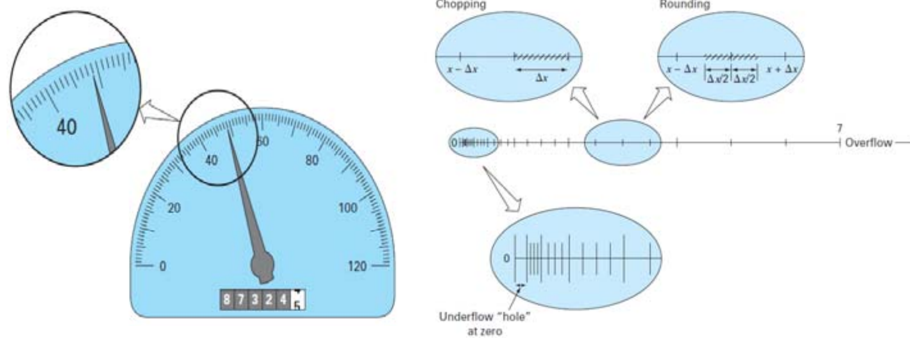
Lecture 03: Approximation and Round-off Errors



Outline

- Understanding the distinction between accuracy and precision.
- Learning how to quantify error.
- Learning how error estimates can be used to decide when to terminate an iterative calculation.
- Understanding how round-off errors occur because digital computers have a limited ability to represent numbers.
- Understanding why floating-point numbers have limits on their range and precision.

Bungee Jumper Problem

$$\frac{dv}{dt} \cong \frac{\Delta v}{\Delta t} = \frac{v(t_{i+1}) - v(t_i)}{t_{i+1} - t_i}$$

- The resulting solution is not exact, i.e., it has **error**.
- In addition, the computer you use to obtain the solution is also an imperfect tool.
- Because it is a digital device, the computer is limited in its ability to represent the magnitudes and precision of numbers.
- Consequently, the machine itself yields results that contain error.

Problems in Numerical Computation

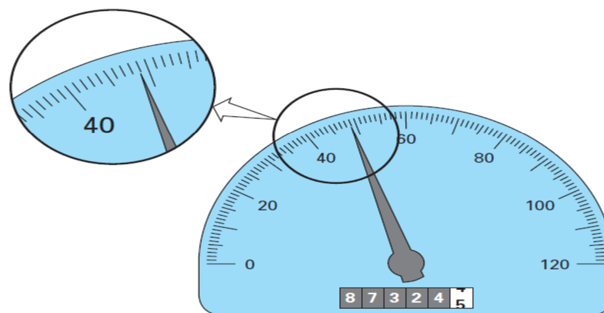
- Both your mathematical approximation and your digital computer cause your resulting model prediction to be uncertain.
- Your problem is: How do you deal with such uncertainty?
  - Is it possible to understand, quantify and control such errors in order to obtain acceptable results?

Errors

- Errors exist in numerical computation.
- Your task is to identify, quantify, and minimize these errors
- Two major forms of numerical error:
  - Round-off error → due to the fact that computers can represent only quantities with a finite number of digits.
  - Truncation error → the discrepancy introduced by the fact that numerical methods may employ approximations to represent exact mathematical operations and quantities.
- Errors not directly connected with the numerical methods themselves: blunders, formulation or model errors, and data uncertainty.

Significant Figures

- The significant digits of a number are those that can be used with confidence.
- They correspond to the number of certain digits plus one estimated digit.



Significant Figures

- Zeros are not always significant figures because they may be necessary just to locate a decimal point.
- The numbers 0.00001845, 0.0001845, and 0.001845 all have four significant figures.
- When trailing zeros are used in large numbers, it is not clear how many, if any, of the zeros are significant.
- For example, at face value the number 45,300 may have three, four, or five significant digits, depending on whether the zeros are known with confidence.
- Such uncertainty can be resolved by using scientific notation, where 4.53×10^4, 4.530×10^4, 4.5300×10^4 designate that the number is known to three, four, and five significant figures, respectively.

