ICS 226: Numerical Analysis

Oliver Mhlanga

HARARE INSTITUTE OF TECHNOLOGY

omhlanga@hit.ac.zw or olivamhlanga@gmail.com; 0712531415

February 13, 2019

Error and Computer Arithmetic

Computer arithmetic and number storage

Numerical Analysis: This refers to the analysis of mathematical problems by numerical means, especially mathematical problems arising from models based on calculus.
Effective numerical analysis requires several things:

- An understanding of the **computational tool** being used, be it a calculator or a computer.
- An understanding of **the problem** to be solved.
- Construction of **an algorithm** which will solve the given mathematical problem to a **given desired accuracy and within the limits** of the resources (time, memory, etc) that are available.

Computer arithmetic and number storage

Definition

A **mathematical model** is a mathematical description of a physical situation.

The main goal of **numerical analysis** is to develop efficient algorithms for computing precise numerical values of mathematical quantities, including functions, integrals, solutions of algebraic equations, solutions of differential equations (both ordinary and partial), solutions of minimization problems, and so on.
Part of this process is the consideration of the errors that arise in these calculations, from the errors in the arithmetic operations or from other sources.

Computer arithmetic and number storage

Computers use **binary arithmetic**, representing each number as a **binary number** (consisting two **bits** 0 and 1): a finite sum of integer powers of 2. Some numbers can be represented exactly, but others, such as $1/10,\ 1/100,\ 1/1000, ...$ cannot.
For example, $2,125 = 2^1 + 2^{-3}$ has an exact representation in binary (base 2), but
$3,1 = 2^1 + 2^0 + 2^{-4} + 2^{-5} + 2^{-8} + ...$ does not.
And, of course, there are the transcendental numbers like π that have no finite representation in either decimal or binary number system.

Computer arithmetic and number storage

A **bit** is the most basic unit of information in a computer, usually labelled 0 and 1. It is a state of 'on' or 'off' in a digital circuit. Sometimes they represent high or low voltage .The term 'bit' is an abbreviation for binary digit.
A **byte** is a unit of information built from bits; one byte equals 8 bits. A **word** is a contiguous group of bytes. The number of bits used by a computers CPU for addressing information represents one measure of a computers speed and power. Computers today often use 32 or 64 bits in groups of 4 and 8 bytes, respectively, in their addressing.
$8 bits = 1 byte$
$2^{10} bytes = 1 Kbyte \equiv 1024 bytes$
$2^{20} bytes = 1 Megabyte$
$2^{30} bytes = 1 Gigabyte$
$2^{40} bytes = 1 Terabyte$

Computer arithmetic and number storage

Characters are letters of the alphabet, both upper and lower case, punctuation marks, and various other symbols. In the ASCII convention (American Standard Code for Information Interchange) one character uses 7 bits. (there are at most $2^7 = 128$ different characters representable with this convention). As a consequence, each character will be stored in exactly one byte of memory.

Computer arithmetic and number storage

Decimal to binary conversion: For the integer part, we divide by 2 repeatedly (using integer division); the remainders are the successive digits of the number in base 2, from least to most significant.

Example 1 Convert $(107)_{10}$ to a binary number:

Quotients	107	53	26	13	6	3	1	0
Remainders		1	1	0	1	0	1	1

Therefore , $(107)_{10} = (1101011)_2$

For the fractional part, multiply the number by 2; take away the integer part, and multiply the fractional part of the result by 2, and so on; the sequence of integer parts are the digits of the base 2 number, from most to least significant.

Example 2 Convert $(0.625)_{10}$ to a binary number:

Quotients	0.625	0.25	0.5	0
Remainders		1	0	1

Therefore , $(0.625)_{10} = (0.101)_2$

Computer arithmetic and number storage

Note that if f has an infinite representation in base 10, its representation in base 2 will also be infinite. However, we could have situations where f is finitely represented in base 10 and infinitely represented in base 2.

Octal representation: A binary number can be easily represented in base 8. Partition the number into groups of 3 binary digits ($2^3 = 8$), from decimal point to the right and to the left (add zeros if needed). Then, replace each group by its octal equivalent.

