CHAPTER I Constraints of Numerical Methods

I. Introduction

- Numerical Analysis is concerned with the creation & study of algorithms dedicated to solving particular mathematical problems

- The rapidly evolving state of computing hardware means that the field of numerical analysis is rapidly evolving too.

  e.g.:
  - constraints on memory storage are not as dramatic as they used to be
  - the development of parallel computing calls for an entirely new generation of algorithms.

- There are many types of problems that one can investigate. In this class, we will focus only on "some"
  1. Numerical methods for Linear Algebra
  2. ________________ solving ODEs
  3. ________________ solving PDEs.

- As we discuss algorithms, we will always bear in mind the 4 standard concerns of numerical analysis:
  1. stability
  2. accuracy
  3. speed
  4. memory storage.

I The representation of numbers

The first two "concerns" of Numerical Analysis are indirectly (and sometimes directly) related to the way numbers are encoded by the computer hardware/architecture, on "bytes". (recall: 1 byte = 8 bits)

① Integer numbers

⇒ The representation of an integer number is (usually) <u>exact</u>.

<u>Recall</u>: The representation of an integer as a sequence of numbers from 0 to 9 can be thought of as an expansion in base 10:

$$a_n a_{n-1} \cdots a_0 = a_n \times 10^n + a_{n-1} \times 10^{n-1} + \cdots + a_0$$

e.g: $144 = 1 \times 100 + 4 \times 10 + 4$

However, the natural base for computing is base 2 (one "bit" is either 0 or 1)

→ write <u>positive</u> integers in base 2 as

$$a_n a_{n-1} \cdots a_0 = a_n \cdot 2^n + a_{n-1} 2^{n-1} + \cdots a_0$$

e.g: $25 = 1 \times 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$
$= 11001$ in base 2

Standard storage: a <u>signed</u> integer is typically stored on a finite number of bytes, usually using 1-bit for the sign (though other conventions also exist)

In Fortran it is common to use
- normal integers on 4 bytes = 32 bits (one for the sign, 31 for the value)

- long integers on 8 bytes = 64 bits
(1 bit for the sign, 63 for the value)

⇒ CONSEQUENCE!

For a given integer type there are only a finite number of integers which can be coded

- for normal 4-byte integers:

from -2^{31} to $+2^{31}$

- for long-integers

from -2^{63} to 2^{63}

Attemps to reach numbers beyond these values can/will create problems. Note that $2^{31} \cong 2.1$ billion → not that big a number.

② Floating point numbers

Again note that the base-10 notation

$$a_n a_{n-1} \cdots a_0 . b_1 b_2 \cdots b_m$$

means $a_n 10^n + a_{n-1} 10^{n-1} + \cdots a_0 + b_1 10^{-1} + b_2 10^{-2} + \cdots b_m 10^{-n}$

→ by analogy we define a base-2 rational number as

$$a_n a_{n-1} \cdots a_0 . b_1 b_2 \cdots b_m = a_n 2^n + a_{n-1} 2^{n-1} + \cdots a_0 + b_1 2^{-1} + b_2 2^{-2} + \cdots b_m 2^{-n}$$

Note: Finite storage possibilities means that only a finite number of $\{a\}$ and $\{b\}$ coefficients can be stored → real numbers can only be approximated ⇒ $\boxed{\text{ROUNDOFF ERROR}}$

Standard notation: in fact, the numbers are not stored as written above: first, either write them as

$$2^n \left[a_n + a_{n-1} 2^{-1} + \cdots + a_0 2^{-n} + b_1 2^{-n-1} + \cdots b_m 2^{-n-m} \right]$$

or $$2^{n+1} \left[a_n 2^{-1} + \cdots\cdots + b_m 2^{-n-m-1} \right]$$

→ in the first case a_n can be chosen to be $\neq 0$ by assumption (and is therefore necessarily $a_n = 1$).

Examples

27.25 in base 10 becomes

$$= 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = (11011.01) \text{ in base 2}$$

↳ two ways of writing it:

In the form of $x = 2^k \cdot f$

$$\begin{cases} = 2^4 \, [1.101101] & \leftarrow \text{IEEE standard} \\ = 2^5 \, [0.1101101] & \leftarrow \text{DEC standard.} \end{cases}$$

k is called the exponent
f is called the mantissa.

