Sect.1 Introduction to numerical analysis

Numerical analysis deals with the process of getting numerical solutions to difficult mathematical problems.

Most of the mathematical problems that arise in science and engineering are very hard and sometime impossible to solve exactly. Thus an approximation to a difficult problem is very important to make it more easy to solve. Therefore, numerical analysis is the study of numerical methods that attempt to find approximate solutions of problems (within specified error bounds) rather than the exact ones. As a result many scientific software are developed (for instance Maple, Mathematica, Matlab, ...) to handle difficult problems in an ease way.

Numerical analysis include three parts:

1. The creation of a problem-solving approach (develop a method to a problem).
2. Analysis of the method (error analysis and efficiency analysis).
3. Construction of an efficient algorithm to implement the method as a computer code.

Sect.2 Computer representation of numbers

Digital computer are the principal means of calculation in numerical analysis and consequently it is very important to understand how they operate.

In decimal system, a number is presented by a list of digits from 0 to 9.

$$276.49 = 2\times 10^2 + 7\times 10^1 + 6\times 10^0 + 4\times 10^{-1} + 9\times 10^{-2}.$$

Called ***base10*** because it is based on 10 digits (0 to 9).

Modern computers read pulses sent by electrical components. The state of an electrical impulse is either on or off. It is therefore convenient to represent numbers in computers in **the binary system** (the digits are 0 or 1) also known as bits.

When using numbers in different bases, we will use $(x)_\beta$ to indicates that the number x is to be evaluated in the base β number system.

$$(N)_{10} = (a_n a_{n-1}...a_1 a_0)_2\,, \qquad N = a_n\times 2^n + a_{n-1}\times 2^{n-1} + ... + a_1\times 2^1 + a_0\times 2^0.$$

$$\big(0.1011\big)_2 = \frac{1}{2} + \frac{0}{4} + \frac{1}{8} + \frac{1}{16} = \left(\frac{11}{16}\right)_{10}$$

In base 16 (hexadecimal), digits are 0, 1, ..., 9, A, B, C, D, E, F.

$$(37C.F)_{16} = 3\times 16^2 + 7\times 16^1 + 12\times 16^0 + 15\times 16^{-1} = 892.9375.$$

13 | 2		
1	6 | 2	
	0	3 | 2
		1 1

$\longleftarrow$

$13 = (1101)_2$

478 | 7	
2	68 | 7
	5 9 | 7
	2 1

$\longleftarrow$

$478 = (1252)_7$

319 | 16		
15	19	16
	3	1 | 16
		1 0

$\longleftarrow$

$319 = (13F)_{16}$

It is worthy to notice that each hex number can be represented by 4 bits. For instance $(1)_{16} = (0001)_2$, $(8)_{16} = (1000)_2$, $(E)_{16} = (1110)_2 = (14)_{10}$.

Sect.3 Storing a number in computers

The number $\frac{1}{3} = 0.333...$ cannot be stored in a computer because it would take an infinite amount of memory. Thus, computers and calculators store just a finite number of digits or bits after the decimal point. Computer scientists and mathematicians have worked since the dawn of electronic computers to find efficient ways to accurately represent, or approximate, real numbers. The common idea is to use **floating point arithmetic**. The latter consists of three parts:

(1) a sign (±)

(2) a mantissa or significand (where most of the digits or bits of interest are),

(3) an exponent

Floating point numbers have a base. Let β the number (base) being used in the computer. Then a nonzero number x in the computer is stored in the form

$$x = (-1)^s \times (.a_1 a_2 \dots a_t)_\beta \times \beta^e \tag{1}$$

where $\boldsymbol{s \in \{0,1\}}$, $(-1)^s = \pm 1$ (s=0 for positive numbers and s=1 for negative numbers).

$$(.a_1 a_2 \dots a_t)_\beta = \frac{a_1}{\beta^1} + \frac{a_2}{\beta^2} + \dots + \frac{a_t}{\beta^t} \tag{2}$$

with $0 \leqslant a_i \leqslant \beta - 1$ is called the **mantissa** of the floating-point number x (also called the **radix**), and the point preceding a_1 is called **the radix point**.
$\boldsymbol{e}$ is an integer called the **exponent**.

