1

Errors in Computation

1.1 Introduction

For solving a mathematical problem by numerical method, an input is provided in the form of some numerical data or it is generated/created as called for by the problem. The input is processed through arithmetic operations together with logical operations, which are performed in a systematic manner and the output is produced in the form of some numbers. Thus the whole exercise in Numerical Analysis is all about manipulation of numbers. Whether we are working by hand or on a computing machine, there is always a constraint in regard to physical size of the numbers, i.e., the number of digits a number can contain. Inside a computer the size of the number is dependent on its word-length (number of bits) which also puts a limit on the range of numbers that can be represented in a particular computer. Further, it may be noted that all numbers are not represented exactly inside the computer and that the input given in the decimal form is converted to binary in the computer. It should also be remembered that fractions cannot be stored in their natural form; they are converted to decimals, for example 2/5 is input as 0.4 and 1/3 as 0.333... up to a finite number of digits acceptable by a computer.

1.2 Floating Point Representation of Number

When a number x is expressed as,

$$x = p \times 10^q$$

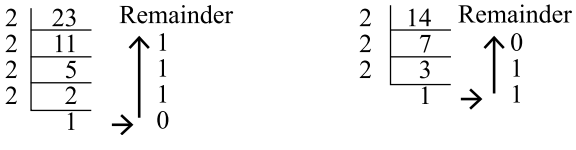
where $0.1 \leq |p| < 1.0$ and q is an integer (positive (+ve) or negative (−ve)), it is called 'floating point' representation of number x. A floating point form consists of two parts; the fractional part p (alongwith the sign) is known as mantissa and the other part q as exponent, a power raised to a radix (in the case of decimal system, 10). At some places it is referred to

as 'normalised floating point' and when $1 \leq p < 10$, the form is called 'scientific notation'. A few examples of floating point representation, $fl(x)$ of number x are given as follows:

x	$fl(x)$	*Mantissa*(p)	*Exponent*(q)
2.0456	0.20456×10^{1}	0.20456	1
−32.7652	-0.327652×10^{2}	−0.327652	2
0.00234	0.234×10^{-2}	0.234	−2
0.000000034	0.34×10^{-7}	0.34	−7
34000000	0.34×10^{8}	0.34	8

1.3 Binary Numbers

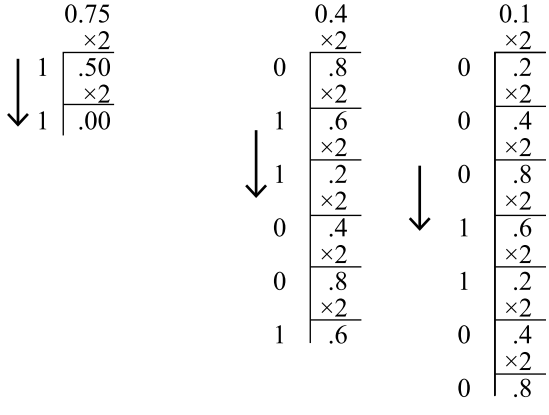
The decimal numbers (radix 10) are converted to binary form with digits 0 and 1 (radix 2) in the computer. An integer decimal number, may be converted to binary equivalent by following procedure:



Divide repeatedly by 2 until last quotient is 1, keeping the remainder against the quotient; read the binary digits in the direction of arrow. Thus we get,

$$23 = 10111;\ 14 = 1110.$$

The fractional decimal number is converted to binary form in the following manner:



Multiply by 2 until the decimal part is zero, saving digit 0 or 1 before the decimal point. Read the digits saved in a top-down manner. Thus the converted numbers are

$$0.75 = 0.11;\ 0.4 = 0.0110011\ (0011\ \text{recurring});\ 0.1 = 0.0001100\ (1100\ \text{recurring})$$

When a decimal number consists of both parts, integral as well as fractional, then both parts are converted to binary forms separately. For example, 23.75 will convert to 10111.11. It should be clear from the above examples that the integer numbers in the decimal system can be converted exactly in the binary system but most of the non-integers may be represented approximately due to non-terminating character of the converted numbers.

