

Accuracy and Reliability

Numerical Precision Is the Very Soul of Science.

*Sir D'Arcy Wentworth Thompson
(1860-1948)*

Fundamental Problems

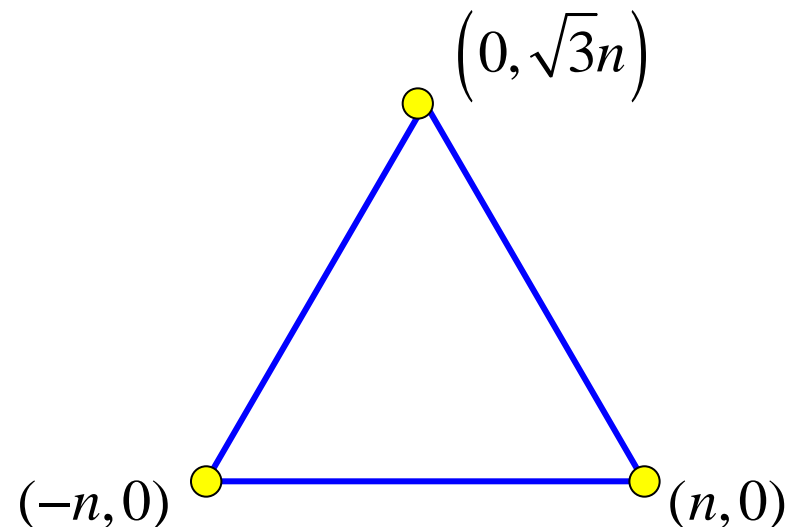
- **There are two fundamental problems in numerical computation:**
 - **The input values are not known exactly or cannot be represented exactly**
 - **Some of the calculations cannot be performed exactly.**
- **Therefore, errors obtained can propagate to later calculations, causing an error growth, which can be very large.**

Inexact Values

- **There are a number of issues with respect to the inexact value problem:**
 - **Input values are not exact**
 - **Finite precision storage**
 - **Number conversion is not exact**
 - **Overflow and underflow**
 - **Some arithmetic laws may not hold**
- **See the next few slides.**

Input Values are Not Exact

- Since computers can only store finite number of digits, some real numbers cannot be input to computer precisely (e.g., $\sqrt{2}$, $\sqrt{3}$, π , $1/3$, $1/7$, etc).
- The following triangle cannot be input precisely because of $\sqrt{3}$.



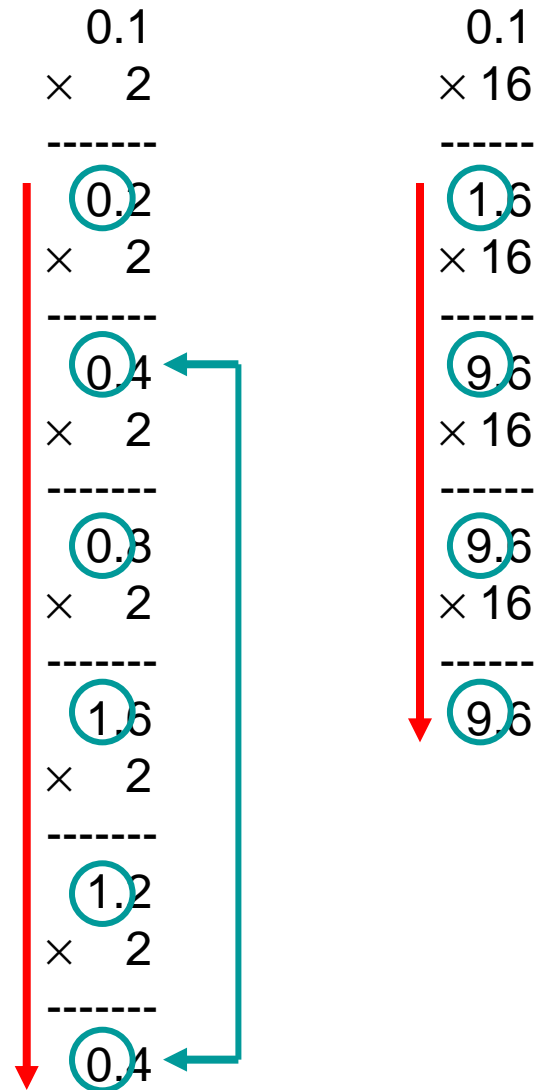
Finite Precision Storage

- Computers store a real number, in normal form $\pm 0.xx\dots x \times 10^{\pm yyy}$, in two parts: *fraction* and *exponent* as shown below.
- Both parts have only finite number of digits.
- Thus, numbers such as $\sqrt{2}$, $\sqrt{3}$, π , $1/3$ and $1/7$ cannot be stored precisely.



