Chapter 1: Introduction

Prof. Alok Shukla

Department of Physics
IIT Bombay, Powai, Mumbai 400076

Course Name: Introduction to Numerical Analysis (PH 307)

Introduction

- Our aim in the present course is to learn about numerical methods and their computer implementation, as relevant to the problems encountered in physics. Therefore, our motto will be

  "THE PURPOSE OF COMPUTING IS INSIGHT, NOT NUMBERS"

  - R. W. HAMMING
  (REPUTED COMPUTER SCIENTIST)

- Therefore in the course, besides learning about numerical methods, and their efficient implementation, we will also spend substantial amount of time on their applications in specific physics problems.

Basics

- Before really starting to study the numerical methods, we will briefly discuss a few fundamental concepts.
- **Error:** The error (or to be more precise, absolute error) associated with the value of a function is defined as

$$error = |True\ value - Calculated\ value\ (Approximate)| \quad (1)$$

The reason this concept is relevant is that most of the time we are able to calculate approximate value of a function by a computer.

Basics

- **Relative Error:** More important than the concept of absolute error, is the concept of relative error. First we will define the relative error

$$Relative\ error = \left| \frac{True\ value - Calculated\ value}{True\ value} \right| \qquad (2)$$

- For example, let us consider functions f(x) and g(x) with true values; $f_{true}(x) = 0.001$ and $g_{true}(x) = 3.000$
- Using some numerical procedure they are calculated to be; $f_{calc}(x) = 0.002$ and $g_{calc}(x) = 3.001$
- Note that in both the cases, absolute error = 0.001, which gives the misleading impression that both the functions are being computed with equal precision.

Basics

- It is the relative error which clarifies the picture.
- Rel. Error (f(x)) = 0.001/0.001 = 1, and Rel. Error (g(x)) = 0.001/3 $\approx$ 0.00033
- Thus, there is hardly any precision in the computation of f(x), while g(x) is being computed very accurately.

Basics

Other Properties of Relative Error

- Relative error is scale invariant, which is not the case with the absolute error.

$$Rel.\ Error(\lambda f(x)) = \frac{|\lambda f_{true} - \lambda f_{calc}|}{|\lambda f_{true}|}$$

$$= Rel.\ Error(f(x))$$

above λ is an arbitrary constant, called a scaling parameter

- The concept of a relative error fails when the true value of the calculated function is 0, e.g. $\sin(x)$ for $x = n\pi$, where n is an integer.
- In such a case the use of absolute error is mandatory which is clearly well defined.

Storage of numbers in a computer

- Normally numbers are stored in a computer in two formats: integers and floating point numbers. We describe them next.
- **Integers:** These, as the name suggests are whole numbers along with a sign. Examples of integers are - 39, 0, 35670 etc.
- This integers are characterized by a sign ($\pm$) and the magnitude of the number.
- **Floating Point Numbers:** In order to represent numbers other than integers, floating point number system is used. It consists of three parts; sign, mantissa and exponent as below:

$$number = sign.mantissa \times 10^{exponent}$$

where mantissa and the exponents are integers.

Storage of numbers in a computer

- This number system is best explained by examples. For example, $1234.0 = +.1234 \times 10^4$; 1234 is mantissa and 4 is the exponent.
- Similarly a number in the binary system

$$d_n d_{n-1} \ldots . d_0 . d_{-1} d_{-2} \ldots . d_{-m} = \sum_{i=-m}^{n} d_i \times 2^i \qquad (3)$$

where $d_i \in 0, 1$, for all i.

Conversion from decimal to binary

- The method to convert a decimal number into a binary becomes obvious when we write Eq. 3 as

$$d_n 2^n + d_{n-1} 2^{n-1} .... d_1 2^1 + d_0 + d_{-1} 2^{-1} + d_{-2} 2^{-2} + ... d_{-m} 2^{-m} \tag{4}$$

- We will demonstrate it for three different cases.
- **Integer:** From Eq. 4 it is obvious that if we keep dividing an integer by 2, the successive remainders will be the binary digits d_i.
- Consider an example, convert 13 to binary representation

$\frac{13}{2}$, Quotient = 6, Remainder = 1; $\frac{6}{2}$, Quotient = 3, Remainder = 0,
$\frac{3}{2}$, Quotient = 1, Remainder = 1

Conversion from decimal to binary

Thus, $13 \equiv 1101 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1$

- The convention adopted to represent negative integers on modern computers is to use one bit (the left most bit) as the sign bit and then the magnitude of the integer is stored as usual.
- Positive sign is denoted by bit 0 and the negative sign by bit 1. Thus, $-100 = \underbrace{1}_{\textit{sign bit}} \underbrace{100100}_{\textit{magnitude}}$ and $100 = \underbrace{0}_{\textit{sign bit}} \underbrace{100100}_{\textit{magnitude}}$
- Thus one extra bit is used to denote the sign of the integer.
- **Fractions:** In order to convert a decimal fraction into a binary one, we keep multiplying the number by 2 and the digits which appear to the left of decimal will be the binary digits d_{-1}, d_{-2},....etc.