Example 3 $(107.625)_{10} = (|1|\ |101|\ |011|\,.\,|101|)_2 = (153.5)_8$

Computer arithmetic and number storage

Hexadecimal representation: To represent a binary number in base 16 proceed as above, but now partition the number into groups of 4 binary digits ($2^4 = 16$). The base 16 digits are $0, ..., 9,\ A = 10, ..., F = 15$.
Example 4 $(107.625)_{10} = (|0110|\ \ |1011|\,.\,|1010|)_2 = (6B.A)_{16}$

Computer arithmetic and number storage

Before storing to the computer, all the numbers are converted into binary numbers and then these converted numbers are stored into computer memory. Generally, two bytes memory space is required to store an integer and four bytes space is required to store a floating point number. So, there is a limitation to store the numbers into computers.

In computer, the numbers are stored mainly in two forms:

(i) **integer or fixed point** form, and

Storing of integers is straight forward while representation of floating point numbers is different from our conventional technique. A k bit word can store a total of $N = 2^k$ different numbers. For example, the **standard single precision** computer uses 32 bit arithmetic, for a total of $N = 2^{32} \approx 4.3 \times 10^9$ different numbers can be stored, while **double precision** uses 64 bits, with $N = 2^{64} \approx 1.84 \times 10^9$ different numbers can be stored.

Computer arithmetic and number storage

The aim is to distribute the N exactly representable numbers over the real line for maximum efficiency and accuracy in practical computations. One choice is to space the numbers closely together, say with uniform gap of 2^{-k}, and so distribute our N numbers uniformly over the interval $-2^{-31} \leq x \leq 2^{31} - 1$ (a.k.a **Two's complement**).
The range for fixed-point numbers is too limited for scientific computing. If in a computation a number outside the range occurs, this is called **underflow** if the number is smaller and **overflow** when it is larger. In the case of underflow the result is usual set to zero. Overflow causes the computer to halt.
(ii) **real or floating point** form.
Is the most common non-uniform distribution of our N numbers and is based on scientific notation.

Errors in Numerical Computations

Definition

An n-digit floating-point number in base β has the form

$$x = \sigma.\bar{x}.\beta^e, \tag{1}$$

where $x \in \Re$,, and $e \in Z$ is an integer called the **exponent**. We call $\bar{x}$ the **significand or mantissa**, $1 \leq \bar{x} < \beta$, $\sigma = -1 or + 1$ is the **sign**.

The allowable number of digits in the mantissa is called the **precision** of the floating-point representation of $\bar{x}$
N.B: *A floating point number is represented as (sign,mantissa, exponent) with a limited number of digits for the mantissa and the exponent*.
For most computers $\beta = 2$, although on some $\beta = 16$, and in hand calculations and on most desk and pocket calculators, $\beta = 10$.

Floating point arithmetic

Definition

The **significant digits or significant figures** of a number are all its digits, except for zeros which appear to the left of the first non-zero digit.

Example 5. (with $n = 8$)

$$\begin{aligned} 12.625 &= +(.11001010)_2 \times 2^4 \\ &= +(1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 0 \times 2^{-4} \\ &\quad +1 \times 2^{-5} + 0 \times 2^{-6} + 1 \times 2^{-7} + 0 \times 2^{-8}) \times 2^4 \end{aligned}$$

Modern computers adopt **IEEE-754** standard for representing floating-point numbers.

- There are two representation schemes: **32-bit single-precision and 64-bit double-precision**.
- The IEEE-754 floating-point arithmetic standard is the format for floating point numbers used in almost all computers.
- IEEE is an abbreviation for the Institute of Electrical and Electronics Engineers.

Floating point arithmetic

IEEE floating point representation for binary real numbers consists of three parts. For a 32-bit (called single precision) number, they are:

- Sign, for which 1 bit is allocated.
- Mantissa (called significand in the standard) is allocated 23 bits.
- Exponent is allocated 8 bits. As both positive and negative numbers are required for the exponent, instead of using a separate sign bit for the exponent, the standard uses a biased representation. The value of the bias is 127.
- Has a precision of 24 binary digits-23 bits of the significand which is stored in the memory and an implied 1 as the most significant 24th bit. The exponent e is limited by $-126 \leq e \leq 127$.

