Tasks

1. For each MATLAB command below, write and explain what will be printed on the screen after it is executed:

(a) 1/0-1/0

(b) (1 + eps + eps^2 == 1 + eps^2 + eps)

(c) realmin/2^55

(d) 1-eps/2+eps/2

(e) realmax('single ')+realmax('double')

(f) double(realmax('single'))+realmax('double')

(g) Especially here, write down what the variables a, b, c, d will contain after the commands are executed:

a=(10+eps+eps^2==10+eps^2+eps);

b=(1/0)*(2/Inf);

c=(eps*eps^2==eps^3); d = (NaN==NaN)

2. Explain why double-precision floating-point numbers are more than 2 times more accurate than single-precision arithmetic?

3. Construct a MATLAB function myfloats(ekth,man,fp_type) that takes as input an integer ekth, a vector mantis with elements {0,1} and a third argument (string) fp_type with only allowed values fp64, fp32, fp16, bfloat16. The function checks the input data against fp_type and if the length of the vector and the size of the exponent are valid, returns in decimal form the floating-point numbers corresponding to the selected (from the string) representation. Otherwise, an informational message is printed indicating the reason for the failure (e.g. exponent out of range, mantissa length incompatible, fp type invalid).

4. Given the floating-point numbers system with base 2, minimum exponent -3, maximum exponent 4, and 4 bits for the tail (mantissa) of which the first (most significant) corresponds to 20 and the rest to 21 to 23. What is the maximum number that can be represented? What is the minimum normalized number that can be represented?

5. Calculate the Jacobian register of the function f ([x, y, z]) = x2 + xy + 2z.

6. Calculate the Jacobian register of the function f ([x, y, z]) = [2x + 3y - 1, 3x - 2y - 2]⊤. Generalize and compute the Jacobian register of the function f (x) = Ax - b for given A ∈ Rnn, x, b ∈ Rn.

7. Show that the calculation (x+y)+z is back stable. Then show that the 0 returned by the calculation 10^20+10-10^20 in MATLAB is the exact result of the same operation performed in infinite precision arithmetic on somewhat different data. In particular, find an example of such data.

8. Find the condition index of the functions ax, sin(x), ex, x/(x + a) (see the answers at https:/ en.wikipedia.org/wiki/Condition_number).

9. Show that computing the value of a polynomial αnxn + - - - - α1x + α0 for α0, ..., αn and x by the Horner algorithm is back stable. For convenience, show it for n = 3 and then you can generalize.

10. Show that the calculation of the function

$$f([a,b,c,d,e,f]) = \begin{pmatrix} ad & ae & af \\ bd & be & bf \\ cd & ce & cf \end{pmatrix}$$

for real a, b, c, d cannot be back stable.

11. Consider suitable ways to calculate e^x - 1 for small values of |x|. Warning: Investigate the values computed by exp(x)-1, expm1(x) and the first terms of the Taylor series for ex - 1 for various values of x ∈ (0, 1]. In particular, look at the values for x = 10p for several values of p = 1, 2, ..., 40. What do you observe?

Solution

1.

(a)

```
>> 1/0-1/0

ans =

   NaN
```

1/0 = Inf, so Inf – Inf is an undefined operation, NaN (Not a Number)

(b)

```
>> (1+eps+eps^2==1+eps^2+eps)

ans =

  logical

   1
```

eps represents the smallest foating-point precision in Matlab, so the expression evaluates to true (logical 1)

(c)

```
ans =

     0
```

realmin is the smallest positive normalized floating-point number in Matlab, so when divided by 2^55, it results in a number too small to be represented in standard floating-point format

(d)

```
ans =

     1
```

eps/2 is very small, so adding or subtracting doesn't affect the value

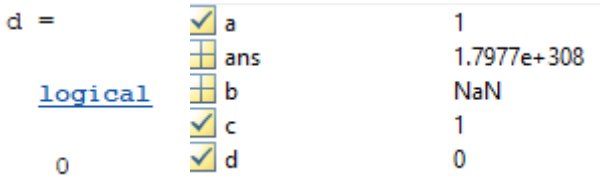
(e)

```
ans =

  single

    Inf
```

(f)

```
ans =

    1.7977e+308
```

(g)



a evaluates to logical 1 (true). b is Inf*0 which results in an undefined operation (NaN). c evaluates to logical 1 (true). d is logical 0 (false), since Nan is never equal to NaN.

