Numerical analysis is concerned with how to solve a problem numerically, i.e., how to develop a sequence of numerical calculations to get a satisfactory answer.

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.

Computers use **binary arithmetic**, representing each number as a **binary number**: a finite sum of integer powers of 2.

Some numbers can be represented exactly, but others, such as $\frac{1}{10}, \frac{1}{100}, \frac{1}{1000}, \ldots$, cannot.
For example,

$$2.125 = 2^1 + 2^{-3}$$

has an exact representation in binary (base 2), but

$$3.1 \approx 2^1 + 2^0 + 2^{-4} + 2^{-5} + 2^{-8} + \cdots$$

does not.
And, of course, there are the transcendental numbers like π that have no finite representation in either decimal or binary number system.

Computers use 2 formats for numbers.

- **Fixed-point** numbers are used to store integers.
  Typically, each number is stored in a **computer word** of 32 binary digits (**bits**) with values of 0 and 1. $\Rightarrow$ at most 2^{32} different numbers can be stored.
  If we allow for negative numbers, we can represent integers in the range $-2^{-31} \leq x \leq 2^{31} - 1$, since there are 2^{32} such numbers.
  Since $2^{31} \approx 2.1 \times 10^9$, the range for fixed-point numbers is too limited for scientific computing. $=$
  - always get an integer answer.
  - the numbers that we can store are equally spaced.
  - **very** limited range of numbers.

  Therefore they are used mostly for indices and counters.
- An alternative to fixed-point, **floating-point** numbers approximate real numbers.
  - the numbers that we can store are **NOT** equally spaced.
  - wide range of variably-spaced numbers that can be representeed exactly.

Numbers must be stores and used for arithmetic operations. Storing:

1. integer format
2. floating-point format

Definition (decimal Floating-point representation)

Let consider $x \neq 0$ written in *decimal* system.
Then it can be written uniquely as

$$x = \sigma \cdot \overline{x} \cdot 10^e \tag{4.1}$$

where

- $\sigma = +1$ or -1 is the **sign**
- e is an integer, **the exponent**
- $1 \leq \overline{x} < 10$, **the significand** or **mantissa**

Example ($124.62 = (\underbrace{1.2462}_{\overline{x}}) \cdot 10^2$)

$\sigma = +1$, the exponent $e = 2$, the significand $\overline{x} = 1.2462$

The decimal floating-point representation of $x \in \mathbb{R}$ is given in (4.1), with limitations on the

1. number of digits in mantissa $\overline{x}$
2. size of e

Example

Suppose we limit

1. number of digits in $\overline{x}$ to 4
2. $-99 \leq e \leq 99$

We say that a computer with such a representation has a **four-digit decimal floating point arithmetic**.
This implies that we cannot store *accurately* more than the first four digits of a number; and even the fourth digit may be changed by rounding.

What is the next smallest number bigger than 1? What is the next smallest number bigger than 100? What are the errors and relative errors?

What is the smallest positive number?

Definition (Floating-point representation of a binary number x)

Let consider x written in *binary* format. Analogous to (4.1)

$$x = \sigma \cdot \overline{x} \cdot 2^e \tag{4.2}$$

where

- $\sigma = +1$ or -1 is the **sign**
- e is an integer, **the exponent**
- $\overline{x}$ is a binary fraction satisfying

$$(1)_2 \leq \overline{x} < (10)_2 \quad \text{(in decimal:} 1 \leq \overline{x} < 2)$$

For example, if

$$x = (11011.0111)_2$$

then $\sigma = +1, e = 4 = (100)_2$ and $\overline{x} = (1.10110111)_2$

The floating-point representation of a binary number x is given by (4.2) with a restriction on

1. number of digits in $\overline{x}$: the **precision** of the binary floating-point representation of x
2. size of e

The IEEE floating-point arithmetic standard is the format for floating point numbers used in almost all computers.

- the **IEEE single precision floating-point representation** of x has
  - a *precision of 24 binary digits*,
  - and the *exponent* e is limited by $-126 \leq e \leq 127$:

$$x = \sigma \cdot (1.a_1 a_2 \ldots a_{23}) \cdot 2^e$$

where, in binary

$$-(1111110)_2 \leq e \leq (1111111)_2$$

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

$$x = \sigma \cdot (1.a_1 a_2 \ldots a_{52}) \cdot 2^e$$

- the **IEEE single precision floating-point representation** of x has a precision of 24 binary digits,
  and the exponent e is limited by $-126 \leq e \leq 127$:

$$x = \sigma \cdot (1.a_1 a_2 \ldots a_{23}) \cdot 2^e$$

  stored on **4 bytes (32 bits)**

$$\underbrace{b_1}_{\sigma} \quad \underbrace{b_2 b_3 \cdots b_9}_{E=e+127} \quad \underbrace{b_{10} b_{11} \cdots b_{32}}_{\overline{x}}$$

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

$$x = \sigma \cdot (1.a_1 a_2 \ldots a_{52}) \cdot 2^e$$

  stored on **8 bytes (64 bits)**

$$\underbrace{b_1}_{\sigma} \quad \underbrace{b_2 b_3 \cdots b_{12}}_{E=e+1023} \quad \underbrace{b_{13} b_{14} \cdots b_{64}}_{\overline{x}}$$

Epsilon machine

How accurate can a number be stored in the floating point representation? How can this be measured?

(1) Machine epsilon

Machine epsilon

For any format, the machine epsilon is the difference between 1 and the next larger number that can be stored in that format.