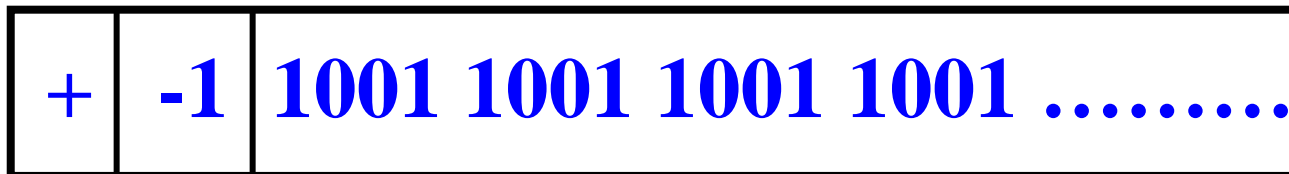
Number Conversion Is Not Exact

- We use decimal system; however, computers use binary or hexadecimal systems for floating-point number representations.
- Conversion from decimal to binary or to hexadecimal cannot be exact.
- For example, $0.1_{10} = 0.0001100110011\dots_2$ and $0.1_{10} = 0.199999\dots_{16}$.



Number Conversion Revisited: 1/2

- In computers, floating points are stored in a “normal” form as mentioned earlier. For example, $0.3_{10} = 0.01001\ 1001\ 10011001\dots_2$.
- In a normal form, the fraction part is converted to the range of $(-1,+1)$. Thus, we have $0.3_{10} = 0.01001\ 1001\ 1001\ 1001\dots_2 = 0.1001\ 1001\ 1001\ 1001\dots \times 2^{-1}$.



only a portion of this part can be stored

Number Conversion Revisited: 2/2

- Moreover, integral and fraction parts should be done separately. Thus, $277.31_{10} = 277_{10} + 0.31_{10}$.
- $277_{10} = 115_{16}$ and $0.31_{10} = 0.4 \text{ F5C28 F5C28 F5C28} \dots \dots \dots_{16}$.
- Hence, $277.31_{10} = 115.4 \text{ F5C28 F5C28 F5C28} \dots \dots \dots_{16} = 0.1154 \text{ F5C28 F6C28 F5C28} \dots \dots \dots \times 16^3$.



← only a portion of this part can be stored →

Overflow and Underflow: 1/2

- **Due to finite precision, floating-point number calculations can cause overflow or underflow.**
- **Overflow and underflow mean the exponent $\pm yyy$ of a number (in normal form) exceeds the limit of hardware's capability.**
- **For example, if the hardware allows the exponent (base 16) in $[-64,+64]$, then 0.1×16^{70} causes overflow and 0.1×16^{-70} causes underflow.**
- **Computer hardware does report floating-point number over- and under- flow; but, *usually integer over- and under- flows are ignored!***

Overflow and Underflow: 2/2

- One may use better formulas to avoid over- or under- flow.
- Suppose we wish to compute $c = \sqrt{a^2 + b^2}$. Even though the magnitude of c may be similar to the larger of a and b , $a^2 + b^2$ may cause overflow if one of a or b is very large.
- One way to overcome this is scaling the numbers down like the following, where $k = \text{MAX}(a,b)$:

$$c = k \times \sqrt{\left(\frac{a}{k}\right)^2 + \left(\frac{b}{k}\right)^2}$$

Note that one of a/k and b/k is 1. If $a > b$, $a/k = 1$ and $|b/k| \leq 1$, and the number in `SQRT()` is small!

Some Arithmetic Laws Do Not Hold

- The commutative law holds; but, the associative and distributive laws do not.
- Examples using *2-digit* calculation.