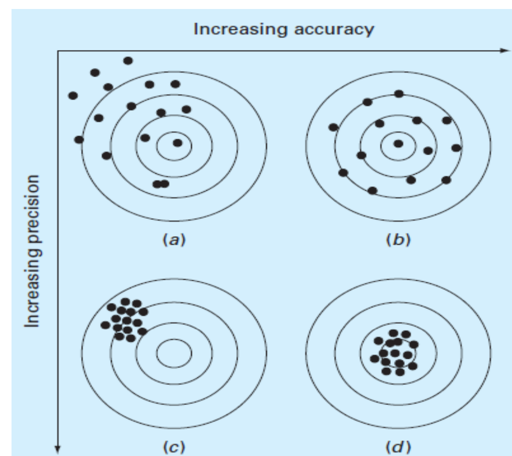
Significant Figures

- The concept of significant figures has two important implications for our study of numerical methods:
  1. We might decide that our approximation is acceptable if it is correct to certain significant figures.
  2. Although quantities such as *π, e, or* √7 represent specific quantities, they cannot be expressed exactly by a limited number of digits.
     - Because computers retain only a finite number of significant figures, such numbers can never be represented exactly. The omission of the remaining significant figures is called round-off error.

Accuracy and Precision

- The errors associated with both calculations and measurements can be characterized with regard to their accuracy and precision.
- **Accuracy** refers to how closely a computed or measured value agrees with the true value.
- **Precision** refers to how closely individual computed or measured values agree with each other.

Accuracy and Precision



Accuracy and Precision

- Inaccuracy (also called bias) is defined as systematic deviation from the truth.
- Imprecision (also called uncertainty) refers to the magnitude of the scatter.
- We will use the collective term error to represent both the inaccuracy and imprecision of our predictions.

Error Definitions

$$\text{True value} = \text{approximation} + \text{error}$$

$$E_t = \text{true value} - \text{approximation}$$

- Note that the true error is commonly expressed as an absolute value and referred to as the *absolute error.*
- A shortcoming of this definition is that it takes no account of the order of magnitude of the value under examination.

$$\text{True fractional relative error} = \frac{\text{true value} - \text{approximation}}{\text{true value}}$$

$$\varepsilon_t = \frac{\text{true value} - \text{approximation}}{\text{true value}} 100\%$$

Example 1

Problem Statement. Suppose that you have the task of measuring the lengths of a bridge and a rivet and come up with 9999 and 9 cm, respectively. If the true values are 10,000 and 10 cm, respectively, compute (*a*) the true error and (*b*) the true percent relative error for each case.

(a) The error for measuring the bridge is

$$E_t = 10{,}000 - 9999 = 1 \text{ cm}$$

and for the rivet it is

$$E_t = 10 - 9 = 1 \text{ cm}$$

(b) The percent relative error for the bridge is

$$\varepsilon_t = \frac{1}{10{,}000} 100\% = 0.01\%$$

and for the rivet it is

$$\varepsilon_t = \frac{1}{10} 100\% = 10\%$$

Error Definitions (Cont'd)

- In actual situations, true solution is rarely available.
- For numerical methods, the true value will only be known when we deal with functions that can be solved analytically.
- However, in real-world applications, we will obviously not know the true answer *a priori.*
- For these situations, an alternative is to normalize the error using the best available estimate of the true value.

$$\varepsilon_a = \frac{\text{approximate error}}{\text{approximation}} 100\%$$

$$\varepsilon_a = \frac{\text{current approximation} - \text{previous approximation}}{\text{current approximation}} 100\%$$

Error Definitions (Cont'd)

- Computation is repeated until $|\varepsilon_a| < \varepsilon_s$
- It is also convenient to relate these errors to the number of significant figures in the approximation.
- It can be shown (Scarborough, 1966) that if the following criterion is met, we can be assured that the result is correct to *at least n significant figures*.

$$\varepsilon_s = (0.5 \times 10^{2-n})\%$$

Example 2

Problem Statement. In mathematics, functions can often be represented by infinite series. For example, the exponential function can be computed using

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!} \qquad \text{(E3.2.1)}$$

Thus, as more terms are added in sequence, the approximation becomes a better and better estimate of the true value of e^x. Equation (E3.2.1) is called a *Maclaurin series expansion.*

Starting with the simplest version, $e^x = 1$, add terms one at a time to estimate $e^{0.5}$. After each new term is added, compute the true and approximate percent relative errors with Eqs. (3.3) and (3.5), respectively. Note that the true value is $e^{0.5} = 1.648721\ldots$. Add terms until the absolute value of the approximate error estimate ε_a falls below a prespecified error criterion ε_s conforming to three significant figures.