Example 1.
Let $x = 0.0625$*:*

$$\begin{aligned} 0.0625 \times 2 = 0.1250 &\longrightarrow d_1 = 0 \\ 0.1250 \times 2 = 0.2500 &\longrightarrow d_2 = 0 \\ 0.2500 \times 2 = 0.5000 &\longrightarrow d_3 = 0 \\ 0.5000 \times 2 = 1.0000 &\longrightarrow d_4 = 1 \end{aligned}$$

$$(0.0625)_{10} = (0.0001)_2.$$

$$\textit{For} \quad \frac{1}{3} = (0.010101\ldots)_2 = (0.\overline{01})_2, \qquad \frac{1}{5} = (0.001001001\ldots)_2 = (0.\overline{001})_2$$

$$\begin{aligned} \frac{1}{3} \times 2 = 0 + \frac{2}{3} &\longrightarrow d_1 = 0 \\ \frac{2}{3} \times 2 = 1 + \frac{1}{3} &\longrightarrow d_2 = 1 \\ \frac{1}{3} \times 2 = 0 + \frac{2}{3} &\longrightarrow d_3 = 0 \\ \frac{2}{3} \times 2 = 1 + \frac{1}{3} &\longrightarrow d_4 = 1 \\ \vdots & \end{aligned}$$

$$\textit{Remark} \quad 0.137 = (0.\overline{0646362510535212234})_7 = (0.001\overline{d_4 \ldots d_{103}})_2$$

The latter example, shows that either the representation of the mantissa's period only is not possible for very large periods. Therefore, the set A of numbers which are representable in a given machine is only finite.

Before 1985, different manufacturers of computers and computer hardware used different formats for storing floating point numbers, making it difficult to use programs written for one computer system on another. In 1985, the Institute for Electronic and Electrical Engineering standard IEEE 754 for floating point arithmetic was adopted and has become the main standard used for floating point arithmetic.

There are three commonly used levels of precision for floating point numbers: single precision, double precision, and extended precision. The three standards use binary representation of numbers, but use different numbers of bits for the different components as well as different numbers of bits for the entire number. The form of a normalized IEEE floating point number is

$$x = \pm 1.a_1 a_2 \ldots a_t \times 2^e, \qquad e = b_1 b_2 \ldots b_d \tag{3}$$

single precision (32 bits)

±	$b_1 b_2 b_3 \ldots b_7 b_8$	$a_1 a_2 a_3 \ldots a_{22} a_{23}$
	exponent	mantissa

double precision (64 bits)

±	$b_1 b_2 b_3 \quad \ldots \quad b_{10} b_{11}$	$a_1 a_2 a_3 \quad \ldots \quad a_{51} a_{52}$
	exponent	mantissa

extended precision (80 bits)

±	$b_1 b_2 b_3 \quad \ldots \quad b_{14} b_{15}$	$a_1 a_2 a_3 a_4 \quad \ldots \quad a_{62} a_{63}$	
	exponent	mantissa	

The single precision and double precision number 1 are, respectively,

+1.	00000000000000000000000	2^0

+1.	00000000000000000000000000000000000000000000000000000000	2^0

In case of double precision, the next floating point number greater than 1 is

+1.	00000000000000000000000000000000000000000000000000000001	2^0

The distance between 1 and the smallest floating point number greater than 1 is called the **number machine epsilon** and denoted ϵ_{mach}. Thus, the machine epsilon for single precision and double precision numbers are, $\epsilon_{\text{mach}} = 2^{-23}$ and $\epsilon_{\text{mach}} = 2^{-52}$, respectively.

Exercise 1:

1. Convert the following base 10 numbers to binary. Use overbar notation for non-terminating binary numbers: 11.5, 5/7, 20.6
2. Convert the following binary numbers to decimal: 1010101, $0.\overline{1011}$, $0.10\overline{1011}$, $0.1\overline{000111}$, $10.\overline{110}$, $1011.\overline{10101}$
3. Convert the following numbers to decimal: $(3.12\overline{3401})_5$, $(0.\overline{3612})_7$, $(75.34\overline{752136})_8$, $(1A5.2E\overline{3FD612})_{16}$.

Sect.4 Errors and their computations

The restriction that the number of mantissa require no more than s bits means that our machine numbers have a limited precision. Thus, numbers expressed with more than s decimal digits will be approximated when given as input to the computer.

Therefore, since the precision or length n of floating-point numbers on any machine (usually determined by **the word length**) is finite, the question to think about now is: how do we fit an infinite binary representation of number in a finite number of bits?

There are two commonly used ways of translating a given real number

$$x = (.a_1 a_2 ... a_k a_{k+1} a_{k+2} ...)_\beta \beta^e$$

into an β-digit floating-point number, chopping and rounding.

The error that results from replacing a number with its floating-point form is called **round-off error** regardless of whether the chopping or rounding method is used.

