

Mathematical Software Programming (02635)

Lecture 2 — September 11, 2025

Instructor: Martin S. Andersen

Fall 2025



Practical information

Autolab exercises

- ▶ due on a rolling basis (approximately 3 weeks after module)
- ▶ 10% of final grade

Assignment

- ▶ individual or group of 2-3 students
- ▶ Autolab and short report
- ▶ will be posted on October 23, 2025
- ▶ due on November 19, 2025
- ▶ 15% of final grade

This week

Topics

- ▶ Control statements and basic types
- ▶ Finite precision arithmetic

Learning objectives

- ▶ Evaluate discrete and continuous mathematical expressions
- ▶ Explain rounding errors and floating-point number representation of real numbers
- ▶ Choose appropriate data types and data structures for a given problem

Example 1: associative property of addition

```
#include <stdio.h>

int main(void) {

    double a = 1.0, b = 1e-16, c = -1.0;
    printf("(a + b) + c = %.4e\n", (a+b)+c);
    printf("a + (b + c) = %.4e\n", a+(b+c));
    printf("(a+c)+b = %.4e\n", (a+c)+b);

    return 0;
}
```

```
$ ./fpadd
(a + b) + c = 0.0000e+00
a + (b + c) = 1.1102e-16
(a + c) + b = 1.0000e-16
```

Example 2: integer arithmetic

```
#include <stdio.h>

int main(void) {

    /* a = 2^30 = 1073741824 */
    int a = (1 << 30);
    printf("a = %d\n", a);
    printf("2*a = %d\n", 2*a);

    return 0;
}
```

```
$ ./intmul
a = 1073741824
2*a = -2147483648
```

What went wrong?

- ▶ Associative property of addition

$$(a + b) + c = a + (b + c)$$

does not hold for *finite-precision* arithmetic

- ▶ We need to learn about floating-point numbers and finite-precision arithmetic
- ▶ Integer operations may *overflow*

Integers

- ▶ Unsigned integer types: modular arithmetic
 - ▶ $a \equiv b \pmod{N}$ if $a = b + kN$ for some $k \in \mathbb{Z}$
 - ▶ $a \bmod N$ is remainder: $a = \lfloor a/N \rfloor N + r$
 - ▶ unsigned integer operations yield remainder (e.g., `a+b` yields $(a + b) \bmod 2^n$)
- ▶ C does not require a specific representation of negative numbers
- ▶ Signed integer overflow is *undefined behavior* in C!
- ▶ Without overflow, signed integer arithmetic satisfies commutative, associative, and distributive properties, e.g., for addition we have
 - ▶ commutative: $x + y = y + x$
 - ▶ associative: $(x + y) + z = x + (y + z)$
 - ▶ left distributive: $x(y + z) = (xy) + (xz)$
 - ▶ right distributive: $(y + z)x = (yx) + (zx)$
- ▶ Integer promotion (no arithmetic operations on `char` and `short`)
- ▶ In C, if `a` and `b` are integers, then `a == (a/b)*b + a%b` is always true
 - ▶ `a/b` rounds towards zero and remainder `a%b` is signed

Floating-point numbers

$$x = s \cdot (d_0.d_1d_2 \dots d_{p-1})_b \cdot b^E = s \sum_{k=0}^{p-1} d_k \cdot b^{E-k}$$

- ▶ b is the base (e.g., 2 or 10)
- ▶ s represents the sign (+1 or -1)
- ▶ $d_0.d_1d_2 \dots d_{p-1}$ is the so-called *mantissa* or *significant*
- ▶ $d_i \in \{0, \dots, b-1\}$ is the i th digit of the mantissa
- ▶ E is the *exponent*
- ▶ p is the *precision*
- ▶ $x \neq 0$ is *normal* if $d_0 \neq 0$; otherwise x is *subnormal*

Floating-point numbers (continued)

Machine epsilon

$$\epsilon = (0.00\dots 01)_b = b^{-(p-1)}$$

Warning: some books/authors use a different definition!