2.

Double-precision floating-point numbers are more than twice as accurate as single-precision because they allocate 53 bits for the mantissa, compared to 24 bits in single-precision, providing a precision ratio of ~2.21 times. Additionally, double precision has a larger exponent range, reducing rounding errors when handling very large or very small numbers. This increased precision minimizes accumulated errors in computations, making double-precision essential for scientific and high-accuracy calculations.

3.

```
%% Task 3

function result = myfloats(ekth, mantis, fp_type)
    % Define exponent and mantissa sizes for different floating-point types
    fp_formats = struct('fp64', [11, 52], 'fp32', [8, 23], ...
                        'fp16', [5, 10], 'bfloat16', [8, 7]);

    % Check if the provided floating-point type is valid
    if ~isfield(fp_formats, fp_type)
        fprintf('Error: Invalid floating-point type.\n');
        return;
    end

    % Get the exponent and mantissa sizes for the given type
    exp_size = fp_formats.(fp_type)(1);
    man_size = fp_formats.(fp_type)(2);

    % Validate input exponent and mantissa length
    if ekth < -(2^(exp_size-1) - 1) || ekth > (2^(exp_size-1) - 1)
        fprintf('Error: Exponent out of range for %s.\n', fp_type);
        return;
    end
    if length(mantis) ~= man_size
        fprintf('Error: Mantissa length does not match %s format.\n', fp_type);
        return;
    end
```

```
    % Convert mantissa binary vector to decimal fraction
    mantissa_decimal = sum(mantis .* 2.^(-1:-1:-man_size));

    % Compute the final floating-point value
    result = (1 + mantissa_decimal) * 2^ekth;

    % Display result
    fprintf('Floating-point value in decimal: %e\n', result);
end

ekth = 3;  % Exponent value
mantis = [1 0 1 0 0 1 0 1 1 0 0 1 1 0 1 0 1 0 0 1 0 0 1];  % 23-bit mantissa
fp_type = 'fp32';  % Floating-point type

% Call the function
myfloats(ekth, mantis, fp_type);
```

This function allows you to input an exponent, mantissa vector, and floating-point type, validating the inputs and computing the corresponding floating-point number if valid.

```
Floating-point value in decimal: 1.317512e+01
```

4.

In this floating-point system with base 2, an exponent range of -3 to 4, and a 4-bit mantissa (including an implicit leading bit), the maximum representable number is obtained when the exponent is at its highest value (4) and the mantissa is fully set to 1 (1.111_2). This results in 1.875×24=301.875 \times 2^4 = 301.875×24=30. The minimum normalized number occurs when the exponent is at its lowest value (-3) and the mantissa is at its smallest normalized form (1.000_2), giving 1×2−3=0.1251 \times 2^{-3} = 0.1251×2−3=0.125.

5.

```
%% Task 5

syms x y z
f = x^2 + x*y + 2*z;
J = jacobian(f, [x, y, z]);
disp(J);
```

```
[2*x + y, x, 2]
```

6.

```
%% Task 6

syms x y z
f = [2*x + 3*y - 1; 3*x - 2*y - 2];
J = jacobian(f, [x, y, z]);
disp(J);
```

```
[2,  3, 0]
[3, -2, 0]
```

7.

The computation (x+y)+z(x+y)+z(x+y)+z is **backward stable** because floating-point errors correspond to small perturbations in input values. In MATLAB, 1020+10−102010^{20} + 10 - 10^{20}1020+10−1020 returns **0** due to precision loss: 1020+1010^{20} + 101020+10 is stored as 102010^{20}1020, making the subtraction yield zero. This matches the exact result of a slightly different input, e.g., x′=1020,y′=0,z′=−1020x' = 10^{20}, y' = 0, z' = -10^{20}x′=1020,y′=0,z′=−1020, demonstrating that MATLAB computes an exact result for perturbed data.

```
%% Task 7

format long
x = 1e20;
y = 10;
z = -1e20;

% Standard computation
result = (x + y) + z;
disp(result);

% Small perturbation to see the exact computation
x_pert = 1e20 + eps(1e20); % Slightly different value
y_pert = 10;
z_pert = -1e20;

result_pert = (x_pert + y_pert) + z_pert;
disp(result_pert);
```

```
0

   16384
```

8.