Floating point arithmetic

- Instead of using a separate sign bit for the exponent, the standard uses a biased representation. The value of the bias is 127. Thus, a floating point number in the IEEE Standard is:

$$\underbrace{b_1}_{Sign} \qquad \underbrace{b_2 b_3 \ldots . b_9}_{Exponent:E=e+127} \qquad \underbrace{b_{10} b_{11} \ldots . b_{32}}_{Mantissa(\bar{x})}$$

  The sign σ is stored in bit b_1 ($b_1 = 0$ for $\sigma = +1$ & $b_1 = 1$ for $\sigma = -1$).
  Rather than e, the positive binary integer E is stored in $b_2 b_3 \ldots . b_9$.
- $x = \sigma.(1.a_1 a_2 \ldots . a_{23}).2^e$, stored on (**4 bytes**).
  in binary $(1111110)_2 \leq e \leq (1111111)_2$

Floating point arithmetic

The **IEEE double precision floating-point representation** of x has a precision of 53 binary digits, and the exponent e is limited by $-1022 \leq e \leq 1023$:

-
- $x = \sigma.(1.a_1 a_2 .... a_{52}).2^e$,
  stored on **8 bytes** (64 bits)
  $$\underbrace{b_1}_{Sign} \quad \underbrace{b_2 b_3 .... b_{12}}_{Exponent: E=e+1023} \quad \underbrace{b_{13} b_{14} .... b_{64}}_{Mantissa(\bar{x})}$$

Floating point arithmetic

Example 6. Represent 52.21875 in IEEE 754 32-bit floating point format.

$52.21875 = 110100.00111 = 1.1010000111 \times 2^5$.

Normalized significand $= .1010000111$.

Exponent: E= e+127=5+127 = 132.

The bit representation in IEEE format is

0	10000100	10100001110000000000000
Sign 1 bit	Exponent 8 bits	Significand 23 bits

Representation of Zero: zero is represented in the IEEE Standard by all 0s for the exponent and all 0s for the significand.

Representation of Infinity: All 1s in the exponent field is assumed to represent infinity (∞). A sign bit 0 represents ($+\infty$) and a sign bit 1 represents ($-\infty$).

Floating point arithmetic

Largest and Smallest Positive Floating Point Numbers: *single precision floating-point representation.*

Largest Positive Number:

0	11111110	11111111111111111111111
Sign	Exponent	Significand
1 bit	8 bits	23 bits

Significand: $1111...1 = 1 + (1 - 2^{-23}) = 2 - 2^{-23}$.

Exponent: $(254 - 127) = 127$.

Largest Number $= (2 - 2^{-23}) \times 2^{127} \cong 3.403 \times 10^{38}$. If the result of a computation exceeds the largest number that can be stored in the computer, then it is called an **overflow**.

Smallest Positive Number:

0	00000001	00000000000000000000000
Sign	Exponent	Significand
1 bit	8 bits	23 bits

Significand $= 1.0$. Exponent $= 1127 = 126$.

The smallest normalized number is $= 2^{-126} \cong 1.1755 \times 10^{-38}$.

Floating point arithmetic

When all the exponent bits are 0 and the leading hidden bit of the siginificand is 0, then the floating point number is called a <u>subnormal number</u>. The smallest positive subnormal number is:

0	00000000	00000000000000000000001
Sign	Exponent	Significand
1 bit	8 bits	23 bits

the value of which is $2^{-23} \times 2^{-126} = 2^{-149} \cong 1.4 \times 10^{-45}$. .
A result that is smaller than the smallest number that can be stored in a computer is called an **underflow**.

Definition

Machine Epsilon is defined as the difference between 1 and the next larger number that can be stored in that format.

This is a conservative definition used by industry.

Floating point arithmetic

- For example, in 32-bit IEEE format with a 23-bit significand, the machine epsilon is 1.00000000000000000000001 which is equivalent to $2^{-23} = 1.19 \times 10^{-7}$. This essentially tells us that the precision of decimal numbers stored in this format is 7 digits.
- The term precision and accuracy are not the same. Accuracy implies correctness whereas precision does not. For 64-bit representation of IEEE floating point numbers the significand length is 52 bits. Thus, machine epsilon is $2^{-52} = 2.22 \times 10^{-16}$. Therefore, decimal calculations with 64 bits give 16 digit precision.
- MATLAB: machine epsilon is available as the constant eps.