Unit round-off

$$u = \frac{\epsilon}{2}$$

Unit in the last place (ulp)

$$\text{ulp}(x) = (0.00\dots 01)_b \cdot b^E = \epsilon \cdot b^E$$

$\text{ulp}(x)$ is the gap between $|x|$ and the next larger floating-point number

Representable positive numbers

Floating-point number system with precision p , base b , and exponent $E \in \{E_{\min}, \dots, E_{\max}\}$



- ▶ largest number (let $E = E_{\max}$ and $d_i = b - 1$)

$$N_{\max} = (b - 1) \sum_{i=0}^{p-1} b^{E_{\max}-i} = b^{E_{\max}}(b - b^{-(p-1)})$$

- ▶ smallest *normal* number (let $E = E_{\min}$, $d_0 = 1$, and $d_i = 0$ for $i > 0$)

$$N_{\min} = b^{E_{\min}}$$

- ▶ smallest *subnormal* number (let $E = E_{\min}$, $d_{p-1} = 1$, and $d_i = 0$ for $i < p - 1$)

$$b^{E_{\min}-(p-1)}$$

Example

Suppose $b = 2$, $p = 3$, and $E \in \{-1, 0, 1\}$



- ▶ machine epsilon: $\epsilon = 2^{-2} = 0.25$
- ▶ largest number: $N_{\max} = 2^2(1 - 2^{-3}) = 3.5$
- ▶ smallest *normal* number: $N_{\min} = 2^{-1} = 0.5$
- ▶ smallest *subnormal* number: $2^{-3} = 0.125$

Rounding

- ▶ Round to nearest
 - ▶ ties to even (aka *round to even*, *banker's rounding*, and *scientific rounding*)
 - ▶ ties away from zero
- ▶ Directed rounding
 - ▶ round toward zero
 - ▶ round toward $+\infty$ (round up)
 - ▶ round toward $-\infty$ (round down)

Example

Suppose $b = 10$ and $p = 2$, hence $\epsilon = 0.1$

x	nearest (even)	nearest (away)	zero	$+\infty$	$-\infty$
1.05	1.0	1.1	1.0	1.1	1.0
1.15	1.2	1.2	1.1	1.2	1.1
-1.05	-1.0	-1.1	-1.0	-1.0	-1.1
-1.15	-1.2	-1.2	-1.1	-1.1	-1.2

Computational model (no underflow/overflow)

Operation “ \odot ” (addition/subtraction/multiplication/division with *round to nearest*)

$$\text{fl}(x \odot y) = (x \odot y)(1 + \delta) \quad | \delta | \leq u$$

Function evaluation (for example, exp, log, sin, or cos)

$$\text{fl}(f(x)) = (1 + \delta_1)f((1 + \delta_2)x)$$

Absolute and relative error

$$e_{\text{abs}} = |\text{fl}(f(x)) - f(x)|$$

$$e_{\text{rel}} = \frac{|\text{fl}(f(x)) - f(x)|}{|f(x)|}, \quad f(x) \neq 0$$

Cancellation

Cancellation is **loss of significance** in calculations with finite-precision arithmetic

Suppose $b = 10$ and $p = 3$ ($\epsilon = 10^{-2}$) and let $x = 1.02$, $y = 1.01$, and $z = 1.23 \cdot 10^{-2}$

$$\text{fl}(x - z) = 1.01 \cdot 10^0 \quad e_{\text{abs}} = 2.3 \cdot 10^{-3} \quad e_{\text{rel}} \approx e_{\text{abs}}$$

$$\text{fl}(x - y) = 0.01 \cdot 10^0 = 1.00 \cdot 10^{-2} \quad e_{\text{abs}} = e_{\text{rel}} = 0$$

$$\text{fl}(\text{fl}(x + z) - y) = 0.02 \cdot 10^0 = 2.00 \cdot 10^{-2} \quad e_{\text{abs}} = 2.3 \cdot 10^{-3} \quad e_{\text{rel}} \approx 10^{-1}$$

Subtraction may cause many significant digits to disappear (can be *benign* or *catastrophic*)

Catastrophic cancellation

Relative error substantially larger than absolute error when subtracting two nearly equal numbers

Loss of precision

Suppose $x > y > 0$ are normalized floating-point numbers with precision p

$$x = m_x \cdot b^{E_x}, \quad y = m_y \cdot b^{E_y}, \quad m_x, m_y \in [1, b)$$

Subtraction amounts to

- ▶ rewrite to get the same exponent as x

$$x - y = (1 - y/x)x = m_x(1 - y/x) \cdot b^{E_x}$$

- ▶ normalize mantissa: find E such that

$$x - y = m \cdot b^{E_x - E}, \quad m = m_x(1 - y/x) \cdot b^E, \quad m \in [1, b)$$

Loss of precision theorem

Suppose $x > y > 0$ are normal floating-point numbers with precision p , and

$$b^{-\beta} \leq 1 - \frac{y}{x} \leq b^{-\alpha}, \quad \alpha, \beta \in \mathbb{N}_0,$$

then at most β and at least α significant digits will be lost in the subtraction $x - y$.