Associative Law Fails

$$\begin{aligned} (0.12 \times 0.34) \times 4 \\ &= 0.0408 \times 4 \\ &\rightarrow 0.041 \times 4 \\ &= 0.164 \\ &\rightarrow 0.16 \end{aligned}$$

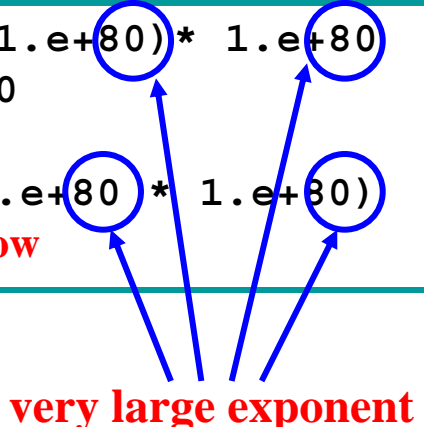
$$\begin{aligned} 0.12 \times (0.34 \times 4) \\ &= 0.12 \times 1.36 \\ &\rightarrow 0.12 \times 1.4 \\ &= 0.168 \\ &\rightarrow 0.17 \end{aligned}$$

Distributive Law Fails

$$\begin{aligned} (0.12 + 0.99) \times 5 \\ &= 1.11 \times 5 \\ &\rightarrow 1.1 \times 5 \\ &= 5.5 \end{aligned}$$

$$\begin{aligned} 0.12 \times 5 + 0.99 \times 5 \\ &= 0.6 + 4.95 \\ &\rightarrow 0.6 + 5.0 \\ &= 5.6 \end{aligned}$$

Associative Law Fails

$$\begin{aligned} (1.e-80 * 1.e+80) * 1.e+80 \\ &= 1.e+80 \\ 1.e-80 * (1.e+80 * 1.e+80) \\ &\rightarrow \text{overflow} \end{aligned}$$


very large exponent

A Serious Failure: 1/2

- In an iterative computation, the next term x_{i+1} is computed from x_i using the following five mathematically equivalent forms, where $x_0 = 0.5$ and $R = 3.0$:

$$x_{i+1} = ((R + 1)x_i - R(x_i x_i))$$

$$x_{i+1} = (R + 1)x_i - (Rx_i)x_i$$

$$x_{i+1} = ((R + 1) - Rx_i)x_i$$

$$x_{i+1} = Rx_i + (1 - Rx_i)x_i$$

$$x_{i+1} = x_i + R(x_i - x_i x_i)$$

A Serious Failure: 2/2

- If we compute the values of x_i up to the 1000th term and display the results every 100 terms, we have the following shocking results due to the failure of the distributive law:

0	0.5	0.5	0.5	0.5	0.5
100	1.6782e-05	1.30277	0.506627	1.14111	0.355481
200	1.28383	0.97182	1.25677	1.04532	0.805295
300	0.745989	0.155282	1.32845	0.295785	0.0590719
400	0.0744125	0.354702	1.00514	0.882786	0.171617
500	0.788592	0.00176679	0.804004	0.496569	1.32348
600	0.0190271	0.35197	0.832942	0.518201	1.20776
700	0.377182	0.525322	0.31786	0.435079	0.525364
800	0.1293	0.000277146	1.33159	0.0130949	0.954542
900	0.375104	0.00743761	1.27548	0.409476	0.00996984
1000	0.680855	1.03939	0.462035	0.0734742	1.26296

Three Commonly Seen Issues

- **The following issues can cause significant problems in numerical computation:**
 - *Rounding*
 - *Cancellation*
 - *Recursion*
- **The next few slides will provide several examples.**

Rounding: 1/4

- **Because of finite number of significant digits, computed results may be rounded or truncated due to hardware design.**
- **A rounding error in the optimal case is less than half of a unit in the last digit.**
- **This introduces an error that may propagate to other computations.**
- **The propagation of rounding errors is usually very complex and difficult to analyze.**

Rounding: 2/4

- The following goes from 1 to 2 with step size 1/3 (*i.e.*, 1, 1 1/3, 1 2/3, 2). However, it shows one more step (*i.e.*, 5 steps)!

```
PROGRAM Rounding
  IMPLICIT NONE
  INTEGER, PARAMETER :: DOUBLE = SELECTED_REAL_KIND(15)
  REAL(KIND=DOUBLE) :: a = 1.0_DOUBLE, b = 2.0_DOUBLE, x, h
  INTEGER :: n = 3

  h = (b - a)/n
  x = a
  WRITE(*,*) "x = ", x
  DO
    IF (x >= b) EXIT
    x = x + h
    WRITE(*,*) "x = ", x
  END DO
END PROGRAM Rounding
```

