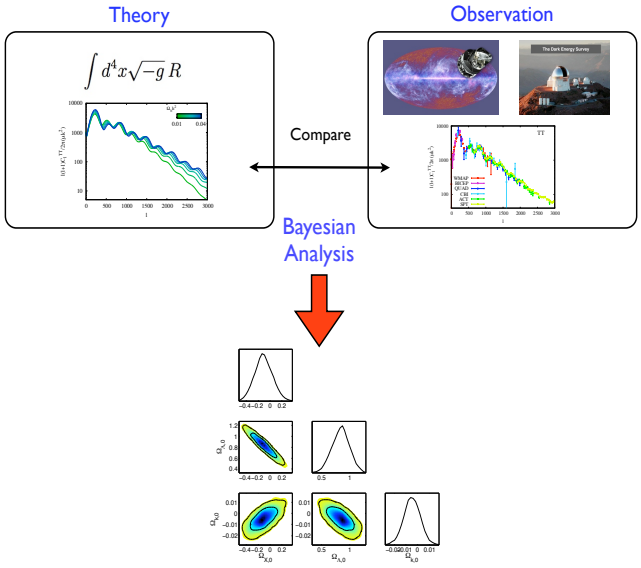
Numerical Methods



José-Alberto Vázquez

ICF-UNAM / Kavli-Cambridge

In progress

August 12, 2021

1

Numerical Analysis

- Computer arithmetic is not the same as 'pencil and paper' arithmetic.
- A hand calculation will usually be short, whereas a computer calculation can involve millions of steps. **Tiny errors** that would be negligible in a short calculation can be devastating when **accumulated** over a long calculation.

Only rational numbers (not all) can be represented exactly: (py: $(\sqrt{3})^2$)

1.0.1 Example 1

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots \tag{1.1}$$

x	$E(X)$	e^x
1	2.718282	2.718282
5	148.4132	148.4132
10	22026.47	22026.46
15	3269017.	3269017.
20	4.8516531×10^8	4.8516520×10^8
-1	.3678794	.3678795
-5	6.7377836×10^{-3}	6.7379470×10^{-3}
-10	$-1.6408609 \times 10^{-4}$	4.5399930×10^{-5}
-15	$-2.2377001 \times 10^{-2}$	3.0590232×10^{-7}
-20	1.202966	2.0611537×10^{-9}

Figure 1.1: (hw: WH- Do the table)

1.0.2 Example 2

The derivative of f at x is defined by

$$f'(x) \simeq \frac{f(x+h) - f(x)}{h} \equiv \Delta_h f(x)$$

Compute $f'(x)|_{x=1}$.

h	$\Delta_h f(1)$	e	error
10^0	4.67077446	2.71828183	1.95×10^0
10^{-1}	2.85884380	2.71828183	1.41×10^{-1}
10^{-2}	2.73191929	2.71828183	1.36×10^{-2}
10^{-3}	2.71987939	2.71828183	1.60×10^{-3}
10^{-4}	2.72035623	2.71828183	2.07×10^{-3}
10^{-5}	2.71797204	2.71828183	3.10×10^{-4}
10^{-6}	2.62260461	2.71828183	9.57×10^{-2}
10^{-7}	4.76837206	2.71828183	2.05×10^0
10^{-8}	0.00000000	2.71828183	2.72×10^0

Figure 1.2: (hw: WH- Do the table)

1.0.3 Example 3

Solve the system (here: Do it now)

$$0.780x + 0.563y = 0.217 \tag{1.2}$$

$$0.457x + 0.330y = 0.127 \tag{1.3}$$

we get the solution

$$x = 1.71 \qquad y = -1.98$$

and put them back we get a the rhs $0.00206, 0.00107$. And the exact solution should be $1, -1$

1.1 Approximations and Round-off Errors

The **significant digits** of a number are those that can be used with confidence (the known to be correct). They correspond to the number of **certain digits** plus **one estimated digit**. It is conventional to set the estimated digit at **one-half of the smallest scale** division on the

measurement device.

Ascertain the significant figures of a number, some cases can lead to confusion: **zeros are not always significant figures** because they may be necessary just to locate a decimal point The numbers 0.00001845, 0.0001845, and 0.001845 all have four significant figures.