In single precision IEEE, the next larger binary number is

$$1.00000000000000000000000\underbrace{1}_{a_{23}}$$

($1 + 2^{-24}$ cannot be stored exactly)

Then **the machine epsilon in single precision IEEE format** is

$$2^{-23} \doteq 1.19 \times 10^{-7}$$

i.e., we can store approximately 7 decimal digits of a number x in decimal format.

Then **the machine epsilon in double precision IEEE format** is

$$2^{-52} \doteq 2.22 \times 10^{-16}$$

IEEE double-precision format can be used to store approximately 16 decimal digits of a number x in decimal format.
MATLAB: machine epsilon is available as the constant eps.

Another way to measure the accuracy of floating-point format:

(2) look for the largest integer M such that any integer x such that $0 \leq x \leq M$ can be stored and represented exactly in floating point form.

If n is the number of binary digits in the significand $\overline{x}$, all integers less or equal to

$$(1.11\ldots1)_2 \cdot 2^{n-1} = \left(1 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + \ldots + 1 \cdot 2^{-(n-1)}\right) \cdot 2^{n-1}$$
$$= \left(\frac{1 - \frac{1}{2^n}}{1 - \frac{1}{2}}\right) \cdot 2^{n-1} = 2^n - 1$$

can be represented exactly.

- In IEEE single precision format
$$M = 2^{24} = 16777216$$
and all 7-digit decimal integers will store exactly.
- In IEEE double precision format
$$M = 2^{53} = 9.0 \times 10^{15}$$
and all 15-digit decimal integers and most 16 digit ones will store exactly.

Let say that a number x has a significand

$$\overline{x} = 1.a_1a_2 \ldots a_{n-1}a_n a_{n+1}$$

but the floating-point representation may contain only n binary digits. Then x must be shortened when stored.

Definition

We denote the **machine floating-point version** of x by $fl(x)$.

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

It can be shown that

$$fl(x) = x \cdot (1 + \epsilon) \tag{4.3}$$

where ϵ is a small number depending of x.

(a) If chopping is used

$$-2^{-n+1} \leq \epsilon \leq 0 \tag{4.4}$$

(b) If rounding is used

$$-2^{-n} \leq \epsilon \leq 2^{-n} \tag{4.5}$$

Characteristics of chopping:

1. the worst possible error is twice as large as when rounding is used
2. the sign of the error $x - fl(x)$ is the same as the sign of x

The worst of the two: no possibility of cancellation of errors.

Characteristics of rounding:

1. the worst possible error is only half as large as when chopping is used
2. More important: the error $x - fl(x)$ is negative for only half the cases, which leads to better error propagation behavior

For single precision IEEE floating-point rounding arithmetic (there are $n = 24$ digits in the significand):

1. chopping ("rounding towards zero"):

$$-2^{-23} \leq \epsilon \leq 0 \tag{4.6}$$

2. standard rounding:

$$-2^{-24} \leq \epsilon \leq 2^{-24} \tag{4.7}$$

For double precision IEEE floating-point rounding arithmetic:

1. chopping:

$$-2^{-52} \leq \epsilon \leq 0 \tag{4.8}$$

2. rounding:

$$-2^{-53} \leq \epsilon \leq 2^{-53} \tag{4.9}$$

Numbers that have finite decimal expressions may have infinite binary expansions. For example

$$(0.1)_{10} = (0.0001100110011001100110011\ldots)_2$$

Hence $(0.1)_{10}$ cannot be represented exactly in binary floating-point arithmetic.
Possible problems:

- Run into infinite loops
- 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

Definition

- The **error** in a computed quantity is defined as

$$\text{Error}(x_A) = x_T \text{ - } x_A$$

where x_T =true value, x_A=approximate value. This is called also **absolute error**.

- The **relative error** $\text{Rel}(x_A)$ is a measure off error related to the size of the true value

$$\text{Rel}(x_A) = \frac{\text{error}}{\text{true value}} = \frac{x_T - x_A}{x_T}$$

For example, for the approximation

$$\pi \doteq \frac{22}{7}$$

we have $x_T = \pi = 3.14159265\ldots$ and $x_A = \frac{22}{7} = 3.1428571$

$$\text{Error}\left(\frac{22}{7}\right) = \pi - \frac{22}{7} \doteq -0.00126$$

$$\text{Rel}\left(\frac{22}{7}\right) = \frac{\pi - \frac{22}{7}}{\pi} \doteq -0.00042$$

The notion of **relative error** is a more intrinsic error measure.

1. The exact distance between 2 cities: $x_T^1 = 100\text{km}$ and the measured distance is $x_A^1 = 99\text{km}$

$$\text{Error}\left(x_T^1\right) = x_T^1 - x_A^1 = \underline{1km}$$

$$\text{Rel}\left(x_T^1\right) = \frac{\text{Error}\left(x_T^1\right)}{x_T^1} = 0.01 = 1\%$$

2. The exact distance between 2 cities: $x_T^2 = 2\text{km}$ and the measured distance is $x_A^2 = 1\text{km}$

$$\text{Error}\left(x_T^2\right) = x_T^2 - x_A^2 = \underline{1km}$$

$$\text{Rel}\left(x_T^2\right) = \frac{\text{Error}\left(x_T^2\right)}{x_T^2} = 0.5 = 50\%$$

Definition (significant digits)

The number of **significant digits** in x_A is number of its leading digits that are correct relative to the corresponding digits in the true value x_T

More precisely, if x_A, x_T are written in decimal form; compute the error

$$\begin{array}{rcccccccc} x_T & = a_1 & a_2 & .a_3 & \cdots & a_m & a_{m+1} & a_{m+2} \\ |x_T - x_A| & = 0 & 0 & .0 & \cdots & 0 & b_{m+1} & b_{m+2} \end{array}$$

If the error is ≤ 5 units in the $(m+1)^{th}$ digit of x_T, counting rightward from the first nonzero digit, then we say that x_A has, at least, m significant digits of accuracy relative to x_T.