Output is

```
x = 1.0
x = 1.3333333333333332
x = 1.6666666666666665
x = 1.9999999999999997
x = 2.3333333333333333
```

rather than $1, 1\frac{1}{3}, 1\frac{2}{3}, 2$

$x < 2$, so goes for one more iteration

Rounding: 3/4

- If $b = 1.1$, we have exactly four iterations!

```
PROGRAM Rounding
  IMPLICIT NONE
  INTEGER, PARAMETER :: DOUBLE = SELECTED_REAL_KIND(15)
  REAL(KIND=DOUBLE) :: a = 1.0_DOUBLE, b = 1.1_DOUBLE, x, h
  INTEGER :: n = 3

  h = (b - a)/n
  x = a
  WRITE(*,*) "x = ", x
  DO
    IF (x >= b) EXIT
    x = x + h
    WRITE(*,*) "x = ", x
  END DO
END PROGRAM Rounding
```

Output is

```
x = 1.0
x = 1.033333333333333334
x = 1.066666666666666668
x = 1.100000000000000003
```

Now, it is correct!

Rounding: 4/4

- **Suggestions:**

- Use **INTEGER** to count the number of iterations. This is the reason that **REAL** for counting **DO**-loop was dropped in Fortran 90.

- If one must use **REAL** for a counting **DO**-loop, consider the use of

```
IF (x >= b-h/2) EXIT
```

- This would avoid the extra iteration.

Cancellation: 1/8

- Cancellation usually occurs from the subtraction of two almost equal numbers.
- Suppose after rounding we have two numbers $a = 1.243 \pm 0.0005$ (i.e., $a = 1.243 \in [1.2425, 1.2435]$) and $b = 1.234 \pm 0.0005$ (i.e., $b = 1.234 \in [1.2335, 1.2345]$).
- Then, $a - b = 0.009 \pm 0.001$ (i.e., $a - b = 0.009 \in [.008, 0.01]$). As a result, the possible range of $a - b$ is very large, which may not be trust worthy.
- Moreover, the original 4-significant digits reduce to 1-significant digit only.

Cancellation: 2/8

- Number representation contributes to cancellation.
- Below we expect the result to be 0.4111×10^{-12} ; but, the subtraction yields $4.110933 \dots \times 10^{-12}$. Only 3 to 4 digits can be trusted.

```
PROGRAM Cancellation
  IMPLICIT NONE
  INTEGER, PARAMETER :: DOUBLE = SELECTED_REAL_KIND(15)
  REAL(KIND=DOUBLE) :: x = 3.141592653589793 DOUBLE
  REAL(KIND=DOUBLE) :: y = 3.141592653585682 DOUBLE
  REAL(KIND=DOUBLE) :: z

  z = x - y
  WRITE(*,*) "x      = ", x
  WRITE(*,*) "y      = ", y
  WRITE(*,*) "x - y = ", z
END PROGRAM Cancellation
```

output

```
x      = 3.141592653589793
y      = 3.141592653585682
x - y = 4.11093381558203E-12
```

Cancellation: 3/8

- When subtracting two nearly equal values, we face **loss of significant digits**, *an old story*.
- Consider the solution to $ax^2 + bx + c = 0$ with the textbook formula, where $a = c = 1$ and $b = 20000$.

```
PROGRAM Eqn_2
  IMPLICIT NONE
  REAL :: a = 1.0, b = 20000.0, c = 1.0, d, r1, r2

  WRITE(*,*) "a      = ", a
  WRITE(*,*) "b      = ", b
  WRITE(*,*) "c      = ", c

  d = SQRT(b*b - 4.0*a*c)
  r1 = (-b + d)/(2.0*a)
  r2 = (-b - d)/(2.0*a)
  WRITE(*,*)
  WRITE(*,*) "d      = ", d
  WRITE(*,*) "root 1 = ", r1
  WRITE(*,*) "root 2 = ", r2
END PROGRAM Eqn_2
```

```
a      = 1.0
b      = 20000.0
c      = 1.0

d      = 20000.0
root 1 = 0.0E+0
root 2 = -20000.0
```

This solution is obviously incorrect. Why?