Floating point arithmetic

Academics define machine epsilon as the upper bound of the relative error when numbers are rounded. Thus, for 32-bit floating point numbers, the machine epsilon is $2^{-23}/2 \cong 5.96 \times 10^{-8}$.

- The machine epsilon is useful in iterative computation. When two successive iterates differ by less than $|epsilon|$ one may assume that the iteration has converged. The IEEE Standard does not define machine epsilon.
- **Tiny or dwarf** is the minimum subnormal number that can be represented using the specified floating point format. Thus, for IEEE 32-bit format, it is

0	00000000	00000000000000000000001
Sign 1 bit	Exponent 8 bits	Significand 23 bits

whose value is $2^{-149} \cong 1.4 \times 10^{-45}$

Rounding and Chopping

Let the mantissa in the floating-point representation contain n binary digits.If the number x in equation (1) has the mantissa $\bar{x}$ that requires more than n binary bits, it must be shortened when x is stored in an computer.
This is done in two ways:

- rounding by **chopping**: truncate or chop x to n binary digits, ignoring the remaining digits.
- optimal **rounding**: round x to n binary digits, based on the size of the part of $\bar{x}$ following digit n:
  (a) if digit n + 1 is 0, chop x to n digits
  (b) if digit n + 1 is 1, chop x to n digits and add 1 to the last digit of the result.

Denote the *machine floating-point version of x by fl(x)*.

Rounding and Chopping

The number fl(x) can be written in the form:

$$fl(x) = x.(1 + \epsilon) \tag{2}$$

where ϵ is a small number depending on x.

- If chopping is used $-2^{-n+1} \leq \epsilon \leq 0$
- If rounding is used $-2^{-n} \leq \epsilon \leq 2^{-n}$

Characteristics of chopping:

- the worst possible error is twice as large as when rounding is used
- the sign of the error $x - fl(x)$ is the same as the sign of x. This property is the worst of the two. It will lead no possibility of cancellation of errors.

Rounding and Chopping

Characteristics of rounding:

- the worst possible error is only one-half as large as for chopping.
- More important: the error $x - fl(x)$ is negative for half the cases and positive for the other half. This leads to better error propagation behavior in calculations involving many arithmetic operations.
- For single precision IEEE floating-point rounding arithmetic (there are n = 24 digits in the significand):
  1. chopping (often called 'rounding towards zero'):

  $$-2^{-23} \leq \epsilon \leq 0$$

  2. standard rounding:

  $$-2^{-24} \leq \epsilon \leq 2^{-24}$$

Rounding and Chopping

- For double precision IEEE floating-point rounding arithmetic:
  1. chopping:
  $$-2^{-52} \leq \epsilon \leq 0$$
  2. rounding:
  $$-2^{-53} \leq \epsilon \leq 2^{-53}$$

Consequences for Programming Floating-Point Arithmetic

Numbers that have finite decimal expressions may have infinite binary expansions.
For example $(0.1)_{10} = (0.0001100110011001100 11...)_2$
Therefore $(0.1)_{10}$ cannot be represented exactly in binary floating-point arithmetic.
Possible problem: Run into infinite loops.
Possible Solution: different kind of looping test should be used, one that is not sensitive to rounding or chopping errors.
N.B: Pay attention to the language used:

- Fortran and C have both single and double precision, specify double precision constants correctly.
- MATLAB does all computations in double precision.

Errors in Numerical Computations

In any applied numerical computation, there are three key sources of error:

1. **Inherent errors**:
   1.1 Inexactness of the mathematical model for the underlying physical phenomenon (often called *Modeling errors*).
   1.2 Errors in measurements of parameters entering the model (often called *Experimental errors*) .

   (1.1) is the domain of mathematical modeling, and will not concern us here. Neither will (1.2), which is the domain of the experimentalists.
2. **Round-off errors**: arises due to the finite numerical precision imposed by the computer.
3. **Truncation errors**: arises due to approximations used to solve the full mathematical system.

