

# EAFIT University - Department of Information and Computer Sciences

Mauricio Escudero Restrepo  
César Esteban Peñuela Cubides  
Diego Mesa Ospina

August 2024

## **1 Objective**

Review the progress of the final project and the work methodologies of the workgroups

## **2 Course**

Numerical Analysis

## **3 Responsible Faculty Member**

Edwar Samir Posada Murillo

## **4 Current Report Delivery Date**

October 1st 2024

## **5 Numerical Methods**

The following numerical methods will be presented in this document in the following manner, first a pseudo-code version of the method algorithm will be presented, after the code for the implementation of the method in both languages selected will be presented, finally proof for the execution and results of the methods will be provided.

## 5.1 Incremental Search

Incremental Search Method is a numerical method used to find the roots of a function (i.e., where the function equals zero). This method works by iteratively narrowing down an interval where a root lies. The goal is to find an approximation of the root with increasing precision.

### 5.1.1 Pseudo-code

```
FUNCTION IncrementalSearch(f, x0, h, Nmax):

    INPUT:
    - f: continuous function for which a sign change is sought.
    - x0: initial point.
    - h: step size (increment in each iteration).
    - Nmax: maximum number of iterations.

    OUTPUT:
    - a: left endpoint of the interval where a sign change occurs.
    - b: right endpoint of the interval where a sign change occurs
      .
    - iter: number of iterations performed.
    - data_frame: table of results with details of each iteration.

    BEGIN:
    1. Initialize:
       - xprev = x0.
       - fprev = f(xprev).
       - xcurr = xprev + h.
       - fcurr = f(xcurr).
       - Create an empty list to store the results.

    2. FOR i = 1 to Nmax:

       a. If fprev * fcurr < 0, a sign change occurs in the
          interval [xprev, xcurr]:
          - Save the iteration result (i, xcurr, fcurr, error).
          - Break the loop.

       b. Save current iteration values:
          - Store iteration values: index i, current x, f(current
            x), and error (difference between current x and
            previous x).

       c. Update values for the next iteration:
          - xprev = xcurr.
```

```

        - fprev = fcurr.
        - xcurr = xprev + h.
        - fcurr = f(xcurr).

3. Create a DataFrame with the stored results.

4. Return:
    - xprev as the left endpoint of the interval.
    - xcurr as the right endpoint of the interval.
    - i as the number of iterations performed.
    - the DataFrame with the results.

END FUNCTION

```

### 5.1.2 Method Implementation

#### Python

```

def incremental_search(f, x0, h, Nmax):
    """
    This program finds an interval where  $f(x)$  has a sign change
    using the incremental search method.
    Inputs:
    f: continuous function
    x0: initial point
    h: step
    Nmax: maximum number of iterations

    Outputs:
    a: left endpoint of the interval
    b: right endpoint of the interval
    iter: number of iterations
    """

    # Initialization

    xant = x0
    fant = f(xant)
    xact = xant + h
    fact = f(xact)
    result_array = []
    # Loop
    for i in range(1, Nmax+1):
        if fant * fact < 0:
            result = {
                'i': i,

```

```

        'x_i': xact,
        'f_xi': fact,
        'e': abs(xact - xant)
    }
    result_array.append(result)
    break
result = {
    'i': i,
    'x_i': xact,
    'f_xi': fact,
    'e': abs(xact - xant)
}
result_array.append(result)
xant = xact
fant = fact
xact = xant + h
fact = f(xact)

# Result delivery

a = xant
b = xact
iter = i
result_data_frame = pd.DataFrame(result_array)
return a, b, iter, result_data_frame

```

## Rust

```

use std::f64::EPSILON;

// Define a struct to hold the result for each iteration
#[derive(Debug)]
struct SearchResult {
    iteration: usize,
    x_i: f64,
    f_xi: f64,
    error: f64,
}

// Function for incremental search in Rust
fn incremental_search<F>(f: F, x0: f64, h: f64, nmax: usize) -> (f64, f64, usize, Vec<SearchResult>)
where
    F: Fn(f64) -> f64,
{
    // Initialization
    let mut xant = x0;

```

```

let mut fant = f(xant);
let mut xact = xant + h;
let mut fact = f(xact);

// Create a Vec to store the results
let mut result_array: Vec<SearchResult> = Vec::new();

// Iteration variable
let mut iterations = 0;

// Loop
for i in 1..=nmax {
    iterations = i;

    // Store the current iteration result
    let result = SearchResult {
        iteration: i,
        x_i: xact,
        f_xi: fact,
        error: (xact - xant).abs(),
    };
    result_array.push(result);

    // Check for sign change
    if fant * fact < 0.0 {
        break;
    }

    // Update
    xant = xact;
    fant = fact;
    xact = xant + h;
    fact = f(xact);

    // Stopping condition if the step size is too small
    if (xact - xant).abs() < EPSILON {
        break;
    }
}

// Return the result: interval endpoints, number of iterations, and the results
(xant, xact, iterations, result_array)
}

```

### 5.1.3 Method Tests

The test parameters are as follows:

- $f = \mathbf{math.log((math.sin(x)**2) + 1) - 1/2}$
- $x_0 = -3$
- $\text{step} = 0.5$
- $\text{Tol} = 1 \times 10^{-7}$
- $N = 100$

**Result:** Interval: [-2.5, -2.0], Iterations: 2

i	x <sub>i</sub>	f <sub>i</sub>	e
1	-2.5	-0.193863	0.5
2	-2.0	0.102578	0.5

## 5.2 Bisection

The bisection method is based on the Intermediate Value Theorem, which states that if a continuous function changes sign over an interval, there is at least one root in that interval. The method systematically reduces the interval in which the root lies by repeatedly bisecting it.

### 5.2.1 Pseudo-code

```
FUNCTION BisectionMethod(f, a, b, tolerance=1e-7, max_iterations
=100):

    INPUT:
    - f: function to evaluate
    - a, b: interval [a, b] where the root is sought
    - tolerance: stopping criterion (optional)
    - max_iterations: maximum number of iterations (optional)

    OUTPUT:
    - c: approximate value of the root
    - iterations: number of iterations performed
    - converged: indicator of whether the algorithm converged
    - df_result: table of results from the iterations

    IF f(a) * f(b) >= 0 THEN:
        Raise an error ("f(a) and f(b) must have opposite signs")
```

```

c = (a + b) / 2 // Initialize midpoint
results_list = [] // Initialize list to store iteration
results

FOR i FROM 0 TO max_iterations DO:
  c = (a + b) / 2 // Calculate the midpoint of [a, b]

  IF (b - a) < tolerance OR abs(f(c)) < tolerance THEN:
    Create a DataFrame with results_list // Convert the
    list to a table
    PRINT the DataFrame // Display table with iteration
    results
    RETURN (c, i + 1, True, DataFrame) // Algorithm has
    converged

  // Determine in which half of the interval the root is
  located
  IF f(a) * f(c) < 0 THEN:
    b = c // The root is in the interval [a, c]
  ELSE:
    a = c // The root is in the interval [c, b]

  // Store the current iteration's results
  result = { 'iteration': i, 'x_i': c, 'f(x_i)': f(c), '
    error': abs(b - a) }
  results_list.ADD(result)

  // If the maximum number of iterations is reached without
  convergence
  RETURN (c, max_iterations, False, results_list)

END FUNCTION

```

### 5.2.2 Method Implementation

#### Python

```

def bisection_method(f, a, b, tolerance=1e-7, max_iterations=100):
    """
    Bisection method to find the root of f(x) in the interval [a, b].

    Parameters:
    f : function
        The function for which we are trying to find a root.
    a, b : float
        The interval [a, b] in which the root is located.
    """

```

```

    tolerance : float, optional
        The stopping criterion for the algorithm.
    max_iterations : int, optional
        The maximum number of iterations to perform.

Returns:
    c : float
        The approximate root.
    iterations : int
        The number of iterations performed.
    converged : bool
        Whether the algorithm converged or not.
"""
if f(a) * f(b) >= 0:
    raise ValueError("f(a) and f(b) must have opposite signs.")
c = (a + b) / 2 # Midpoint initial
result_array = []
for i in range(max_iterations):
    c = (a + b) / 2 # Midpoint
    if abs(b - a) < tolerance or abs(f(c)) < tolerance:
        df_result = pd.DataFrame(data=result_array)
        print(df_result)
        return c, i + 1, True, df_result # Converged

    # Narrow the interval based on the sign of f(c)
    if f(a) * f(c) < 0:
        b = c # The root is in [a, c]
    else:
        a = c # The root is in [c, b]

    result = {'i': i, 'x_i': c, 'f_x_i': f(c), 'e': abs(b - a)}
    result_array.append(result)

return c, max_iterations, False # Did not converge within max_iterations