For conversion from binary to decimal, we simply multiply the binary digits by their respective place-value and add. For example, 10111.11 can be converted to decimal form as,

$$\begin{array}{cccccccc} 2^4 & 2^3 & 2^2 & 2^1 & 2^0 & 2^{-1} & 2^{-2} & \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & = 1\times 2^4+0\times 2^3+1\times 2^2+1\times 2^1+1\times 2^0+1\times 2^{-1}+1\times 2^{-2} \\ & & & & & & & = 16+0+4+2+1+.5+.25 \\ & & & & & & & = 23.75 \end{array}$$

It may also be noted that largest k-digit binary integer will have the value $2^k - 1$ in decimal. For example, the largest 2-digit binary number will be $11 = 2^2 - 1 = 3$ and a 3-digit largest binary number will be $111 = 2^3 - 1 = 7$ and so on. Obviously all the k digits will be binary 1's. A k-digit binary number can represent 2^k decimal numbers from 0 to $2^k - 1$.

1.3.1 Binary number representation in computer

As stated earlier, all the input data is converted to binary inside the computer; while the decimal integers are represented exactly in the computer memory, the non-integers are represented in floating point form. We would like to explain very briefly as how the floating point numbers are stored in the computer memory. Consider the floating point representation of binary numbers given below:

Binary number	*Floating point form*	*Mantissa*	*Exponent*
0.0111	0.1110×10^{-01}	+0.1110	−01
−1.101	$-0.1101\times 10^{+01}$	−0.1101	+01
11.1	$0.1110\times 10^{+10}$	+0.1110	+10

It may be noted that all numbers are in binary so that 10 is equal to 2 in decimal. The other thing to be noted is that mantissa is expressed in four digits and exponent in two digits, in each case.

Let us now consider a hypothetical case of a computer having a word length of 8 bits only. Out of eight bits, the left-most bit is used for storing the sign of mantissa. Let 0 denote positive (+ve) and 1 denote negative (−ve) sign of mantissa. The next four bits are used for storing the binary digits of mantissa. The right-most 3 bits are used for storing the exponent part; the first bit for storing its sign and last two bits for its value, digit 0 showing positive (+ve) and digit 1 showing negative (−ve) exponent (See Fig. 1.1).

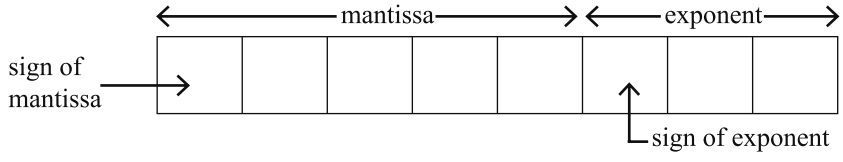


Figure 1.1 Floating point representation in 8-bit computer memory.

According to the memory configuration of Fig. 1.1 the binary numbers given above will be represented in the floating point form as follows:

	Binary number with decimal equivalent	*Representation in 8-bit memory*
(*a*)	0.0111 (0.4375)	01110101
(*b*)	−1.101 (−1.875)	11101001
(*c*)	11.1 (3.5)	01110010

It may be stated that the positive exponent varies from 000 to 011, i.e., from 0 to 3 in decimal. The negative exponent should vary from 101 to 111, i.e., from −1 to −3. But 100 may be considered as −4, since 000 is already zero, hence negative exponent varies from −1 to −4 in decimal.

It may be noted that the largest positive number that can be stored under present configuration would be, $0.1111 \times 10^{11} = 111.1$ (binary) $= 7.5$ (decimal). The algebraically smallest number that can be stored would be −7.5 (in decimal). Thus the range of numbers that can be represented in the computer memory would be $-7.5 \leq x \leq 7.5$. The smallest positive non-zero number represented in the above memory configuration would be, $0.1000 \times 10^{100} = 0.00001$ (binary) $= 2^{-5} = 0.03125$ (decimal). However, it may also be mentioned that even the simplest computer has a memory of 32-bit word and two or more words can be adjoined to store a number in floating point. Thus the space (number of bits) occupied by the mantissa and the exponent would be manifolds that of shown in Fig. 1.1

but the logic remains same. When a fixed number of decimal digits are kept in all numbers, it is called 'Fixed Point' representation.

