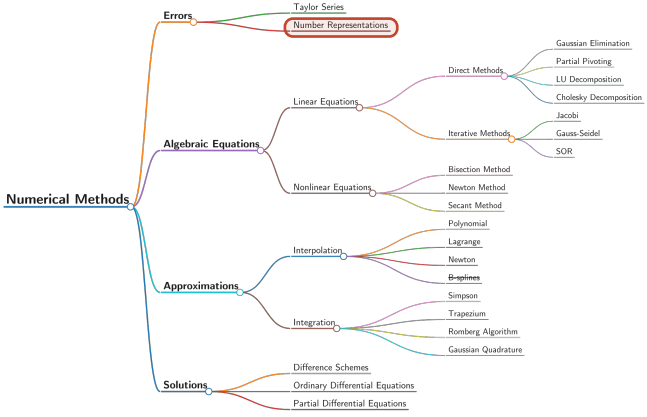
Number Representations

CTMS-MAT-13 · Numerical Methods

Dr. David Sinden

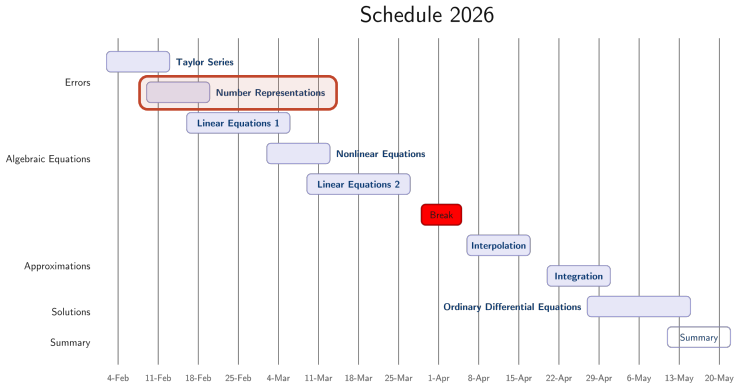
September 10, 2026

Where this sits in the course



Full outline: *djps.github.io/docs/numericalmethods/intro/intro/*

Schedule



Notes: *djps.github.io/teaching/numerical_methods.pdf*

Number Representations

Types of Error

Bases

Floating point

Types of Error

Three kinds of error

1. **Errors in the data** — the inputs were never exact.
2. **Round-off error** — the machine cannot store what it computed.
3. **Truncation error** — an (possibly infinite) process was stopped early.

TAKEAWAY

Only the third is under your control as an analyst; the second is a property of the representation, and this lecture is about that representation.

Absolute and relative error

Let $\tilde{a}$ approximate a.

absolute: $|\tilde{a} - a|$

relative: $\dfrac{|\tilde{a} - a|}{|a|}$

Error bound

The magnitude of the *admissible* error — a range imposed on one of the two quantities above, agreed before the computation starts.

Relative error is the one that matters on a computer: floating-point precision is constant in *relative* terms, not absolute.

Error propagation

If $y(x)$ is differentiable, $|y'(x)|$ is the sensitivity of y to error in x:

$$|\Delta y| \approx |y'(x)|\,|\Delta x|.$$

Several variables, $\vec{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$

$$|\Delta y| = \sum_{i=1}^{n} \left|\frac{\partial y}{\partial x_i}\right| |\Delta x_i|, \qquad \Delta y = \tilde{y} - y, \quad \Delta x_i = \tilde{x}_i - x_i.$$

A large derivative amplifies small input errors — the model, not the arithmetic, is then the problem.

Bases

Expansion in base b

Let $b \in \mathbb{N} \setminus \{1\}$. Every $x \in \mathbb{N}_0$ has a unique expansion

$$x = \sum_{i=0}^{n} a_i b^i, \qquad a_i \in \{0, 1, \ldots, b-1\}.$$

For real numbers a fractional part is appended:

$$x = \sum_{i=0}^{n} a_i b^i + \sum_{i=1}^{\infty} \alpha_i b^{-i} = \underbrace{a_n \cdots a_0}_{\text{integer part}} \, . \, \underbrace{\alpha_1 \alpha_2 \cdots}_{\text{fractional part}}$$

Three bases, one rule

Base 10

$$37294 = 4 \cdot 10^0 + 9 \cdot 10^1 + 2 \cdot 10^2 + 7 \cdot 10^3 + 3 \cdot 10^4$$

Base 2

$$(1011)_2 = 1 + 2 + 0 + 8 = (11)_{10}$$

Base 16

Digits $0 \ldots 9$, *a*, *b*, *c*, *d*, *e*, *f*. A web colour *#a0f954* is three two-digit base-16 numbers:

$$16 \times 16 = 256 \;\Rightarrow\; R \in \{0, \ldots, 255\}.$$

TAKEAWAY

The base changes the digits, never the number. Only the *cost* of writing it down changes.

Euclid's algorithm: digits from the top down

Input: $(x)_{10}$

1. find the smallest n with $x < b^{n+1}$;
2. for $i = n$ down to 0:

$$a_i = x \operatorname{\mathbf{div}} b^i$$
$$x = x \operatorname{\mathbf{mod}} b^i$$

3. $(x)_b = \overline{a_n \cdots a_0}$.