The chopping, is simply to throw away the bits that fall off the end, that is to say, discarding excess bits $a_{24} a_{25}$.... The resulting number is

$$\tilde{x} = (.a_1 a_2 ... a_{22} a_{23})_\beta \beta^e$$

The alternative method is **rounding**, this corresponds to the familiar rounding up or rounding down in decimal arithmetic, based on the first discarded decimal digit: if it is larger than or equal to 5, one rounds up; if it is less than 5, one rounds down.

In base β the rounded machine machine approximation of x is given by

$$\tilde{x} = \begin{cases} (-1)^s \left(.a_1 a_2 ... a_k\right)_\beta \beta^e, & \text{if} \quad 0 \leqslant a_{k+1} < \dfrac{\beta}{2} \\ (-1)^s \left(.a_1 a_2 ... (a_k + 1)\right)_\beta \beta^e, & \text{if} \quad \dfrac{\beta}{2} \leqslant a_{k+1} < \beta \end{cases}$$

In binary arithmetic, the procedure is somewhat simpler, since there are only two possibilities: either the first discarded binary digit is 1, in which case we add 1 to bit k (round up), or it is 0, in which case we do nothing (round down) to bit k.

To represent a real number as a single precision (32-bits) floating point number, convert the number to binary, and carry out two steps:

(1) **Justify**. Shift radix point to the right of the leftmost 1, and compensate with the exponent.

(2) **Round**. Apply a rounding-off error rule to reduce the mantissa to 23 digits.

Example 2.
Determine the five-digit ***(a)*** *chopping and* ***(b)*** *rounding values of the numbers e, x and* $e - x$.

$$e = 2.7182818284... \qquad x = 19/7 = 2.7142857142...$$

The numbers has an infinite decimal, we have

(a) Chopping

$$e = 2.7182818284... \times 10^1 = 2.71828 \times 10^1$$

$$x = 19/7 = 2.7142857142... \times 10^1 = 2.71428 \times 10^1$$

$$e - x = 2.71828 \times 10^1 - 2.71428 \times 10^1 = 0.004 \times 10^1 = 4 \times 10^{-2}$$

(b) Rounding

$$e = 2.7182818284... \times 10^1 = 2.71828 \times 10^1$$

$$x = 19/7 = 2.7142857142... \times 10^1 = 2.71429 \times 10^1$$

$$e - x = 2.71828 \times 10^1 - 2.71429 \times 10^1 = 0.00399 \times 10^1 = 3.99 \times 10^{-2}$$

Example 3.
Find the binary representation of $y = 11.3$ *in a machine of double precision.*
We have

$$(11.3)_2 = 1011.0\overline{1001}$$

then

(a) Justify *The radix point is moved three places to the left, to obtain the justified number*

$$+1.\boxed{0110100110011001100110011001100110011001100110011001}1001... \times 2^3$$

(b) Rounding *Bit* 53 *of the justified number is* 1, *so round up, which means adding* 1 *to bit* 52. *Notice that the addition causes carrying to bit* 51.

$$+1.\boxed{0110100110011001100110011001100110011001100110011010} \times 2^3$$

Exercise 2:

Find the binary representation in 23-digits of the following decimal numbers

$$0.1, \quad 1/3, \quad 1/5, \quad 5/3, \quad 4/3, \quad 22/7, \quad 17/7, \quad 13/29, \quad 8/33, \quad 5/124$$

Since rounding-off error is inherent in floating-point computation, it is important to have a way to measure this error. We shall next examine these errors a tittle bit more.

Definition 1.
We define the ***error*** *associated with an approximate value $\tilde{x}$ as the result of subtracting that approximation from the true value X*

$$\Delta := X - \tilde{x} = \textbf{\textit{error}} = \textit{true value} - \textit{approximation}$$

The ***absolute error*** *is the absolute value of the error defined above.*

$$\Delta_a := |X - \tilde{x}|$$

The ***relative error*** *is a measure of the error in relation to the size of the true value as given by*

$$\delta_r := \frac{\Delta_a}{|X|} = \textbf{\textit{Relative error}} = \frac{\textbf{\textit{error}}}{\textbf{\textit{true value}}}$$

The ***percentage error*** *is defined as* 100 *times the relative error.*

Example 4.
Find absolute error and relative error in the following three cases:

(a) $a = 3.1415926536$, $\tilde{a} = 22/7$. *We have*

$$\Delta_a = |a - \tilde{a}| = 0.001264489. \qquad \delta_a = \frac{|a - \tilde{a}|}{a} = 0.000402499$$

(b) $b = 1000000$, $\tilde{b} = 1000002$. *We have*

$$\Delta_b = |b - \tilde{b}| = 2 \qquad \delta_b = \frac{|b - \tilde{b}|}{b} = 0.000002$$

(c) $c = 0.000016$, $\tilde{c} = 0.000019$. *We have*

$$\Delta_c = |c - \tilde{c}| = 0.000003. \qquad \delta_c = \frac{|c - \tilde{c}|}{c} = 0.1875 = 18.75\%$$

Exercise 3:

Convert the numbers $(u)_2 = 1.0110101$ and $(v)_2 = 11.0010010001$ to decimal form, then find the error in the following approximations $\sqrt{2} - u$ and $\pi - v$.