```

## Rust

```

#[derive(Debug)]
struct BisectionResult {
    iteration: usize,
    x_i: f64,
    f_x_i: f64,
    error: f64,
}

```



```

// Bisection method function in Rust
fn bisection_method<F>(
    f: F,
    mut a: f64,
    mut b: f64,
    tolerance: f64,
    max_iterations: usize,
) -> (f64, usize, bool, Vec<BisectionResult>)
where
    F: Fn(f64) -> f64,
{
    // Check if f(a) and f(b) have opposite signs
    if f(a) * f(b) >= 0.0 {
        panic!("f(a) and f(b) must have opposite signs.");
    }

    // Create a Vec to store the results for each iteration
    let mut result_array: Vec<BisectionResult> = Vec::new();

    let mut c = (a + b) / 2.0; // Midpoint initialization

    // Bisection loop
    for i in 0..max_iterations {
        c = (a + b) / 2.0; // Midpoint

        // Check if the solution has converged
        if (b - a).abs() < tolerance || f(c).abs() < tolerance {
            return (c, i + 1, true, result_array); // Converged
        }

        // Narrow the interval based on the sign of f(c)
        if f(a) * f(c) < 0.0 {
            b = c; // The root is in [a, c]
        } else {
            a = c; // The root is in [c, b]
        }

        // Store the result of the current iteration
        let result = BisectionResult {
            iteration: i + 1,
            x_i: c,
            f_x_i: f(c),
            error: (b - a).abs(),
        };
        result_array.push(result);
    }
}

```

```

        // If the loop finishes without converging, return the last midpoint and results
        (c, max_iterations, false, result_array) // Did not converge within max_iterations
    }

```

### 5.2.3 Method Test

The test parameters are as follows:

- $f = \mathbf{math.log((math.sin(x)**2) + 1) - 1/2}$
- $a = 0$
- $b = 1$
- $Tol = 1 \times 10^{-7}$
- $N = 100$

Result: - Root: 0.9364047050476074 - Iterations: 21 - Converged: True

i	x <sub>i</sub>	f <sub>x<sub>i</sub></sub>	e
0	0.5	-0.2931087267313766	0.5
1	0.75	-0.11839639385347844	0.25
2	0.875	-0.036817690757380506	0.125
3	0.9375	0.0006339161592386899	0.0625
4	0.90625	-0.017772289226861138	0.03125
5	0.921875	-0.008486582211768012	0.015625
6	0.9296875	-0.0039053586270640928	0.0078125
7	0.93359375	-0.0016304381170096915	0.00390625
8	0.935546875	-0.0004969353153196909	0.001953125
9	0.9365234375	6.882244496264622e-05	0.0009765625
10	0.93603515625	-0.00021397350516405567	0.00048828125
11	0.936279296875	-7.255478812057126e-05	0.000244140625
12	0.9364013671875	-1.860984900181606e-06	0.0001220703125
13	0.93646240234375	3.348202684883006e-05	6.103515625e-05
14	0.936431884765625	1.581084516011355e-05	3.0517578125e-05
15	0.9364166259765625	6.975011174192858e-06	1.52587890625e-05
16	0.9364089965820312	2.5570333977986692e-06	7.62939453125e-06
17	0.9364051818847656	3.4802931392352576e-07	3.814697265625e-06
18	0.9364032745361328	-7.564765268641693e-07	1.9073486328125e-06
19	0.9364042282104492	-2.042232898902263e-07	9.5367431640625e-07

### 5.3 False Rule

The false position method (also known as the regula falsi method) is a root-finding technique used in numerical analysis. It is similar to the bisection method in that it iteratively narrows down an interval where a root of a function exists. However, instead of using the midpoint of the interval as in the bisection method, the false position method uses a more refined estimate by linearly interpolating the function between the endpoints.

#### 5.3.1 Pseudo-code

```
FUNCTION FalsePosition(f, a, b, tol, Nmax):

    INPUT:
    - f: continuous function whose root is sought.
    - a: left endpoint of the initial interval.
    - b: right endpoint of the initial interval.
    - tol: tolerance that defines the stopping criterion (when the
          error is sufficiently small).
    - Nmax: maximum number of iterations allowed.

    OUTPUT:
    - x: approximate solution.
    - iter: number of iterations performed.
    - err: estimated error.
    - result_array: list of results with values for each iteration
                    (point, f(x) value, and error).

    BEGIN:
    1. Compute f(a) and f(b) // fa = f(a), fb = f(b)

    2. Compute the first midpoint using the false position formula
       :
       pm = (fb * a - fa * b) / (fb - fa) // pm is the first trial
       point

    3. Evaluate the function at the midpoint:
       fpm = f(pm)

    4. Initialize a large error value (E = 1000) to start the loop
       .

    5. Initialize an iteration counter (cont = 1).

    6. Create an empty list to store the results of each iteration
       (result_array).
```

7. While the error (E) is greater than the tolerance (tol) AND the number of iterations is less than Nmax:
  - a. If  $f(a) * f(pm) < 0$ :
    - The root is in the interval  $[a, pm]$ , so update  $b = pm$  and  $fb = fpm$ .
  - b. Else:
    - The root is in the interval  $[pm, b]$ , so update  $a = pm$  and  $fa = fpm$ .
  - c. Store the previous value of  $pm$  ( $p0 = pm$ ).
  - d. Compute a new midpoint using the false position formula:
 
$$pm = (fb * a - fa * b) / (fb - fa)$$
  - e. Evaluate the function at the new midpoint:
 
$$fpm = f(pm)$$
  - f. Compute the error as the difference between the new and previous midpoint:
 
$$E = abs(pm - p0)$$
  - g. Store the current iteration results (iteration number, point value,  $f(pm)$ , and error) in `result_array`.
  - h. Increment the iteration counter ( $cont += 1$ ).
8. When the loop ends (when the tolerance is met or the maximum number of iterations is reached):
  - The approximate solution is  $x = pm$ .
  - The number of iterations is  $iter = cont$ .
  - The estimated error is  $err = E$ .
9. Return the approximate solution, the number of iterations, the error, and the set of iteration results.

END FUNCTION

### 5.3.2 Method Implementation

#### Python

```
import math
import pandas as pd
```

```

def false_rule(f, a, b, tol, Nmax):
    """
        This program finds the solution to the equation  $f(x) = 0$  en el interval  $[a, b]$  using
        the false rule method.

        Inputs:
        f: continious function
        a: left endpoint of the initial interval
        b: right endpoint of the initial interval
        tol: tolerance
        Nmax: maximum number of iterations

        Outputs:
        x: aproximate solution
        iter: number of iterations
        err: estimated error
    """
    # Initialization
    fa = f(a)
    fb = f(b)
    pm = (fb * a - fa * b) / (fb - fa)
    fpm = f(pm)
    E = 1000 # Initial large error
    cont = 1 # Iteration counted

    # loop of the false rule method
    result_array = []
    while E > tol and cont < Nmax:
        if fa * fpm < 0:
            b = pm
            fb = fpm
        else:
            a = pm
            fa = fpm

        # Updating previous point
        p0 = pm
        pm = (fb * a - fa * b) / (fb - fa)
        fpm = f(pm)
        E = abs(pm - p0)

    result = {
        'i': cont,
        'x_i': pm,
        'f_xi': fpm,
    }

```

```

        'e': abs(pm - p0)
    }
    result_array.append(result)

    cont += 1

# Results
x = pm
iter = cont
err = E
return x, iter, err, result_array

```

## Rust

```

// Define a struct to store results for each iteration
#[derive(Debug)]
struct FalsePositionResult {
    iteration: usize,
    a: f64,
    b: f64,
    x_new: f64,
    f_x_new: f64,
    error: f64,
}

// False Position method in Rust
fn false_position<F>(
    f: F,
    mut a: f64,
    mut b: f64,
    tol: f64,
    max_iter: usize,
) -> (f64, Vec<FalsePositionResult>)
where
    F: Fn(f64) -> f64,
{
    // Check if f(a) and f(b) have opposite signs
    if f(a) * f(b) >= 0.0 {
        panic!("The function must have opposite signs at the endpoints a and b.");
    }

    // Create a Vec to store the results for each iteration
    let mut result_array: Vec<FalsePositionResult> = Vec::new();

    // False position method loop
    for i in 0..max_iter {