Example: $x = (13)_{10}$, $b = 2$

$13 < 2^4$, so $n = 3$.

$$a_3 = 13 \operatorname{\mathbf{div}} 8 = 1, \quad r = 5$$
$$a_2 = 5 \operatorname{\mathbf{div}} 4 = 1, \quad r = 1$$
$$a_1 = 1 \operatorname{\mathbf{div}} 2 = 0, \quad r = 1$$
$$a_0 = 1 \operatorname{\mathbf{div}} 1 = 1$$

$(13)_{10} = (1101)_2 = 8 + 4 + 1.$

Horner's algorithm: digits from the bottom up

Euclid's method has two weaknesses: finding n is awkward, and the loop over $i = n, \ldots, 0$ is wasteful. Horner (1786–1827) factors the number instead:

$$(a_n \cdots a_0)_b = a_0 + b\big(a_1 + b(a_2 + b(a_3 + \cdots))\big).$$

Input: $(x)_{10}$, $i = 0$
while $x > 0$:

$$a_i = x \bmod b$$
$$x = x \operatorname{div} b$$
$$i = i + 1$$

No n is needed — the loop stops when x runs out. The digits emerge in reverse order.

In Python, `//` is **div** and `%` is **mod**.

Simple in one base $\neq$ simple in another

The decimal that isn't

$$(0.1)_{10} = (0.000\overline{1100})_2$$

A finite decimal becomes a *repeating* binary fraction — so `0.1` is not stored exactly by any binary machine.

- Base 2 is what the hardware uses; bases 8 and 16 are compact ways of writing it.
- Powers of two convert digit-wise: $(551.624)_8 = (101\,101\,001.\cdots)_2$.

Floating point

The standard representation

$$x = 0.a_1 a_2 \cdots a_k \cdot b^n, \qquad a_i \in \{0, \ldots, b-1\}$$

- k — the precision, the number of stored digits;
- n — the exponent, setting the scale;
- $a_1 \cdots a_k$ — the mantissa.

Requiring $a_1 \neq 0$ is **normalisation**; it makes the representation unique. In base 10, $x = 32.213$ becomes $0.\underbrace{32213}_{a_1 \neq 0} \times 10^2$.

Single precision (32 bits)

1 bit	sign of the number
1 bit	sign of the exponent
7 bits	exponent
23 bits	mantissa

$2^7 = 127$ and $2^{127} \approx 10^{38}$, so

$$10^{-38} < |x| < 10^{+38}.$$

Double precision spends 64 bits and reaches $\approx 10^{308}$.

TAKEAWAY

Fixed k means fixed *relative* precision: the gap between representable numbers grows with their magnitude.

Issue (i): addition is not associative

Commutative, yes. Associative, no.

$a + b = b + a$ always, but

$$(x + y) + z \neq x + (y + z)$$

in finite precision.

Base 10 with $k = 7$ significant figures, $x = y = 0.0000033$, $z = 0.00000034$, $w = 1.000000$:

$$\big((x + y) + z\big) + w = 1.000007, \qquad \big((w + z) + y\big) + x = 1.000000.$$

Why the order matters

Adding smallest-first keeps the small quantities together until they are large enough to survive:

$$\begin{aligned} & 0.3300000 \cdot 10^{-5} \\ + \; & 0.3300000 \cdot 10^{-5} \\ + \; & 0.3400000 \cdot 10^{-6} \\ + \; & 0.1000000 \cdot 10^{+1} \quad (w) \end{aligned}$$

TAKEAWAY

Adding a small number to a large one discards the small number's significant digits. Sum from smallest to largest.

Issue (ii): cancellation in subtraction

Let $x > y > 0$ be normalised, base 2. If $p, q \in \mathbb{N}_0$ satisfy

$$2^{-p} < 1 - y/x \;\le\; 2^{-q},$$

then at most p and at least q significant bits are lost in $x - y$.

The closer y is to x, the more of the answer is noise: leading digits cancel and the trailing digits promoted to take their place were never accurate.

Cancellation in practice: $x - \sin x$ near 0

Take $x = 1/15$ with $k = 10$:

$$\begin{aligned} x &= 0.6666666667 \times 10^{-1} \\ \sin x &= 0.6661729492 \times 10^{-1} \\ x - \sin x &= 0.0004937175 \times 10^{-1} = 0.4937175000 \times 10^{-4} \end{aligned}$$

Seven digits agreed and cancelled. The result carries three fabricated zeros: three decimal places of precision were destroyed by a single subtraction.

The fix: rewrite the expression

Use the series for $\sin x$ before subtracting, not after:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

$$\Longrightarrow \quad x - \sin x = \frac{x^3}{3!} - \frac{x^5}{5!} + \frac{x^7}{7!} - \cdots$$

TAKEAWAY

The subtraction is gone: nothing nearly-equal is ever differenced, so every digit of the answer is earned. Same mathematics, different arithmetic.

The number you typed,
the number stored,
and the number computed
are three different numbers.

Summary

- Error comes from the data, from round-off, and from truncation — name which one you are fighting before you fix it.
- A base changes only the digits. Binary is the machine's base, and finite decimals need not be finite in it.
- Euclid's algorithm converts from the top down; Horner's from the bottom up, and cheaper.
- Floating point stores a normalised mantissa and an exponent: precision is fixed in *relative* terms.
- Finite precision breaks associativity, and subtracting nearly equal numbers destroys significance.

Take-home messages

1. **Judge accuracy relatively.** A 10^{-6} absolute error means nothing until you know the size of the answer.
2. **Sum from smallest to largest.** Order of operations is part of the algorithm, not a detail of the code.
3. **Never subtract nearly equal quantities.** Rewrite the expression — by series, by factoring, by conjugates — so the cancellation never happens.
4. **Never test floating-point numbers for equality.** Compare against a tolerance; `0.1 + 0.2 == 0.3` is false.

Next: root finding and iterative methods.