The number 45,300 may have three, four, or five significant digits, depending on whether the **zeros are known with confidence**.

Numerical methods yield approximate results. we might **decide** that our approximation is acceptable if it is correct to certain significant figures.

$\pi, e, \sqrt{7}$ represent specific quantities, they **cannot be expressed exactly** by a limited number of digits: such numbers can never be represented exactly. The omission of the remaining significant figures is called **round-off error**. (8.49 universidad)

1.2 Accuracy and Precision

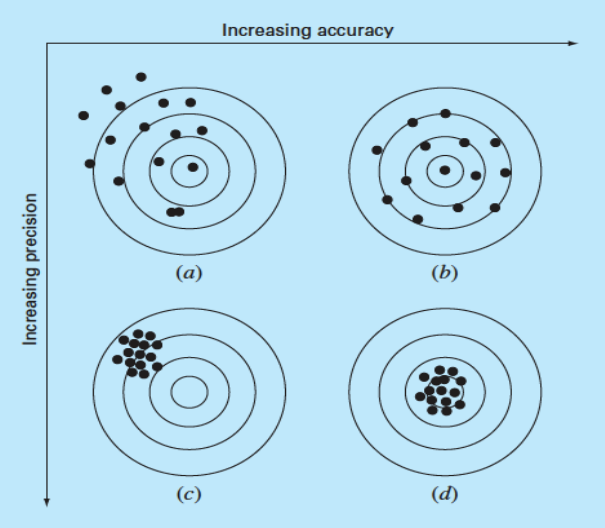
- **Accuracy** refers to how closely a computed or measured value agrees **with the true value** (is governed by the errors in the numerical approximation).
- **Precision** refers to how closely individual computed or measured values agree **with each other** (is governed by the number of digits being carried in the numerical calculations).
- **Inaccuracy** (also called bias) is defined as systematic **deviation from the truth**.
- **Imprecision** (also called uncertainty), on the other hand, refers to the magnitude of the **scatter**.

We will use the collective term **error** to represent both the inaccuracy and the imprecision of our predictions.

1.3 Errors

(here: Error_estimates.ipynb)

1. Errors in the parameters of the problem (units, assumed nonexistent).



2. Algebraic errors in the calculations (assumed nonexistent).
3. Iteration errors.
4. Approximation errors (truncation or round off).
5. Roundoff errors.

$$\text{True value} = \text{approximation} + \text{error}$$

rearranging

$$E_t = \text{true value} - \text{approximation}$$

something absolute value. E_t is designated the **'true' (absolute) error**.

It is not taking into account the order of magnitude, hence, normalize the error to the true value.

$$\text{True fractional relative error} = \frac{\text{true error}}{\text{true value}}.$$

The **true percent relative error**:

$$\epsilon_t = \frac{\text{true error}}{\text{true value}} \times 100\%.$$

However, in real-world applications, we will obviously **not know the true answer** a priori. The error of the approximation

$$\epsilon_a = \frac{\text{approximate error}}{\text{approximation}} \times 100\%.$$

Certain numerical methods use an **iterative** approach to compute answers. This process is performed repeatedly, or iteratively, to successively compute (we hope) better and better approximations.

$$\epsilon_a = \frac{\text{current approximation} - \text{previous approximation}}{\text{current approximation}} \times 100\%.$$

We are interested in whether the **percent absolute value is lower than a prespecified percent tolerance** ϵ_s.

$$|\epsilon_a| < \epsilon_s.$$

It is also convenient to relate these errors to the number of significant figures (n) in the approximation (Scarborough, 1966).

$$\epsilon_s = (0.5 \times 10^{2-n}\%)$$

1.4 Round-off errors

Numbers such as $\pi, e, \sqrt{7}$ cannot be expressed by a fixed number of significant figures. Therefore, they cannot be represented exactly by the computer (py: 0.1+0.2).

Computers use a base-2 representation, they cannot precisely represent certain exact base-10 numbers.