```

```

// Calculate the new point using the false position formula
let x_new = b - (f(b) * (b - a)) / (f(b) - f(a));
let f_x_new = f(x_new);

// Store the result of the current iteration
let result = FalsePositionResult {
    iteration: i + 1,
    a,
    b,
    x_new,
    f_x_new,
    error: f_x_new.abs(),
};
result_array.push(result);

// Check if the result is within the tolerance
if f_x_new.abs() < tol {
    return (x_new, result_array); // Converged
}

// Update the interval based on the sign of f(x_new)
if f(a) * f_x_new < 0.0 {
    b = x_new;
} else {
    a = x_new;
}

// If no solution is found within the given number of iterations
panic!("Maximum number of iterations reached without convergence.");
}

```

### 5.3.3 Method Tests

Approximate solution:

- $x = 0.9364045808798893$
- Iterations = 5
- Error =  $2.2097967899981086 \times 10^{-10}$

i	$x_i$	$f_{x_i}$	e
1	0.9365060516656253	5.875600835791861e-05	0.0025656709474095596
2	0.9364047307426415	8.67825411532408e-08	0.00010132092298376083
3	0.936404581100869	1.2815393191090152e-10	1.4964177252885236e-07
4	0.9364045808798893	1.894040480010517e-13	2.2097967899981086e-10

## 5.4 Fixed Point

The fixed-point method is an iterative numerical technique used to solve equations of the form  $x = g(x)$ . In this method, the goal is to find a value  $x$  such that  $g(x) = x$ , which is known as a fixed point of the function  $g(x)$ .

### 5.4.1 Pseudo-code

```

FUNCTION FixedPointMethod(g, x0, tol=1e-7, max_iter=1000):

    INPUT:
    - g: function for fixed-point iteration, where we seek  $x = g(x)$ .
    - x0: initial guess or starting value for the iteration.
    - tol: tolerance to determine if the method has converged (optional, default is 1e-7).
    - max_iter: maximum number of iterations allowed (optional, default is 1000).

    OUTPUT:
    - x: approximate solution to the fixed point.
    - iterations: number of iterations performed.
    - converged: a boolean indicating whether the method converged.
    - result_array: list containing details of each iteration (values of  $x$ ,  $g(x)$ , and error).

    BEGIN:
    1. Create an empty list to store results of each iteration (result_array).

    2. Initialize  $x$  with the initial value  $x_0$ .

    3. FOR  $i$  FROM 0 TO  $\text{max\_iter}$  DO:
        a. Compute the new value  $x_{\text{new}} = g(x)$ .

```



```

b. Compute the error as error = abs(x_new - x).

c. Create a dictionary with the current iteration results:
    - i: iteration number.
    - x: value of x_new.
    - g_x: value of g(x_new).
    - error: computed error.
    Add this dictionary to result_array.

d. IF the error is less than the tolerance (error < tol):
    - The method has converged.
    - Return x_new, number of iterations (i + 1), True (
      converged), and result_array.

e. Update the value of x with x_new.

4. IF the maximum number of iterations is reached without
convergence:
    - Return the last value of x, max_iter, False (not
      converged), and result_array.

END FUNCTION

```

#### 5.4.2 Method Implementation

##### Python

```

import pandas as pd

def fixed_point_method(g, x0, tol=1e-7, max_iter=1000):
    """
    Fixed-Point Iteration Method to find a solution to  $x = g(x)$ .

    Parameters:
    g : function
        The function to apply in the fixed-point iteration ( $x = g(x)$ ).
    x0 : float
        Initial guess for the fixed-point.
    tol : float, optional
        Tolerance for convergence. Default is 1e-7.
    max_iter : int, optional
        Maximum number of iterations. Default is 1000.

    Returns:

```

```

x : float
    The approximate fixed point.
iterations : int
    The number of iterations performed.
converged : bool
    Whether the method converged.
result_array : list of dict
    A list containing the iteration details.
"""
result_array = []
x = x0

for i in range(max_iter):
    x_new = g(x)
    error = abs(x_new - x)

    result = {
        'i': i + 1,
        'x': x_new,
        'g_x': g(x_new),
        'error': error
    }
    result_array.append(result)

    if error < tol:
        return x_new, i + 1, True, result_array # Converged

    x = x_new # Update for the next iteration

return x, max_iter, False, pd.DataFrame(result_array) # Did not converge within max

```

## Rust

```

use std::f64;

fn fixed_point_method<F>(g: F, x0: f64, tol: f64, max_iter: usize) -> (f64, usize, bool)
where
    F: Fn(f64) -> f64,
{
    let mut result_array = Vec::new();
    let mut x = x0;

    for i in 0..max_iter {
        let x_new = g(x);
        let error = (x_new - x).abs();
    }
}

```

```

        result_array.push((i + 1, x_new, g(x_new), error));

        if error < tol {
            return (x_new, i + 1, true, result_array); // Converged
        }

        x = x_new; // Update for the next iteration
    }

    // Did not converge within max_iter
    (x, max_iter, false, result_array)
}

fn print_results(result_array: &Vec<usize, f64, f64, f64>) {
    println!("{:<10} {:<15} {:<15} {:<15}", "Iter", "x", "g(x)", "Error");

    for (i, x, g_x, error) in result_array {
        println!("{:<10} {:<15.8} {:<15.8} {:<15.8}", i, x, g_x, error);
    }
}

fn main() {
    // Example usage:
    let g = |x: f64| x.cos(); // Example function g(x) = cos(x)
    let x0 = 0.5; // Initial guess
    let tol = 1e-7; // Tolerance for convergence
    let max_iter = 1000; // Maximum number of iterations

    let (x, iterations, converged, result_array) = fixed_point_method(g, x0, tol, max_iter);

    println!("Fixed point: {}", x);
    println!("Iterations: {}", iterations);
    println!("Converged: {}", converged);

    // Print results in table-like format
    print_results(&result_array);
}

```

### 5.4.3 Method Test

Approximate solution:

- Root found: -0.37444505296105535
- Iterations: 30

- Error: 7.726074024994034e-08

Iteration	x <sub>i</sub>	f(x <sub>i</sub> )	g(x <sub>i</sub> )	Error
1	-0.2931087267313766	-0.12671281687488073	-0.41982154360625734	0.2068912732686234
2	-0.41982154360625734	0.07351702442859243	-0.3463045191776649	0.12671281687488073
3	-0.3463045191776649	-0.044653937364644736	-0.39095845654230965	0.07351702442859243
4	-0.39095845654230965	0.026553421648170428	-0.3644050348941392	0.044653937364644736
5	-0.3644050348941392	-0.016021268273817058	-0.3804263031679563	0.026553421648170428
6	-0.3804263031679563	0.009589507887747428	-0.37083679528020885	0.016021268273817058
7	-0.37083679528020885	-0.005768850083372357	-0.3766056453635812	0.009589507887747428
8	-0.3766056453635812	0.003460227756392209	-0.373145417607189	0.005768850083372357
9	-0.373145417607189	-0.002079223579867173	-0.3752246411870562	0.003460227756392209
10	-0.3752246411870562	0.00124805513874654	-0.37397658604830963	0.002079223579867173
11	-0.37397658604830963	-0.0007496296601224861	-0.3747262157084321	0.00124805513874654
12	-0.3747262157084321	0.00045008239797816874	-0.37427613331045395	0.0007496296601224861
13	-0.37427613331045395	-0.00027029514763832196	-0.3745464284580923	0.00045008239797816874
14	-0.3745464284580923	0.0001623020232475736	-0.3743841264348447	0.00027029514763832196
15	-0.3743841264348447	-9.746439711039168e-05	-0.3744815908319551	0.0001623020232475736
16	-0.3744815908319551	5.8525648058027624e-05	-0.37442306518389706	9.746439711039168e-05
17	-0.37442306518389706	-3.514467880877392e-05	-0.37445820986270584	5.8525648058027624e-05
18	-0.37445820986270584	2.110401325022826e-05	-0.3744371058494556	3.514467880877392e-05
19	-0.3744371058494556	-1.2672877957420337e-05	-0.37444977872741303	2.110401325022826e-05
20	-0.37444977872741303	7.609964212673681e-06	-0.37444216876320036	1.2672877957420337e-05
21	-0.37444216876320036	-4.5697420043566694e-06	-0.3744467385052047	7.609964212673681e-06
22	-0.3744467385052047	2.744098679452467e-06	-0.37444399440652526	4.5697420043566694e-06
23	-0.37444399440652526	-1.647814738270359e-06	-0.37444564222126353	2.744098679452467e-06
24	-0.37444564222126353	9.895019896788426e-07	-0.37444465271927385	1.647814738270359e-06
25	-0.37444465271927385	-5.941897863737111e-07	-0.3744452469090602	9.895019896788426e-07
26	-0.3744452469090602	3.568071592630062e-07	-0.37444489010190096	5.941897863737111e-07
27	-0.37444489010190096	-2.1426045238026603e-07	-0.37444510436235334	3.568071592630062e-07
28	-0.37444510436235334	1.28662038245686e-07	-0.3744449757003151	2.1426045238026603e-07
29	-0.3744449757003151	-7.726074024994034e-08	-0.37444505296105535	1.28662038245686e-07
30	-0.37444505296105535	4.63945839523916e-08	-0.3744450065664714	7.726074024994034e-08