Errors in Numerical Computations

(3) is the true domain of numerical analysis, and refers to the fact that most systems of equations are too complicated to solve explicitly, or, even in cases when an analytic solution formula is known, directly obtaining the precise numerical values may be difficult.

Definition

Let x be a real number and let x^* be an approximation. The *absolute error* in the approximation $x^* \approx x$ is defined as $\Delta x = |x^* - x|$. The *relative error* is defined as the ratio of the absolute error to the size of x, i.e., $\frac{|x^* - x|}{|x|} = \frac{\Delta x}{|x|}$, provided $x \neq 0$; otherwise relative error is not defined.

If $x^* < x$, then we say that the number x^* is an approximate value of the number x by defect and if $x^* > x$, then it is an approximate value of x by excess.

Errors in Numerical Computations

The percentage error (a.k.a relative percentage error)of an approximate number is *Relative error* $\times$ 100%.

NB: Absolute error measures only quantity of error and it is the total amount of error incurred by approximate value. While the relative error measures both the quantity and quality of the measurement. It is the total error while measuring one unit. The absolute error depends on the measuring unit, but, relative error does not depend on measuring unit.

Example 7. Find the absolute, relative and percentage error in x^* when $x = 1/3$ and $x^* = 0.333$

Solution. The absolute error:

$\Delta x = |x^* - x| = |0.333 - 1/3| = 0.00033.$

Relative error :$\frac{\Delta x}{|x|} = \frac{0.00033}{1/3} = 0.001.$

Percentage error:*Relative error* $\times 100\% = 0.001 \times 100 = 0.1\%.$

Errors in Numerical Computations

H/W 1. Determine the absolute error and the exact number corresponding to the approximate number $x^* = 5.373$ if percentage error is 0.01%.

H/W 2.An exact number x is in the interval [28.03, 28.08]. Assuming an approximate value, find the absolute and the percentage errors. [*Hint*: The middle of the given interval is taken as its approximate value]

Errors in Numerical Computations

Significant error occurs due to the **loss of significant digits** during arithmetic computation. This error occurs mainly due to the finite representation of the numbers in computers or calculators. Let x^* be an approximation to x, we say that x^* approximates x to r significant $\beta - digits$ provided the absolute error $|x - x^*|$ is at most $\frac{1}{2}$ in the r^{th} significant $\beta - digits$ of x. This can be written as

$$|x - x^*| \leq \frac{1}{2}\beta^{s-r+1} \tag{3}$$

with s the largest integer such that $\beta^s \leq |x|$.
For instance, $x^* = 3$ agrees with $x = \pi$ to one significant (decimal) digit, while $x^* = \frac{22}{7} = 3.1428...$ is correct to three significant digits (as an approximation to π).

Errors in Numerical Computations

The loss of significant digits occurs due to the following two reasons:

- when two nearly equal numbers are subtracted and
- when division is made by a very small divisor compared to the dividend.

Significant error is more serious than round-off error, which are illustrated in the following examples:
Example 8. Consider the evaluation of

$$f(x) = x[\sqrt{x+1} - \sqrt{x}] \tag{4}$$

for an increasing sequence of values of x. The results of using 6-digit decimal calculator are shown in the next slide.

Errors in Numerical Computations

x	*Computed* $f(x)$	*True* $f(x)$
1	0.414210	0.414214
10	1.54340	1.54347
100	4.99000	4.98756
1000	15.8000	15.8074
10, 000	50.0000	49.9988
100, 000	100.000	158.113

As x increases, there are fewer digits of accuracy in the computed value f(x).

For $x = 100 : \sqrt{100} = \underbrace{10.0000}_{exact}$ $\& \sqrt{101} = \underbrace{10.04999}_{rounded}$

where $\sqrt{101}$ is correctly rounded to 6 significant digits of accuracy.

$$\sqrt{x+1} - \sqrt{x} = \sqrt{101} - \sqrt{100} = .04999000 \tag{5}$$

Errors in Numerical Computations

while the true value should be 0.0498756. The calculation (5) has a **loss-of-significance error**.

Three digits of accuracy in $\sqrt{x+1} = \sqrt{101}$ were canceled by subtraction of the corresponding digits in $\sqrt{x} = \sqrt{100}$.

The loss of accuracy was a by-product of the form of f(x) and the finite precision 6-digit decimal arithmetic being used.