Example 2

$$\varepsilon_s = (0.5 \times 10^{2-3})\% = 0.05\%$$

The first estimate is simply equal to Eq. (E3.2.1) with a single term. Thus, the first estimate is equal to 1. The second estimate is then generated by adding the second term, as in

$$e^x = 1 + x$$

or for $x = 0.5$,

$$e^{0.5} = 1 + 0.5 = 1.5$$

$$\varepsilon_t = \frac{1.648721 - 1.5}{1.648721} 100\% = 9.02\%$$

$$\varepsilon_a = \frac{1.5 - 1}{1.5} 100\% = 33.3\%$$

Example 2

Terms	Result	ε_t (%)	ε_a (%)
1	1	39.3	
2	1.5	9.02	33.3
3	1.625	1.44	7.69
4	1.645833333	0.175	1.27
5	1.648437500	0.0172	0.158
6	1.648697917	0.00142	0.0158

Computer Algorithm for Iterative Calculations

$$e^x \cong \sum_{i=0}^{n} \frac{x^n}{n!}$$

```
FUNCTION IterMeth(val, es, maxit)
iter = 1
sol = val
ea = 100
DO
  solold = sol
  sol = ...
  iter = iter + 1
  IF sol ≠ 0 ea=abs((sol − solold)/sol)*100
  IF ea ≤ es OR iter ≥ maxit EXIT
END DO
IterMeth = sol
END IterMeth
```


Computer Algorithm for Iterative Calculations

$$e^x \cong \sum_{i=0}^{n} \frac{x^n}{n!}$$

```
function [fx,ea,iter] = IterMeth(x,es,maxit)
% Maclaurin series of exponential function
%   [fx,ea,iter] = IterMeth(x,es,maxit)
% input:
%   x = value at which series evaluated
%   es = stopping criterion (default = 0.0001)
%   maxit = maximum iterations (default = 50)
% output:
%   fx = estimated value
%   ea = approximate relative error (%)
%   iter = number of iterations

% defaults:
if nargin<2|isempty(es),es=0.0001;end
if nargin<3|isempty(maxit),maxit=50;end
% initialization
iter = 1; sol = 1; ea = 100;
% iterative calculation
while (1)
  solold = sol;
  sol = sol + x ^ iter / factorial(iter);
  iter = iter + 1;
  if sol~=0
    ea=abs((sol - solold)/sol)*100;
  end
  if ea<=es | iter>=maxit,break,end
end
fx = sol;
end
```

Round-off Errors

- Round-off errors arise because digital computers cannot represent some quantities exactly.
- They are important to engineering and scientific problem solving because they can lead to erroneous results.
- In certain cases, they can actually lead to a calculation going unstable and yielding obviously erroneous results.
- Such calculations are said to be ill-conditioned.
- Worse still, they can lead to subtler discrepancies that are difficult to detect.

Round-off Errors

- There are two major facets of round-off errors involved in numerical calculations:
  - Digital computers have magnitude and precision limits on their ability to represent numbers.
  - Certain numerical manipulations are highly sensitive to round-off errors.

Round-off Errors: Computer Number Representation

- The fundamental unit whereby information is represented is called a *word.*
- A word is an entity that consists of a string of *binary digits, or bits.*
- Numbers are typically stored in one or more words.

Computer Number Representation: A number system

- A number system is merely a convention for representing quantities.
- Because we have 10 fingers and 10 toes, the number system that we are most familiar with is the decimal, or base-10, number system.
- A base is the number used as the reference for constructing the system.
- The base-10 system uses the 10 digits - 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 - to represent numbers.