## 5.5 Newton

### 5.5.1 Pseudo-code

```
FUNCTION NewtonRaphson(f, df, x0, tol=1e-7, max_iter=30):
```

  INPUT:

- f: function whose zero (root) is to be found.
- df: derivative of the function f.
- x0: initial guess for the root.

- tol: tolerance for convergence (default value is  $1e-7$ ).
- max\_iter: maximum number of iterations allowed (default value is 30).

OUTPUT:

- x: approximate root of the function f.
- n: number of iterations performed.
- result\_array: table with details of each iteration.

BEGIN:

1. Initialize  $x_n = x_0$  (initial estimate).
2. Create an empty list result\_array to store details of the iterations.
3. For n from 0 to max\_iter - 1 DO:
  - a. Compute  $f_{xn} = f(x_n)$  and  $df_{xn} = df(x_n)$  (the function and its derivative evaluated at  $x_n$ ).
  - b. IF  $|x_n - (x_n - f_{xn} / df_{xn})| < \text{tol}$ :
    - Store in result\_array the details of the current iteration (n,  $x_i$ ,  $f(x_i)$ , error).
    - Print the DataFrame with iteration results.
    - Return  $x_n$  as the approximate root.
    - End the function.
  - c. IF  $df_{xn} == 0$  (the derivative is zero):
    - Print "Derivative is zero. No solution found."
    - Return None (indicating no solution found).
  - d. Store in result\_array the details of the current iteration (n,  $x_i$ ,  $f(x_i)$ , error).
  - e. Update  $x_n$  using the Newton-Raphson formula:
    - $x_n = x_n - f_{xn} / df_{xn}$ .
4. IF the maximum number of iterations is exceeded:
  - Print "Maximum number of iterations exceeded. No solution found."
  - Return None.

END FUNCTION

## 5.5.2 Method Implementation

### Python

```
def newton_raphson(f, df, x0, tol=1e-7, max_iter=30):
    """
    Solves  $f(x) = 0$  using the Newton-Raphson method.

    Parameters:
    - f: The function whose root we want to find.
    - df: The derivative of the function f.
    - x0: Initial guess for the root.
    - tol: The tolerance for convergence (default is 1e-6).
    - max_iter: Maximum number of iterations (default is 100).

    Returns:
    - The estimated root and the number of iterations taken.
    """
    xn = x0
    result_array = []
    for n in range(0, max_iter):
        fxn = f(xn)
        dfxn = df(xn)

        if abs(xn - (xn - fxn / dfxn)) < tol:
            result = {
                'i': n,
                'xi': xn,
                'f_xi': fxn,
                'E': abs(xn - (xn - fxn / dfxn))
            }
            result_array.append(result)
            print(f"Found solution after {n} iterations.")
            print(pd.DataFrame(result_array))
            return xn

        if dfxn == 0:
            print("Zero derivative. No solution found.")
            return None

    result = {
        'i': n,
        'xi': xn,
        'f_xi': fxn,
        'E': abs(xn - (xn - fxn / dfxn))
    }
    result_array.append(result)
```

```

        # Update the next approximation using Newton-Raphson formula
        xn = xn - fxn / dfxn
    print("Exceeded maximum iterations. No solution found.")
    return None

```

## Rust

```

use std::f64::EPSILON;

fn newton_raphson<F, DF>(f: F, df: DF, x0: f64, tol: f64, max_iter: usize)
-> Option<(f64, Vec<(usize, f64, f64, f64)>)>
where
    F: Fn(f64) -> f64,
    DF: Fn(f64) -> f64,
{
    let mut xn = x0;
    let mut result_array = Vec::new();

    for n in 0..max_iter {
        let fxn = f(xn);
        let dfxn = df(xn);

        // Avoid division by zero
        if dfxn.abs() < EPSILON {
            println!("Zero derivative encountered. No solution found.");
            return None;
        }

        // Calculate the error
        let error = (xn - (xn - fxn / dfxn)).abs();

        // Append iteration details to result_array
        result_array.push((n, xn, fxn, error));

        // Check if the solution has converged
        if error < tol {
            println!("Found solution after {} iterations.", n);
            return Some((xn, result_array));
        }

        // Update xn using Newton-Raphson formula
        xn = xn - fxn / dfxn;
    }

    println!("Exceeded maximum iterations. No solution found.");
    None
}

```

```

}

fn main() {
    // Example function  $f(x) = x^2 - 2$  (finding square root of 2)
    let f = |x: f64| x * x - 2.0;
    let df = |x: f64| 2.0 * x;

    let x0 = 1.0;
    let tol = 1e-7;
    let max_iter = 30;

    match newton_raphson(f, df, x0, tol, max_iter) {
        Some((root, result_array)) => {
            println!("Root: {}", root);
            println!("Iterations:");
            println!("| Iter | x | f(x) | Error |");
            println!("-----");
            for (i, xi, f_xi, e) in result_array {
                println!("| {:4} | {:5.4} | {:5.4} | {:7.4} |", i, xi, f_xi, e);
            }
        }
        None => println!("No solution found."),
    }
}

```

### 5.5.3 Method Test

- The root is: 0.9364045808526077
- Found solution after 4 iterations.

i:	xi:	f_xi:	E
0	1.2	0.12524132043913228	0.3464852988384717
1	0.8535147011615283	-0.05025900826059804	0.07953835548275334
2	0.9330530566442816	-0.0019446986420873502	0.003344827656164062
3	0.9363978843004457	-3.877863362977685e-06	6.696552162011038e-06
4	0.9364045808526077	-1.5608903058961232e-11	2.6954660725664326e-11

## 5.6 Secant

### 5.6.1 Pseudo-code

```
FUNCTION SecantMethod(f, x0, x1, tol=1e-7, max_iter=100)
```

```
    INPUT:
```



- f: function for which we want to find a root.
- x0: first initial value.
- x1: second initial value.
- tol: tolerance to determine convergence (optional, default value 1e-7).
- max\_iter: maximum number of iterations allowed (optional, default value 100).

OUTPUT:

- The approximate root of the function f or an error message if it does not converge.

Initialize an empty list called results.

FOR i FROM 0 TO max\_iter - 1 DO:

1. Compute f(x0) and f(x1).
2. IF f(x1) is equal to f(x0), throw an error: "Division by zero".
3. Use the secant method formula to compute x2:  

$$x2 = x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0)).$$
4. IF  $|x2 - x1| < tol$ , DO the following:
  - Add to the results list the index i, the value x1, the value f(x2), and the error  $|x2 - x1|$ .
  - Print the results list.
  - Print: "Converged after i + 1 iterations."
  - Return x2 (the root).
5. IF not converged, add the current result to the list with the index i, the value x1, the value f(x2), and the error  $|x2 - x1|$ .
6. Update  $x0 = x1$  and  $x1 = x2$  for the next iteration.

IF the maximum number of iterations is reached, throw an error:  
 "The secant method did not converge within the maximum number of iterations."