From **Example** 4, we remark in case **(c)** that c is of magnitude 10^{-6} and $\tilde{c}$ is the smallest of all three cases, but the relative error δ_c is the largest. In terms of percentage, it amounts to 18.75% and thus $\tilde{c}$ is a bad approximation to c.

Observe that as $|X|$ moves away from 1 the relative error δ_x is a better indicator of the accuracy of the approximation than Δ_x. Relative error is preferred for floating-point representations since it deals directly with the mantissa.

Definition 2.
The number $\tilde{x}$ is said to approximate X to d significant digits if d is the largest positive integer for which

$$\frac{|X - \tilde{x}|}{|X|} \leqslant \frac{10^{-d}}{2}.$$

In general, we can prove the following

Theorem 1.
if a machine operates with base β and carries n places in the mantissa of its floating-point numbers, then $\epsilon_{mach} = \frac{1}{2}\beta^{1-n}$ in the case of correct rounding and $\epsilon_{mach} = \beta^{1-n}$ in the case of chopping.

Proof. **(1)** In fact, **if k decimal digits** and chopping are used for the machine representation of we have

$$x = .d_1 d_2 ... d_k d_{k+1} ... \times 10^e$$

then

$$\begin{aligned}\epsilon_{\text{mach}} = \frac{|x - \tilde{x}|}{|x|} &= \frac{|.d_1 d_2 ... d_k d_{k+1} ... \times 10^e - .d_0 d_1 ... d_k \times 10^e|}{|.d_1 d_2 ... d_k d_{k+1} ... \times 10^e|} \\ &= \frac{|.d_{k+1} d_{k+2} ... \times 10^{e-k}}{|.d_1 d_2 ... d_k d_{k+1} ... \times 10^e|} = \frac{|.d_{k+1} d_{k+2} ...|}{|.d_1 d_2 ... d_k d_{k+1} ...|} \times 10^{-k}\end{aligned}$$

Since $d_1 \neq 0$, the minimal value of the denominator is 0.1. The numerator is bounded above by 1. As a consequence,

$$\frac{|x - \tilde{x}|}{|x|} \leqslant \frac{1}{0.1} \times 10^{-k} = 10^{1-k}.$$

In a similar manner, a bound for the relative error when using k-digit rounding arithmetic is $0.5 \times 10^{1-k}$.

(2) In binary representation ($b = 2$), we can show the bound of machine epsilon more easily. Indeed, since $0 \leqslant d_i \leqslant 1 = 2 - 1 = \beta - 1$ and

$$.d_1 d_2 ... = \frac{1}{2} + ... + \frac{1}{2^i} + ... \geqslant \frac{1}{2} \quad \Longrightarrow \quad \frac{1}{.d_1 d_2 ...} \leqslant 2.$$

then

$$\epsilon_{\text{mach}} = \frac{|x - \tilde{x}|}{|x|} = \frac{|.d_{k+1} d_{k+2} ...|}{|.d_1 d_2 ... d_k d_{k+1} ...|} \times 2^{-k} \leqslant |.(\beta - 1)(\beta - 1) ...| \times 2 \times 2^{-k} \leqslant 2^{1-k}.$$

□

Example 5.
Determine the number of significant digits for the approximations in Example 4.

(a) *If* $a = 3.1415926536$ *and* $\tilde{a} = 22/7$, *then* $\dfrac{|a - \tilde{a}|}{a} = 0.000402499 < \dfrac{10^{-2}}{2}$. *Therefore,* $\tilde{a}$ *approximates* a *to two significant digits.*

(b) *If* $b = 1000000$ *and* $\tilde{b} = 1000002$, *then* $\dfrac{|b - \tilde{b}|}{b} = 0.000002 < \dfrac{10^{-5}}{2}$. *Therefore,* $\tilde{b}$ *approximates* b *to five significant digits.*

(c) *If* $c = 0.000016$ *and* $\tilde{c} = 0.000019$, *then* $\dfrac{|c - \tilde{c}|}{c} = 0.1875 < \dfrac{10^{-0}}{2}$. *Therefore,* $\tilde{c}$ *approximates* c *to no significant digits.*

Since the true errors cannot, in most cases, be calculated (the true solution is unknown), other means are used for estimating the accuracy of a numerical solution. In some methods the numerical error can be bounded, while in others an estimate of the order of magnitude of the error is determined. In practical applications, numerical solutions can also be compared to experimental results, but it is important to remember that experimental data have errors and uncertainties as well.