Cancellation: 4/8

- Since $a = 1$, $b = 20000$ and $c = 1$, $b^2 - 4ac = 400000000 - 4 \approx 400000000$ and $(b^2 - 4ac)^{1/2} \approx b$.
- As a result, $-b + (b^2 - 4ac)^{1/2}$ would have a cancellation issue.
- We can avoid the above cancellation to a large degree; but, due to the nature of the solution, cancellation in $b^2 - 4ac$ is unavoidable if b^2 and $4ac$ are very close.

Cancellation: 5/8

- We may take the following steps to avoid the subtraction, and hence cancellation:
 - If $b > 0$, use $-b - (b^2 - 4ac)^{1/2}$
 - If $b < 0$, use $-b + (b^2 - 4ac)^{1/2}$
- In this way, no subtraction is performed, and one root is computed in a more reliable way.
- Since the product of the roots is c/a in $ax^2 + bx + c = 0$, the other root can be computed with $(c/a)/r_1$, where r_1 is the root computed above.

Cancellation: 6/8

- Here is a possible program:

```
PROGRAM Eqn_2
  IMPLICIT NONE
  REAL :: a = 1.0, b = 20000.0, c = 1.0, d, r1, r2

  WRITE(*,*) "a      = ", a
  WRITE(*,*) "b      = ", b
  WRITE(*,*) "c      = ", c

  d = SQRT(b*b - 4.0*a*c)
  IF (b >= 0.0) THEN
    r1 = (-b - d)/(2.0*a)
  ELSE
    r1 = (-b + d)/(2.0*a)
  END IF
  r2 = (c/a)/r1
  WRITE(*,*)
  WRITE(*,*) "d      = ", d
  WRITE(*,*) "root 1 = ", r1
  WRITE(*,*) "root 2 = ", r2
END PROGRAM Eqn_2
```

```
a      = 1.0
b      = 20000.0
c      = 1.0

d      = 20000.0
root 1 = -20000.0
root 2 = -5.0E-5
```

Cancellation: 7/8

- A very useful formula to avoid cancellation is $a^2 - b^2 = (a-b)(a+b)$.
- For example, if $x \approx 0$, then $(x + 1)^{1/2} - 1$ will have cancellation because $(x+1)^{1/2} \approx 1$.
- The following eliminates the subtraction:

$$\sqrt{x+1} - 1 = (\sqrt{x+1} - 1) \times \frac{\sqrt{x+1} + 1}{\sqrt{x+1} + 1} = \frac{(\sqrt{x+1})^2 - 1}{\sqrt{x+1} + 1} = \frac{x}{\sqrt{x+1} + 1}$$

- If $x = 0.0000001$, on a 7-digit computer the original form yields 0 (why?) and the transformed form has 0.00000005!

Cancellation: 8/8

- Let us try one more example.
- Evaluating $(1-\cos(x))/\sin(x)$ when $x \approx 0$ would cause cancellation because $\cos(x) \approx 1$.
- Applying $a^2 - b^2 = (a-b)(a+b)$ yields the following:

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

- Since there is no subtraction when $x \approx 0$, there is no cancellation.
- *Always do your best to get rid of subtractions in your program when the operands can be very close!*

Recursion

- **Many numerical algorithms compute a new entity based on the previous ones with either an iterative or a recursive method.**
- **In both cases, errors can accumulate and eventually destroy the computation.**
- **We have seen this when discussing the failure of the associative and distributive laws.**

Examples (Variance): 1/3

- The sample variance of n values x_1, x_2, \dots, x_n is defined as follows, where m is the sample mean:

$$s^2 = \frac{1}{n-1} \sum_{i=1}^n (x_i - m)^2$$

- An alternative, theoretically equivalent, *faster one-pass* formula is:

$$s^2 = \frac{1}{n-1} \left[\sum_{i=1}^n x_i^2 - \frac{1}{n} \left(\sum_{i=1}^n x_i \right)^2 \right]$$

- Which one is *better*?