Example

1. $x_A = 0.222, x_T = \frac{2}{9}$:

$$\begin{array}{rcccccccc} x_T & = 0. & 2 & 2 & 2 & 2 & 2 & 2 \\ |x_T - x_A| & = 0. & 0 & 0 & 0 & 2 & 2 & 2 \end{array} \Rightarrow 3$$

Example

1. $x_A = 23.496, x_T = 23.494$:

$$\begin{aligned} x_T &= 2 \quad 3. \quad 4 \quad 9 \quad 6 \\ |x_T - x_A| &= 0 \quad 0. \quad 0 \quad 0 \quad 1 \quad 9 \quad 9 \end{aligned} \Rightarrow 4$$

2. $x_A = 0.02138, x_T = 0.02144$:

$$\begin{aligned} x_T &= 0. \quad 0 \quad 2 \quad 1 \quad 4 \quad 4 \\ |x_T - x_A| &= 0. \quad 0 \quad 0 \quad 0 \quad 0 \quad 6 \end{aligned} \Rightarrow 2$$

3. $x_A = \frac{22}{7} = 3.1428571\ldots, x_T = \pi = 3.14159265\ldots$:

$$\begin{aligned} x_T &= 3. \quad 1 \quad 4 \quad 1 \quad 5 \quad 9 \quad 2 \quad 6 \quad 5 \\ |x_T - x_A| &= 0. \quad 0 \quad 0 \quad 1 \quad 2 \quad 6 \quad 4 \quad 4 \quad 8 \end{aligned} \Rightarrow 3$$

Errors in a scientific-mathematical-computational problem:

1. Original errors
   - (E1) Modeling errors
   - (E2) Blunders and mistakes
   - (E3) Physical measurement errors
   - (E4) Machine representation and arithmetic errors
   - (E5) Mathematical approximation errors
2. Consequences of errors
   - (F1) Loss-of-significance errors
   - (F2) Noise in function evaluation
   - (F3) Underflow and overflow erros

(E1) Modeling errors: the mathematical equations are used to represent physical reality - mathematical model.

- Malthusian growth model (it can be accurate for some stages of growth of a population, with unlimited resources)

$$N(t) = N_0 e^{kt}, \quad N_0, k \geq 0$$

where $N(t) =$ population at time t. For large t the model overestimates the actual population:

  - accurately models the growth of US population for $1790 \leq t \leq 1860$, with $k = 0.02975$, $N_0 = 3,929,000 \times e^{-1790k}$
  - but considerably overestimates the actual population for 1870

(E2) *Blunders and mistakes:* mostly programming errors

- test by using cases where you know the solution
- break into small subprograms that can be tested separately

(E2) *Physical measurement errors.* For example, the speed of light in vacuum

$$c = (2.997925 + \epsilon) \cdot 10^{10} cm/sec, \quad |\epsilon| \leq 0.000003$$

Due to the error in data, the calculations will contain the effect of this observational error. Numerical analysis cannot remove the error in the data, but it can

- look at its propagated effect in a calculation and
- suggest the best form for a calculation that will minimize the propagated effect of errors in data

(E4) *Machine representation and arithmetics errors.* For example errors from rounding and chopping.

- they are inevitable when using floating-point arithmetic
- they form the main source of errors with some problems (solving systems of linear equations). We will look at the effect of rounding errors for some summation procedures

(E4) *Mathematical approximation errors*: major forms of error that we will look at.

For example, when evaluating the integral

$$I = \int_0^1 e^{-x^2} dx,$$

since there is no antiderivative for e^{-x^2}, I cannot be evaluated explicitly. Instead, we approximate it with a quantity that can be computed.

Using the Taylor approximation

$$e^{-x^2} \approx 1 - x^2 + \frac{x^4}{2!} - \frac{x^6}{3!} + \frac{x^8}{4!}$$

we can easily evaluate

$$I \approx \int_0^1 \left(1 - x^2 + \frac{x^4}{2!} - \frac{x^6}{3!} + \frac{x^8}{4!}\right) dx$$

with the **truncation error** being evaluated by (3.10)

Loss of significant digits

Example

Consider the evaluation of

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

for an increasing sequence of values of x.

x_0	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

Table: results of using a 6-digit decimal calculator

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

What happens?
For $x = 100$:

$$\sqrt{100} = \underbrace{10.0000}_{\text{exact}}, \quad \sqrt{101} = \underbrace{10.04999}_{\text{rounded}}$$

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

$$\sqrt{x+1} - \sqrt{x} = \sqrt{101} - \sqrt{100} = 0.0499000$$

while the true value should be 0.0498756.
The calculation 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

For this particular f, there is a simple way to reformulate it and avoid the loss-of-significance error:

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

which on a 6 digit decimal calculator will imply

$$f(100) = 4.98756$$

the correct answer to six digits.