Definition 3.
We call an ***upper bound*** *of the absolute error (resp. relative error), any positive number* Δ_a *(resp.* δ_a*) satisfying*

(1) $\Delta = |A - a| \leqslant \Delta_a \quad \Longleftrightarrow \quad a - \Delta_a \leqslant A \leqslant a + \Delta_a$

(2) $\delta = \dfrac{|\Delta|}{|A|} \leqslant \delta_a$

Example 6.
Consider 5-digit chopping of $e = 2.71828182....$ *We have*

$$\delta_e = \frac{|e-\tilde{e}|}{e} = \frac{|0.271828182... \times 10^1 - 0.27183 \times 10^1|}{0.271828182... \times 10^1}$$
$$= \frac{0.000001818... \times 10^1}{0.271828182... \times 10^1} = \frac{0.1818...}{0.271828182...} \times 10^{-5} \leqslant 0.5 \times 10^{-5} \leqslant \mathbf{10^{-4}}.$$

In practice, the quantities E_a and δ_r have to be chosen as small as possible. In the other hand, if we know an absolute error (resp. relative error), we write

$$a = \tilde{a} \pm \Delta_a, \qquad a = \tilde{a}\big(1 \pm \delta_a\big), \quad \text{respectively.}$$

Proposition 1.
If the number of significant number of x *are at least two, then we can take as relative error bound of* x*, the number*

$$\eta_x = \frac{1}{2d_1} \times 10^{1-k}$$

Sect.5 Propagation of Error

Once an error is committed, it affects subsequent results as this error propagates through subsequent calculations. We first study how the results are affected by using approximate numbers instead of actual numbers and then will take up function evaluation. We will now see how error might be propagated in successive computations with the four basic arithmetic operations ($\pm$, $\times$, $\div$). If the symbol $\odot$ stand for any one of the four basic arithmetic operations, in any computer, it is desirable to know that the four arithmetic operations satisfy equations like

$$\tilde{x} \odot \tilde{y} = (x \odot y)(1+\delta), \qquad |\delta| \leqslant \epsilon_{\text{mach}}.$$

Addition (and subtraction) Let A_1 and A_2 exact positives quantities, a_1 and a_2 be their respective approximate values, Δ_1 and Δ_2 the corresponding errors in these approximations. We have

$$A_1 = a_1 + \Delta_1,$$
$$A_2 = a_2 + \Delta_2, \qquad A_1 + A_2 = (a_1 + a_2) + (\Delta_1 + \Delta_2)$$

Therefore, the sum $a_1 + a_2$ of the approximate values approximates the sun $A_1 + A_2$ of the two exact values with an error $\Delta_1 + \Delta_2$ the sum of the individual errors. Then

$$|(A_1 \pm A_2) - (a_1 \pm a_2)| = |\Delta_1 \pm \Delta_2| \leqslant |\Delta_1| + |\Delta_2|$$

Thus, the absolute error in the sum of two numbers is less or equal to the sum of the sum of the absolute errors in the individual numbers.

The relative error satisfy

$$\delta_{a_1 \pm a_2} = \frac{\Delta_{a_1 \pm a_2}}{|A_1 \pm A_2|} \leqslant \frac{\Delta_{a_1} + \Delta_{a_2}}{|A_1 \pm A_2|} = \frac{|A_1| \frac{\Delta_{a_1}}{|A_1|} + |A_2| \frac{\Delta_{a_2}}{|A_2|}}{|A_1 \pm A_2|} = \frac{|A_1| \delta_{a_1} + |A_2| \delta_{a_2}}{|A_1 \pm A_2|}$$

Multiplication We have

$$\delta_{a_1 a_2} = \left| \frac{A_1 A_2 - a_1 a_2}{A_1 A_2} \right| = \left| \frac{(A_1 A_2) - (A_1 - a_1)(A_2 - a_2)}{A_1 A_2} \right| = \left| \frac{A_1 a_2 + A_2 a_1 - a_1 a_2}{A_1 A_2} \right|$$

But $a_1 a_2$ can be assumed negligible with respect to the other terms in the numerator and we omit it. As a result,

$$\delta_{a_1 a_2} = \left| \frac{A_1 A_2 - a_1 a_2}{A_1 A_2} \right| \approx \left| \frac{A_2 a_1 + A_1 a_2}{A_1 A_2} \right| = \left| \frac{a_1}{A_1} + \frac{a_2}{A_2} \right| \leqslant \delta_{a_1} + \delta_{a_2}$$

