Universidad EAFIT
Escuela de Ciencias Aplicadas e Ingeniería
Análisis Numérico 1
Fecha: 29 de mayo de 2024

Final Project

Cristian Camilo Zapata
Felipe Arcila

1. Introduction

Corolla is a Python application implemented through Django with the aim of it being a tool that allows the application of a wide variety of useful numerical methods for different purposes. The methods implemented within Corolla are:

1. Root-finding methods:
   - Incremental Search
   - Bisection Method
   - False Position Method
   - Secant Method
   - Fixed Point Method
   - Newton-Raphson Method
   - Multiple Roots Method
   - Secant Method
2. Methods for Solving Systems of Equations:
   - Jacobi
   - Gauss Seidel
   - Successive Over-relaxation
3. Matrix Factorization Methods
   - LU
   - Crout
   - Dolittle
   - Cholesky
4. Polynomial interpolation methods:
   - Vandermonde

This document delves into the use of each one and their respective implementation.

2. User Manual

2.1. Execution of the Application

The following is an explanation of how to run the program on Windows. To run the Corolla application, it is necessary to have Python installed and added to the PATH on your machine, along with the Django, numpy, sympy, math, and pandas libraries. In addition, Git must be installed on the machine to follow the procedure described below:

1. Open the terminal and clone the repository using the command:

   ```
   git clone https://github.com/DhyzerZ/Corolla
   ```

2. Open the folder where the project was saved locally with:

   ```
   cd Corolla
   ```

3. Run the Django project in the browser using:

   ```
   python manage.py runserver
   ```

Once the three steps are completed, go to the *METHODS* menu located in the upper right corner of the interface, unfold it and select the method of interest. The use of each one is explained in the following subsections.

2.2. Root-finding methods

- **Incremental search:** The following is an image of the interface:

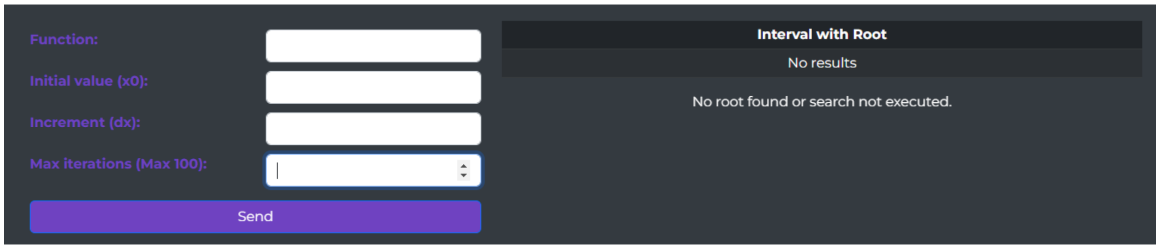


Figura 1: Image of the interface of the Incremental Search method within the Corolla application.

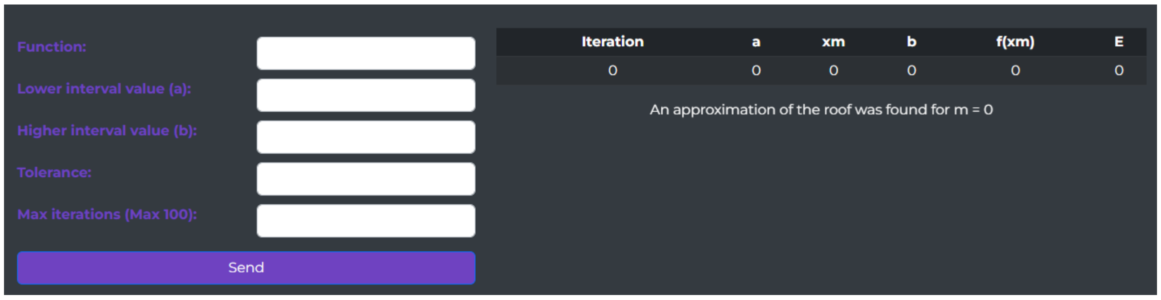
To execute the method, the boxes must be filled with the following information:

- ⋆ **Function:** In this box, the user must input the target function where the algorithm will search for roots.
- ⋆ **Initial value (x0):** Here, the user should input the initial value from where the search will start.

- ⋆ **Increment (dx):** Here, the user should input the increment value for each step of the search.
- ⋆ **Max iterations:** This is the maximum number of times the algorithm will run to approximate the root. The entered value should not exceed 100.

On the right side of the screen, you can see the iterations of the method in execution with the iteration number, the interval with root, and the message if no root is found.