1.4 Significant Digits

All the digits from 0 to 9 in a number, except the zeros which are used for fixation of decimal point, are called significant digits (or figures). For example, in the number .003456, the first two zeros are not significant since we can also express the number as $.3456 \times 10^{-2}$, while the other four digits, namely, 3, 4, 5 and 6 are significant. But in the number 20.003456, all the eight digits are significant. In order to find the number of significant digits in a number, express it in floating point; the mantissa part gives the number of significant digits.Whether the last zeros in a number are significant or not may depend on the context. For example, in measuring the heights of the students, in 168.00 cm, the last zero may not be significant and we can express the height as 168.0 cm, showing that the height is being measured nearest to the $\frac{1}{10}$th part of the centimeter, so that zero in 168.0 cm is significant.

1.5 Rounding and Chopping a Number

In scientific computing we are encountered by numbers with too many digits. More often than not, we have to shorten/reduce them to a size which may not affect the end result within a desired accuracy. There are two ways of reducing the size of or truncating the number, viz., (*i*) rounding (*ii*) chopping. Let us first discuss the procedure for rounding off a number x in decimal system.

Let the number x be expressed in floating point form with s digits in mantissa and with exponent q, as

$$x = \cdot d_1 d_2 \ldots d_n d_{n+1} \ldots . d_s \times 10^q.$$

If the number x is to be rounded to n significant digits, following procedure would be adopted:

(*i*) if $d_{n+1} < 5$, then no change in any of the digits from d_1 to d_n and the rounded number would be,

$$x \simeq .d_1 d_2 \ldots . \, d_n \times 10^q.$$

(*ii*) if $d_{n+1} > 5$, then digit d_n is incremented by 1, i.e. d_n becomes $d_n + 1$; as a cosequence of this other digits may get affected and even the exponent may have to be adjusted accordingly.

(*iii*) if $d_{n+1} = 5$, then there will be two ways for rounding, depending upon d_n being an even digit $(0, 2, 4, 6, 8)$ or an odd digit (1, 3, 5, 7, 9). If d_n is even, then case (i) applies and if d_n is odd then case (ii) applies. Thus probability of both cases is $\frac{1}{2}$ when $d_{n+1} = 5$.

Given below are some examples of rounding the numbers to four places of decimal (four significant digits):

	Floating point number	*Rounded to four decimals*
(a)	0.245684×10^2	0.2457×10^2
(b)	0.245629×10^{-2}	0.2456×10^{-2}
(c)	0.245659×10^2	0.2456×10^2
(d)	0.245750×10^2	0.2458×10^2
(e)	0.999951×10^2	0.1000×10^3
(f)	0.999858×10^2	0.9998×10^2

The difference between examples (e) and (f) may be noted. It may also be observed that in example (e) all zeros in the rounded number are significant.

However, a more conventional way for rounding, is straight in that if $d_{n+1} < 5$, then all the digits from d_1 to d_n remain unaltered [case (i)] while if $d_{n+1} \geq 5$, then d_n is incremented by 1 and necessary changes are made in the digits d_1 to d_n and also in the exponent, if necessary [case (ii)].

When all the digits after d_n are ignored, irrespective of whatever value d_{n+1} has, the procedure for truncating the number is known as 'chopping off' the number or simply 'chopping'. If there are sufficient number of significant digits in a number, like in a computer, the process of chopping may not affect the result in normal circumstances.

1.6 Errors due to Rounding/Chopping

Suppose a number x is rounded to x^*, then the modulus of the difference between x and x^*, i.e. $|x - x^*|$ is known as rounding error or error due to rounding in x^*.

Let x be a number which has been rounded to 4 decimals, say $x^* = 0.4387$. Then lower and upper bounds for the actual number x would be,

$$0.43865 \leq x < 0.43875$$

or $\quad 0.43865 - 0.4387 \leq x - x^* < 0.43875 - 0.4387$

or $$-0.00005 \leq x - x^* < 0.00005$$

or $$|x - x^*| \leq 0.00005 = \frac{1}{2} \times 10^{-4}$$

$$= \frac{1}{2} \times \text{ unit at } 4^{\text{th}} \text{ decimal place.}$$

The above result can be generalised for a number x represented in the floating point form as,

$$x = \cdot d_1 d_2 \ldots . d_n d_{n+1} \ldots . d_s \times 10^q.$$

If x is rounded to n decimals, then the maximum rounding error would be,

$$|x - x^*| \leq \frac{1}{2} \times 10^{-n} \times 10^q$$

$$= \frac{1}{2} \times 10^{q-n}. \tag{1.1}$$

If the number x is chopped off to n decimals, then it is easy to see that the maximum error due to chopping would be,

$$|x - x^*| \leq 10^{q-n}. \tag{1.2}$$

That is, the error in chopping a number is twice that in the rounding.