Consider 4 as fraction with a denominator of 1, the simple way is to reformulate (rationalise the numerator) it and avoid the loss-of-significance error:

$$f(x) = x\frac{\sqrt{x+1}-\sqrt{x}}{1} \times \frac{\sqrt{x+1}+\sqrt{x}}{\sqrt{x+1}+\sqrt{x}} = \frac{x}{\sqrt{x+1}+\sqrt{x}} \qquad (6)$$

on a 6-digit decimal calculator (6) gives

$f(100) = 4.98756$

the correct answer to six digits.

Errors in Numerical Computations

Consider the function

$$f(x) = 1 - \cos x$$

in six-decimal-digit arithmetic. Since for x near zero, there will be loss of significant digits for x near zero if we calculate f(x) by first finding cos x and then subtracting the calculated value from 1. We cannot calculate cos x to more than six digits for x near zero. To compute the value of f(x) near zero to about six significant digits using six-digit, use an alternative formula for f(x), such as

$$f(x) = 1 - \cos x = \frac{1 - \cos^2 x}{1 + \cos x} = \frac{\sin^2 x}{1 + \cos x}$$

which can be evaluated quite accurately for small x; else, one could make use of the **Taylor expansion** for f(x).

Errors in Numerical Computations

Definition

Accuracy, is defined as the number of correct digits in some approximate quantity, and **precision**, is defined as the accuracy with which single operations with (or storage of) exact numbers are performed

Once an error is committed, it contaminates subsequent results. This **error propagation** through subsequent calculations is conveniently studied in terms of the two related concepts of **condition** and **instability**.
The word *condition* is used to describe the sensitivity of the function value f(x) to changes in the argument x and is usually measured by the maximum relative change in the function value f(x) caused by a unit relative change in the argument.
The larger the condition, the more *ill-conditioned* the function is said to be.

Errors in Numerical Computations

If all round off errors of an **algorithm** are harmless, then the algorithm is said to be *well behaved or well-conditioned* or *numerically stable*.
Instability describes the sensitivity of a numerical process for the calculation of f(x) from x to the inevitable rounding errors committed during its execution in finite precision arithmetic.

Errors in Numerical Computations

For a 32-bit word-length computer, we established previously that if x is any real number within the range of the computer, then

$$fl(x) = x(1+\epsilon), \qquad |\epsilon| \leq 2^{-24}, \quad \epsilon = \frac{fl(x)-x}{x}. \tag{7}$$

Now let the symbol ω denote any one of the arithmetic operations : $'+'$, $'-'$, $'\div'$, $'\times'$.
Suppose a 32-binary computer has been designed so that whenever two machine numbers x and y are to be combined arithmetically, the computer will produce $fl(x)$ instead of $x\omega y$. Note that $x\omega y$ is first correctly formed, then normalized, and finally rounded to become a machine number. So equation (7) can be rewritten as

$$fl(x\omega y) = (x\omega y)(1+\epsilon), \ |\epsilon| \leq 2^{-24}. \tag{8}$$

N.B: It is possible for x and y to be machine numbers and for $x\omega y$ to overflow or underflow.

Errors in Numerical Computations

Thus, the machine version of $x + y$, which is $fl(x + y)$, is the exact sum of a slightly perturbed x and a slightly perturbed y.
This is called **backward error analysis** and it attempts to determine what perturbation of the original data would cause the computer results to be the exact results for a perturbed problem.
In contrast, a **direct error analysis** attempts to determine how computed answers differ from exact answers based on the same data.
Example 9. If x, y, and z are machine numbers in a 32-binary computer, what upper bound can be given for the relative roundoff error in computing z(x + y)?
In the computer, the calculation of x + y will be done first. This arithmetic operation produces the machine number $fl(x + y)$, which differs from $x + y$ because of round off. By the principles established in (8), there is a ϵ_1:

Errors in Numerical Computations

$$fl(x) = x(1+\epsilon_1), \qquad |\epsilon_1| \leq 2^{-24}$$

Now z is already a machine number. When it multiplies the machine number $fl(x+y)$, the result is the machine number $fl[zfl(x+y)]$. This, too, differs from its exact counterpart, and we have, for some ϵ_2,