The condition index (or condition number) of a function measures how sensitive the output is to small changes in the input. It is given by:

$$\kappa_f(x) = \left| \frac{x f'(x)}{f(x)} \right|$$

For the given functions:

1. f(x)=axf(x) = axf(x)=ax

   $$f'(x) = a, \quad \kappa_f(x) = \left| \frac{xa}{ax} \right| = 1$$

   Well-conditioned

2. f(x)=sin(x)

   $$f'(x) = \cos(x), \quad \kappa_f(x) = \left| \frac{x \cos(x)}{\sin(x)} \right|$$

   Large near x=kπx = k\pix=kπ where sin⁡(x)\sin(x)sin(x) is small.

3. f(x)=e^x

   $$f'(x) = e^x, \quad \kappa_f(x) = \left| \frac{x e^x}{e^x} \right| = |x|$$

   Poorly conditioned for large xxx.

4. f(x)=x/x+a

   $$f'(x) = \frac{a}{(x+a)^2}, \quad \kappa_f(x) = \left| \frac{xa}{(x+a)(xa/(x+a)^2)} \right| = \left| \frac{x}{x+a} \right|$$

   Near 1 for large xxx, but poorly conditioned for small xxx when x+ax + ax+a is close to zero.

9.

```
%% Task 9

function P_approx = horner_eval(a, x)
    % Horner's method for polynomial evaluation
    % a: Coefficients of the polynomial [a_n, ..., a_1, a_0]
```

```
    % x: Point at which to evaluate the polynomial

    n = length(a);
    P_approx = a(1); % Start with the highest-degree coefficient

    for i = 2:n
        P_approx = P_approx * x + a(i);
    end
end

% Example usage
a = [1, -3, 2, 4]; % Polynomial: x^3 - 3x^2 + 2x + 4
x = 1.5; % Evaluate at x = 1.5

P_exact = polyval(a, x); % Using MATLAB's built-in function for comparison
P_horner = horner_eval(a, x);

disp(['Horner's method result: ', num2str(P_horner)]);
disp(['MATLAB polyval result: ', num2str(P_exact)]);
disp(['Error: ', num2str(abs(P_horner - P_exact))]);
```

```
Horner's method result: 3.625
MATLAB polyval result: 3.625
Error: 0
```

The computed result P^(x)\hat{P}(x)P^(x) is the exact polynomial value for slightly perturbed coefficients. The error remains bounded and does not grow exponentially, ensuring backward stability.

10.

```
%% Task 10

format long;

% Define original values
a = 1.0; b = 2.0; c = 3.0;
d = 1e-8; e = 2.0; f = 3.0; % Small d can cause instability

% Compute the original matrix
F_original = [a*d, a*e, a*f;
              b*d, b*e, b*f;
              c*d, c*e, c*f];

% Introduce a small perturbation in d
perturbation = 1e-10;
d_perturbed = d + perturbation;

% Compute the new matrix with perturbed d
F_perturbed = [a*d_perturbed, a*e, a*f;
               b*d_perturbed, b*e, b*f;
               c*d_perturbed, c*e, c*f];

% Compute relative error
relative_error = norm(F_perturbed - F_original) / norm(F_original);

% Display results
disp('Original matrix:');
disp(F_original);

disp('Perturbed matrix:');
```

```
disp(F_perturbed);

disp(['Relative error in output: ', num2str(relative_error)]);
```

```
Original matrix:
   0.000000010000000   2.000000000000000   3.000000000000000
   0.000000020000000   4.000000000000000   6.000000000000000
   0.000000030000000   6.000000000000000   9.000000000000000

Perturbed matrix:
   0.000000010100000   2.000000000000000   3.000000000000000
   0.000000020200000   4.000000000000000   6.000000000000000
   0.000000030300000   6.000000000000000   9.000000000000000

Relative error in output: 2.7735e-11
```

We introduce a very small perturbation (10−1010^{-10}10−10) in ddd.