There is another result concerns the bound of the Theorem 1. Indeed, if we take a product of n factors ($n \leqslant 10$)

$$p = A_1 \times A_2 \times \ldots \times A_n,$$

then according to Theorem 1, we have

$$\delta_{a_i} = \frac{1}{2 d_i} 10^{1-m}, \qquad i = \overline{1, n}$$

where d_i are the first significant digits of A_i. Therefore, the relative of the product is

$$\delta_p = \frac{1}{2} \left(\frac{1}{d_1} + \frac{1}{d_2} + \ldots + \frac{1}{d_n} \right) 10^{1-m}$$

and since $\frac{1}{d_1} + \frac{1}{d_2} + \ldots + \frac{1}{d_n} \leqslant \frac{1}{10}$, its follows then that

$$\delta_p \leqslant \frac{1}{2} 10^{2-m}$$

Division We have

$$\begin{aligned} \Delta_{\frac{a_1}{a_2}} &= \left| \frac{A_1}{A_2} - \frac{a_1}{a_2} \right| = \left| \frac{a_1 + \Delta_{a_1}}{a_2 + \Delta_{a_2}} - \frac{a_1}{a_2} \right| = \left| \frac{a_2 \Delta_{a_1} - a_1 \Delta_{a_2}}{a_2 \left(a_2 + \Delta_{a_2} \right)} \right| \\ &= \left| \frac{a_2 \Delta_{a_1} - a_1 \Delta_{a_2}}{a_2^2 \left(1 + \frac{\Delta_{a_2}}{a_2} \right)} \right| \leqslant \left| \frac{a_2 \Delta_{a_1} - a_1 \Delta_{a_2}}{a_2^2} \right| = \frac{a_1}{a_2} \left| \frac{\Delta_{a_1}}{a_1} - \frac{\Delta_{a_2}}{a_2} \right|. \end{aligned}$$

☛ In relation with the bound of the Theorem 1, we have

$$\delta_{\frac{x}{y}} = \frac{1}{2} \times \left(\frac{1}{d_1} + \frac{1}{b_1} \right) \times 10^{1-m}$$

where d_1 and b_1 are the first significant digits of x and y, respectively.

Example 7.
(1) *Evaluate the sum* $\sqrt{3} + \sqrt{7} + \sqrt{17}$ *to* 4 *significant digits and find the absolute and relative errors.*

We have

$$\sqrt{3} = 1.732, \quad \sqrt{7} = 2.646, \quad \sqrt{17} = 4.123$$

then $S = 8.501$ *and*

$$\Delta_s = 0.0005 + 0.0005 + 0.0005 = 0.0015 < 0.5 \times 10^{-2} = 0.5 \times 10^{\mathbf{1-3}}$$

The total absolute error shows that the sum is correct to 3 *significant digits only. Hence we take* $S = 8.50$ *and then*

$$\delta_s = \frac{0.0015}{8.50} = 0.0002.$$

(2) *Calculate the value of* $\sqrt{102} - \sqrt{101}$ *correct to four significant digits.*

We have $\sqrt{102} = 10.099504938$, $\sqrt{102} = 10.049875621$, *thus*

$$\sqrt{102} - \sqrt{101} = 0.049629317 = 4.963 \times 10^{-2}$$

correct to 4 *significant digits.*

Alternatively, we have

$$\sqrt{102} - \sqrt{101} = \frac{102 - 101}{\sqrt{102} + \sqrt{101}} = \frac{1}{10.10 + 10.05} = 0.04963 = 4.963 \times 10^{-2}$$

which is the same result as obtained before.

We present next a theorem that illustrates how the relative round-off error in long calculations can be analyzed. The theorem states roughly that in adding $n+1$ positive machine numbers, the relative round-off error should not exceed $n\epsilon$.

Theorem 2.
Let $x_0, x_1, \ldots, x_n$ *be positive machine numbers in a computer whose unit round-off error is* ϵ. *Then the relative round-off error in computing* $x_0 + \cdots + x_n$ *in the usual way is at most* $(1+\epsilon)^n - 1$. *This quantity is approximately* $n\epsilon$.