END FUNCTION

## 5.6.2 Method Implementation

Python

```
def secant_method(f, x0, x1, tol=1e-7, max_iter=100):
    """
    Secant method to find the root of a function f.

    Parameters:
```

```

f : function
    The function for which we are trying to find a root.
x0 : float
    Initial guess 1.
x1 : float
    Initial guess 2.
tol : float, optional
    Tolerance for stopping the iteration. Default is 1e-6.
max_iter : int, optional
    Maximum number of iterations. Default is 100.

Returns:
float
    The root of the function f.
"""
result_array = []

for i in range(max_iter):
    # Calculate the value of f at the two initial guesses
    f_x0 = f(x0)
    f_x1 = f(x1)

    # Avoid division by zero
    if f_x1 == f_x0:
        raise ValueError("Division by zero encountered in the secant method.")

    # Secant method formula
    x2 = x1 - f_x1 * (x1 - x0) / (f_x1 - f_x0)

    # Check for convergence
    if abs(x2 - x1) < tol:
        result = {'i': i,
                  'x_i': x1,
                  'f_xi': x2,
                  'e': abs(x2 - x1)}
        result_array.append(result)
        print(DataFrame(result_array))
        print(f"Converged after {i + 1} iterations.")
        return x2

    # add results to list
    result = {'i': i,
              'x_i': x1,
              'f_xi': x2,
              'e': abs(x2 - x1)}
    result_array.append(result)

```

```

        # Update guesses
        x0, x1 = x1, x2

    raise ValueError("Secant method did not converge within the maximum number of iterations")

```

## Rust

```

fn secant_method<F>(f: F, x0: f64, x1: f64, tol: f64, max_iter: usize) -> Result<(f64, Vec<(i32, f64, f64, f64)>)> {
    where
        F: Fn(f64) -> f64,
    {
        let mut x0 = x0;
        let mut x1 = x1;
        let mut result_array = Vec::new();

        for i in 0..max_iter {
            let f_x0 = f(x0);
            let f_x1 = f(x1);

            // Avoid division by zero
            if f_x1 == f_x0 {
                return Err(String::from("Division by zero encountered in the secant method"));
            }

            // Secant method formula
            let x2 = x1 - f_x1 * (x1 - x0) / (f_x1 - f_x0);

            // Check for convergence
            let error = (x2 - x1).abs();
            result_array.push((i, x1, x2, error));

            if error < tol {
                println!("Converged after {} iterations.", i + 1);
                return Ok((x2, result_array));
            }

            // Update guesses for the next iteration
            x0 = x1;
            x1 = x2;
        }

        Err(String::from("Secant method did not converge within the maximum number of iterations"))
    }
}

fn main() {

```

```

// Example function f(x) = x^2 - 2 (finding square root of 2)
let f = |x: f64| x * x - 2.0;

let x0 = 1.0;
let x1 = 2.0;
let tol = 1e-7;
let max_iter = 100;

match secant_method(f, x0, x1, tol, max_iter) {
  Ok((root, result_array)) => {
    println!("Root: {}", root);
    println!("Iterations:");
    println!("| Iter | x_i | f(x_i) | Error |");
    println!("-----");
    for (i, xi, f_xi, e) in result_array {
      println!("| {:4} | {:.7.4} | {:.7.4} | {:.7.4} |", i, xi, f_xi, e);
    }
    Err(err) => println!("{}", err),
  }
}

```

### 5.6.3 Method Test

- Converged after 5 iterations.
- The root is: 0.9364045808795616

i	x <sub>i</sub>	f <sub>xi</sub>	e
0	1.0	0.946166222306525	0.05383377769347497
1	0.946166222306525	0.9359965807911726	0.010169641515352379
2	0.9359965807911726	0.9364070023767038	0.00041042158553117325
3	0.9364070023767038	0.9364045814731196	2.420903584265943e-06
4	0.9364045814731196	0.9364045808795616	5.935579805438351e-10

## 5.7 Multiple Roots

### 5.7.1 Pseudo-code

```

FUNCTION MultipleRootMethod(f, df, ddf, x0, tol=1e-7, max_iter
=100):

INPUT:
- f: function for which the root is sought.
- df: derivative of the function f.

```

- ddf: second derivative of the function f.
- x0: initial approximation of the root.
- tol: tolerance for convergence (optional, default value  $1e-7$ ).
- max\_iter: maximum number of iterations allowed (optional, default value 100).

OUTPUT:

- x: approximate root of the function f.
- iterations: number of iterations performed.
- data\_frame: table with details of each iteration (optional).

BEGIN:

1. Create an empty list to store results of each iteration (result\_array).
2. FOR i from 0 to max\_iter DO:
  - a. Evaluate  $f(x_0)$ ,  $f'(x_0)$  and  $f''(x_0)$ :
    - $f_{x0} = f(x_0)$
    - $df_{x0} = df(x_0)$
    - $ddf_{x0} = ddf(x_0)$
  - b. Check if the denominator is zero:
    - IF  $[df_{x0}^2 - f_{x0} * ddf_{x0}]$  equals 0, throw an error ("Division by zero in multiple root method").
  - c. Calculate the new value of x using the multiple root method formula:
    - $x_1 = x_0 - (f_{x0} * df_{x0}) / (df_{x0}^2 - f_{x0} * ddf_{x0})$
  - d. Calculate the error as the absolute difference:
    - $error = abs(x_1 - x_0)$
  - e. Store the results of the current iteration in the list:
    - Save (i, x0, f\_x0, df\_x0, ddf\_x0, x1, error) in result\_array.
  - f. Check if the error is less than the tolerance (tol):
    - IF error < tol:
      - Create a DataFrame with the results from result\_array.
      - Print the DataFrame.
      - Return x1, number of iterations (i + 1), and the DataFrame.

```

g. Update x0 for the next iteration:
    - x0 = x1

3. IF the maximum number of iterations is reached without
   converging, throw an error ("The multiple root method did
   not converge within the maximum number of iterations").

END FUNCTION

```

## 5.7.2 Method Implementation

### Python

```

def multiple_root_method(f, df, ddf, x0, tol=1e-7, max_iter=100):
    """
    Multiple Root Method to find the root of a function f where the root has multiplicity.

    Parameters:
    f : function
        The function whose root we want to find.
    df : function
        The derivative of the function f.
    ddf : function
        The second derivative of the function f.
    x0 : float
        Initial guess for the root.
    tol : float, optional
        Tolerance for stopping the iteration. Default is 1e-7.
    max_iter : int, optional
        Maximum number of iterations. Default is 100.

    Returns:
    float
        The root of the function f.
    int
        The number of iterations performed.
    pd.DataFrame
        DataFrame with iteration details.
    """
    result_array = []

    for i in range(max_iter):
        f_x0 = f(x0)
        df_x0 = df(x0)
        ddf_x0 = ddf(x0)

```

```

if df_x0**2 - f_x0 * ddf_x0 == 0:
    raise ValueError("Division by zero encountered in the multiple root method.")

# Multiple Root Method formula
x1 = x0 - (f_x0 * df_x0) / (df_x0**2 - f_x0 * ddf_x0)

# Check for convergence
error = abs(x1 - x0)

# Store iteration details in result_array
result = {
    'i': i,
    'x_i': x0,
    'f(x_i)': f_x0,
    'df(x_i)': df_x0,
    'ddf(x_i)': ddf_x0,
    'x_(i+1)': x1,
    'Error': error
}
result_array.append(result)

if error < tol:
    df_result = pd.DataFrame(result_array)
    print(df_result)
    print(f"Converged after {i + 1} iterations.")
    return x1, i + 1, df_result

# Update x0 for next iteration
x0 = x1

raise ValueError("Multiple root method did not converge within the maximum number of

```

## Rust

```

use std::vec::Vec;

#[derive(Debug)]
struct IterationResult {
    i: usize,
    x_i: f64,
    f_xi: f64,
    df_xi: f64,
    ddf_xi: f64,
    x_next: f64,
    error: f64,
}

```

```

}

fn multiple_root_method(
    f: &dyn Fn(f64) -> f64,
    df: &dyn Fn(f64) -> f64,
    ddf: &dyn Fn(f64) -> f64,
    x0: f64,
    tol: f64,
    max_iter: usize,
) -> Result<(f64, usize, Vec<IterationResult>), &'static str> {
    let mut x0 = x0;
    let mut result_array: Vec<IterationResult> = Vec::new();

    for i in 0..max_iter {
        let f_x0 = f(x0);
        let df_x0 = df(x0);
        let ddf_x0 = ddf(x0);

        if df_x0.powi(2) - f_x0 * ddf_x0 == 0.0 {
            return Err("Division by zero encountered in the multiple root method.");
        }

        // Multiple Root Method formula
        let x1 = x0 - (f_x0 * df_x0) / (df_x0.powi(2) - f_x0 * ddf_x0);

        // Check for convergence
        let error = (x1 - x0).abs();

        // Store iteration details in result_array
        let result = IterationResult {
            i,
            x_i: x0,
            f_xi: f_x0,
            df_xi: df_x0,
            ddf_xi: ddf_x0,
            x_next: x1,
            error,
        };
        result_array.push(result);

        if error < tol {
            println!("Converged after {} iterations.", i + 1);
            return Ok((x1, i + 1, result_array));
        }

        // Update x0 for next iteration
    }
}

