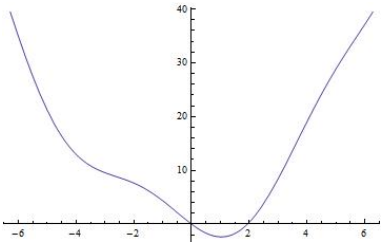
1 Introduction

- Numerical methods are mathematical techniques used for solving mathematical problems that cannot be solved or are difficult to solve (example: eq.1). The numerical solution is an approximate numerical value for the solution. Although numerical solutions are an approximation, they can be very accurate.
  - Example: Find the roots of the following equation

$$f(x) = x^2 - 4\sin(x) = 0$$



- In many numerical methods, the calculations are executed in an iterative manner until a desired accuracy is achieved.
  - Example: start at one value of x then change its value in small increment. A change in the sign of $f(x)$ indicates that there is a root within the last increment.
- Today, numerical methods are used in fast electronic digital computers that make it possible to execute many tedious and repetitive calculations that produce accurate (even though not exact) solutions in a very short time.
- For every type of mathematical problem there are several numerical techniques that can be used. The techniques differ in accuracy, length of calculations, and difficulty in programming.
  - Example: to solve a nonlinear equation of the form $f(x) = 0$ one can use among others
    - Bisection method
    - Regula Falsi method
    - Newton's method
    - Secant method

  These techniques and others will be discussed in the next chapters.

2 Errors in numerical solutions

Since numerical solutions are an approximation, and since the computer program that executes the numerical method might have errors, a numerical solution needs to be examined closely. There are three major sources of error in computation: **human errors**, **truncation errors**, and **round-off errors**.

2.1 Human errors

Typical human errors are arithmetic errors, and/or programming errors: These errors can be very hard to detect unless they give obviously incorrect solution. In discussing errors, we shall assume that human errors are not present.

- Example of arithmetic errors: When parentheses or the rules about orders of operation are misunderstood or ignored:
  - You can remember the correct order of operations rules by the mnemonic **PEMDAS**, which says to compute anything inside Parentheses first, then compute Exponential expressions (powers) next, then compute Multiplications and Divisions from left to right, and finally compute Additions and Subtractions from left to right. The highest priority for parentheses means that you should follow the remaining rules for anything inside the parentheses to arrive at a result for that part of the calculation.

2.2 Truncation and Round-off errors

- Error in computation is the difference between the exact answer X_{ex} and he computed answer X_{cp}. This is also known as true error

$$true\ error = X_{cp} - X_{ex}$$

- Since we are usually interested in the magnitude or absolute value of the error we define

$$true\ absolute\ error = \mid X_{cp} - X_{ex} \mid$$

- The absolute error can sometimes be misleading. Even if the absolute error is small in magnitude, the approximation may still be grossly inaccurate if the exact value X_{ex} is even smaller in magnitude. For this reason, it is preferable to measure accuracy in terms of the relative error.

$$true\ realative\ error = \frac{\mid X_{cp} - X_{ex} \mid}{\mid X_{ex} \mid}$$

- Note that the errors defined above cannot be determined in problems that require numerical methods for their solution, this is because the exact solution X_{ex} is not known. These error quantities are useful for evaluating the accuracy of different numerical methods when the exact solution is known (problem solved analytically).
- Since the true errors cannot, in most cases, be calculated, other means are used for estimating the accuracy of a numerical solution. This will be discussed in more details in later chapters

2.2.1 Truncation error:

Numerical methods use approximations for solving problems. The errors introduced by the approximations are the truncation errors. For example, consider the Taylor series expansion

$$e^x = 1 + x + \frac{x^2}{2!} + ... + \frac{x^n}{n!} + ...$$

If the formula is used to calculate $e^{0.3}$ we get

$$e^{0.3} = 1 + 0.3 + \frac{0.3^2}{2!} + ... + \frac{0.3^n}{n!} + ...$$

Where do we stop the calculation? How many terms do we include? Theoretically the calculation will never stop. There are always more terms to add on. If we do stop after a finite number of terms, we will not get the exact answer. For example if we do take the first four terms as the approximation we get

$$r = e^{0.3} \approx 1 + 0.3 + \frac{0.3^2}{2!} + . \frac{0.3^3}{3!} = \bar{r}$$

For this calculation, the truncation error is

$$truncation\ error = r - \bar{r}$$

Another example of truncation error that is the approximate calculation of derivatives. The value of the derivative of a function at a point can be approximated by the expression:

$$\frac{df}{dx} |_{x=x_1} = \frac{f(x_2) - f(x_1)}{x_2 - x_1}$$

where x_2 is a point near x_1. The difference between the value of the true derivative and the value that is calculated with this equation is called a truncation error. The truncation error is dependent on the specific numerical method or algorithm used to solve a problem. The truncation error is independent of round-off error.

