Ch. 02
Arithmetic Precision

Andrea Mignone
Physics Department, University of Torino
AA 2022-2023

Float and Double precision datatype

- *Singles* or *floats* is shorthand for *single- precision floating-point numbers and* occupy 32 bits: 1 bit for the sign, 8 bits for the exponent, and 23 bits for the fractional mantissa:

	s	e		f	
Bit position	31	30	23	22	0

```
EXAMPLE: IEEE-754 Single-Precision representation of: 3.141590

 0 1 0 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 0 1 1 1 1 1 1 0 1 0 0 0 0
|- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -|
|s|      exp      |                  mantissa                   |
```

- The sign bit s is in bit position 31, the biased exponent e is in bits 30–23, and the fractional part of the mantissa f is in bits 22–0. Since 8 bits are used to store the exponent e and since $2^8 = 256 \rightarrow 0 \leq e \leq 255$.
- Likewise $-126 \leq e \leq 127$.
- In summary, single-precision (32-bit or 4-byte) numbers have six or seven decimal places of significance and magnitudes in the range

$$1.4 \times 10^{-45} \leq \text{single precision} \leq 3.4 \times 10^{38}$$

Float and Double precision datatype

- Doubles are stored as two 32-bit words, for a total of 64 bits (8 B). The sign occupies 1 bit, the exponent e, 11 bits, and the fractional mantissa, 52 bits:

	s	e		f		f (cont.)	
Bit position	63	62	52	51	32	31	0

- The fields are stored contiguously, with part of the mantissa f stored in separate 32-bit words.
- Doubles have approximately 16 decimal places of precision (1 part in 252) and magnitudes in the range

$$4.9 \times 10^{-324} \leq \text{double precision} \leq 1.8 \times 10^{308}.$$

C and C++ Data-Type Range

In 1987, the Institute of Electrical and Electronics Engineers (IEEE) and the American National Standards Institute (ANSI) adopted the IEEE 754 standard for floating-point arithmetic. When the standard is followed, you can expect the primitive data types to have the precision and ranges given by the following table

Key word	Size in bytes	Interpretation	Possible values
bool	1	boolean	true and false
unsigned char	1	Unsigned character	0 to 255
char (or signed char)	1	Signed character	-128 to 127
wchar_t	2	Wide character (in windows, same as unsigned short)	0 to 2^{16}-1
short (or signed short)	2	Signed integer	-2^{15} to 2^{15} - 1
unsigned short	2	Unsigned short integer	0 to 2^{16}-1
int (or signed int)	4	Signed integer	-2^{31} to 2^{31} -1
unsigned int	4	Unsigned integer	0 to 2^{32} - 1
Long (or long int or signed long)	4	signed long integer	-2^{31} to 2^{31} -1
unsigned long	4	unsigned long integer	0 to 2^{32} -1
float	4	Signed single precision floating point (23 bits of significand, 8 bits of exponent, and 1 sign bit.)	$3.4*10^{-38}$ to $3.4*10^{38}$ (both positive and negative)
long long	8	Signed long long integer	-2^{63} to 2^{63} -1
unsigned long long	8	Unsigned long long integer	0 to 2^{64} -1
double	8	Signed double precision floating point(52 bits of significand, 11 bits of exponent, and 1 sign bit.)	$1.7*10^{-308}$ to $1.7*10^{308}$ (both positive and negative)
long double	8	Signed double precision floating point(52 bits of significand, 11 bits of exponent, and 1 sign bit.)	$1.7*10^{-308}$ to $1.7*10^{308}$ (both positive and negative)

Overflow and Underflow

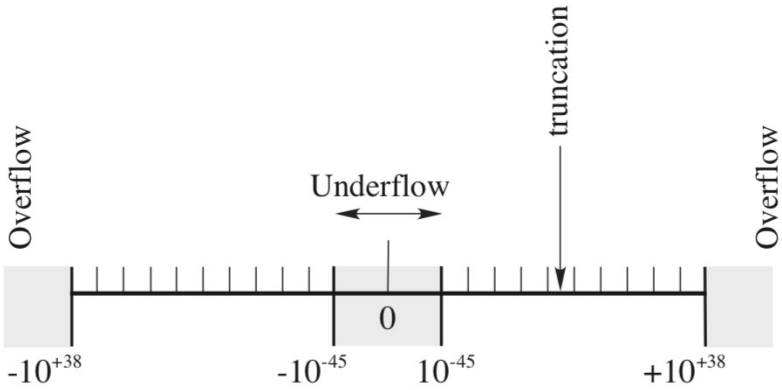

Figure 1.7 The limits of single-precision floating-point numbers and the consequences of exceeding these limits. The hash marks represent the values of numbers that can be stored; storing a number in between these values leads to truncation error. The shaded areas correspond to over- and underflow.