Computer Number Representation: A number system

- For larger quantities, combinations of these basic digits are used, with the position or place value specifying the magnitude.
- For example, if we have the number 8642.9, then we have eight groups of 1000, six groups of 100, four groups of 10, two groups of 1, and nine groups of 0.1, or

$$(8 \times 10^3) + (6 \times 10^2) + (4 \times 10^1) + (2 \times 10^0) + (9 \times 10^{-1}) = 8642.9$$

- This type of representation is called *positional notation.*

Computer Number Representation: A number system

- Computer is like a two-fingered animal who is limited to two states →either 0 or 1.
- This relates to the fact that the primary logic units of digital computers are on/off electronic components.
- Hence, numbers on the computer are represented with a binary, or base-2, system.
- For example, in the decimal system, the binary number 101.1 is equivalent to

$$(1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) + (1 \times 2^{-1}) = 4 + 0 + 1 + 0.5 = 5.5$$

Computer Number Representation: Integer Representation

- The most straightforward approach, called the signed magnitude method.
- It employs the first bit of a word to indicate the sign, with a 0 for positive and a 1 for negative.
- The remaining bits are used to store the number.
- For example, the integer value of 173 is represented in binary as 10101101:

$$(10101101)_2 = 2^7 + 2^5 + 2^3 + 2^2 + 2^0 = 128 + 32 + 8 + 4 + 1 = (173)_{10}$$

Computer Number Representation: Integer Representation

- The binary representation of the decimal integer –173 on a 16-bit computer using the signed magnitude method:

1	0	0	0	0	0	0	0	1	0	1	0	1	1	0	1

Sign (first bit); Magnitude (remaining bits)

Example 4

Problem Statement. Determine the range of integers in base-10 that can be represented on a 16-bit computer.

Solution. Of the 16 bits, the first bit holds the sign. The remaining 15 bits can hold binary numbers from 0 to 111111111111111. The upper limit can be converted to a decimal integer, as in

$$(1 \times 2^{14}) + (1 \times 2^{13}) + \cdots + (1 \times 2^{1}) + (1 \times 2^{0})$$

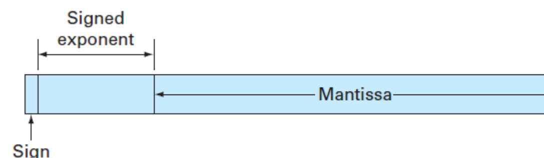
which equals 32,767 (note that this expression can be simply evaluated as $2^{15} - 1$). Thus, a 16-bit computer word can store decimal integers ranging from −32,767 to 32,767. In addition, because zero is already defined as 0000000000000000, it is redundant to use the number 1000000000000000 to define a "minus zero." Therefore, it is usually employed to represent an additional negative number: −32,768, and the range is from −32,768 to 32,767.

Computer Number Representation: Integer Representation

- For an *n*-bit word, the range would be from -2^{n-1} to $2^{n-1} - 1$.
- Thus, 32-bit integers would range from −2,147,483,648 to +2,147,483,647.
- The signed magnitude method is not actually used to represent integers for conventional computers.
- A preferred approach called the *2s complement technique* → *directly incorporates the sign into the* number's magnitude rather than providing a separate bit to represent plus or minus.

Computer Number Representation: Floating-Point Representation

- Fractional quantities are typically represented in computers using *floating-point format*.
- In this approach, the number is expressed as ***m.b^e^***.
- where *m* = the mantissa (or significand), *b* = the base of the number system being used, and *e* = the exponent.
- For instance, the number 156.78 could be represented as 0.15678×10^3 in a floating point base-10 system.



Computer Number Representation: Floating-Point Representation

- The mantissa is usually normalized if it has leading zero digits.
- For example, suppose the quantity 1/34 = 0.029411765 . . . was stored in a floating-point base-10 system that allowed only four decimal places to be stored.
- Thus, 1/34 would be stored as 0.0294×10^0.
- The number can be normalized to remove the leading zero by multiplying the mantissa by 10 and lowering the exponent by 1 to give 0.2941×10^{-1}.
- The consequence of normalization is that the absolute value of *m* is limited.

$$\frac{1}{b} \le m < 1$$

Computer Number Representation: Floating-Point Representation

- Floating-point representation allows both fractions and very large numbers to be expressed on the computer.
- However, it has some disadvantages:
  - Floating- point numbers take up more room and take longer to process than integer numbers.
  - More significantly, their use introduces a source of error because the mantissa holds only a finite number of significant figures. Thus, a round-off error is introduced.