Conversion from decimal to binary

- Consider the example of converting 0.1 to binary representation

$$
\begin{array}{lll}
2 \times 0.1 & \rightarrow 0.2 & \stackrel{\times 2}{\rightarrow} 0.4 \\
\stackrel{\times 2}{\rightarrow} 0.8 & \stackrel{\times 2}{\rightarrow} 1.6 & \rightarrow 0.6 \\
\stackrel{\times 2}{\rightarrow} 1.2 & \rightarrow 0.2 & \stackrel{\times 2}{\rightarrow} 0.4 \\
\stackrel{\times 2}{\rightarrow} 0.8 & \stackrel{\times 2}{\rightarrow} 1.6 & \rightarrow 0.6
\end{array}
$$

Thus, 0.1 in the decimal system is 0.0001100110011... in the binary representation

- Clearly, 0.1 cannot be represented exactly in the binary representation.
- How exactly can such a fraction be represented in binary number systems will depend on the size of memory used to represent a real number.
- Therefore, it is clear that in the process of representation of fractional numbers, such precision will be lost.

Conversion from decimal to binary

- **Floating Point Numbers:** In order to represent a real number in the binary floating point format, we use the following

$$(sign)d_2d_1d_0.d_{-1}d_{-2}d_{-3} = sign(0.d_2d_1d_0.d_{-1}d_{-2}d_{-3})2^3$$

$$\equiv (\underbrace{sign\ \ bit}_{0\ or\ 1})(\underbrace{d_2d_1d_0.d_{-1}d_{-2}d_{-3}}_{mantissa})(\underbrace{\underbrace{0}_{sign}11}_{exponent})$$

- Example: 13.75, Using the representation of 13 and 0.75, we immediately get $13.75 \equiv 1101.11 = 1.10111 \times 2^3$. So the following point representation will be

$$13.75 \equiv (\underbrace{0}_{sign})(\underbrace{1.10111}_{mantissa})(\underbrace{\underbrace{0}_{sign}011}_{exponent})$$

- Mantissa is normalized by removing the leading 1, i.e., making it 0.10111 which is stored as 10111

Conversion from decimal to binary

- From these examples it is clear that in order to store a floating point number in binary format, we need

1. One sign bit for mantissa.
2. Some bits for the magnitude of mantissa.
3. One bit for the sign of exponent.
4. Some bits for the exponent.

- For single precision arithmetic normally 4 bytes are used to store a floating point number and for double precision calculations 8 bytes are used.

Conclusions

- Based upon the preceding discussion, we arrive at following important conclusions:
- The largest integer that can be stored in a computer depends upon the size of the memory locations used to store them. In Fortran, the default storage size for integers is 4 bytes while 8 byte storage is also allowed.
- The largest, the smallest (other than zero), and the precision with which the floating point numbers can be stored in a computer will also depend upon the size of the memory location allocated to store floating point variables.
- The default storage of such variables in Fortran is 4 byte words while, for double precision arithmetic 8 byte words are normally used.

Finite Precision Errors

- By now it is clear to us that the precision of the machine arithmetic is finite, unlike the analytic mode arithmetic.
- This finite precision of computers leads to the following errors:
- **Round off Error:** This error is in general present in the representation of all floating point numbers involving fractions.
- In an example earlier we saw that the precision with which we can express 0.1 in binary system will depend upon size of the memory used to store floating point variables.
- Thus, depending upon the memory size, a given fraction is rounded off to the nearest approximate number.

Finite Precision Errors

- Suppose our computer system is such that it allowed precision up to 4 places of decimal, then it will round off 1/3 to 0.3333.
- **Examples:** Round off error begins to play an important role, in the multiplication of small numbers, and in the subtraction of two fractional numbers of approximately the same magnitude.
- If two numbers x = 0.1492 and y=0.1498 are correct up to fourth digit, i.e. absolute error is no larger than 5 units in the fifth place. So relative error in the numbers is

$$Rel.\ Error \approx \frac{0.00005}{0.15000} = \frac{5}{150000} \approx 0.03\%$$

Finite Precision Errors

- However the relative error in the difference is

$$Rel.\ Error \approx \frac{0.0001}{0.0006} \times 100 \approx 17\%$$

- Thus small relative error in the individual numbers can lead to large relative error in their difference if the two numbers are approximately of the same size. If one were to compute $\frac{1}{x-y}$, even larger relative errors will be found.
- **Overflow:** Overflow is said to occur when during the course of arithmetic a number is generated which is larger than the largest number representable on that computer.

Finite Precision Errors

- Most often overflow occurs when some number is divided by very small number such as zero.
- Thus one has to be careful in calculations involving $f(x) = \frac{x}{\varepsilon}$ with $\varepsilon \to 0$.
- **Underflow:** This occurs when one encounters a number which is smaller than the smallest number, other than zero, representable on the machine.
- Many machines automatically set such a number to zero or otherwise the programmer himself can do it.
- However, this error is not as severe as the overflow.

Speed of Calculations

- Due to reasons related to hardware, the speed of calculations involving integer arithmetic is much faster than those involving floating point operations.
- By arithmetic operations we imply simple operations such as addition/subtraction, multiplication/division.
- That is why while developing algorithms and programs our aim must be to minimize the number of floating point operations involved.