Example

Consider the evaluation of

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

for a sequence approaching 0.

x_0	Computed $f(x)$	True f(x)
0.1	0.4995834700	0.4995834722
0.01	0.4999960000	0.4999958333
0.001	0.5000000000	0.4999999583
0.0001	0.5000000000	0.4999999996
0.00001	0.0	0.5000000000

Table: results of using a 10-digit decimal calculator

Look at the calculation when $x = 0.01$:

$$\cos(0.01) = 0.9999500004 \quad (= 0.999950000416665)$$

has *nine* significant digits of accuracy, being off in the tenth digit by two units. Next

$$1 - \cos(0.01) = 0.0000499996 \quad (= 4.999958333495869e - 05)$$

which has only *five* significant digits, with four digits being lost in the subtraction.

To avoid the loss of significant digits, due to the subtraction of nearly equal quantities, we use the Taylor approximation (3.7) for $\cos(x)$ about $x = 0$:

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + R_6(x)$$

$$R_6(x) = \frac{x^8}{8!}\cos(\xi), \qquad \xi \text{ unknown number between } 0 \text{ and } x.$$

Hence

$$f(x) = \frac{1}{x^2}\left\{1 - \left[1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + R_6(x)\right]\right\}$$
$$= \frac{1}{2!} - \frac{x^2}{4!} + \frac{x^4}{6!} - \frac{x^6}{8!}\cos(\xi)$$

giving $f(0) = \frac{1}{2}$. For $|x| \leq 0.1$

$$\left|\frac{x^6}{8!}\cos(\xi)\right| \leq \frac{10^{-6}}{8!} \doteq 2.5 \cdot 10^{-11}$$

Therefore, with this accuracy

$$f(x) \approx \frac{1}{2!} - \frac{x^2}{4!} + \frac{x^4}{6!}, \quad |x| < 0.1$$

Remark

When two nearly equal quantities are subtracted, leading significant digits will be lost.

In the previous two examples, this was easy to recognize and we found ways to avoid the loss of significance.
More often, the loss of significance is subtle and difficult to detect, as in calculating sums (for example, in approximating a function $f(x)$ by a Taylor polynomial). If the value of the sum is relatively small compared to the terms being summed, then there are probably some significant digits of accuracy being lost in the summation process.

Example

Consider using the Taylor series approximation for e^x to evaluate e^{-5}:

$$e^{-5} = 1 + \frac{(-5)}{1!} + \frac{(-5)^2}{2!} + \frac{(-5)^3}{3!} + \frac{(-5)^4}{4!} + \dots$$

Degree	Term	Sum	Degree	Term	Sum
0	1.000	1.000	13	-0.1960	-0.04230
1	-5.000	-4.000	14	0.7001E-1	0.02771
2	12.50	8.500	15	-0.2334E-1	0.004370
3	-20.83	-12.33	16	0.7293E-2	0.01166
4	26.04	13.71	17	-0.2145E-2	0.009518
5	-26.04	-12.33	18	0.5958E-3	0.01011
6	21.70	9.370	19	-0.1568E-3	0.009957
7	-15.50	-6.130	20	0.3920E-4	0.009996
8	9.688	3.558	21	-0.9333E-5	0.009987
9	-5.382	-1.824	22	0.2121E-5	0.009989
10	2.691	0.8670	23	-0.4611 E-6	0.009989
11	-1.223	-0.3560	24	0.9607 E-7	0.009989
12	0.5097	0.1537	25	-0.1921 E-7	0.009989

Table. Calculation of $e^{-5} = 0.006738$ using four-digit decimal arithmetic

There are loss-of-significance errors in the calculation of the sum. To avoid the loss of significance is simple in this case:

$$e^{-5} = \frac{1}{e^5} = \frac{1}{\text{series for } e^5}$$

and form e^5 with a series not involving cancellation of positive and negative terms.

Consider evaluating a continuous function f for all $x \in [a, b]$. The graph is a continuous curve.
When evaluating f on a computer using floating-point arithmetic (with rounding or chopping), the errors from arithmetic operations cause the graph to cease being a continuous curve.
Let look at

$$f(x) = (x-1)^3 = -1 + x(3 + x(-3 + x))$$

on $[0, 2]$.