Examples (Variance): 2/3


- Consider $x_1 = 100000000$, $x_2 = 1000000001$ and $x_3 = 1000000002$. Since the mean value is $m = 1000000001$, variance is $s^2 = 1$.
- If you try these values with a statistical package, say Microsoft Excel, the result may be $s^2 = 0$!
- *The ghost of cancellation.* The system may use the one-pass formula. Since $\sum x_i^2$ and $(\sum x_i)^2 / n$ are similar in magnitude when the input values are large (don't forget rounding/truncation), the subtraction can yield a *zero*!

Examples (Variance): 3/3

- Use the two-pass formula if possible, because it always delivers good results unless n is very large.
- The one-pass formula, although more efficient than the two-pass one, can suffer from cancellation.
- It may also suffer from overflow! **Where?**
- This is an important lesson for you to rewrite and use formulas wisely and carefully.

Example (Average)

- Computing average may also have problems.
- The following shows the result of computing the average of 5.01 and 5.03 with a 3-digit computer.
- The result is outside of the input range, and is obviously wrong, although it is accurate! 😞
- Finite precision is not always friendly.


$$\begin{aligned} &(5.01 + 5.03) / 2 \\ &= 10.04 / 2 \\ &\rightarrow 10.0 / 2 \\ &= 5.0 \end{aligned}$$

Example (Sum): 1/3

- In fact, computing $x_1 + x_2 + \dots + x_n$ accurately and reliably is not easy.
- There was an intensive research in late 1960's and early 1970's.
- Here are some interesting and easy summation methods and comments without proofs.
- A typical summation procedure looks like this:

```
sum = 0.0
DO i = 1, n
    sum = sum + xi
END DO
```

Example (Sum): 2/3

- Sort the data values so that $|x_1| \leq |x_2| \leq |x_3| \leq \dots \leq |x_n|$ or $|x_1| \geq |x_2| \geq |x_3| \geq \dots \geq |x_n|$, and use this order for summation.
- *Pair-wise* summation looks like a form of binary tree. Adjacent values are summed until only one value remains. Thus, $x_1 + x_2 + x_3 + x_4 + x_5 + x_6$ is computed as $y_1 = x_1 + x_2$, $y_2 = x_3 + x_4$ and $y_3 = x_5 + x_6$, followed by $z_1 = y_1 + y_2$, followed by $s = z_1 + y_3$.
- *Insertion* summation: If data values are sorted in some other, the sum of the first two values is inserted into the list until only one value remains.

Example (Sum): 3/3

- **There are other more advanced and more complex summation algorithms.**
- **In general, if all values are non-negative, the increasing order with a simple **DO**-loop, perhaps using higher precision, is good enough.**
- **The pair-wise summation has a smaller error bound than that of the simple **DO**-loop.**

"Real" Problems: 1-1/2

- The Patriot missile is designed to operate a few hours at one location to avoid detection.
- The incoming missile's velocity is a floating-point (*i.e.*, **REAL**) number.
- The internal clock of the Patriot missile is an **INTEGER** with unit of 1/10 sec. This **INTEGER** value is multiplied by 0.1_{10} as a 24-bit binary value before its use. The precision error is 9.5×10^{-8} in the conversion factor. (*Do you recall this fact: $0.1_{10} = 0.0001100110011\dots_2$?*)

"Real" Problems: 1-2/2

- **The inaccuracy of the target's position is proportional to the product of the target velocity and the length of time the system has been running.**
- **With the system up and running for 100 hours and a velocity of 1676 meters per second, *an error of 573 meters is obtained.***
- **As a result, a Scud missile was not intercepted successfully and killed 28 soldiers on February 25, 1991, at Dhahran, Saudi Arabia.**

"Real" Problems: 2

- **Vancouver Stock Exchange updated its index after each transaction.**
- **The index, with 3 decimals, was updated and *truncated*.**
- **After 22 months, the index had fallen from the initial value **1000.000** to **524.881**; but, the correctly evaluated index was **1098.811**.**
- **This happened in 1982.**

"Real" Problems: 3

- On June, 1996, an unmanned Ariane 5 rocket launched by the European Space Agency exploded 40 seconds after lift-off from Kourou, French Guiana.
- The failure was caused by *the conversion of a 64-bit floating-point number to a 16-bit signed integer.*
- A 16-bit signed integer is less than or equal to 32767, and a 64 floating-point number may be larger than 32767!