1.7 Measures of Error in Approximate Numbers

Let x^* be an approximation of exact number x, then we can measure the magnitude of error in three different forms:

(*i*) absolute error (a.e.) $= |x - x^*|$ (1.3a)

(*ii*) relative error (r.e.) $= \left|\frac{x - x^*}{x}\right|$ or $\left|\frac{x - x^*}{x^*}\right|$ (1.3b)

(*iii*) percentage error (p.e.) $=$ r.e. $\times 100$ (1.3c)

1.8 Errors in Arithmetic Operations

Let x_1 and x_2 be two numbers which are rounded to x_1^* and x_2^* respectively and let ε_1 and ε_2 be the corresponding rounding errors in them, such that $x_1 = x_1^* + \varepsilon_1$ and $x_2 = x_2^* + \varepsilon_2$. We are going to study below, the effects of rounding errors on arithmetic operations, viz., addition, subtraction, multiplication and division.

(*i*) Addition

$$x_1 + x_2 = x_1^* + \varepsilon_1 + x_2^* + \varepsilon_2$$

$$= x_1^* + x_2^* + \varepsilon_1 + \varepsilon_2$$

or $\quad |(x_1 + x_2) - (x_1^* + x_2^*)| = |(x_1 - x_1^*) + (x_2 - x_2^*)| = |\varepsilon_1 + \varepsilon_2| \leq |\varepsilon_1| + |\varepsilon_2|.$

This can be generalised to n numbers as,

$$\left|\sum_{i=1}^{n} x_i - \sum_{i=1}^{n} x_i^*\right| = \left|\sum_{i=1}^{n} (x_i - x_i^*)\right| = \left|\sum_{i=1}^{n} \varepsilon_i\right| \leq \sum_{i=1}^{n} |\varepsilon_i| \tag{1.4}$$

That is, the total absolute error in the sum of n numbers will be less than or equal to the sum of the absolute errors in each of them. Although it gives the upper bound for the absolute error in the sum, the actual error will be much smaller since some of the errors may be positive and some negative so that cumulative effect would be much reduced.

(*ii*) Subtraction

$$x_1 - x_2 = x_1^* + \varepsilon_1 - x_2^* - \varepsilon_2$$

or $\quad |(x_1 - x_2) - (x_1^* - x_2^*)| = |(x_1 - x_1^*) - (x_2 - x_2^*)| = |\varepsilon_1 - \varepsilon_2| \leq |\varepsilon_1| + |\varepsilon_2|.$ (1.5)

Thus the absolute error in subtraction of two approximate numbers can be as great as the sum of their individual absolute errors; since the errors can be positive or negative and if ε_1 and ε_2 are of opposite signs, then under subtraction, they will be added up.

(*iii*) Multiplication

$$x_1 \cdot x_2 = (x_1^* + \varepsilon_1) \cdot (x_2^* + \varepsilon_2)$$

or $\quad x_1 \cdot x_2 - x_1^* \cdot x_2^* = x_1^* \varepsilon_2 + x_2^* \varepsilon_1$, neglecting $\varepsilon_1 \varepsilon_2$

$$\text{a.e.} = |x_1 x_2 - x_1^* \cdot x_2^*| \leq |x_1^* \varepsilon_2| + |x_2^* \varepsilon_1| \tag{1.6a}$$

$$\text{r.e.} = \left|\frac{x_1x_2 - x_1^*x_2^*}{x_1^*x_2^*}\right| \leq \left|\frac{\varepsilon_1}{x_1^*}\right| + \left|\frac{\varepsilon_2}{x_2^*}\right|. \tag{1.6b}$$

Thus the maximum relative error in the product of two approximate numbers will be less than or equal to the sum their individual relative errors. This can be generalised to n numbers.