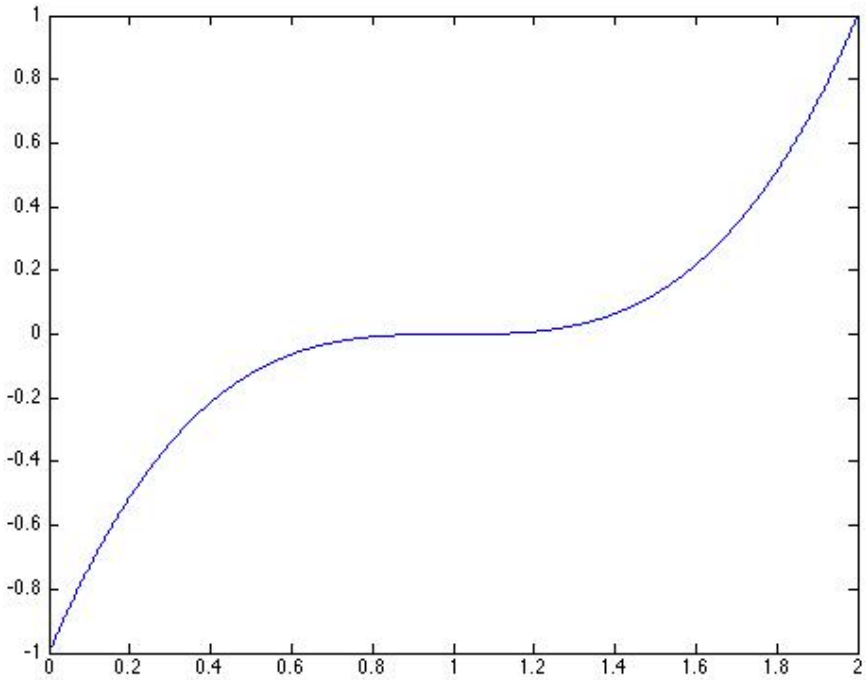

Figure: $f(x) = x^3 - 3x^2 + 3x - 1$

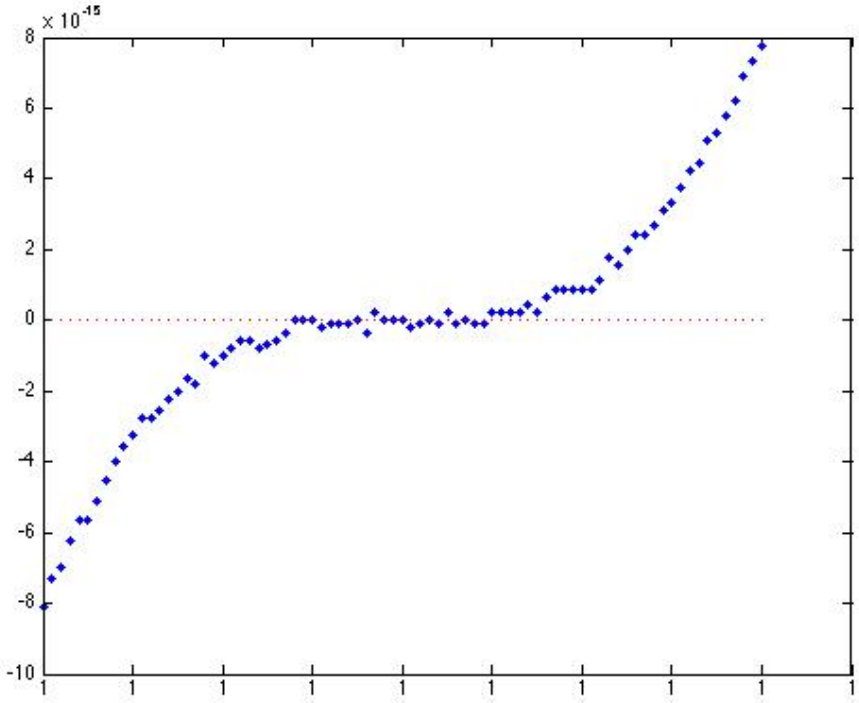

Figure: Detailed graph of $f(x) = x^3 - 3x^2 + 3x - 1$ near $x = 1$

Here is a plot of the computed values of $f(x)$ for 81 evenly spaced values of $x \in [0.99998, 1.00002]$.
A rootfinding program might consider $f(x)$ t have a very large number of solutions near 1 based on the many sign changes!!!

From the definition of floating-point number, there are **upper** and **lower** limits for the magnitudes of the numbers that can be expressed in a floating-point form.
Attempts to create numbers

- that are too small $\Rightarrow$ **underflow** errors: the default option is to set the number to zero and proceed
- that are too large $\Rightarrow$ **overflow** errors: generally fatal errors on most computers. With the IEEE floating-point format, overflow errors can be carried along as having a value of $\pm\infty$ or NaN, depending on the context. Usually, an overflow error is an indication of a more significant problem or error in the program.

Example (underflow errors)

Consider evaluating $f(x) = x^{10}$ for x near 0.

With the IEEE single precision arithmetic, the smallest nonzero positive number expressible in *normalized* floating point arithmetic is

$$m = 2^{-126} \doteq 1.18 \times 10^{-38}$$

So $f(x)$ is set to zero if

$$\begin{aligned} x^{10} &< m \\ |x| &< \sqrt[10]{m} \doteq 1.61 \times 10^{-4} \\ -0.000161 &< x < 0.000161 \end{aligned}$$

Sometimes is possible to eliminate the overflow error by just reformulating the expression being evaluated.

Example (overflow errors)

Consider evaluating $z = \sqrt{x^2 + y^2}$.

If x or y is very large, then $x^2 + y^2$ might create an overflow error, even though z might be within the floating-point range of the machine.

$$z = \begin{cases} |x|\sqrt{1 + \left(\frac{y}{x}\right)^2}, & 0 \leq |y| \leq |x| \\ |y|\sqrt{1 + \left(\frac{x}{y}\right)^2}, & 0 \leq |x| \leq |y| \end{cases}$$

In both cases, the argument of $\sqrt{1 + w^2}$ has $|w| \leq 1$, which will not cause any overflow error (except when z is too large to be expressed in the floating-point format used).

When doing calculations with numbers that contain an error, the result will be effected by these errors.
If x_A, y_A denote numbers used in a calculation, corresponding to x_T, y_T, we wish to bound the **propagated error**

$$E = (x_T \omega y_T) - (x_A \omega y_A)$$

where ω denotes: "$+$", "$-$", "$\cdot$", "$\div$".
The first technique used to bound E is **interval arithmetic**.
Suppose we know bounds on $x_T - x_A$ and $y_T - y_A$. Using these bound and $x_A \omega y_A$, we look for an interval guaranteed to contain $x_T \omega y_T$.