"Real" Problems: 4

- **In September, 1997, a crew member of the USS Yorktown mistakenly entered a zero for a data value.**
- ***This caused a division by zero*, and the error cascaded and eventually shut down the ship's propulsion system.**
- **The ship was dead in the water for 2 hours and 45 minutes.**

Some Commonly Used Terms

- **Precision**: the number of digits used for arithmetic and I/O
- **Accuracy**: the absolute or relative error of an approximate quantity. **More on this later.**
- **Reliability**: how “often” (as a percentage) the computation fails, in the sense that the true error is larger than what is requested.
- **Robustness**: how “gracefully” the computation fails and its sensitivity to small changes in the problem.
- **Stability**: a method is **stable** if it guarantees as accurate a solution as the data warrants.

Absolute/Relative Error

- Let x and x^* be the computed quality and true value.
- Absolute error is $|x - x^*|$
- Relative error is $|x - x^*|/|x^*|$
- If the relative error is less than 1, $-\log_{10}(|x - x^*|/|x^*|)$ gives the number of significant decimal digits in the computed value.
- Normally, relative error is not very useful if x^* is close to zero. Thus, use relative error if $|x - x^*| \geq 1$. Otherwise, use absolute error.

Condition: 1/3

- The *condition* of a problem is the sensitivity of the problem to perturbations in the data.
- A Problem is *ill-conditioned* if small changes in the data can cause relatively large changes in the solution. Otherwise, the problem is *well-conditioned*.
- Note that condition is concerned with the *sensitivity of the problem* and is independent of the method being used to solve the problem.

Condition: 2/3

- The cubic equation $x^3 - 21x^2 + 120x - 100 = 0$ has roots $x_1 = 1, x_2 = x_3 = 10$.
- If the coefficient of x^3 is perturbed to become $0.99x^3 - 21x^2 + 120x - 100 = 0$, the roots are $x_1 = 1, x_2 \approx 11.17, x_3 \approx 9.041$.
- If the coefficient of x^3 is perturbed to become $1.01x^3 - 21x^2 + 120x - 100 = 0$, the roots are $x_1 = 1, x_2, x_3 \approx 9.896 \pm 1.044i$.
- So, roots x_2 and x_3 are *ill-conditioned*; but, we cannot deduce the condition of root x_1 .

Condition: 3/3

- The following system of linear equations has solutions $x_1 = x_2 = 1$

$$\begin{bmatrix} 99 & 98 \\ 100 & 99 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 197 \\ 199 \end{bmatrix}$$

- If the upper-left corner value is perturbed as follows, the solutions are $x_1=100, x_2=-99$.

$$\begin{bmatrix} 98.99 & 98 \\ 100 & 99 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 197 \\ 199 \end{bmatrix}$$

- The problem is ill-conditioned.

Food for Thought (1)

- The following compensated summation algorithm is due to Kahan (1965).
- There are other more advanced and better methods.

```
sum = 0.0           ! computed sum
c = 0.0            ! correction (i.e., a+b=sum+c)
DO i = 1, n
  temp = sum       ! save the sum
  y =  $x_i$  + c     ! compensate  $x_i$ 
  sum = temp + y   ! add the compensated to sum
  c = (temp - sum) + y ! compute the new correction
END DO
```

Why don't you write a program to give it a try!

Food for Thought (2): 1/2

- In computing the sample mean $(\sum_{i=1}^n x_i) / n$, it is possible that the summation could cause overflow.
- There are so-called “updating formulas” for computing the mean so that overflow and other problems can be avoided to a large degree.
- Let $m_k = (\sum_{i=1}^k x_i) / k$ (i.e., m_k is the mean of x_1, x_2, \dots, x_k).
- The following is a possible updating formula. Prove it yourself.

$$m_k = m_{k-1} + \frac{1}{k} (x_k - m_{k-1})$$

Food for Thought (2): 2/2

- There are updating formulas for computing the sample variance.
- Let S_k be the sample variance of x_1, x_2, \dots, x_k . Here is a possible updating formula:

$$S_k = S_{k-1} + (k-1)(x_k - m_{k-1}) \left(\frac{x_k - m_{k-1}}{k} \right)$$

- Prove the correctness yourself.
- These updating formulas for computing the sample mean and variance are more stable than the on-pass textbook formulas.

The End