- If a single-precision number $x > 2^{128}$, a fault condition known as an *overflow* occurs. The resulting number x_c may end up being a machine-dependent pattern, not a number (NAN), or unpredictable.
- If $x < 2^{-128}$, an *underflow* occurs. The resulting number x_c is usually set to zero, although this can usually be changed via a compiler option.
- In our experience, *serious scientific calculations almost always require at least 64-bit (double-precision) floats*. And if you need double precision in one part of your calculation, you probably need it all over, which means double-precision library routines for methods and functions.

Practice Session #1: determining machine precision

- The loss of precision is categorized by defining the *machine precision* ε_m as the maximum positive number that can be added unity without changing it:

$$1_c + \epsilon_m \overset{\text{def}}{=} 1_c,$$

where the subscript *c* is a reminder that this is a computer representation of 1.

- Consequently, an arbitrary number *x* can be thought of as related to its floating- point representation x_c by

$$x_c = x(1 \pm \epsilon), \quad |\epsilon| \leq \epsilon_m,$$

but the actual value for ε is not known.

- In other words, except for powers of 2 that are represented exactly, we should assume that all single-precision numbers contain an error in the sixth decimal place and that all doubles have an error in the fifteenth place.
- `precision.cpp:` write a computer program to determine the machine precision. Define 1 in float (or double) precision arithmetic and keep adding epsilon (→epsilon/10) until 1+eps = 1.

Quadratic Equation Solver

- Finite precision arithmetic may lead to loss of accuracy when computing the roots of a quadratic polynomial with the standard formula,

$$ax^2 + bx + c = 0 \quad \rightarrow \quad x_{\pm} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- When quantities of the same sign are subtracted, some precision loss may occur. In particular, if b > 0, the root with the plus sign may become inaccurate when ac is relatively small compared to b^2. If this is the case, we can rationalize the previous expression and find

$$x_{\pm} = -\frac{2c}{b \pm \sqrt{b^2 - 4ac}}$$

- This suggests that we can use the standard representation when we sum and the second representation when we subtract terms:

$$x_1 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \quad \& \quad x_2 = \frac{2c}{-b - \sqrt{b^2 - 4ac}} \quad \text{when } b \geq 0$$

and

$$x_1 = \frac{2c}{-b + \sqrt{b^2 - 4ac}} \quad \& \quad x_2 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad \text{when } b < 0$$

Practice Session #2

- `quadratic.cpp:` using double precision arithmetic, write a computer program to solver the quadratic quadratic equation.

$$ax^2 + bx + c = 0$$

using, at first, the standard formula.

- Test your solver on the following cases:

a	b	c	x1	x2
1	$-(x_1+x_2)$	x_1*x_2	2	-3
1	$-(x_1+x_2)$	x_1*x_2	10^{-5}	10^{8}
1	$-(x_1+x_2)$	x_1*x_2	10^{-12}	10^{12}

- what do you see ?
- In order to avoid catastrophic cancellation, implement the selective expressions depending on the sign of the b coefficient.