Number Systems. A number system is merely a convention for representing quantities

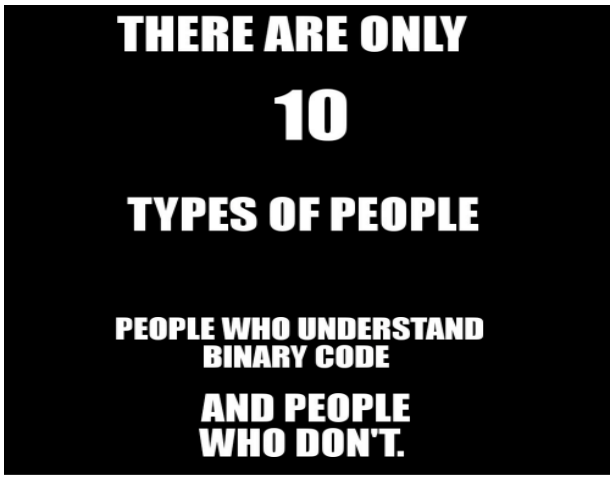
The base 10 system uses the 10 digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 to represent numbers. For larger quantities, combinations of these basic digits are used, with the **position** or place value specifying the magnitude. For example, if we have the number 86,409 (here: do it base 10) then we have eight groups of 10,000, six groups of 1000, four groups of 100, zero groups of 10, and nine more units, or

$$(8 \times 10^4) + (6 \times 10^3) + (4 \times 10^2) + (0 \times 10^1) + (9 \times 10^0) = 86409$$

This type of representation is called **positional notation**.

The fact that the primary logic units of digital computers are **on/off electronic components**. Hence, numbers on the computer are represented with a binary, or base-2, system.

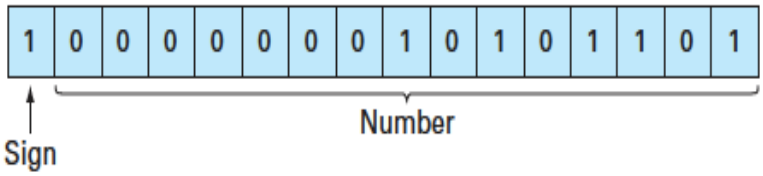
The number 11 is equivalent to $(1 \times 2^1) + (1 \times 2^0) = 2 + 1 = 3$ (here: do it), (hw: -175 in binary)



A nerd joke

1.5 Integer representation

The signed magnitude method: the first bit of a word to indicate the sign, with a 0 for positive and a 1 for negative. The remaining bits are used to store the number. For instance the number -173 on a 16 bit computer.



Exercise: Determine the range of integers in base-10 that can be represented on a 16-bit computer. (here: do it) – Cual es el numero mas grande que puede ser escrito en una computadora como estas?

The first bit holds the sign. The remaining 15 bits can hold binary numbers from 0 to 111111111111111, which is

$$(1 \times 2^{14}) + (1 \times 2^{13}) + \cdots + (1 \times 2^{1}) + (1 \times 2^{0}) = 32767$$

can be simplified to $2^{15} - 1$.

Hence, can store decimal integers ranging from -32,767 to 32,767. However, **zero is already defined** as 0000000000000000 (1000000000000000, minus zero). The range is from -32,768 to 32,767.

(here: do it.) In a 32 binary digits, the largest integer is $2^{31} - 1 = 2,147,647$ and the smallest -2^{31}.

1.6 Floating-Point Representation.

The number is expressed as a fractional part, called a **mantissa or significand**, and an integer part, called an **exponent or characteristic**, as in

$$m \cdot b^e$$

where m the mantissa, b the base of the number system being used, and e the exponent. For instance, the number 156.78 could be represented as 0.15678×10^3 in a floating-point base-10 system.

The first digit in the mantissa be non-zero (this is called **normalization**) to avoid ambiguity ($1.000 \times 10^{00} = 0.100 \times 10^{01}$).

In a 32 binary digits or bits (IEEE Standard), 24 bits are for the mantissa, 8 for the exponent.
The 64-bit : 11bit for the exponent, 52 bit for the mantissa (52 binary corresponds to 16/17 decimal digits.). Larger $2^{-1023}(1 + 2^{-52}) \sim 10^{-308}$, smaller $2^{1024}(2 + 2^{-52}) \sim 10^{308}$.

Several aspects of floating-point representation:

- There Is a Limited Range of Quantities That May Be Represented. **overflow** (goes outside the range) / **underflow** (the calculation result is too close to zero) error.