Standard Fortran storage: there are two standard types, in addition to the possibility of defining any new type we want (F90 and above only)

Typically: "single" precision (type REAL(SP))
→ storage on 4 bytes
$= \begin{cases} 1 \text{ bit for sign} \\ 8 \text{ bit for the exponent} \\ 23 \text{ bits for the mantissa} \end{cases}$

"double" precision (type REAL(DP))
→ storage on 8 bytes
$= \begin{cases} 1 \text{ bit for the sign} \\ 11 \text{ bits for the exponent} \\ 52 \text{ bits for the mantissa.} \end{cases}$

Note: the bits in the exponent store integers from L to U (L = lowest exponent, a negative integer usually)
U = highest exponent, a positive ....)

with $U - L = 2^8$ for SP
$= 2^{11}$ for DP

(3) Floating point arithmetic & roundoff errors

Arithmetic using floating pt numbers is "quite" different from real arithmetic. To understand this, let's work in base 10 for simplicity

- in, say, DEC standard, π to
  - 2 decimal places (significant digits) is 0.31
  - 6 ——————————— is 0.314159

- for a given number of significant digits (i.e a given length of mantissa), the "distance" between 2 consecutive numbers is dependent on the value of the exponent.

e.g : suppose we had 3 possible values of the exponent ($k = -2, -1$ and 0) and worked with a 2-digit mantissa. The positive numbers we can create are

$$\left.\begin{array}{l} 0.10 \times 10^{-2} \\ 0.11 \times 10^{-2} \\ \vdots \\ 0.98 \times 10^{-2} \\ 0.99 \times 10^{-2} \end{array}\right\} \text{all separated by } 10^{-4}$$

$$\left.\begin{array}{l} 0.10 \times 10^{-1} \\ 0.11 \times 10^{-1} \\ \vdots \\ 0.98 \times 10^{-1} \\ 0.99 \times 10^{-1} \end{array}\right\} \text{all separated by } 10^{-3}$$

$$\left.\begin{array}{l} 0.10 \times 10^{-0} \\ 0.11 \times 10^{-0} \\ \vdots \\ 0.98 \times 10^{-0} \\ 0.99 \times 10^{-0} \end{array}\right\} \text{all separated by } 10^{-2}.$$

$\Rightarrow$ the real axis has been discretized $\rightarrow$ roundoff errors

$\Rightarrow$ the discretization is <u>NOT</u> uniform $\rightarrow$ depend on the absolute value of the numbers considered

<u>Note</u> As a result of roundoff errors, FP arithmetic is not associative

<u>Example</u> : with a 6-digit mantissa

$$\begin{cases} a = 472635 & (= 0.472635 \times 10^6) \\ b = 472630 & (= 0.472630 \times 10^6) \\ c = 27.5013 & (= 0.275013 \times 10^2) \end{cases}$$

then $(a-b)+c \neq a-(b-c)$ in FP arithmetic

In first case

$$\begin{aligned} (a-b)+c &= (472635 - 472630) + 27.5013 \\ &= 5.00000 + 27.5013 \\ &= 32.5013 \end{aligned}$$

but

$$\begin{aligned} a-(b-c) &= 472635 - (472630 - 27.5013) \\ &= 472635 - \underbrace{472602.4987}_{\text{more than 6-digits so must be rounded off to}} \\ &= 472635 - 472602 \\ &= 33.0000 \end{aligned}$$

→ the error on the calculation is large!
it is of the order of the discretization error for the largest of the numbers considered (a and b)

4 <u>Machine accuracy ϵ</u>

It is a similar concept : the question is, what is the <u>largest</u> number which can be added to 1 such that, in FP arithmetic

$$1 + \epsilon = 1$$

<u>Example</u> Again, consider 6-digit mantissa. Then

$$1 = 0.100000 \times 10^1$$

$$1 + 10^{-7} = 0.1000001 \times 10^1 \rightarrow \text{rounded down to } 0.100000 \times 10^1$$

↳ the machine accuracy here is $\simeq 4 \cdot 10^{-7}$

For FP arithmetic in base 2, with a mantissa of size ℓ,

$$\boxed{\epsilon \simeq 2^{-\ell}} \rightarrow \begin{array}{ll} \text{in real, SP}: & \epsilon \simeq 10^{-7} \\ \text{in real DP}: & \epsilon \simeq 10^{-16}. \end{array}$$

⑤ Overflow and underflow problems

There exists a smallest and largest number (in absolute value), that can be represented in FP notation:

<u>Example</u> Suppose the exponent k ranges from -4 to 4 and the mantissa has 8 significant digits

→ the smallest possible number is

$$x_{min} = 0.1 \times 10^{-4} \quad (\text{in base } 10)$$

→ the largest number is

$$x_{max} = 0.99999999 \times 10^{4} \simeq 10^{4} \quad (\text{in base } 10)$$

So in general, in base 2:

$$x_{min} = 2^{L-1}$$
$$x_{max} = 2^{u}$$

→ for real SP: $\begin{cases} x_{min} \cong 10^{-38} \\ x_{max} \simeq 10^{38} \end{cases}$

DP: $\begin{cases} x_{min} \simeq 10^{-308} \\ x_{max} \simeq 10^{308} \end{cases}$

If the outcome of a FD operation yields $|x| < x_{min}$ then an <u>underflow</u> error occurs. If $|x| > x_{max}$ then an <u>overflow</u> error occurs.