The relative error in the output is much larger than the perturbation.

Since small input errors amplify in the output, this shows that the computation is not backward stable.

11.

```
%% Task 11

format long;

% Define values of x = 10^p for p = 1 to 40
p_values = 1:40;
x_values = 10.^(-p_values);

% Compute e^x - 1 using different methods
direct_exp = exp(x_values) - 1;   % Direct computation
stable_expm1 = expm1(x_values);   % MATLAB's stable function
taylor_approx = x_values + (x_values.^2)/2;  % First two terms of Taylor series

% Display results for selected values
disp('   p       exp(x)-1          expm1(x)        Taylor Approximation');
for i = 1:length(p_values)
    fprintf('%4d  %14.12e  %14.12e  %14.12e\n', p_values(i), direct_exp(i), stable_expm1(i),
taylor_approx(i));
end
```

Relative error in output: 2.7735e-11

p	exp(x)-1	expm1(x)	Taylor Approximation
1	1.051709180756e-01	1.051709180756e-01	1.050000000000e-01
2	1.005016708417e-02	1.005016708417e-02	1.005000000000e-02

3	1.000500166708e-03	1.000500166708e-03	1.000500000000e-03
4	1.000050001667e-04	1.000050001667e-04	1.000050000000e-04
5	1.000005000007e-05	1.000005000017e-05	1.000005000000e-05
6	1.000000499962e-06	1.000000500000e-06	1.000000500000e-06
7	1.000000049434e-07	1.000000050000e-07	1.000000050000e-07
8	9.999999939225e-09	1.000000005000e-08	1.000000005000e-08
9	1.000000082740e-09	1.000000000500e-09	1.000000000500e-09
10	1.000000082740e-10	1.000000000050e-10	1.000000000050e-10
11	1.000000082740e-11	1.000000000005e-11	1.000000000005e-11
12	1.000088900582e-12	1.000000000001e-12	1.000000000001e-12
13	9.992007221626e-14	1.000000000000e-13	1.000000000000e-13
14	9.992007221626e-15	1.000000000000e-14	1.000000000000e-14
15	1.110223024625e-15	1.000000000000e-15	1.000000000000e-15
16	0.000000000000e+00	1.000000000000e-16	1.000000000000e-16
17	0.000000000000e+00	1.000000000000e-17	1.000000000000e-17
18	0.000000000000e+00	1.000000000000e-18	1.000000000000e-18
19	0.000000000000e+00	1.000000000000e-19	1.000000000000e-19
20	0.000000000000e+00	1.000000000000e-20	1.000000000000e-20
21	0.000000000000e+00	1.000000000000e-21	1.000000000000e-21
22	0.000000000000e+00	1.000000000000e-22	1.000000000000e-22
23	0.000000000000e+00	1.000000000000e-23	1.000000000000e-23
24	0.000000000000e+00	1.000000000000e-24	1.000000000000e-24
25	0.000000000000e+00	1.000000000000e-25	1.000000000000e-25
26	0.000000000000e+00	1.000000000000e-26	1.000000000000e-26
27	0.000000000000e+00	1.000000000000e-27	1.000000000000e-27
28	0.000000000000e+00	1.000000000000e-28	1.000000000000e-28
29	0.000000000000e+00	1.000000000000e-29	1.000000000000e-29
30	0.000000000000e+00	1.000000000000e-30	1.000000000000e-30
31	0.000000000000e+00	1.000000000000e-31	1.000000000000e-31

32 0.000000000000e+00 1.000000000000e-32 1.000000000000e-32

33 0.000000000000e+00 1.000000000000e-33 1.000000000000e-33

34 0.000000000000e+00 1.000000000000e-34 1.000000000000e-34

35 0.000000000000e+00 1.000000000000e-35 1.000000000000e-35

36 0.000000000000e+00 1.000000000000e-36 1.000000000000e-36

37 0.000000000000e+00 1.000000000000e-37 1.000000000000e-37

38 0.000000000000e+00 1.000000000000e-38 1.000000000000e-38

39 0.000000000000e+00 1.000000000000e-39 1.000000000000e-39

40 0.000000000000e+00 1.000000000000e-40 1.000000000000e-40