerations
- Abstracting away error handling when it doesn't affect the main flow
- Focusing on the "what" and "why" rather than the "how" of each step

Structure

Well-structured pseudocode provides clear visual cues about program flow through:

- Consistent indentation to show nested blocks and scope
- Clear delineation of conditionals, loops, and function definitions
- Visual separation between major algorithm components
- Sequential numbering for steps that must occur in a specific order

Level of Abstraction

Choose an appropriate level of abstraction based on your audience:

- Higher abstraction for conceptual discussions
- Lower abstraction when implementation details matter

Conventions

Establish consistent conventions for:

- Representing input/output operations.
- Indicating assignment versus comparison
- Denoting loops and conditionals

Converting Mathematical Formulations into Algorithmic Steps

Often, scientific problems start as mathematical equations. The challenge is to translate these equations into a series of computational steps. For instance, consider a basic numerical integration problem where you need to approximate the integral of a function $f(x)$ over an interval $[a, b]$. The mathematical formulation might look like:

$$\int_a^b f(x)\,dx$$

The algorithmic approach involves discretizing the interval and summing the contributions:

1. Define n subdivisions.
2. Calculate $\Delta x = \frac{b-a}{n}$.
3. Sum the areas: Area $\approx \sum_{i=0}^{n-1} f\left(a + i\,\Delta x\right)\Delta x$.

Analyzing Algorithm Complexity and Efficiency

Understanding and evaluating the performance of your algorithms is as important as designing them. If you write a super complicated algorithm or a very simple one, what matters most then is how well is it performing. Algorithm complexity measures the resources that your algorithm consumes as a function of input size.

- **Time Complexity** gauges how the running time scales, often expressed in terms of Big O notation.
- **Space Complexity** looks at memory usage.

These metrics let you compare different approaches and decide which algorithm will remain efficient as your problem scales.

Big O Notation and Asymptotic Behavior

Big O notation gives you a high-level view of how an algorithm scales by focusing on its dominant term

- Constant Time: $O(1)$
- Logarithmic Time: $O\left(\log n\right)$
- Linear Time: $O(n)$
- Linearithmic Time: $O\left(n \log n\right)$
- Quadratic Time: $O\left(n^2\right)$, and so on.

Opinionated Note:

Don't get hung up on constant factors—what really matters is the order of growth. A quadratic algorithm might seem fine for small inputs, but as the dataset grows, its performance will tank compared to a linearithmic one.

Example:

Consider sorting algorithms. Quicksort has an average-case complexity of $O\left(n \log n\right)$, which makes it excellent for most practical scenarios. However, its worst-case can degrade to $O(n^2)$, so understanding the input characteristics is key.

Worst-Case vs Average-Case vs Best-Case

Worst-Case Analysis

This measures the maximum amount of work an algorithm will perform on any input of size n

Average-Case Analysis

Here, you take a probabilistic view, averaging the running time over all inputs.

Best-Case Analysis

Although less frequently used, this tells you the minimum work an algorithm could do.

Trade-offs Between Time and Space Complexity

Sometimes you can trade extra memory for faster running time and vice versa.

- Caching and Memoization: Storing intermediate results can drastically reduce time complexity, but at the cost of increased memory usage.
- Loop Unrolling: By unrolling loops, you reduce the overhead of loop control. This speeds up execution but may increase code size, potentially affecting cache performance.

Example

In numerical integration, using a refined approach (more subdivisions) increases precision (space complexity in terms of data storage) but also increases the number of computations (time complexity). Balancing these factors is key.

Data Structures for Scientific Computing

Arrays, Matrices, and Tensors for Physical Quantities

Fundamental Data Structures

Other than primitive types such as integers, double, string; Arrays, and matrices form the backbone of scientific computing, providing structured ways to represent physical quantities across various dimensions:

Arrays

Arrays store sequences of elements, typically of the same data type, indexed by one or more integers. In scientific computing, they often represent:

- Time series data
- Discretized spatial coordinates
- Collections of measured values

One-dimensional arrays (vectors) represent quantities with magnitude and direction, such as velocity, force, or electric field at a point.