Proof. Let $S_k = x_0 + \cdots + x_k$ and let $\tilde{S}_k$ be what the computer calculates instead of S_k. The recursive formulæ for these quantities are

$$\begin{cases} S_0 = x_0 \\ S_{k+1} = S_k + x_{k+1} \end{cases} \qquad \begin{cases} \tilde{S}_0 = x_0 \\ \tilde{S}_{k+1} = fl\left(\tilde{S}_k + x_{k+1}\right) \end{cases}$$

For the analysis define

$$\rho_k = \frac{\tilde{S}_k - S_k}{S_k}, \qquad \delta_k = \frac{\tilde{S}_{k+1} - \left(\tilde{S}_k + x_{k+1}\right)}{S_k + x_{k+1}}$$

Thus, ρ_k is the relative error in approximating the k$^{\text{th}}$ partial sum S_k by the computed partial sum $\tilde{S}_k$, and $|\delta_k|$ is the relative error in approximating $\tilde{S}_k + x_{k+1}$ by the quantity $fl\left(\tilde{S}_k + x_{k+1}\right)$. Using the equations that define ρ_k and δ_k, we have

$$\begin{aligned} \rho_{k+1} &= \frac{\tilde{S}_{k+1} - S_{k+1}}{S_{k+1}} = \frac{\left(\tilde{S}_k + x_{k+1}\right)(1+\delta_k) - (S_k + x_{k+1})}{S_{k+1}} \\ &= \frac{\left[S_k\left(1+\rho_k\right) + x_{k+1}\right](1+\delta_k) - (S_k + x_{k+1})}{S_{k+1}} \\ &= \frac{\rho_k S_k(1+\delta_k) + \delta_k\left(S_k + x_{k+1}\right)}{S_{k+1}} = \frac{\rho_k S_k(1+\delta_k) + \delta_k S_{k+1}}{S_{k+1}} \\ &= \delta_k + \rho_k(1+\delta_k)\frac{S_k}{S_{k+1}}. \end{aligned}$$

Since $S_k < S_{k+1}$ and $|\delta_k| \leqslant \epsilon$, we deduce that

$$|\rho_{k+1}| \leqslant \epsilon + |\rho_k|(1+\epsilon) = \epsilon + \theta|\rho_k|.$$

Thus, we have the following successive inequalities

$$\begin{aligned} |\rho_0| &\leqslant 0 \\ |\rho_1| &\leqslant \epsilon \\ |\rho_2| &\leqslant \theta\epsilon \\ |\rho_3| &\leqslant \epsilon + \theta(\epsilon + \theta\epsilon) \leqslant \epsilon + \theta\epsilon + \theta^2\epsilon \\ &\vdots \end{aligned}$$

The general recurrence will be

$$|\rho_n| \leqslant \epsilon + \theta\epsilon + \theta^2\epsilon + \cdots + \theta^{n-1}\epsilon = \epsilon\left(\frac{\theta^n - 1}{\theta - 1}\right) = \epsilon\left(\frac{(1+\epsilon)^n - 1}{\epsilon}\right) = (1+\epsilon)^n - 1.$$

Now, by binomial theorem we have

$$(1+\epsilon)^n - 1 = 1 + \binom{n}{1}\epsilon + \binom{n}{2}\epsilon^2 + \binom{n}{3}\epsilon^3 + \cdots - 1 \approx n\epsilon.$$

□

Exercise 4:

Suppose $a, b \in \mathbb{R}$. Their floating-point approximations are $\tilde{a}$ and $\tilde{b}$, respectively. Let $x = a - b$ and its floating-point approximation $\tilde{a} - \tilde{b}$, show that the relative error satisfy

$$\delta_x = \left|\frac{x - \tilde{x}}{x}\right| \leqslant \eta \frac{|a| + |b|}{|a - b|}$$

what is η?

Sect.6 General error formula

We now derive a general formula for the error committed in using a certain formula or a functional relation. Let

$$u = f(x_1, x_2, ..., x_n)$$

and let the errors in x_1, ..., x_n be Δ_{x_1}, ... Δ_{x_n}, respectively. Then the error Δ_u of u is given by

$$\Delta_u = \left|f\left(x + \Delta_x, y + \Delta_y, z + \Delta_z\right) - f(x, y, x)\right|$$

If grad $f(x, y, z) \neq 0$, we have

$$\Delta_u = \sum_{i=1}^{n} \left|\frac{\partial u}{\partial x_i}\right| \Delta_{x_i}, \qquad \delta_u = \sum_{i=1}^{n} \left|\frac{\partial}{\partial x_i} \ln(u)\right| \Delta_{x_i}$$

Example 8.
Determine the absolute error and the relative error of the cylinder's volume $v = \pi h r^2$ *with* $\pi = 3.145$, $r = .25$, $h = 16.16$. *We have* $v = 3.17645$.

Example 9.
Given that $u = 7x^2y^3/z^4$, *find the relative error at* $x = y = z = 1$ *when the errors in each of* x, y, z *is* 0.001.