Example

Suppose $b = 10$ and $p = 3$ ($\epsilon = 10^{-2}$) and let $x = 1.02$ and $y = 1.01$

$$1 - \frac{y}{x} = \frac{x - y}{x} = \frac{0.01}{1.02} \approx 9.8 \cdot 10^{-3}$$

Thus, at most 3 and at least 2 significant digits will be lost in the subtraction $x - y$.

Error analysis: subtraction

Suppose $\hat{x} = \text{fl}(x)$ and y are finite-precision floating-point numbers

Floating-point subtraction satisfies

$$\begin{aligned}\text{fl}(\hat{x} - y) &= (x(1 + \delta_1) - y)(1 + \delta_2) \\ &= (x - y) + (\delta_1 + \delta_2 + \delta_1\delta_2)x - \delta_2y, \quad \text{for some } |\delta_1| \leq u, |\delta_2| \leq u\end{aligned}$$

Relative error bound

$$e_{\text{rel}} = \frac{|(\delta_1 + \delta_2 + \delta_1\delta_2)x - \delta_2y|}{|x - y|} \leq \frac{|x| + |y|}{|x - y|}(2 + u)u, \quad x \neq y$$

IEEE 754: Standard for FP Arithmetic

- ▶ Technical standard for floating-point formats and arithmetic
- ▶ Adopted in 1985, latest revision from 2019
- ▶ Several binary and decimal formats (i.e., $b = 2$ or $b = 10$)
- ▶ Defines rounding modes and required operations (arith., conversions, total ordering, ...)
- ▶ The four arithmetic operations and the square root should be correctly rounded

binary32 (single precision)

- ▶ base 2
- ▶ 32 bits: 1 sign, 8 exponent, 23 mantissa ($p = 24$, d_0 is implicit)
- ▶ typically float in C

binary64 (double precision)

- ▶ base 2
- ▶ 64 bits: 1 sign, 11 exponent, 52 mantissa ($p = 53$, d_0 is implicit)
- ▶ typically double in C

Example: binary64 (double precision) floating-point numbers

Floating-point number system with $b = 2$, $p = 53$, and $E \in \{-1022, \dots, 1023\}$

- ▶ $\epsilon = 2^{-52} \approx 2.22 \cdot 10^{-16}$
- ▶ $N_{\max} = 2^{1023}(2 - 2^{-52}) \approx 1.8 \cdot 10^{308}$
- ▶ $N_{\min} = 2^{-1022} \approx 2.2 \cdot 10^{-308}$

Representing special values

11 bits for exponent: 2048 possible values (normal numbers use only 2046 of these)

- ▶ Signed zero and subnormal numbers
- ▶ Signed INFINITY (divide-by-zero, overflow, $\log(0)$, ...) and NAN (not a number)

Invalid operations yield NAN, e.g.,

$$\sqrt{-1}, \quad 0 \cdot \infty, \quad 0/0, \quad \infty/\infty, \quad \infty - \infty$$

Decimal floating-point numbers

- ▶ included 2008 revision
- ▶ limited hardware support (software implementation: libdfp)
- ▶ emulate exact decimal rounding (e.g., in finance)
- ▶ different binary representations allowed (binary coded / decimal coded)

decimal32 (32 bits)

Floating-point number system with $b = 10$, $p = 7$, and $E \in \{-95, \dots, 96\}$

decimal64 (64 bits)

Floating-point number system with $b = 10$, $p = 16$, and $E \in \{-383, \dots, 384\}$

Floating-point unit (FPU)

Floating-point operations (+, -, *, /, square root, bit shifting) may be carried out by

- ▶ Integrated FPU
- ▶ Add-on FPU
- ▶ FPU emulator (floating-point library)

Some FPUs also support transcendental functions (e.g., log., exp., trig.)