Example 5

Problem Statement. Suppose that we had a hypothetical base-10 computer with a 5-digit word size. Assume that one digit is used for the sign, two for the exponent, and two for the mantissa. For simplicity, assume that one of the exponent digits is used for its sign, leaving a single digit for its magnitude.

Solution. A general representation of the number following normalization would be

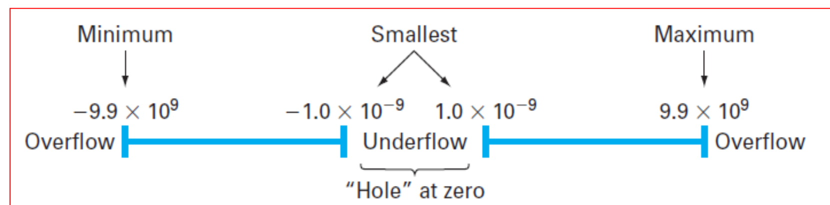
$$s_1 d_1.d_2 \times 10^{s_0 d_0}$$

where s_0 and s_1 = the signs, d_0 = the magnitude of the exponent, and d_1 and d_2 = the magnitude of the significand digits.

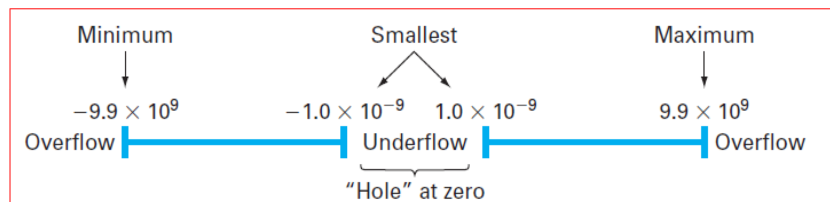
Example 5

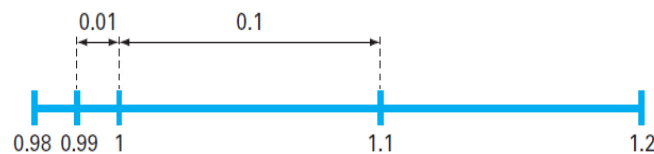
Largest value = $+9.9 \times 10^{+9}$

Smallest value = $+1.0 \times 10^{-9}$



Example 5



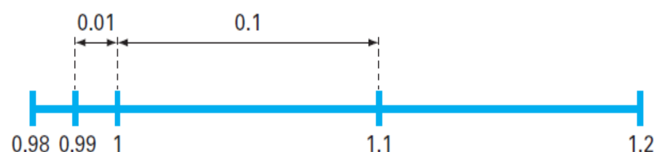


Computer Number Representation: Floating-Point Representation

- Large positive and negative numbers that fall outside the range would cause an overflow error.
- In a similar sense, for very small quantities there is a “hole” at zero, and very small quantities would usually be converted to zero.
- Recognize that the exponent overwhelmingly determines these range limitations.
- For example, if we increase the mantissa by one digit, the maximum value increases slightly to 9.99×10^{9}.
- In contrast, a one-digit increase in the exponent raises the maximum by 90 orders of magnitude to 9.9×10^{99}!

Computer Number Representation: Floating-Point Representation

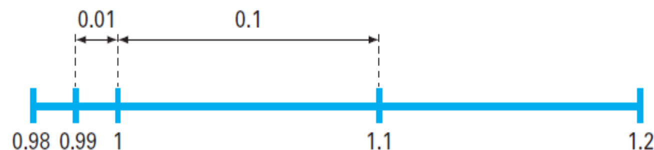
- When it comes to precision, however, the situation is reversed.
- Whereas the significand plays a minor role in defining the range, it has a profound effect on specifying the precision.
- There are “holes” at zero as well as between values.