Example

Let $x_A = 3.14$ and $y_A = 2.651$ be correctly rounded from x_T and y_T, to the number of digits shown. Then

$$\begin{aligned} |x_A - x_T| \le 0.005, \qquad & |y_A - y_T| \le 0.0005 \\ 3.135 \le x_T \le 3.145 \quad & 2.6505 \le y_T \le 2.6515 \end{aligned} \tag{4.10}$$

- For addition (ω : "$+$")

$$x_A + y_A = 5.791 \tag{4.11}$$

The true value, from (4.10)

$$5.7855 = 3.135 + 2.6505 \le x_T + y_T \le 3.145 + 2.6515 = 5.7965 \tag{4.12}$$

To bound E, subtract (4.11) from (4.12) to get

$$-0.0055 \le (x_T + y_T) - (x_A + y_A) \le 0.0055$$

Example

- For division (ω : "$\div$")

$$\frac{x_A}{y_A} = \frac{3.14}{2.651} \doteq 1.184459 \tag{4.13}$$

The true value, from (4.10),

$$\frac{3.135}{2.6515} \leq \frac{x_T}{y_T} \leq \frac{3.145}{2.6505}$$

dividing and rounding to 7 digits:

$$1.182350 \leq \frac{x_T}{y_T} \leq 1.186569$$

The bound on E

$$-0.002109 \leq \frac{x_T}{y_T} - 1.184459 \leq 0.002110$$

Propagated error in multiplication

The relative error in $x_A y_A$ compared to $x_T y_T$ is

$$Rel(x_A y_A) = \frac{x_T y_T - x_A y_A}{x_T y_T}$$

If $x_T = x_A + \epsilon$, $y_T = y_A + \eta$, then

$$\begin{aligned} Rel(x_A y_A) = \frac{x_T y_T - x_A y_A}{x_T y_T} &= \frac{x_T y_T - (x_T - \epsilon)(y_T - \eta)}{x_T y_T} \\ &= \frac{\eta x_T + \epsilon y_T - \epsilon\eta}{x_T y_T} = \frac{\epsilon}{x_T} + \frac{\eta}{y_T} - \frac{\epsilon}{x_T}\frac{\eta}{y_T} \\ &= Rel(x_A) + Rel(y_A) - Rel(x_A)Rel(y_A) \end{aligned}$$

If $Rel(x_A) \ll 1$ and $Rel(y_A) \ll 1$, then

$$Rel(x_A y_A) \approx Rel(x_A) + Rel(y_A)$$

Propagated error in division

The relative error in $\frac{x_A}{y_A}$ compared to $\frac{x_T}{y_T}$ is

$$Rel\left(\frac{x_A}{y_A}\right) = \frac{\frac{x_T}{y_T} - \frac{x_A}{y_A}}{\frac{x_T}{y_T}}$$

If $x_T = x_A + \epsilon$, $y_T = y_A + \eta$, then

$$Rel\left(\frac{x_A}{y_A}\right) = \frac{\frac{x_T}{y_T} - \frac{x_A}{y_A}}{\frac{x_T}{y_T}} = \frac{x_T y_A - x_A y_T}{x_T y_A} = \frac{x_T(y_T - \eta) - (x_T - \epsilon)y_T}{x_T(y_T - \eta)}$$

$$= \frac{-x_T\eta + \epsilon y_T}{x_T(y_T - \eta)} = \frac{\frac{\epsilon}{x_T} - \frac{\eta}{y_T}}{1 - \frac{\eta}{x_T}}$$

$$= \frac{Rel(x_A) - Rel(y_A)}{1 - Rel(y_A)}$$

If $Rel(y_A) \ll 1$, then

$$Rel\left(\frac{x_A}{y_A}\right) \approx Rel(x_A) - Rel(y_A)$$

Propagated error in addition and subtraction

$$(x_T \pm y_T) - (x_A \pm y_A) = (x_T - x_A) \pm (y_T - y_A) = \epsilon \pm \eta$$

$$Error(x_A \pm y_A) = Error(x_A) \pm Error(y_A)$$

Misleading: we can have **much larger** $Rel(x_A \pm y_A)$ for small values of $Rel(x_A)$ and $Rel(y_A)$ (this source of error is connected closely to loss-of-significance errors).

Example

1. $x_T = x_A = 13$
   $y_T = \sqrt{168}, y_A = 12.961$
   $Rel(x_A) = 0, Rel(y_A) \doteq 0.0000371$
   $Error(x_A - y_A) \doteq -0.0004814$
   $Rel(x_A - y_A) \doteq -0.0125$
2. $x_T = \pi x_A = 3.1416$
   $y_T = \frac{22}{7}, y_A = 3.1429$
   $x_T - x_A \doteq -7.35 \times 10^{-6} \quad Rel(x_A) \doteq -2.34 \times 10^{-6},$
   $(y_T - y_A) \doteq -4.29 \times 10^{-5} \quad Rel(y_A) \doteq -1.36 \times 10^{-5}$
   $(x_T - y_T) - (x_A - y_A) \doteq -0.0012645 - (-.0013) \doteq 3.55 \times 10^{-5}$
   $Rel(x_A - y_A) \doteq -0.28$

Total calculation error

With floating-point arithmetic on a computer, $x_A \omega y_A$ involves an additional rounding or chopping error, as in (4.13). Hence the total error in computing $x_A \hat{\omega} y_A$ (the complete operation in computer, involving the propagated error plus the rounding or chopping error):

$$x_T \omega y_T - x_A \hat{\omega} y_A = \underbrace{(x_T \omega y_T - x_A \omega y_A)}_{\text{propagated error}} + \underbrace{(x_A \omega y_A - x_A \hat{\omega} y_A)}_{\text{error in computing } x_A \omega y_A}$$