Intel x87 and extended precision

- ▶ Initially add-on FPU (Intel 8087)
- ▶ Floating-point related subset of x86 instruction set
- ▶ 80-bit *extended double* may be used internally ($p = 64$, $\epsilon \approx 10^{-19}$)
- ▶ Some systems implement long double as “extended double” on the x86 architecture

Classifying floating-point numbers (C99)

Header file `math.h` includes macros and functions:

- ▶ `isfinite(x)`, `isnormal(x)`, `isnan(x)`, `isinf(x)`, `fpclassify(x)`

`fpclassify(x)` returns one of the following values:

- ▶ `FP_NAN`, `FP_INFINITE`, `FP_ZERO`, `FP_SUBNORMAL`, `FP_NORMAL`

Example

Check if x is NAN

```
double x = 0.0/0.0;
if (fpclassify(x) == FP_NAN)
    printf("x is not a number\n");
if (isnan(x))
    printf("x is not a number\n");
if (x != x) // Required by IEEE 754; may not work with some compilers
    printf("x is not a number\n");
```

Floating-point environment (C99)

Header file `fenv.h` includes macros and functions for controlling FP environment

- ▶ `fesetround(int mode)`, `fegetround(void)`, ...
- ▶ rounding modes: `FE_DOWNWARD`, `FE_TONEAREST`, `FE_TOWARDZERO`, `FE_UPWARD`

Inform compiler that application might access the floating-point environment

```
#pragma STDC FENV_ACCESS ON // C99 (support is limited)
```

Example

Set rounding mode to *round toward zero*

```
if (fesetround(FE_TOWARDZERO)) {  
    fprintf(stderr, "Failed to set rounding mode\n");  
}
```

GCC/Clang compiler option: `-frounding-math`

Avoiding cancellation/underflow/overflow

Use special functions (`expm1()`, `log1p()` `hypot()`, `lgamma()`, ...)

Examples

- ▶ $f(x) = e^x - 1$ (loss of significance as $x \rightarrow 0$): use `expm1(x)`
- ▶ $f(x, y) = \sqrt{x^2 + y^2}$ (may underflow/overflow): use `hypot(x, y)` or rewrite as

$$f(x, y) = |x| \sqrt{1 + (y/x)^2}, \quad \text{if } |x| > |y|$$

- ▶ $f(n) = \log(n!)$ with $n \in \mathbb{N}_0$ ($n!$ may overflow): use `lgamma(n+1)` or rewrite as

$$f(n) = \sum_{k=2}^n \log(k)$$

Avoiding cancellation/underflow/overflow (continued)

Rewrite expression (trigonometric identities, radical simplification, Taylor series, . . .)

Examples

- ▶ $f(x) = 1 - \cos(x)$ (loss of significance as $x \rightarrow 0$): rewrite as

$$f(x) = 1 - (1 - 2 \sin^2(x/2)) = 2 \sin^2(x/2)$$

- ▶ $f(x) = \sqrt{x+2} - \sqrt{x}$ (loss of significance as $x \rightarrow \infty$): rewrite as

$$f(x) = (\sqrt{x+2} - \sqrt{x}) \frac{\sqrt{x+2} + \sqrt{x}}{\sqrt{x+2} + \sqrt{x}} = \frac{2}{\sqrt{x+2} + \sqrt{x}}$$

Exercises

Autolab exercise

Evaluate the function

$$f(x) = \begin{cases} \frac{1-\cos(x)}{x^2}, & x \neq 0, \\ \frac{1}{2}, & x = 0. \end{cases}$$

Taylor expansion of $\cos(x)$ around 0:

$$\cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \dots$$

Double angle identity:

$$\cos(2x) = 1 - 2\sin^2(x)$$

Things to avoid with floating-point numbers

- ▶ Avoid testing for equality between floating-point numbers (with some exceptions)
- ▶ Avoid using floating-point numbers as loop counters
- ▶ If possible, avoid subtracting nearly equal quantities

```
#include <stdio.h>
int main(void) {
    float test = 4.1;
    if (test == 4.1) printf("Equal!\n");
    else print("Yikes!\n");
    return 0;
}
```

GCC/Clang compiler option: -Wfloat-equal