Matrices

Matrices are two - dimensional arrays that represent linear transformations or systems of linear equations. Key operations include :

- Addition and subtraction: Element-wise operations
- Matrix multiplication: Represents composition of linear transformations
- Inversion: Finding A^{-1} such that $A^{-1} A = I$
- Decompositions: Factoring matrices into products of simpler matrices
  - LU decomposition: $A = L U$ (L lower triangular, U upper triangular)

- QR decomposition: $A = Q\,R$ (Q orthogonal, R upper triangular)
- Singular Value Decomposition (SVD): A = $U\,\Sigma\,V^*$ (powerful for analysis)
- Eigendecomposition: $A = P\,D\,P^{-1}$ (for diagonalizable matrices)

Implementation Considerations

Memory Layout

- **Row-Major vs. Column-Major Storage**
  - Row-major (C/C++): Elements in the same row are stored contiguously
  - Column-major (FORTRAN, MATLAB): Elements in the same column are stored contiguously

This choice affects cache performance when accessing elements in different patterns.

- **Memory Alignment**
  - Aligned data allows for vectorized operations (SIMD)
  - Misaligned data may cause cache line splits and performance degradation
- **Padding and Striding**
  - Padding adds extra bytes between elements for alignment
  - Striding allows for efficient subsampling or working with non-contiguous data

Performance Implications

Cache Efficiency

Traversing arrays in the order they're stored in memory maximizes cache hits:

- For row-major: iterate through rows first, then columns
- For column-major: iterate through columns first, then rows

SIMD Vectorization

Single Instruction Multiple Data operations can perform calculations on multiple elements simultaneously:

- Requires properly aligned contiguous memory
- Can achieve 4-16× speedup for floating-point operations

Data Transfer Minimization

- Minimize data movement between CPU and memory

- Use techniques like loop tiling/blocking to improve data locality

---

Sparse Data Structures For Efficiency

Need for Sparsity

In many scientific applications, matrices contain mostly zeros. For example:

- Finite element discretizations of PDEs
- Molecular dynamics simulations
- Network adjacency matrices
- Quantum mechanical systems

Advantages of Sparse Representation

- Reduced memory usage (only store non-zero elements)
- Faster computation (avoid operations with zeros)
- Ability to handle larger problems that wouldn't fit in memory using dense formats

A matrix is typically considered “sparse” when $< 10 - 15$ percent of elements are non-zero.

Types of Sparse Representations

Coordinate Format (COO)

Stores Three Arrays:

- Values: non-zero values
- Rows: row indices for each value
- Cols: column indices for each value

Advantages:

- Simple implementation
- Efficient for construction and modification
- Good for incremental building

Disadvantages:

- Less efficient for computation

- Requires more storage than other formats

Compressed Sparse Row (CSR) Format

Stores Three Arrays:

- Values: non-zero values
- Columns: column indices for each value
- Column pointers: indexes into values where each column starts

For matrix with n rows and n_z non-zeros:

- Values: array of length n_z
- Columns: array of length n_z
- Row pointers: array of length $n+1$

Advantages:

- Efficient row-wise operations
- Compact storage
- Fast matrix-vector multiplication

Disadvantages

- Column access is inefficient
- Modification requires restructuring

Compressed Sparse Column (CSC) Format

Similar to CSR but oriented by columns:

- Values: non-zero values
- Rows: row indices for each value
- Row pointers: indexes into values where each row starts

Advantages:

- Efficient column-wise operations
- Natural for certain algorithms (e.g., sparse Cholesky factorization)

Disadvantages

- Row access is inefficient
- Modification requires restructuring

These formats exploit patterns where non-zeros occur in blocks rather than individually.

Further Learning

- **Data Structures Tutorial:** *https://www.geeksforgeeks.org/data-structures/*
- **Computer Science Concepts:** *https://www.youtube.com/playlist?list=PLWKjhJtqVAbn5emQ3RRG8gEBqkhf_5vxD*
- **Best Data Structures and Algorithms Books:** *https://hackr.io/blog/best-data-structures-and-algorithms-books*