When using IEEE arithmetic with the basic arithmetic operations

$$x_A \hat{\omega} y_A = fl(x_A \omega y_A) \overset{(4.3)}{=} (1+\epsilon)(x_A \omega y_A)$$

where ϵ as in (4.4)-(4.5), we get

$$\begin{aligned} x_A \omega y_A - x_A \hat{\omega} y_A &= -\epsilon(x_A \omega y_A) \\ \frac{x_A \omega y_A - x_A \hat{\omega} y_A}{x_A \omega y_A} &= -\epsilon \end{aligned}$$

Hence the process of rounding or chopping introduces a relatively small new error into $x_A \hat{\omega} y_A$ as compared with $x_A \omega y_A$.

Evaluate $f(x) \in C^1[a,b]$ at the approximate value x_A instead of x_T. Using the mean value theorem

$$\begin{aligned} f(x_T)-f(x_A) &= f'(c)(x_T-x_A), \quad c \text{ between } x_T \text{ and } x_A \\ &\approx f'(x_T)(x_T - x_A) \approx f'(x_A)(x_T - x_A) \end{aligned} \tag{4.14}$$

and

$$Rel(f(x_A)) \approx \frac{f'(x_T)}{f(x_T)}(x_T - x_A) = \frac{f'(x_T)}{f(x_T)} x_T Rel(x_A) \tag{4.15}$$

Example: ideal gas law

$$PV = nRT$$

with R a constant for all gases:

$$R = 8.3143 + \epsilon, \qquad |\epsilon| \leq 0.0012$$

Let evaluate T assuming $P = V = n = 1 : T = \frac{1}{R}$,
i.e., evaluate $f(x) = \frac{1}{x}$ at $x = R$. Let

$$x_T = R, \quad x_A = 8.3143, \quad |x_T - x_A| \leq 0.0012$$

For the error

$$E = \frac{1}{R} - \frac{1}{8.3143}$$

we have

$$|E| = |f(x_T) - f(x_A)| \overset{(4.14)}{\approx} |f'(x_A)||x_T - x_A| \leq \left(\frac{1}{x_A^2}\right) 0.0012 \doteq 0.000144$$

Hence , the uncertainty ϵ in $R \Rightarrow$ relatively small error in the computed

$$\frac{1}{R} \doteq \frac{1}{8.3143}$$

Example: evaluate b^x, $b > 0$

Since $f'(x) = (\ln b)b^x$, by (4.15)

$$b^{x_T} - b^{x_A} \approx (\ln b)b^{x_T}(x_T - x_A)$$
$$Rel(b^{x_A}) \approx \underbrace{(\ln b)x_T}_{K=\text{conditioning number}} Rel(x_A)$$

If the conditioning number $K \gg 1$ is large in size $\Rightarrow$

$$Rel(b^{x_A}) \gg Rel(x_A)$$

For example, if

$$Rel(x_A) = 10^{-7}, \quad K = 10^4$$

then

$$Rel(b^{x_A}) \doteq 10^{-3}$$

independent of how b^{x_A} is actually computed.

Let denote the sum

$$S = a_1 + a_2 + \cdots + a_n = \sum_{j=1}^{n} a_j,$$

where each a_j is a floating-point number. It takes $(n-1)$ additions, each of which will probably involve a rounding or chopping error:

$$\begin{aligned}
S_2 &= fl(a_1 + a_2) \overset{(4.3)}{=} (a_1 + a_2)(1 + \epsilon_2) \\
S_3 &= fl(a_3 + S_2) \overset{(4.3)}{=} (a_3 + S_2)(1 + \epsilon_3) \\
S_4 &= fl(a_4 + S_3) \overset{(4.3)}{=} (a_4 + S_3)(1 + \epsilon_4) \\
&\vdots \\
S_n &= fl(a_n + S_{n-1}) \overset{(4.3)}{=} (a_n + S_{n-1})(1 + \epsilon_n)
\end{aligned}$$

with S_n the computed version of S.
Each ϵ_j satisfies the bounds (4.6)- (4.9), assuming the IEEE arithmetic is used.

$$\begin{aligned}
S - S_n \approx -a_1(\epsilon_2 + \ldots + \epsilon_n) - a_2(\epsilon_2 + \ldots + \epsilon_n) - a_3(\epsilon_3 + \ldots + \epsilon_n) \\
- a_4(\epsilon_4 + \ldots + \epsilon_n) - \cdots - a_n\epsilon_n
\end{aligned} \tag{4.16}$$

Trying to minimize the error $S - S_N$:

1. before summing, arrange the terms $a_1, a_2, \ldots, a_n$ so they are **increasing** in size

$$|a_1| \leq |a_2| \leq |a_3| \leq \cdots |a_n|$$

2. then summate

Then the terms in (4.16) with the largest numbers of ϵ_j's are multiplied by the smaller values among a_j's.