- There Are Only a Finite Number of Quantities That Can Be Represented within the Range. (here: 0.1 + 0.2)

  The actual approximation is accomplished in either of two ways: **chopping or rounding**. The value of $\pi = 3.14159265358\ldots$ is to be stored on a base- 10 number system carrying seven significant figures. Therefore, chop off, the eighth and higher terms, as in π =3.141592, with the introduction of an associated error of

$$\epsilon_t = 0.00000065$$

Rounding yields a lower absolute error than chopping. Computers that use IEEE format allow 24 bits to be used for the mantissa, which translates into about seven significant base-10 digits of precision1 with a range of about 10^{-38} to 10^{-39}.

Machine epsilon: The smallest floating point number we can addd to 1.0 and obtain a floating point results larger than 1.0. ϵmach $= 0.001 = 1.000 \times 10^3$ on a 4 digit machine, or ϵmach $= 0.0005$ is it is rounded.

1.7 Arithmetic manipulation of Computer Numbers

1.7.1 Arithmetic Operations

Because of their familiarity, normalized base-10 numbers will be employed to illustrate the effect of round-off errors on simple addition, etc..
To simplify the discussion, we will employ a hypothetical decimal computer with *a 4-digit mantissa and a 1-digit exponent*. In addition, chopping is used. Rounding would lead to similar though less dramatic errors.

When two floating-point numbers are added, **the mantissa of the number with the smaller exponent is modified so that the exponents are the same**. This has the effect of aligning the decimal points. This has the effect of aligning the decimal points.

For example, adding $0.1557 \cdot 10^1 + 0.4381 \cdot 10^{-1}$. First

$$0.4381 \cdot 10^{-1} \rightarrow 0.004381 \cdot 10^1$$

$$\begin{array}{r} 0.1557 \quad \cdot 10^1 \\ 0.004381 \cdot 10^1 \\ \hline 0.160081 \cdot 10^1 \end{array}$$

Because is a computer with a 4-digit mantissa, we're losing information. Even more dramatic results would be obtained when the numbers are very close, as in

$$\begin{array}{r} 0.7642 \cdot 10^3 \\ -0.7641 \cdot 10^3 \\ \hline 0.0001 \cdot 10^3 \end{array}$$

which would be converted to $0.1000 \cdot 10^0 = 0.1000$. Thus, for this case, three nonsignificant zeros are appended.

Multiplication and division are somewhat more straightforward than addition or subtraction. **The exponents are added and the mantissas multiplied**.

$$0.1363 \cdot 10^3 \times 0.6423 \cdot 10^{-1} = 0.08754549 \cdot 10^2$$

the result is normalized

$$0.08754549 \cdot 10^2 \rightarrow 0.8754549 \cdot 10^1$$

and chopped to give

$$0.8754 \cdot 10^1$$

Consequently, even though an individual round-off error could be small, the **cumulative effect** over the course of a large computation can be significant.

`https://numpy.org/doc/stable/user/basics.types.html`

see overflow-errors on how errors come out for very long numbers.

1.7.2 Adding a Large and Small Number

Using a hypothetical computer with the 4-digit mantissa and the 1-digit exponent. Adding 4000 and 0.0010

$$\begin{array}{r} 0.4000 \quad \cdot 10^4 \\ 0.0000001 \cdot 10^4 \\ \hline 0.4000001 \cdot 10^4 \end{array}$$

which is chopped to $0.4000 \cdot 10^4$. Thus, we might as well have not performed the addition!

(py: Cumulative effect)

1.8 Taylor Series

Then e^{10} and e^{-10}.