Practice Session #3

- `roundoff.cpp`: using single precision arithmetic, obtain a numerical approximation to sqrt(x^2 + 1) – x (valid for large x) and 1 - cos(x) (valid for x ≈ 0). Write your code such that the output looks like

```
Example #1: compute sqrt(x^2 + 1) - x for large x
=================================================
x = 1.000000e+04; fx1 = 5.000000e-05; fx2 = 5.000000e-05; f(taylor) = 5.000000e-05
x = 1.000000e+05; fx1 = 4.999994e-06; fx2 = 5.000000e-06; f(taylor) = 5.000000e-06
x = 1.000000e+06; fx1 = -2.047500e-03; fx2 = 5.000000e-07; f(taylor) = 5.000000e-07
x = 1.000000e+07; fx1 = 1.884165e-02; fx2 = 5.000000e-08; f(taylor) = 5.000000e-08
x = 1.000000e+08; fx1 = 1.362821e+00; fx2 = 5.000000e-09; f(taylor) = 5.000000e-09
x = 1.000000e+09; fx1 = -7.846625e+00; fx2 = 5.000000e-10; f(taylor) = 5.000000e-10
x = 1.000000e+10; fx1 = 1.002044e+02; fx2 = 5.000000e-11; f(taylor) = 5.000000e-11


Example #2: compute 1-cos(x) for small x
=================================================
x = 1.000000e-01; fx1 = 4.995823e-03; fx2 = 4.995835e-03; f(taylor) = 4.995834e-03
x = 1.000000e-02; fx1 = 5.000830e-05; fx2 = 4.999958e-05; f(taylor) = 4.999958e-05
x = 9.999999e-04; fx1 = 4.768372e-07; fx2 = 4.999999e-07; f(taylor) = 4.999999e-07
x = 9.999999e-05; fx1 = 0.000000e+00; fx2 = 4.999999e-09; f(taylor) = 4.999999e-09
x = 9.999999e-06; fx1 = 0.000000e+00; fx2 = 4.999999e-11; f(taylor) = 4.999999e-11
x = 9.999999e-07; fx1 = 0.000000e+00; fx2 = 4.999999e-13; f(taylor) = 4.999999e-13
x = 9.999999e-08; fx1 = 0.000000e+00; fx2 = 4.999999e-15; f(taylor) = 4.999999e-15
x = 9.999999e-09; fx1 = 0.000000e+00; fx2 = 4.999999e-17; f(taylor) = 4.999999e-17
```

- Here:
  - `fx1` is the function itself, `fx1 = sqrt(x^2 + 1)-x` or `fx1 = 1-cos(x);`
  - `fx2` is the rationalized version;
  - `f(taylor)` the Taylor expansion around the desired point (x = ∞ or x = 0);

Practice Session #4 [SKIP]

- `series.cpp`: compute taylor series of sin(x) for x = 1, up to a precision of 10^{-8} (last term in the series should contribute < 10^{-8}) using term by term summation and recurrence relation;

- Iterative schemes:
  - `babyl.cpp`: compute the square root of a number using Babylonian (or Heron's) method (see next page)
  - `unstable_roundoff.cpp`: not all recurrence relation are numerically stable!

Practice Session #04: Computing the square root

- `heron.cpp`: Compute the square root using Heron's (or Babylonian) method: finding `sqrt(s)` is the same as solving the equation

$$f(x) = x^2 - S = 0 \quad \rightarrow \quad x^{(n+1)} = 0.5*(x^{(n)} + S/x^{(n)})$$

- The basic idea is that if x is an overestimate to the square root of a non-negative real number S then S/x, will be an underestimate and so the average of these two numbers may reasonably be expected to provide a better approximation.
- This is also known as "Heron's method", named after the 1st-century Greek mathematician Heron of Alexandria who gave the first explicit description of the method.
- Your code should take, as inputs, the value of S and a guess $x^{(0)}$ to its square root. The code output should look like

```
 Enter a realnumber:
13
 Enter your guess :
3
--------------------------------
Iteration # 1; x =   3.66666666666667e+00; err = 6.66666666666667e-01
Iteration # 2; x =   3.60606060606061e+00; err = 6.06060606060606e-02
Iteration # 3; x =   3.60555131143366e+00; err = 5.09294626941603e-04
Iteration # 4; x =   3.60555127546399e+00; err = 3.59696747942451e-08
Iteration # 5; x =   3.60555127546399e+00; err = 0.00000000000000e+00

 The SQRT of 1.30000000000000e+01 is: 3.60555127546399e+00
 The Exact values is:                 3.60555127546399e+00
```

- Here the error is computed as the difference between two successive iterates , $\varepsilon = |x^{(n+1)} - x^{(n)}|$.
- Using Arrays is not necessary.