```



```

        x0 = x1;
    }

    Err("Multiple root method did not converge within the maximum number of iterations.")
}

fn main() {
    // Example function:  $f(x) = x^3 - 3x^2 + 3x - 1$ 
    // (has a root with multiplicity at  $x = 1$ )
    let f = |x: f64| x.powi(3) - 3.0 * x.powi(2) + 3.0 * x - 1.0;
    let df = |x: f64| 3.0 * x.powi(2) - 6.0 * x + 3.0; // First derivative of  $f(x)$ 
    let ddf = |x: f64| 6.0 * x - 6.0; // Second derivative of  $f(x)$ 

    let x0 = 0.5; // Initial guess
    let tol = 1e-7;
    let max_iter = 100;

    match multiple_root_method(&f, &df, &ddf, x0, tol, max_iter) {
        Ok((root, iterations, result_array)) => {
            println!("Root found: {}", root);
            println!("Number of iterations: {}", iterations);

            // Print result_array like a DataFrame
            println!("{: <4} {: <10} {: <10} {: <10} {: <10} {: <10} {: <10}",
                "i", "x_i", "f(x_i)", "df(x_i)", "ddf(x_i)", "x_(i+1)", "Error");
            for result in result_array {
                println!("{: <4} {: <10.6} {: <10.6} {: <10.6} {: <10.6} {: <10.6} {: <10.6}",
                    result.i, result.x_i, result.f_xi, result.df_xi, result.ddf_xi, result.x_i1, result.e);
            }
        }
        Err(e) => {
            println!("{}", e);
        }
    }
}

```

### 5.7.3 Method Test

- Converged after 5 iterations.
- Root found: -4.218590698935789e-11

i	$x_i$	$f(x_i)$	Error
0	1.0	0.7182818284590451	1.2342106135535142
1	-0.23421061355351425	0.025405775475345838	0.22575233364275316
2	-0.00845827991076109	3.567060801401567e-05	0.008446389726952502
3	-1.1890183808588653e-05	7.068789997788372e-11	1.1890141622681664e-05
4	-4.218590698935789e-11	0.0	0.0

## 5.8 Gaussian Elimination No Pivot

### 5.8.1 Pseudo-code

```

FUNCTION GaussianEliminationWithoutPivoting(A, b):

    INPUT:
    - A: coefficient matrix (nxn).
    - b: right-hand side vector (nx1).

    OUTPUT:
    - x: solution vector to the system of equations Ax = b.

    BEGIN:
    1. Convert A and b to float matrices, if necessary.

    2. Create the augmented matrix by combining A with b:
       - AugmentedMatrix = [A | b]

    3. FOR i from 0 to n-1 DO: (n is the length of b)

        a. Check if the diagonal pivot AugmentedMatrix[i, i] is 0.
           If so, throw an error ("Zero pivot found in row i").

        b. **Forward elimination**: For each row j from i+1 to n
           -1:
           - Calculate the elimination factor: factor =
             AugmentedMatrix[j, i] / AugmentedMatrix[i, i].
           - Subtract factor * row i from row j to eliminate the
             elements below the diagonal in column i.

    4. **Back substitution**:

        a. Initialize the solution vector x of length n with zeros
           .

        b. FOR i from n-1 to 0 DO: (traverse from bottom to top)
           - Calculate x[i] using the formula:

```

```

        x[i] = (AugmentedMatrix[i, -1] - sum of the products
                of the already solved elements) / AugmentedMatrix[
                i, i]

    5. Return the solution vector x.

END FUNCTION

```

## 5.8.2 Method Implementation

### Python

```

def gaussian_elimination_no_pivoting(A, b):
    """
    Solves the system of linear equations  $Ax = b$  using Gaussian Elimination without Pivoting.

    Parameters:
    A (list of lists or np.ndarray): Coefficient matrix.
    b (list or np.ndarray): Right-hand side vector.

    Returns:
    np.ndarray: Solution vector x.
    """
    # Convert A and b into numpy arrays
    A = np.array(A, float)
    b = np.array(b, float)
    n = len(b)

    # Augment A with b to form the augmented matrix
    augmented_matrix = np.hstack([A, b.reshape(-1, 1)])

    # Forward elimination (without pivoting)
    for i in range(n):
        # Check if the diagonal element is zero, which would lead to division by zero
        if augmented_matrix[i, i] == 0:
            raise ValueError(f"Zero pivot encountered at row {i}. No pivoting applied.")

        # Eliminate entries below the pivot
        for j in range(i + 1, n):
            factor = augmented_matrix[j, i] / augmented_matrix[i, i]
            augmented_matrix[j, i:] -= factor * augmented_matrix[i, i:]

    # Back substitution to solve for x
    x = np.zeros(n)
    for i in range(n - 1, -1, -1):
        x[i] = (augmented_matrix[i, -1] - np.dot(augmented_matrix[i, i + 1:], x[i + 1:]))

```

```
return x
```

## Rust

```
fn gaussian_elimination_no_pivoting(a: &mut Vec<Vec<f64>>, b: &mut Vec<f64>) -> Vec<f64> {
    let n = b.len();

    // Forward elimination (without pivoting)
    for i in 0..n {
        // Check if the diagonal element is zero, which would lead to division by zero
        if a[i][i] == 0.0 {
            panic!("Zero pivot encountered at row {}. No pivoting applied.", i);
        }

        // Eliminate entries below the pivot
        for j in i + 1..n {
            let factor = a[j][i] / a[i][i];
            for k in i..n {
                a[j][k] -= factor * a[i][k];
            }
            b[j] -= factor * b[i];
        }
    }

    // Back substitution to solve for x
    let mut x = vec![0.0; n];
    for i in (0..n).rev() {
        x[i] = b[i];
        for j in i + 1..n {
            x[i] -= a[i][j] * x[j];
        }
        x[i] /= a[i][i];
    }

    x
}

fn main() {
    // Example usage:
    let mut a = vec![
        vec![2.0, 1.0, -1.0],
        vec![-3.0, -1.0, 2.0],
        vec![-2.0, 1.0, 2.0],
    ];
    let mut b = vec![8.0, -11.0, -3.0];
}
```

```

    let solution = gaussian_elimination_no_pivoting(&mut a, &mut b);
    println!("Solution: {:?}", solution);
}

```

### 5.8.3 Method Test

```

Initial augmented matrix:
[[ 2. -1. 0. 3. 1. ]
 [ 1. 0.5 3. 8. 1. ]
 [ 0. 13. -2. 11. 1. ]
 [14. 5. -2. 3. 1. ]]
=====
Augmented matrix after eliminating row 0:
[[ 2. -1. 0. 3. 1. ]
 [ 0. 1. 3. 6.5 0.5]
 [ 0. 13. -2. 11. 1. ]
 [ 0. 12. -2. -18. -6. ]]
=====
Augmented matrix after eliminating row 1:
[[ 2. -1. 0. 3. 1. ]
 [ 0. 1. 3. 6.5 0.5]
 [ 0. 0. -41. -73.5 -5.5]
 [ 0. 0. -38. -96. -12. ]]
=====
Augmented matrix after eliminating row 2:
[[ 2. -1. 0. 3. 1. ]
 [ 0. 1. 3. 6.5 0.5 ]
 [ 0. 0. -41. -73.5 -5.5 ]
 [ 0. 0. 0. -27.87804878 -6.90243902]]
=====
Augmented matrix after eliminating row 3:
[[ 2. -1. 0. 3. 1. ]
 [ 0. 1. 3. 6.5 0.5 ]
 [ 0. 0. -41. -73.5 -5.5 ]
 [ 0. 0. 0. -27.87804878 -6.90243902]]
=====
Solution vector x:
[ 0.03849519 -0.18022747 -0.30971129 0.24759405]
=====