$fl[zfl(x+y)] = zfl(x+y)(1+\epsilon_2), \quad |\epsilon_2| \leq 2^{-24}$

Putting both of our equations together, we have

$$\begin{aligned} fl[zfl(x+y)] &= z(x+y)(1+\epsilon_1)(1+\epsilon_2) \\ &= z(x+y)(1+\epsilon_1+\epsilon_2+\epsilon_1\epsilon_2) \\ &\approx z(x+y)(1+\epsilon_1+\epsilon_2) \\ &= z(x+y)(1+\epsilon), \quad |\epsilon| \leq 2^{-23} \end{aligned}$$

Errors in Numerical Computations

In this calculation, $|\epsilon_1 \epsilon_2| \leq 2^{-48}$, and so we ignore it. Also, we put $\epsilon = \epsilon_1 + \epsilon_2$ and then reason that

$$\begin{aligned} |\epsilon| &= |\epsilon_1 + \epsilon_2| \\ &\leq |\epsilon_1| + |\epsilon_2| \\ &\leq 2^{-24} + 2^{-24} = 2^{-23} \end{aligned}$$

Historical Notes

In the 1991 Gulf War, a failure of the Patriot missile defense system as a result of a software conversion error resulted in the death of 28 American soldiers in a barracks in Dhahran, Saudi Arabia, because it failed to intercept an incoming Iraqi Scud missile. The system clock measured time in tenths of a second, but it was stored as a 24-bit floating-point number, resulting in rounding errors. Since the number 0.1 has an infinite binary expansion, the value in the 24-bit register was in error by $(1.1001100...)_2 \times 2^{-24} \approx 0.95 \times 10^{-7}$. The resulting time error was approximately thirty-four one-hundreds of a second after running for 100 hours.

Errors in Numerical Computations

In 1996, the Ariane 5 rocket (with a cargo valued at USD500 million) launched by the European Space Agency exploded 40 seconds after lift-off from Kourou, French Guiana. An investigation determined that the horizontal velocity required the conversion of a 64-bit floating-point number to a 16-bit signed integer. It failed because the number was larger than 32,767, which was the largest integer of this type that could be stored in memory.

Review of Calculus and Taylor series

Definition

Limit of a Function at a Number: Let f be a function defined on an open interval containing a, with the possible exception of a itself. Then the limit of *f(x)* as x approaches a is the number L, written

$$\lim_{x \to a} f(x) = L \tag{9}$$

if $\forall \epsilon > 0,\ \exists \delta > 0 \textit{ such that } |x - a| < \delta \Rightarrow |f(x) - L| < \epsilon$.
If there is no L with this property,then we say the limit of *f(x)* at a does not exist.

Review of Calculus and Taylor series

Definition

Continuity at a Number: Let f be a function defined on an open interval containing all values of x close to a. Then f is continuous at a if

$$\lim_{x \to a} f(x) = f(a) \tag{10}$$

Definition

The Intermediate Value Theorem: If f is a continuous function on a closed interval $[a, b]$ and M is any number between $f(a)$ and $f(b)$, inclusive, then there is at least one number c in $[a, b]$ such that $f(c) = M$.

Review of Calculus and Taylor series

Definition

The Derivative: The derivative of a function f with respect to x is the function f' defined by the rule

$$f'(x) =_{h\to 0}^{lim} \frac{f(x+h)-f(x)}{h} \tag{11}$$

The domain of f' consists of all values of x for which the limit exists.

Theorem

If f is differentiable at a, then f is continuous at a. But the converse is not true.

Review of Calculus and Taylor series

Theorem

Rolle's Theorem: *Let f be continuous on $[a, b]$ and differentiable on (a, b). If $f(a) = f(b)$, then there exists at least one number c in (a, b) such that $f'(c) = 0$.*

Theorem

The Mean Value Theorem: *Let f be continuous on $[a, b]$ and differentiable on (a, b). Then there exists at least one number c in (a, b) such that*

$$f'(c) = \frac{f(b) - f(a)}{b - a} \tag{12}$$

N.B: *Rolle's theorem is a special case of the Mean Value Theorem.*

Review of Calculus and Taylor series

Theorem

The Extreme Value Theorem: *If f is continuous on a closed interval $[a, b]$, then f attains an absolute maximum value $f(c)$ for some number c in $[a, b]$ and an absolute minimum value $f(d)$ for some number d in $[a, b]$.*