(hw: do it) Use

$$e^{-x} = \frac{1}{e^x} = \frac{1}{1 + x + \frac{x^2}{2!} + \cdots}$$

the problem was with the algorithm we choose. (py: do it)

Solve the quadratic equation a=1, b=3000.001, c=3. The true roots are $x_1 = -0.001$ and $x_2 = -3000$ (py: do it)

Here, b^2 is much larger than $4ac$, so the numerator in the calculation for x_1 involves the **subtraction of nearly equal numbers**. To obtain a more accurate four-digit rounding approx for x_1, we change the form of the quadratic formula by *rationalizing the numerator*

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}\left(\frac{-b - \sqrt{b^2 - 4ac}}{-b - \sqrt{b^2 - 4ac}}\right) = \frac{b^2 - (b^2 - 4ac)}{2a(-b - \sqrt{b^2 - 4ac})} \tag{1.4}$$

which simplifies to

$$x_1 = \frac{-2c}{b + \sqrt{b^2 - 4ac}}. \tag{1.5}$$

Compute the four roots of $x^4-4x^3+8x^2-16x+15.99999999$ (hw: do it). is $(x-2)^2 = \pm 10^{-4}$ and has roots $x_1 = 2.01, x_2 = 1.99$. However if the machine epsilon $> 10^{-10}$ the constant will be rounded to 16, and the problem would be

$$(x-2)^4 = 0,$$

with a 0.5% difference.

1.9 Numerical Differentiation

Finite divided difference:

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} + \mathcal{O}(x_{i+1} - x_i) \tag{1.6}$$

or

$$f'(x_i) = \frac{\Delta f_i}{h} + \mathcal{O}(h) \tag{1.7}$$

Δf_i is referred to as the **first forward difference** and h is called the step size More accurate approximations of the first derivative can be developed by including higher-order terms of the Taylor series.

Backward Difference Approximation of the First Derivative

The Taylor series can be expanded backward to calculate a previous value on the basis of a present value

$$f(x_{i-1}) = f(x_i) - f'(x_i)h + \frac{f''(x_i)}{2!}h^2 - \cdots \tag{1.8}$$

Truncating this equation after the first derivative and rearranging yields

$$f'(x_i) \simeq \frac{f(x_i) - f(x_{i-1})}{h} = \frac{\nabla f_i}{h}$$

first backward difference, where the error is $\mathcal{O}(h)$.

Centered Difference Approximation

Subtract eq. (**??**) from forward Taylor:

$$f(x_{i+1}) = f(x_i) + f'(x_i)h + \frac{f''(x_i)}{2!}h^2 - \cdots \tag{1.9}$$

to yield

$$f(x_{i+1}) = f(x_{i-1}) + 2f'(x_i)h + \frac{2f^{(3)}(x_i)}{3!}h^2 - \cdots \tag{1.10}$$

which can be solved for

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1})}{2h} - \mathcal{O}(h^2) \tag{1.11}$$

Is a centered difference representation of the first derivative. Notice that the truncation error is of the order of h^2 in contrast to the forward and backward approximations that were of the order of h.

Halving the step size approximately halves the error of the backward and forward differences and quarters the error of the centered difference. (py: do it)

Higher Derivatives

We write a forward Taylor series expansion for $f(x_{i+2})$ in terms of $f(x_i)$.

$$f(x_{i+2}) = f(x_i) + f'(x_i)(2h) + \frac{f''(x_i)}{2!}(2h)^2 - \cdots \tag{1.12}$$

Eq, (**??**) can be multiplied by 2 and subtracted from the previous one to give

$$f(x_{i+2}) - 2f(x_{i+1}) = -f(x_i) + f''(x_i)h^2 - \cdots \tag{1.13}$$

which can be solved for (*second forward finite divided difference*)

$$f''(x_i) = \frac{f(x_{i+2}) - 2f(x_{i+1}) + f(x_i)}{h^2} + \mathcal{O}(h) \tag{1.14}$$

(*a backward version*)

$$f''(x_i) = \frac{f(x_i) - 2f(x_{i-1}) + f(x_{i-2})}{h^2} + \mathcal{O}(h) \tag{1.15}$$

(*a centered version*)

$$f''(x_i) = \frac{f(x_{i+1}) - 2f(x_i) + f(x_{i-1})}{h^2} + \mathcal{O}(h^2) \tag{1.16}$$

Alternatively can be expressed as

$$f''(x_i) = \frac{\frac{f(x_{i+1})-f(x_i)}{h} - \frac{f(x_i)-f(x_{i-1})}{h}}{h} \tag{1.17}$$

The second derivative is a derivative of a derivative.

1.10 Total numerical error

The total numerical error is the summation of the truncation and round-off errors. The truncation error can be reduced by decreasing the step size. (py: in the polynomial, reduce h by 10.)