Computer Number Representation: Floating-Point Representation

- For example, a simple rational number with a finite number of digits like $2^{-5} = 0.03125$ would have to be stored as 3.1×10^{-2} or 0.031.
- Relative error: $\frac{0.03125 - 0.031}{0.03125} = 0.008$
- π (= 3.14159….) would have to be represented as 3.1×10^{0} or 3.1.
- Relative error: $\frac{3.14159 - 3.1}{3.14159} = 0.0132$

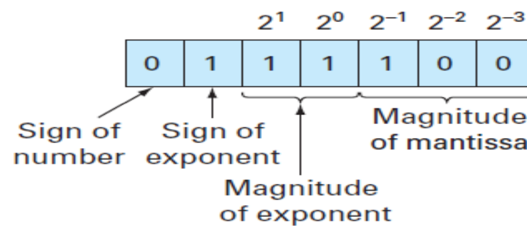
Computer Number Representation: Floating-Point Representation



- The interval between numbers increases as we move between orders of magnitude.
- This means that the round-off error of a number will be proportional to its magnitude.
- In addition, it means that the relative error will have an upper bound.
- For this example, the maximum relative error would be 0.05 → machine epsilon (or machine precision).

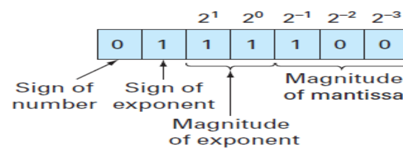
Example 6

Problem Statement. Create a hypothetical floating-point number set for a machine that stores information using 7-bit words. Employ the first bit for the sign of the number, the next three for the sign and the magnitude of the exponent, and the last three for the magnitude of the mantissa (Fig. 3.8).



Smallest possible positive floating-point number

Example 6



$1 \times 2^1 + 1 \times 2^0 = 3$

$1 \times 2^{-1} + 0 \times 2^{-2} + 0 \times 2^{-3} = 0.5$

$+0.5 \times 2^{-3}$

$$\frac{1}{b} \leq m < 1$$

$$0111101 = (1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}) \times 2^{-3} = (0.078125)_{10}$$
$$0111110 = (1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3}) \times 2^{-3} = (0.093750)_{10}$$
$$0111111 = (1 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}) \times 2^{-3} = (0.109375)_{10}$$

Example 6

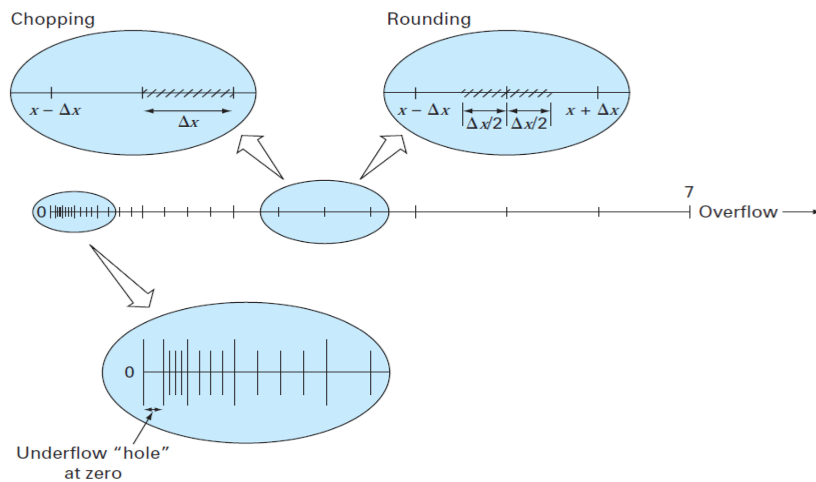
$$1 \times 2^1 + 0 \times 2^0 = 2$$

$$0110100 = (1 \times 2^{-1} + 0 \times 2^{-2} + 0 \times 2^{-3}) \times 2^{-2} = (0.125000)_{10}$$

$$0110101 = (1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}) \times 2^{-2} = (0.156250)_{10}$$
$$0110110 = (1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3}) \times 2^{-2} = (0.187500)_{10}$$
$$0110111 = (1 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}) \times 2^{-2} = (0.218750)_{10}$$

$$0011111 = (1 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}) \times 2^{3} = (7)_{10}$$

Example 6



Floating-point Representation

1. There is a limited range of quantities that may be represented.
2. There are only a finite number of quantities that can be represented within the range.
3. The interval between numbers, Δ*x*, Increases as the numbers grow in magnitude.