Sect.7 Loss of significance

Although round-off errors are inevitable and difficult to control, there are other types of errors in computation that are under our control. The subject of numerical analysis is largely preoccupied with understanding and controlling errors of various kinds. Here we shall take up one type of error that is often the result of careless programming.

An interesting question is the following: Exactly how many significant binary bits are lost in the subtraction $x-y$ when x is close to y? The precise answer depends on the particular values of x and y. However, we can obtain bounds in terms of the quantity $|1-y/x|$, which is a convenient measure for the closeness of x and y. The following theorem contains useful upper and lower bounds.

Theorem 3.
If x and y are positive normalized floating-point binary machine numbers such that $x>y$ and

$$2^{-q} \leqslant 1-\frac{y}{x} \leqslant 2^{-p}$$

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

Proof. We shall prove the lower bound. The normalized binary floating-point forms for x and y are

$$x = r\times 2^m, \qquad y = s\times 2^n$$

Since x is larger than y, the computer may have to shift y so that y has the same exponent as x before performing the subtraction of y from x. Hence, we must write y as

$$y = \left(s\times 2^{n-m}\right)\times 2^m$$

and then we shall have

$$x-y = \left(r-s\times 2^{n-m}\right)\times 2^m$$

The mantissa of this number satisfies

$$r-s\times 2^{n-m} = r\left(1-\frac{s\times 2^n}{r\times 2^m}\right) = r\left(1-\frac{y}{x}\right).$$

To normalize the computer representation of $x-y$, a shift of at least p bits to the left is required. Then at least p spurious zeros are attached to the right end of the mantissa, which means that at least p bits of precision have been lost. □

In some calculations, a loss of significance can be avoided or ameliorated by use of double precision. In this mode of computation, each real number is allotted two words of storage. This at least doubles the number of bits in the mantissa.

Sect.8 Stable and Unstable Computations (Conditioning)

In most situations the introduction of rounding errors into the calculations does not significantly affect the final results. However, in certain cases it can lead to a serious loss of accuracy so that computed results are very different from those obtained using exact arithmetic. This theme occurs repeatedly in numerical analysis and refereed to as: the distinction between numerical processes that are stable and those that are not.

Speaking informally, we say that a numerical process (an algorithm) is stable if small changes in the initial data produce correspondingly small changes in the final results, otherwise it is unstable, i.e. the algorithm is unstable if small errors made at one stage of the process are magnified in subsequent stages and seriously degrade the accuracy of the overall calculation. Some algorithms are stable only for certain choices of initial data, and are called conditionally stable.

Example 10.
We consider the computation of values of the quadratic

$$p(x) = x^2 + x - 1150$$

This is an ill-conditioned problem if x is near a root of the quadratic. For example, for $x = 100/3$ we have $p(100/3) = 50/9$. However, for small changes, if we take for instance $x = 33$, which differs from $100/3$ only by 1, we get $p(33) = 28$ which is approximately five times the value of $p(100/3)$.

Example 11.
If $\{y_n\}_n$ satisfies the recurrence relation

$$y_{n+1} = 100.01\, y_n - y_{n-1}$$

with $y_0 = 1$ and $y_1 = 0.01$, then $y_5 = 10^{-5}$, whereas if $y_0 = 1 + 10^{-6}$ and $y_1 = 0.01$, then $y_5 \approx -1.0001$. The large difference in the value of y_5 shows that the process is very ill-conditioned. In fact we say that the recurrence relation is numerically unstable.

Example 12.
Consider the simultaneous linear equations

$$\begin{cases} x + y &= 2 \\ x + 1.01\, y &= 2.01 \end{cases}$$

which have solution $x = y = 1$. *If the number* 2.01 *is changed to* 2.02, *the corresponding solution is* $x = 0$, $y = 2$. *We see that a* 0.5 *change in the data produces a* 100% *change in the solution.*

We now consider a different type of instability which is a consequence of the method of solution rather than the problem itself. To further discuss the idea and its connection to algorithm stability, suppose an error with magnitude $\Delta_0 > 0$ is introduced at some stage in the calculations and that the magnitude of the error after n subsequent operations is denoted by Δ_n. The two cases that arise most often in practice are defied as follows

☞ **If $\Delta_n \approx Cn\Delta_0$**, where C is a constant independent of n, then the growth of error is said to be **linear**.

☞ **If $\Delta_n \approx C^n\Delta_0$**, where $C > 1$, then the growth of error is called **exponential**.

Linear growth of error is usually unavoidable, and when C and Δ_0 are small the results are generally acceptable. Exponential growth of error should be avoided, because the term C^n becomes large for even relatively small values of n. In other words, an algorithm that exhibits linear growth of error is stable, whereas an algorithm exhibiting exponential error growth is unstable.