Example

$\frac{1}{j} \rightsquigarrow$ decimal fraction $\rightsquigarrow$ round it to 4 significant digits $:= a_j$

Use a decimal machine with 4 significant digits.
SL = adding S from smallest to largest,
LS = adding S from largest to smallest.

n	True	SL	Error	LS	Error
10	2.929	2.929	0.001	2.927	0.002
25	3.816	3.813	0.003	3.806	0.010
50	4.499	4.491	0.008	4.479	0.020
100	5.187	5.170	0.017	5.142	0.045
200	5.878	5.841	0.037	5.786	0.092
500	6.793	6.692	0.101	6.569	0.224
1000	7.486	7.284	0.202	7.069	0.417

Table: Calculating S on a machine using chopping

n	True	SL	Error	LS	Error
10	2.929	2.929	0	2.929	0
25	3.816	3.816	0	3.817	-0.001
50	4.499	4.500	-0.001	4.498	0.001
100	5.187	5.187	0	5.187	0
200	5.878	5.878	0	5.876	0.002
500	6.793	6.794	0.001	6.783	0.010
1000	7.486	7.486	0	7.449	0.037

Table: Calculating S on a machine using rounding

A more important difference in the errors in the previous *Tables*: between rounding and chopping.

Rounding $\Rightarrow$ a far **smaller** error in the calculated sum than does chopping.

Let go back to the first term in (4.16):

$$T \equiv -a_1(\epsilon_1 + \epsilon_2 + \ldots + \epsilon_n) \tag{4.17}$$

(1) Assuming that the previous example used rounding with four-digit decimal machine, we know that all ϵ_j satisfy

$$-0.005 \leq \epsilon_j \leq 0.005 \tag{4.18}$$

Treating rounding errors as being random, then the positive and negative values of the ϵ_j's will tend to cancel and the sum T will be nearly zero. Moreover, from probability theory:

$$|T| \leq 1.49 \cdot 0.0005 \cdot \sqrt{n}|a_1|$$

hence T is proportional to $\sqrt{n}$ and is small until n becomes quite large. Similarly for the total error in (4.16).

(2) For chopping with the four-digit decimal machine, (4.18) is replaced by

$$-0.001 \leq \epsilon_j \leq 0$$

and all errors have one sign.

Again, the errors will be random in this interval, with the average -0.0005 and (4.17) will likely be

$$-a_1 \cdot (n-1) \cdot (-0.0005)$$

hence T is proportional to n, which increases more rapidly than $\sqrt{n}$. Thus the error (4.17) and (4.16) will grow more rapidly when chopping is used rather than rounding.

Example (difference between rounding and chopping on summation)

Consider

$$S = \sum_{j=1}^{n} \frac{1}{j}$$

in single precision accuracy of six decimal digits.

Errors occur both in calculation of $\frac{1}{j}$ and in the summation process.

n	True	Rounding error	Chopping Error
10	2.92896825	-1.76E-7	3.01E-7
50	4.49920534	7.00E-7	3.56E-6
100	5.18737752	-4.12E-7	6.26E-6
500	6.79282343	-1.32E-6	3.59E-5
1000	7.48547086	8.88E-8	7.35E-5

Table: Calculation of S: rounding versus chopping

The true values were calculated using double precision arithmetic to evaluate S; all sums were performed from the smallest to the largest.

Suppose we wish to calculate:

$$x = a + jh, \tag{4.19}$$

for $j = 0, 1, 2, \ldots, n$, for given $h > 0$, which is used later to evaluate a function $f(x)$.

Question

Should we compute x using (4.19) or by using

$$x = x + h \tag{4.20}$$

in the loop, having initially set $x = a$ before beginning the loop?

Remark: These are mathematically equivalent ways to compute x, but they are usually **not** computationally equivalent.
The difficulty arises when h does not have a finite binary expansion that can be stored in the given floating-point significand; for example

$$h = 0.1$$

The computation (4.19)

$$x = a + jh$$

will involve two arithmetic operations, hence only two chopping or rounding errors, for each value of x.
In contrast, the repeated use of (4.20)

$$x = x + h$$

will involve a succession of j additions, for the x of (4.19). As x increases in size, the use of (4.20) involves a larger number of rounding or chopping errors, leading to a different quantity than in (4.19).

Usually (4.19) is the preferred way to evaluate x

Example

Compute x in the two ways, with $a = 0$, $h = 0.1$ and verify with the true value computed using a double precision computation.

j	x	Error using (4.19)	Error using (4.20)
10	1	1.49E-8	-1.04 E-7
20	2	2.98E-8	-2.09E-7
30	3	4.47E-8	7.60 E-7
40	4	5.96E-8	1.73 E-6
50	5	7.45E-8	2.46 E-6
60	6	8.94E-8	3.43 E-6
70	7	1.04E-7	4.40 E-6
80	8	1.19E-7	5.36 E-6
90	9	1.34E-7	2.04 E-6
100	10	1.49E-7	-1.76 E-6

Table: Evaluation of $x = j \cdot h, h = 0.1$

Definition (inner product)

A sum of the form

$$S = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n = \sum_{j=1}^{n} a_j b_j \tag{4.21}$$

is called a **dot product** or **inner product**.

(1) If we calculate S in single precision:
$\Rightarrow$ a single precision rounding error for each multiplication and each addition
$\Rightarrow 2n-1$ single precision rounding errors to calculate S.
The consequences of these errors can be analyzed as for (4.16) and derive an optimal strategy of calculating (4.21).

(2) Using double precision:

1. convert each a_j and b_j to double precision by extending their significands with zeros
2. multiply in double precision
3. sum in double precision
4. round to single precision to obtain the calculated value of S

For machines with IEEE arithmetic, this is a simple and rapid procedure to obtain more accurate inner products in single precision.

There is no increase in storage space for the arrays $A = [a_1, a_2, \ldots, a_n]$ and B.

The accuracy is improved since there is only **one** single precision rounding error, regardless of the size n.