(*iv*) Division

$$\frac{x_1}{x_2} = \frac{x_1^* + \varepsilon_1}{x_2^* + \varepsilon_2} = \frac{x_1^*\left(1 + \dfrac{\varepsilon_1}{x_1^*}\right)}{x_2^*\left(1 + \dfrac{\varepsilon_2}{x_2^*}\right)} = \frac{x_1^*}{x_2^*}\left(1 + \frac{\varepsilon_1}{x_1^*}\right)\left(1 + \frac{\varepsilon_2}{x_2^*}\right)^{-1}$$

$$= \frac{x_1^*}{x_2^*}\left(1 + \frac{\varepsilon_1}{x_1^*}\right)\left(1 - \frac{\varepsilon_2}{x_2^*}\right), \text{ neglecting } {\varepsilon_2}^2 \text{ and higher powers}$$

$$= \frac{x_1^*}{x_2^*}\left(1 + \frac{\varepsilon_1}{x_1^*} - \frac{\varepsilon_2}{x_2^*}\right), \text{ neglecting } \varepsilon_1\varepsilon_2 \text{ term.}$$

$$\text{a.e.} = \left|\frac{x_1}{x_2} - \frac{x_1^*}{x_2^*}\right| \leq \left|\frac{\varepsilon_1}{x_2^*}\right| + \left|\frac{\varepsilon_2 x_1^*}{{x_2^*}^2}\right|. \tag{1.7a}$$

$$\text{r.e.} = \left|\left(\frac{x_1}{x_2} - \frac{x_1^*}{x_2^*}\right) \div \frac{x_1^*}{x_2^*}\right| \leq \left|\frac{\varepsilon_1}{x_1^*}\right| + \left|\frac{\varepsilon_2}{x_2^*}\right|. \tag{1.7b}$$

Like multiplication, the relative error in the division of a number by another number cannot exceed the sum of their individual relative errors.

1.9 Computation of Errors Using Differentials

Let z be a function of two variables x and y defined as $z = f(x,\ y)$. If increments δx and δy are given to x and y respectively, then the corresponding increment δz in z is given by,

$$\delta z = f(x + \delta x,\ y + \delta y) - f(x,\ y).$$

Expanding the first term by Taylor's series (see Appendix A),

$$\delta z = \left[f(x,\ y) + \frac{\partial f}{\partial x}\delta x + \frac{\partial f}{\partial y}\delta y + \frac{1}{2}\left(\frac{\partial^2 f}{\partial x^2}\delta x^2 + 2\frac{\partial^2 f}{\partial x \partial y}\delta x \cdot \delta y + \frac{\partial^2 f}{\partial y^2}\delta y^2\right) + \cdots\right] - f(x,\ y).$$

Neglecting higher powers of δx and δy and their products, assuming they are small, above may be written as,

$$\delta z \simeq \frac{\partial f}{\partial x}\delta x + \frac{\partial f}{\partial y}\delta y. \tag{1.8}$$

The error in arithmetic operations can be explained with the help of formula (1.8) considering δx and δy as errors in x and y respectively:

(*i*) & (*ii*) Addition/Subtraction

$$z = f(x, y) = x \pm y; \quad \frac{\partial f}{\partial x} = 1, \ \frac{\partial f}{\partial y} = \pm 1.$$

a.e. $$= |\delta z| \leq |\delta x| + |\delta y|$$

(*iii*) Multiplication

$$z = f(x, y) = xy; \quad \frac{\partial f}{\partial x} = y, \ \frac{\partial f}{\partial y} = x.$$

a.e. $$= |\delta z| \leq |y\delta x| + |x\delta y| \text{ and r.e.} = \left|\frac{\delta z}{z}\right| \leq \left|\frac{\delta x}{x}\right| + \left|\frac{\delta y}{y}\right|.$$

(*iv*) Division

$$z = f(x, y) = \frac{x}{y}; \quad \frac{\partial f}{\partial z} = \frac{1}{y}, \ \frac{\partial f}{\partial y} = -\frac{x}{y^2}.$$

$$\delta z = \frac{\delta x}{y} - \frac{x\delta y}{y^2} \text{ or } \frac{\delta z}{z} = \frac{\delta x}{x} - \frac{\delta y}{y}$$

$$\text{r.e.} = \left|\frac{\delta z}{z}\right| \leq \left|\frac{\delta x}{x}\right| + \left|\frac{\delta y}{y}\right|.$$

Note: The analysis can be extended for n variables $x_1, x_2, \ldots x_n$.

If $z = f(x_1, x_2, \ldots x_n)$ then

$$\delta z = \frac{\partial f}{\partial x_1} \cdot \delta x_1 + \frac{\partial f}{\partial x_2} \cdot \delta x_2 + \ldots + \frac{\partial f}{\partial x_n} \cdot \delta x_n.$$