Definition

Suppose f is a function defined on $[a, b]$. The **Riemann integral** of f on $[a, b]$ is the limit , if it exists,

$$\int_a^b f(x)dx =_{max} \lim_{\Delta x_i \to 0} \sum_{i=1}^{n} f(z_i)\Delta x_i, \tag{13}$$

where $\Delta x_i = x_i - x_{i-1},\ z_i \in [x_{i-1}, x_i]$ is arbitrary, and $a = x_0 \leq x_1 \leq .... \leq x_n = b$ is a partition of $[a, b]$.

Review of Calculus and Taylor series

Theorem

The Mean Value Theorem for Integrals: *If f is continuous on $[a, b]$, then there exists a number c in $[a, b]$ such that*

$$f(c) = \frac{1}{b-a}\int_a^b f(x)dx \tag{14}$$

N.B: Define $C^n[a, b]$ to be the set of all functions f defined on $[a, b]$ such that f^n exists and is continuous everywhere in the interval (a, b).

Review of Calculus and Taylor series

Theorem

Taylor's Theorem with Lagrange Formula: *Suppose $f \in C^n[a,b]$ and f^{n+1} exits on (a,b). Then for any x and c in $[a,b]$, there exist a number ϵ, depending on x, between x and c such that*

$$f(x) = P_n(x) + E_n(x) \tag{15}$$

where

$$P_n(x) = \sum_{k=0}^{n} \frac{1}{k!} f^k(c)(x-c)^k \tag{16}$$

is called the n^{th} Taylor polynomial for f centered at c, and

Review of Calculus and Taylor series

$$E_n(x) = \frac{1}{(n+1)!} f^{n+1}(\epsilon)(x-c)^{n+1} \tag{17}$$

is called the remainder term, error term, or truncation error associated with $P_n(x)$. the infinite series by taking the limit of $P_n(x)$as $n \to \infty$ is called the Taylor series of $f(x)$.
N.B: *The Mean-Value Theorem is a special case of the Taylor's Theorem by taking $n = 0$.*

Theorem

Taylor's Theorem with Integral Form: *If $f \in C^{n+1}[a, b]$, then for any x and c in $[a, b]$,*

$$f(x) = \sum_{k=0}^{n} \frac{1}{k!} f^k(c)(x-c)^k + R_n(x) \tag{18}$$

where

Review of Calculus and Taylor series

$$R_n(x) = \frac{1}{n!}\int_c^x f^{n+1}(t)(x-t)^n dt \tag{19}$$

Example 10. The function $f(x) = e^x$ has the Taylor expansion

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

for some ϵ between 0 and x about c - 0.
The expansion of $f(x) = lnx$ about $c = 1$ is
$\ln x = (x-1) - \frac{(x-1)^2}{2} + ... - \frac{(-1)^n(x-1)^n}{n} + \frac{(-1)^n(x-1)^{n+1}\epsilon^{-(n+1)}}{n+1}$
where $0 < x < 2$, and ϵ is between 1 and x.

References

[1]. S.T Chapra and R. P. Candle: (2015) Numerical Methods For Engineers, Seventh Edition.
[2] N. R. Nassif and D. K. Fayyad: (2014) Introduction to Numerical Analysis and Scientific Computing.
[3]. Richard L. Burden, J. Douglas Faires. Numerical Analysis Ninth Edition, Youngstown State University 2011 Brooks Cole, Cengage Learning, www.cengage.com.
[4]. K.Atkinson ,W Han :(2004) Elementary Numerical Analysis Wiley India.
[5]. Madhumangal Pal. Numerical Analysis for Scientists and Engineers: Theory and C Programs
[6]. E.Kreyszig: Advanced Engineering mathematics. Copyright (2006) John Wiley & Sons, Inc.
[7]. C. Trenchea: Numerical Mathematical Analysis (2010). Dpt of Mathematics , University of Pittsburgh.
[8]. P. J. Olver: AIMS lecture notes on Numerical Analysis (1995).