2.2.2 Round-off error

Numbers can be represented in various forms. The familiar decimal system (base 10) uses ten digits 0, 1 , ..., 9. A number is written by a sequence of digits that correspond to multiples of powers of 10.

* Example:

$$\begin{array}{cccccccccc} 10^4 & 10^3 & 10^2 & 10^1 & 10^0 & & 10^{-1} & 10^{-2} & 10^{-3} & 10^{-4} \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & & \downarrow & \downarrow & \downarrow & \downarrow \\ 6 & 0 & 7 & 2 & 4 & . & 3 & 1 & 2 & 5 \end{array}$$

$$6\times10^4 + 0\times10^3 + 7\times10^2 + 2\times10^1 + 4\times10^0 + 3\times10^{-1} + 1\times10^{-2} + 2\times10^{-3} + 5\times10^{-4} = 60{,}724.3125$$

In general, however, a number can be represented using other bases. A form that can be easily implemented in computers is the binary (base 2) system. In the binary system, a number is represented by using the two digits 0 and 1. A number is then written as a sequence of zeros and ones that correspond to multiples of powers of 2.

* Example:

$$\begin{array}{cccccccccc} 2^4 & 2^3 & 2^2 & 2^1 & 2^0 & & 2^{-1} & 2^{-2} & 2^{-3} \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & & \downarrow & \downarrow & \downarrow \\ 1 & 0 & 0 & 1 & 1 & . & 1 & 0 & 1 \end{array}$$

$$1\times2^4 + 0\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0 + 1\times2^{-1} + 0\times2^{-2} + 1\times2^{-3}$$

$$1\times16 + 0\times8 + 0\times4 + 1\times2 + 1\times1 + 1\times0.5 + 0\times0.25 + 1\times0.125 = 19.625$$

Computers store and process numbers in binary (base 2) form. Each binary digit (one or zero) is called a bit (for binary digit). Binary arithmetic is used by computers because modern transistors can be used as extremely fast switches. Therefore, a network of these may be used to represent strings of numbers with the 1 referring to the switch being in the on position and 0 referring to the off position. Various operations are then performed on these sequences of ones and zeros.

Computer memory has a finite capacity. This obvious fact has far reaching implications on the representation of real numbers, which in general do not have a finite uniform representation. To accommodate large and small numbers, real numbers are written in floating point representation. A floating-point number is typically expressed in the scientific notation, with a fraction (F), and an exponent (e) of a certain radix (r), in the form of $F \times r^e$. Decimal numbers use radix of 10 ($F \times 10^e$); while binary numbers use radix of 2 ($F \times 2^e$).

Any real number x is accurately representable by an infinite sequence of digits of the form

$$x = \pm(1.d_1d_2d_3....) \times 2^e$$

where e is an integer exponent. The (possibly infinite) set of binary digits d_i each have a value 0 or 1 is called mantissa. The decimal value of this mantissa is

$$1.d_1d_2d_3.... = 1 + \frac{d_1}{2} + \frac{d_2}{2^2} + \frac{d_3}{2^3} + ...$$

For instance, the binary representation $x = (1.10100) \times 2^1$ has in decimal the value

$$x = (1 + 1/2 + 1/8) \times 2 = 3.25$$

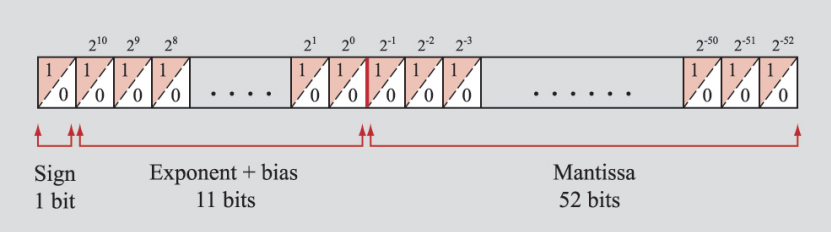
Of course, it should be clear that the choice of a binary representation is just one of many possibilities, indeed a convenient choice when it comes to computers. To represent any real number on the computer, we associate to x a floating point representation $fl(x)$ of a form similar to that of x but with only t digits, so

$$fl(x) = sign(x) \times (1.d_1d_2d_3d_{t-1}d_t) \times 2^e$$

for some t that is fixed in advance. Storing this fraction in memory thus requires t bits. The exponent e must also be stored in a fixed number of bits and therefore must be bounded, say between a lower bound L and an upper bound U.

Without being aware of any further details it should already be clear that representing x by $fl(x)$ necessarily causes an error also known as round-off error. Given a real number x, the machine number obtained by rounding x is denoted by $fl(x)$.

According to the IEEE-754 (Institute of Electrical and Electronics Engineers) standard, computers store numbers and carry out calculations in single precision or in double precision. In single precision, the numbers are stored in a string of 32 bits (4 bytes), and in double precision in a string of 64 bits (8 bytes). In both cases the first bit stores the sign (0 corresponds to + and 1 corresponds to -) of the number. The next 8 bits in single precision (11 bits in double precision) are used for storing the exponent. The following 23 bits in single precision (52 bits in double precision) are used for storing the mantissa.