Finite Number of Quantities

- Both rational and irrational numbers cannot be represented exactly.
- The errors introduced by this approximation are referred to as quantizing errors.
- The actual approximation is accomplished in either of two ways: chopping or rounding.
- **Chopping** means that any quantity falling within an interval of length Δ*x will* be stored as the quantity at the lower end of the interval.
- Thus, the upper error bound for chopping is Δ*x*.

Finite Number of Quantities

- Additionally, a bias is introduced because all errors are positive.
- The shortcomings of chopping → the higher terms in the complete decimal representation have no impact on the shortened version.
- **Rounding** means that any quantity falling within an interval of length Δ*x will be represented as the nearest allowable number.*
- Thus, the upper error bound for rounding is Δ*x/2.*
- Additionally, no bias is introduced because some errors are positive and some are negative.

The Interval Δ*x* and Magnitude of Numbers

- Quantizing errors will be proportional to the magnitude of the number being represented.
- Chopping: $$\frac{|\Delta x|}{|x|} \leq \mathscr{E}$$
- Rounding: $$\frac{|\Delta x|}{|x|} \leq \frac{\mathscr{E}}{2}$$
- Where ε is referred to as the machine epsilon, which can be computed as

$$\mathscr{E} = b^{1-t}$$

Where *b* is the number base and *t* is the number of significant digits in the mantissa.

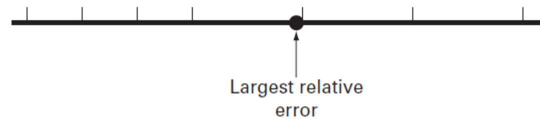
The Interval Δx and Magnitude of Numbers

- The magnitude dependence of quantizing errors has a number of practical applications in numerical methods.
- Most of these relate to the commonly employed operation of testing whether two numbers are equal.
- This occurs when testing convergence of quantities as well as in the stopping mechanism for iterative processes.
- For these cases, it should be clear that, rather than test whether the two quantities are equal, it is advisable to test whether their difference is less than an acceptably small tolerance.
- In addition, the machine epsilon can be employed in formulating stopping or convergence criteria.

Example 7

Problem Statement. Determine the machine epsilon and verify its effectiveness in characterizing the errors of the number system from Example 3.5. Assume that chopping is used.

$$\mathscr{E} = 2^{1-3} = 0.25$$



An example of a maximum error would be a value falling just below the upper bound of the interval between $(0.125000)_{10}$ and $(0.156250)_{10}$. For this case, the error would be less than

$$\frac{0.03125}{0.125000} = 0.25$$

Extended Precision

- The number of significant digits carried on most computers allows most engineering computations to be performed with more than acceptable precision.
- For example, computers that use IEEE format allow 24 bits to be used for the mantissa, which translates into about seven significant base-10 digits of precision with a range of about 10^{-38} to 10^{39}.
- Most computers also allow the specification of extended precision.
- The most common of these is double precision, in which the number of words used to store floating point numbers is doubled.
- It provides about 15 to 16 decimal digits of precision and a range of approximately 10^{-308} to 10^{308}.
- However, a price is paid for such remedies in that they also require more memory and execution time.

Arithmetic Manipulations of Computer Numbers

- Aside from the limitations of a computer's number system, the actual arithmetic manipulations involving these numbers can also result in round-off error.
- Common Arithmetic Operations: simple addition, subtraction, multiplication, and division.
- To simplify, we use a hypothetical decimal computer with a 4-digit mantissa and a 1-digit exponent.
- In addition, chopping is used.

Arithmetic Manipulations of Computer Numbers

- For example, suppose we want to add $0.1557 \cdot 10^1 + 0.4381 \cdot 10^{-1}$.

$$0.4381 \cdot 10^{-1} \rightarrow 0.004381 \cdot 10^1$$

$$\begin{array}{l} 0.1557 \quad \cdot 10^1 \\ \underline{0.004381 \cdot 10^1} \\ 0.160081 \cdot 10^1 \end{array}$$

- The result is chopped to $0.1600 \cdot 10^1$.

