POLITECNICO MILANO 1863

DEPARTMENT OF CHEMISTRY MATERIALS AND CHEMICAL ENGINEERING

Numbers, errors and computers.

Calcoli di Processo dell’ Ingegneria Chimica

Timoteo Dinelli

07th of October 2025.

Department of Chemistry, Materials and Chemical Enginering, “Giulio Natta”, Politecnico di Milano.

email: timoteo.dinelli@polimi.it

Numbers representation.

"*In computing, floating-point arithmetic (**FP**) is arithmetic that represents subsets of real numbers using an integer with a fixed precision, called the significand, scaled by an integer exponent of a fixed base. Numbers of this form are called floating-point numbers. For example, 12.345 is a floating-point number in base ten with five digits of precision.*" Wikipedia.

$$12.345 = \underbrace{12345}_{\text{Significand (Mantissa)}} \times \underbrace{10^{-3}}_{\text{Exponent}}$$

Additional RECOMMENDED read:

- Floating point representations.
- LLM quantization.

Elementary operations

Initialize a single precision variable in MATLAB (e.g. x = 1e^{+25}) using the function:

```
x = single(1.e+25)
```

Working in single precision predict and calculate from x = 1.e+25 and y = 1.e+18 the following values of z:

	Single Precision	Double Precision
$z = x * y$	inf	1.0000e+43
$z = x/y$	10000000	10000000
$z = y/x$	1.0000e-07	1.0000e-07
$z = x^2$	inf	1.0000e+50
$z = y^2$	1.0000e+36	1.0000e+36
$z = 1./(x * y)$	0	1.0000e-43
$z = 1./x/y$	9.9492e-44	1.0000e-43
$z = y + 1e10$	1.0000e+18	1.0000e+18
$z = x * y/(x * y + 1)$	NaN	1

Exercises

Compute the Machine Epsilon (MACHEPS)

Machine epsilon (denoted as $\varepsilon_{\text{mach}}$) is the smallest positive floating-point number such that $1.0 + \varepsilon_{\text{mach}} \neq 1.0$ in floating-point arithmetic. It represents the relative precision of floating-point numbers and is a fundamental constant for a given floating-point format (e.g., approximately 2.22×10^{-16} for double precision).

Machine epsilon is important because it quantifies the limit of precision in numerical computations and helps estimate round-off errors in calculations.

- The algorithm iterates by halving epsilon until $1.0 + \varepsilon/2$ is indistinguishable from 1.0 in floating-point arithmetic. At this point, we've found the machine epsilon.

Implementation

```
function epsilon = macheps_implementation()
    epsilon = 1.0; % Initialize epsilon to 1
    while (1.0 + epsilon/2) > 1.0
        % Iterate as long as 1.0 + epsilon/2 is distinguishable from 1.0
        epsilon = epsilon / 2; % Halve epsilon at each iteration
    end
    % At the end of the loop, epsilon is the machine epsilon
    % (the smallest value where 1.0 + epsilon > 1.0)
end
```

Usage:

```
eps_computed = macheps_implementation()
eps_builtin = eps(1.0)  % Compare with built-in function
```

Sum of the inverse of numbers (order matters!)

Write a script which computes:

$$\sum_{n=1}^{1000000} \frac{1}{n}$$

in single and double precision. Then compare with the results obtained inverting the order of the sum, so by computing:

$$\sum_{n=1000000}^{1} \frac{1}{n}$$

Implementation

```
% Sum from 1 to 1000000 (forward)
sum_forward_single = single(0);
sum_forward_double = 0;
for n = 1:1000000
    sum_forward_single = sum_forward_single + single(1/n);
    sum_forward_double = sum_forward_double + 1/n;
end

% Sum from 1000000 to 1 (backward)
sum_backward_single = single(0);
sum_backward_double = 0;
for n = 1000000:-1:1
    sum_backward_single = sum_backward_single + single(1/n);
    sum_backward_double = sum_backward_double + 1/n;
end
```

```

% Display results
fprintf('Forward sum (single):  %.10f\n', sum_forward_single);
fprintf('Forward sum (double):  %.15f\n', sum_forward_double);
fprintf('Backward sum (single): %.10f\n', sum_backward_single);
fprintf('Backward sum (double): %.15f\n', sum_backward_double);

fprintf('\nDifferences:\n');
fprintf('Single precision:  %.10e\n', ...
    abs(sum_forward_single - sum_backward_single));
fprintf('Double precision:  %.15e\n', ...
    abs(sum_forward_double - sum_backward_double));
```

Vancouver: a nickel at a time

Analogously to what happened on the Vancouver stock market (reference), starting from a stock value of 1000, check what happens when a random variation of $\pm 1\%$ in its value is iterated for 10000 times. Try to round or truncate to two decimal places using floor, ceil, and round. Compare the results with the number obtained using full computer precision.

Please verify in the MATLAB help how the functions *rand, floor, ceil, fix, round* work.

Solution - Part 1: Setup

```
% Initial stock value
initial_value = 1000;
n_iterations = 10000;

% Initialize values for different rounding methods
value_full_precision = initial_value;
value_floor = initial_value;
value_ceil = initial_value;
value_round = initial_value;

% Set random seed for reproducibility (optional)
rng(42);
```

Solution - Part 2: Iteration Loop

```
% Iterate 10000 times with random ±1% variation
for i = 1:n_iterations
    % Generate random variation: ±1%
    % rand() gives [0,1], so 2*rand()-1 gives [-1,1]
    variation = (2 * rand() - 1) * 0.01;

    % Full precision (no rounding)
    value_full_precision = value_full_precision * (1 + variation);
```

Solution - Part 3: Rounding Methods

```
    % Floor (round down to 2 decimal places)
    temp = value_floor * (1 + variation);
    value_floor = floor(temp * 100) / 100;

    % Ceil (round up to 2 decimal places)
    temp = value_ceil * (1 + variation);
    value_ceil = ceil(temp * 100) / 100;

    % Round (round to nearest 2 decimal places)
    temp = value_round * (1 + variation);
    value_round = round(temp * 100) / 100;
end
```

Solution - Part 4: Display Results

```
% Display results
fprintf('Results after %d iterations:\n', n_iterations);
fprintf('=========================================\n');
fprintf('Initial value:        %.2f\n', initial_value);
fprintf('Full precision:       %.6f\n', value_full_precision);
fprintf('Floor (round down):   %.2f\n', value_floor);
fprintf('Ceil (round up):      %.2f\n', value_ceil);
fprintf('Round (to nearest):   %.2f\n', value_round);
fprintf('=========================================\n');
```

Solution - Part 5: Differences Analysis

```
% Calculate differences from full precision
fprintf('\nDifferences from full precision:\n');
fprintf('Floor:  %.6f (%.2f%%)\n', ...
    value_full_precision - value_floor, ...
    100*(value_full_precision - value_floor)/value_full_precision);
fprintf('Ceil:   %.6f (%.2f%%)\n', ...
    value_full_precision - value_ceil, ...
    100*(value_full_precision - value_ceil)/value_full_precision);
fprintf('Round:  %.6f (%.2f%%)\n', ...
    value_full_precision - value_round, ...
    100*(value_full_precision - value_round)/value_full_precision);
```

Expected observations

- Floor will consistently lose value (downward bias, similar to Vancouver bug)
- Ceil will consistently gain value (upward bias)
- Round should be closer to full precision (unbiased rounding)
- The cumulative effect over 10,000 iterations will be significant, demonstrating the Vancouver Stock Exchange issue where the index lost about 50% of its value due to truncation errors!

Thank you for the attention!