The value of the mantissa is entered as is in a binary form. The value of the exponent is entered with a bias. A bias means that a constant is added to the value of the exponent. The bias is introduced in order to avoid using one of the bits for the sign of the exponent (since the exponent can be positive or negative). In binary notation, the largest number that can be written with 11 bits is 2047 (when all 11 digits are 1). The bias that is used is 1023, which means that if, for example, the exponent is 4, then the value that is stored is $4 + 1023 = 1027$. Thus, the smallest exponent that can be stored by the computer is -1023 (which will be stored as 0, and the largest is 1024 (which will be stored as 2047). In single precision, 8 bits are allocated to the value of the exponent and the bias is 127.

Example:

0100000001111110100000000000000000000000000000000000000000000000, considered as a double precision word can be interpreted as follows:

* Based on the first digit 0 the sign is positive by convention.
* The next 11 digits 10000000111 form the exponent: in decimal, $b = 1 \times 2^{10} + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 1031$. So in decimal,

$$e = b - 1023 = 1031 - 1023 = 8$$

.
* The next 52 bits 1110100000000000000000000000000000000000000000000000 form the decimal fraction:

$$f = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{32} = 0.90625$$

.
* The number in decimal is therefore

$$s \cdot (1 + f) \times 2^e = 1.90625 \times 2^8 = 488.$$

A central question is how accurate the floating point representation of the real numbers is. Specifically, we ask how large the relative error in such a representation, defined by

$$\frac{| \, fl(x) - x \, |}{| \, x \, |}$$

Modern floating point systems guarantee that this relative error is bounded by

$$\eta = \frac{\sum_{i=t+1}^{\infty} (\frac{1}{2})^i}{1 + \sum_{i=1}^{\infty} (\frac{1}{2})^i} = \frac{1}{2} \times 2^{-t}$$

The important quantity η been called rounding unit, machine precision, machine epsilon and more. The usual floating point word in the standard floating point system, which is what Matlab uses by default, has 64 bits. Of these, 52 bits are devoted to the mantissa while the rest store the sign and the exponent. Hence the rounding unit is

$$\eta = 2^{-53} \approx 1.1 \times 10^{-16}$$

Example:

$$0.1 = 1.6 \times 2^{-4}$$

The exponent $e = -4$ can be stored exactly in a binary format that uses single precision as

$$b = -4 + 127 = 123$$

. The mantissa 0.6 however, cannot be written exactly in a binary format that uses 23 bits (see Matlab code).

Notes:

* Even if number representations were exact in our floating point system, arithmetic operations involving such numbers introduce roundoff errors. These errors can be quite large in the relative sense. The IEEE standard however requires exact rounding, which yields that the relative error in each arithmetic operation is also bounded by η.

* Upon carefully examining the possible range of exponents you will notice that it is not fully utilized. For double precision $2^{11} = 2048$ different exponents could be distinguished, and for single precision $2^8 = 256$ are available. But only 2046 and 254, respectively, are used in practice. This is because the IEEE standard reserves the endpoints of the exponent range for special purposes. The largest number precisely representable in a standard long word is therefore.

$$\left[1 + \sum_{i=1}^{52} \frac{1}{2^i}\right] \times 2^{1023} \approx 1.8 \times 10^{308}$$

attempts to define a number with | *number* | larger than this causes overflow error and the smallest positive number is

$$[1 + 0] \times 2^{-1022} \approx 2.2 \times 10^{-308}$$

attempts to define a number with | *number* | smaller than than this causes underflow error How are 0 and 1 stored? Here is where the endpoints of the exponent are used, and the conventions are simple and straightforward:

- For 0, set all binaries in the exponent and mantissa to zero, with the sign s arbitrary; i.e., the minimal positive value representable in the system is considered 0.

$$[1 + 0] \times 2^{-1023} \approx 1.1 \times 10^{-308}$$

- For infinity, set the binaries in the exponent and mantissa to 1

$$\left[1 + \sum_{i=1}^{52} \frac{1}{2^i}\right] \times 2^{1024} \approx 1.8 \times 10^{308}$$

- The pattern in which all binaries in the exponent are 1 and one or more binaries in the mantissa is zero is by convention NaN.
- In single precision, or short word, the largest number precisely representable is

$$\left[1 + \sum_{i=1}^{23} \frac{1}{2^i}\right] \times 2^{127} \approx 3.4 \times 10^{38}$$

and the smallest positive number is

$$[1 + 0]2^{-126} \approx 1.210^{-38}$$

* When overflow occurs in the course of a calculation, this is generally fatal. But underflow is non-fatal: the system usually sets the number to 0 and continues. (Matlab does this, quietly.)