```

## 5.9 Gaussian Elimination Partial Pivot

### 5.9.1 Pseudo-code

```
FUNCTION GaussianEliminationWithPartialPivoting(A, b):

    INPUT:
    - A: coefficient matrix (nxn).
    - b: right-hand side vector (nx1).

    OUTPUT:
    - x: solution vector to the system of equations  $Ax = b$ .

    BEGIN:
    1. Convert A and b to float matrices, if necessary.

    2. Create the augmented matrix by combining A with b:
       - AugmentedMatrix = [A | b]

    3. FOR i from 0 to n-1 DO: (n is the length of b)

        a. **Partial Pivoting**: Find the row with the largest
           absolute value in column i from row i to the last.
           - max_row = index of the row with the maximum value in
             column i.
           - If the value in AugmentedMatrix[max_row, i] is 0,
             throw an error ("The matrix is singular or nearly
             singular").

        b. If the row max_row is not equal to i:
           - Swap row i with row max_row.

        c. **Forward elimination**: For each row j from i+1 to n
           -1:
           - Calculate the elimination factor: factor =
             AugmentedMatrix[j, i] / AugmentedMatrix[i, i].
           - Subtract factor * row i from row j to eliminate the
             elements below the diagonal in column i.

    4. **Back substitution**:

        a. Initialize the solution vector x of length n with zeros
           .

        b. FOR i from n-1 to 0 DO: (traverse from bottom to top)
           - Calculate x[i] using the formula:
```

```

        x[i] = (AugmentedMatrix[i, -1] - sum of the products
                of the already solved elements) / AugmentedMatrix[
                i, i]

    5. Return the solution vector x.

END FUNCTION

```

### 5.9.2 Method Implementation

#### Python

```

def gaussian_elimination_with_partial_pivoting(A, b):
    """
    Solves the system of linear equations  $Ax = b$  using Gaussian Elimination with Partial Pivoting.

    Parameters:
    A (list of lists or np.ndarray): Coefficient matrix.
    b (list or np.ndarray): Right-hand side vector.

    Returns:
    np.ndarray: Solution vector x.
    """
    # Convert A and b to numpy arrays
    A = np.array(A, float)
    b = np.array(b, float)
    n = len(b)

    # Augment A with b to form the augmented matrix
    augmented_matrix = np.hstack([A, b.reshape(-1, 1)])

    # Perform Gaussian elimination with partial pivoting
    for i in range(n):
        # Partial Pivoting: Find the row with the largest value in the current column
        max_row = np.argmax(abs(augmented_matrix[i:, i])) + i
        if augmented_matrix[max_row, i] == 0:
            raise ValueError("Matrix is singular or nearly singular")

        # Swap the current row with the row having the largest pivot element
        if max_row != i:
            augmented_matrix[[i, max_row]] = augmented_matrix[[max_row, i]]

        # Eliminate values below the pivot
        for j in range(i + 1, n):
            factor = augmented_matrix[j, i] / augmented_matrix[i, i]
            augmented_matrix[j, i:] -= factor * augmented_matrix[i, i:]

```

```

# Back substitution to solve for x
x = np.zeros(n)
for i in range(n - 1, -1, -1):
    x[i] = (augmented_matrix[i, -1] - np.dot(augmented_matrix[i, i + 1:n],
        x[i + 1:])) / augmented_matrix[i, i]

return x

```

## Rust

```

fn gaussian_elimination_with_partial_pivoting(a: &mut Vec<Vec<f64>>, b: &mut Vec<f64>) {
    let n = b.len();

    // Perform Gaussian elimination with partial pivoting
    for i in 0..n {
        // Partial Pivoting: Find the row with the largest value in the current column
        let mut max_row = i;
        for k in i + 1..n {
            if a[k][i].abs() > a[max_row][i].abs() {
                max_row = k;
            }
        }

        // Check if the matrix is singular or nearly singular
        if a[max_row][i].abs() == 0.0 {
            panic!("Matrix is singular or nearly singular");
        }

        // Swap the current row with the row having the largest pivot element
        if max_row != i {
            a.swap(i, max_row);
            b.swap(i, max_row);
        }

        // Eliminate values below the pivot
        for j in i + 1..n {
            let factor = a[j][i] / a[i][i];
            for k in i..n {
                a[j][k] -= factor * a[i][k];
            }
            b[j] -= factor * b[i];
        }
    }

    // Back substitution to solve for x

```



```

    let mut x = vec![0.0; n];
    for i in (0..n).rev() {
        x[i] = b[i];
        for j in i + 1..n {
            x[i] -= a[i][j] * x[j];
        }
        x[i] /= a[i][i];
    }

    x

}

fn main() {
    // Example usage:
    let mut a = vec![
        vec![2.0, 1.0, -1.0],
        vec![-3.0, -1.0, 2.0],
        vec![-2.0, 1.0, 2.0],
    ];
    let mut b = vec![8.0, -11.0, -3.0];

    let solution = gaussian_elimination_with_partial_pivoting(&mut a, &mut b);
    println!("Solution: {:?}", solution);
}

```

### 5.9.3 Method Test

```

[[ 2. -1. 0. 3. 1. ]
[ 1. 0.5 3. 8. 1. ]
[ 0. 13. -2. 11. 1. ]
[14. 5. -2. 3. 1. ]]

=====
Augmented matrix after swapping row 0 with row 3:
[[14. 5. -2. 3. 1. ]
[ 1. 0.5 3. 8. 1. ]
[ 0. 13. -2. 11. 1. ]
[ 2. -1. 0. 3. 1. ]]

=====
Augmented matrix after eliminating row 3:
[[14. 5. -2. 3. 1. ]
[ 0. 0.14285714 3.14285714 7.78571429 0.92857143]
[ 0. 13. -2. 11. 1. ]
[ 0. -1.71428571 0.28571429 2.57142857 0.85714286]]

=====
Augmented matrix after swapping row 1 with row 2:

```

```

[[14. 5. -2. 3. 1. ]
 [ 0. 13. -2. 11. 1. ]
 [ 0. 0.14285714 3.14285714 7.78571429 0.92857143]
 [ 0. -1.71428571 0.28571429 2.57142857 0.85714286]]
=====
Augmented matrix after eliminating row 3:
[[ 1.40000000e+01 5.00000000e+00 -2.00000000e+00 3.00000000e+00
  1.00000000e+00]
 [ 0.00000000e+00 1.30000000e+01 -2.00000000e+00 1.10000000e+01
  1.00000000e+00]
 [ 0.00000000e+00 0.00000000e+00 3.16483516e+00 7.66483516e+00
  9.17582418e-01]
 [ 0.00000000e+00 2.22044605e-16 2.19780220e-02 4.02197802e+00
  9.89010989e-01]]
=====
Augmented matrix after swapping row 2 with row 2:
[[ 1.40000000e+01 5.00000000e+00 -2.00000000e+00 3.00000000e+00
  1.00000000e+00]
 [ 0.00000000e+00 1.30000000e+01 -2.00000000e+00 1.10000000e+01
  1.00000000e+00]
 [ 0.00000000e+00 0.00000000e+00 3.16483516e+00 7.66483516e+00
  9.17582418e-01]
 [ 0.00000000e+00 2.22044605e-16 2.19780220e-02 4.02197802e+00
  9.89010989e-01]]
=====
Augmented matrix after eliminating row 3:
[[ 1.40000000e+01 5.00000000e+00 -2.00000000e+00 3.00000000e+00
  1.00000000e+00]
 [ 0.00000000e+00 1.30000000e+01 -2.00000000e+00 1.10000000e+01
  1.00000000e+00]
 [ 0.00000000e+00 0.00000000e+00 3.16483516e+00 7.66483516e+00
  9.17582418e-01]
 [ 0.00000000e+00 2.22044605e-16 0.00000000e+00 3.96875000e+00
  9.82638889e-01]]
=====
Augmented matrix after swapping row 3 with row 3:
[[ 1.40000000e+01 5.00000000e+00 -2.00000000e+00 3.00000000e+00
  1.00000000e+00]
 [ 0.00000000e+00 1.30000000e+01 -2.00000000e+00 1.10000000e+01
  1.00000000e+00]
 [ 0.00000000e+00 0.00000000e+00 3.16483516e+00 7.66483516e+00
  9.17582418e-01]
 [ 0.00000000e+00 2.22044605e-16 0.00000000e+00 3.96875000e+00
  9.82638889e-01]]
=====
Augmented matrix after eliminating row 3:

```

```

[[ 1.40000000e+01  5.00000000e+00 -2.00000000e+00  3.00000000e+00
  1.00000000e+00]
 [ 0.00000000e+00  1.30000000e+01 -2.00000000e+00  1.10000000e+01
  1.00000000e+00]
 [ 0.00000000e+00  0.00000000e+00  3.16483516e+00  7.66483516e+00
  9.17582418e-01]
 [ 0.00000000e+00  2.22044605e-16  0.00000000e+00  3.96875000e+00
  9.82638889e-01]]
=====
Solution vector x:
[ 0.03849519 -0.18022747 -0.30971129  0.24759405]
=====
Solution: [ 0.03849519 -0.18022747 -0.30971129  0.24759405]

```

## 5.10 Gaussian Elimination

### 5.10.1 Pseudo-code

### 5.10.2 Method Implementation

#### Python

```

def gaussian_elimination(A, b):
    """
    Solves the system of linear equations  $Ax = b$  using Gaussian Elimination.

    Parameters:
    A (list of list or np.ndarray): Coefficient matrix.
    b (list or np.ndarray): Right-hand side vector.

    Returns:
    np.ndarray: Solution vector x.
    """
    # Convert A and b into augmented matrix
    A = np.array(A, float)
    b = np.array(b, float)
    n = len(b)

    # Augment A with b
    augmented_matrix = np.hstack([A, b.reshape(-1, 1)])

    # Forward Elimination: Transform to upper triangular form
    for i in range(n):
        # Partial Pivoting: Swap rows if needed
        max_row = np.argmax(abs(augmented_matrix[i:, i])) + i
        if augmented_matrix[max_row, i] == 0:

```

```

        raise ValueError("Matrix is singular or nearly singular")
    augmented_matrix[[i, max_row]] = augmented_matrix[[max_row, i]]

    # Eliminate the below rows
    for j in range(i + 1, n):
        factor = augmented_matrix[j, i] / augmented_matrix[i, i]
        augmented_matrix[j, i:] -= factor * augmented_matrix[i, i:]

    # Back Substitution: Solve the upper triangular system
    x = np.zeros(n)
    for i in range(n - 1, -1, -1):
        x[i] = (augmented_matrix[i, -1] - np.dot(augmented_matrix[i, i + 1:n], x[i + 1:n])) / augmented_matrix[i, i]

    return x

```

## Rust

```

fn gaussian_elimination(a: &mut Vec<Vec<f64>>, b: &mut Vec<f64>) -> Vec<f64> {
    let n = b.len();

    // Forward Elimination: Transform to upper triangular form
    for i in 0..n {
        // Partial Pivoting: Swap rows if needed
        let mut max_row = i;
        for k in i + 1..n {
            if a[k][i].abs() > a[max_row][i].abs() {
                max_row = k;
            }
        }

        if a[max_row][i].abs() == 0.0 {
            panic!("Matrix is singular or nearly singular");
        }

        // Swap rows in both A and b
        a.swap(i, max_row);
        b.swap(i, max_row);

        // Eliminate the below rows
        for j in i + 1..n {
            let factor = a[j][i] / a[i][i];
            for k in i..n {
                a[j][k] -= factor * a[i][k];
            }
            b[j] -= factor * b[i];
        }
    }
}

```

```

    }

    // Back Substitution: Solve the upper triangular system
    let mut x = vec![0.0; n];
    for i in (0..n).rev() {
        x[i] = b[i];
        for j in i + 1..n {
            x[i] -= a[i][j] * x[j];
        }
        x[i] /= a[i][i];
    }

    x
}

fn main() {
    // Example usage:
    let mut a = vec![
        vec![2.0, 1.0, -1.0],
        vec![-3.0, -1.0, 2.0],
        vec![-2.0, 1.0, 2.0],
    ];
    let mut b = vec![8.0, -11.0, -3.0];

    let solution = gaussian_elimination(&mut a, &mut b);
    println!("Solution: {:?}", solution);
}

```

### 5.10.3 Method Test

```

Initial augmented matrix:
[[ 2. -1. 0. 3. 1. ]
 [ 1. 0.5 3. 8. 1. ]
 [ 0. 13. -2. 11. 1. ]
 [14. 5. -2. 3. 1. ]]

=====
Augmented matrix after swapping row 0 with row 3:
[[14. 5. -2. 3. 1. ]
 [ 1. 0.5 3. 8. 1. ]
 [ 0. 13. -2. 11. 1. ]
 [ 2. -1. 0. 3. 1. ]]

=====
Augmented matrix after eliminating row 1:
[[14. 5. -2. 3. 1. ]
 [ 0. 0.14285714 3.14285714 7.78571429 0.92857143]

```

```

[ 0. 13. -2. 11. 1. ]
[ 2. -1. 0. 3. 1. ]]
=====
Augmented matrix after eliminating row 2:
[[14. 5. -2. 3. 1. ]
 [ 0. 0.14285714 3.14285714 7.78571429 0.92857143]
 [ 0. 13. -2. 11. 1. ]
 [ 2. -1. 0. 3. 1. ]]
=====
Augmented matrix after eliminating row 3:
[[14. 5. -2. 3. 1. ]
 [ 0. 0.14285714 3.14285714 7.78571429 0.92857143]
 [ 0. 13. -2. 11. 1. ]
 [ 0. -1.71428571 0.28571429 2.57142857 0.85714286]]
=====
Augmented matrix after swapping row 1 with row 2:
[[14. 5. -2. 3. 1. ]
 [ 0. 13. -2. 11. 1. ]
 [ 0. 0.14285714 3.14285714 7.78571429 0.92857143]
 [ 0. -1.71428571 0.28571429 2.57142857 0.85714286]]
=====
Augmented matrix after eliminating row 2:
[[14. 5. -2. 3. 1. ]
 [ 0. 13. -2. 11. 1. ]
 [ 0. 0. 3.16483516 7.66483516 0.91758242]
 [ 0. -1.71428571 0.28571429 2.57142857 0.85714286]]
=====
Augmented matrix after eliminating row 3:
[[ 1.40000000e+01 5.00000000e+00 -2.00000000e+00 3.00000000e+00
 1.00000000e+00]
 [ 0.00000000e+00 1.30000000e+01 -2.00000000e+00 1.10000000e+01
 1.00000000e+00]
 [ 0.00000000e+00 0.00000000e+00 3.16483516e+00 7.66483516e+00
 9.17582418e-01]
 [ 0.00000000e+00 2.22044605e-16 2.19780220e-02 4.02197802e+00
 9.89010989e-01]]
=====
Augmented matrix after swapping row 2 with row 2:
[[ 1.40000000e+01 5.00000000e+00 -2.00000000e+00 3.00000000e+00
 1.00000000e+00]
 [ 0.00000000e+00 1.30000000e+01 -2.00000000e+00 1.10000000e+01
 1.00000000e+00]
 [ 0.00000000e+00 0.00000000e+00 3.16483516e+00 7.66483516e+00
 9.17582418e-01]
 [ 0.00000000e+00 2.22044605e-16 2.19780220e-02 4.02197802e+00
 9.89010989e-01]]

```

```

=====
Augmented matrix after eliminating row 3:
[[ 1.40000000e+01 5.00000000e+00 -2.00000000e+00 3.00000000e+00
  1.00000000e+00]
 [ 0.00000000e+00 1.30000000e+01 -2.00000000e+00 1.10000000e+01
  1.00000000e+00]
 [ 0.00000000e+00 0.00000000e+00 3.16483516e+00 7.66483516e+00
  9.17582418e-01]
 [ 0.00000000e+00 2.22044605e-16 0.00000000e+00 3.96875000e+00
  9.82638889e-01]]
=====
Augmented matrix after swapping row 3 with row 3:
[[ 1.40000000e+01 5.00000000e+00 -2.00000000e+00 3.00000000e+00
  1.00000000e+00]
 [ 0.00000000e+00 1.30000000e+01 -2.00000000e+00 1.10000000e+01
  1.00000000e+00]
 [ 0.00000000e+00 0.00000000e+00 3.16483516e+00 7.66483516e+00
  9.17582418e-01]
 [ 0.00000000e+00 2.22044605e-16 0.00000000e+00 3.96875000e+00
  9.82638889e-01]]
=====
Solution vector x:
[ 0.03849519 -0.18022747 -0.30971129 0.24759405]
=====
Solution: [ 0.03849519 -0.18022747 -0.30971129 0.24759405]

```