Arithmetic Manipulations of Computer Numbers

- For example, suppose that we are subtracting 26.86 from 36.41.

$$\begin{array}{r} 0.3641 \cdot 10^2 \\ \underline{-\ 0.2686 \cdot 10^2} \\ 0.0955 \cdot 10^2 \end{array}$$

- After normalization: $0.9550 \cdot 10^1 = 9.550$.
- Notice that the zero added to the end of the mantissa is not significant.

Arithmetic Manipulations of Computer Numbers

- Even more dramatic results would be obtained when the numbers are very close.

$$\begin{array}{r} 0.7642 \cdot 10^3 \\ -\, 0.7641 \cdot 10^3 \\ \hline 0.0001 \cdot 10^3 \end{array}$$

- After normalization: $0.1000 \cdot 10^0 = 0.1000$.
- Notice that three zeros added to the end of the mantissa are not significant.
- This introduces a substantial computational error because subsequent manipulations would act as if these zeros were significant.

Arithmetic Manipulations of Computer Numbers

- Multiplication and division:

$$0.1363 \cdot 10^3 \times 0.6423 \cdot 10^{-1} = 0.08754549 \cdot 10^2$$

- After normalization: $0.08754549 \cdot 10^2 \rightarrow 0.8754549 \cdot 10^1$
- After chopping: $0.8754 \cdot 10^1$

Large Computations

- Certain methods require extremely large numbers of arithmetic manipulations to arrive at their final results.
- In addition, these computations are often interdependent.
- Consequently, even though an individual round-off error could be small, the cumulative effect over the course of a large computation can be significant.

Example 8

- Sum a round base-10 number 10,000 times that is not round in base-2.

```
function sout = sumdemo()
s = 0;
for i = 1:10000
  s = s + 0.0001;
end
sout = s;
```

Adding a Large and a Small Number

- Suppose we add a small number, 0.0010, to a large number, 4000, using a hypothetical computer with the 4-digit mantissa and the 1-digit exponent.

$$\begin{array}{l} 0.4000 \quad\quad \cdot 10^4 \\ 0.0000001 \cdot 10^4 \\ \hline 0.4000001 \cdot 10^4 \end{array}$$

- After chopping: $0.4000 \cdot 10^4$.
- This type of error can occur in the computation of an infinite series.
- One way to mitigate this type of error is to sum the series in reverse order.

Subtractive Cancellation

- This term refers to the round-off induced when subtracting two nearly equal floating-point numbers.
- One common instance where this can occur involves finding the roots of a quadratic equation or parabola with the quadratic formula:

$$\begin{matrix} x_1 \\ x_2 \end{matrix} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- For cases where $b^2 >> 4ac$, the difference in the numerator can be very small.
- In such cases, double precision can mitigate the problem.
- In addition, an alternative formulation can be used to minimize subtractive cancellation:

$$\begin{matrix} x_1 \\ x_2 \end{matrix} = \frac{-2c}{b \pm \sqrt{b^2 - 4ac}}$$

Example 9

Problem Statement. Compute the values of the roots of a quadratic equation with $a = 1$, $b = 3000.001$, and $c = 3$. Check the computed values versus the true roots of $x_1 = -0.001$ and $x_2 = -3000$.

Smearing

- Smearing occurs whenever the individual terms in a summation are larger than the summation itself.
- For example, it occurs in series of mixed signs.

Example 10

Problem Statement. The exponential function $y = e^x$ is given by the infinite series

$$y = 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \cdots$$

Evaluate this function for $x = 10$ and $x = -10$, and be attentive to the problems of round-off error.

Inner Products

- Calculation of inner products:

$$\sum_{i=1}^{n} x_i y_i = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n$$

- Such summations are prone to round-off error.

Assignment-03

1. Investigate the effect of round-off error on large numbers of interdependent computations. Develop a MATLAB program to sum a number 100,000 times. Sum the number 1 in single precision, and 0.00001 in single and double precision.
2. Problems: 3.5, 3.6, 3.7, 3.11, 3.13.