- **Bisection and False position:** Both methods are used in the same way in the application, although the internal operation and results change (see implementation for more details). The following is an image of the interface:


Function:
Lower interval value (a):
Higher interval value (b):
Tolerance:
Max iterations (Max 100):
Send

Iteration	a	xm	b	f(xm)	E
0	0	0	0	0	0

An approximation of the roof was found for m = 0

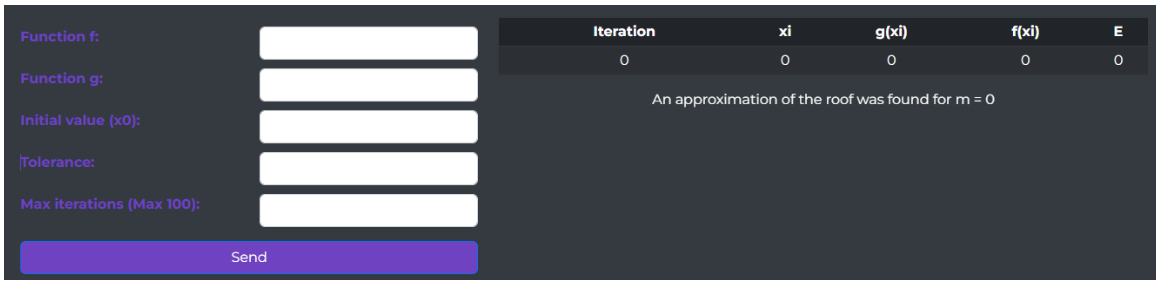
Figura 2: Image of the interface of the Bisection and False Rule methods within the Corolla application.

To execute the method, the boxes must be filled with the following information

- ⋆ **Function:** In this box, the user must put the target function on which the algorithm will search for roots.
- ⋆ **Lower interval value (a):** Here, the minimum value of the interval of interest should be put.
- ⋆ **Higher interval value (b):** Here, the upper end of the interval should be put. The function values for the ends of the interval must have different sign to guarantee the existence of a root inside.
- ⋆ **Tolerance:** It is the level of tolerance to the error with which it is expected to approximate the root.
- ⋆ **Max iterations:** It is the maximum number of times the algorithm will run approximating the root. The entered value should not exceed 100.

On the right side of the screen, you can see the iterations of the method in execution with the iteration number, the lower limit of the new interval a, the midpoint of the interval that will be used as an approximation of the root xm, and the right endpoint b, the function evaluated at xm f(xm), and finally the absolute error E.

- **Fixed point:** This method is used in the Corolla application to find roots of a function through successive iterations. The following is an image of the interface:


Function f:

Function g:

Initial value (x0):

Tolerance:

Max iterations (Max 100):

Send

Iteration	xi	g(xi)	f(xi)	E
0	0	0	0	0

An approximation of the roof was found for m = 0

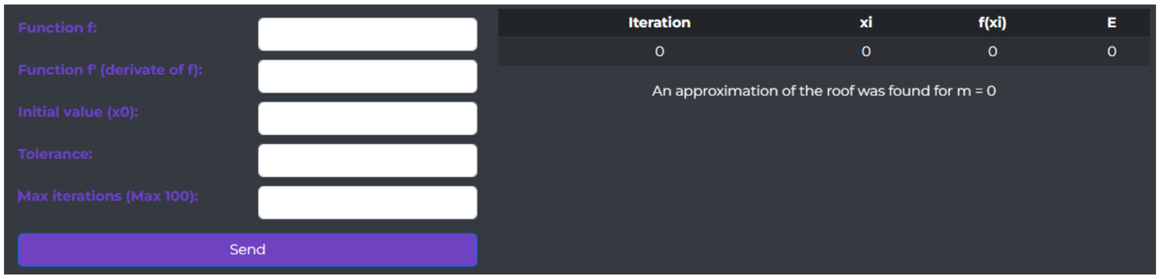
Figura 3: Image of the interface of the Fixed Point method within the Corolla application

To execute the method, the boxes must be filled with the following information:

- ⋆ **Function f:** In this box, the user must input the target function $f(x)$ where the algorithm will search for roots.
- ⋆ **Function g:** In this box, the user must input the iteration function $g(x)$ that defines the sequence for root approximation. The function $g(x)$ is typically obtained by rearranging the equation $f(x) = 0$ to the form $x = g(x)$. approximation.
- ⋆ **Initial value (x0):** Here, the user should input the initial value from where the iteration will start.
- ⋆ **Tolerance:** This is the level of tolerance to the error expected for root approximation.
- ⋆ **Max iterations:** This is the maximum number of times the algorithm will run to approximate the root. The entered value should not exceed 100.

On the right side of the screen, you can see the iterations of the method in execution with the iteration number, the current value xi, the iteration function evaluated at $g(xi)$, the target function evaluated at $f(xi)$, and finally the absolute error E.

- **Newton-Raphson:** This method is used in the Corolla application to find roots of a function through successive iterations. The following is an image of the interface:


Function f:

Function f' (derivate of f):

Initial value (x0):

Tolerance:

Max iterations (Max 100):

Send

Iteration	xi	f(xi)	E
0	0	0	0

An approximation of the roof was found for m = 0

Figura 4: Image of the interface of the Newton-Raphson method within the Corolla application

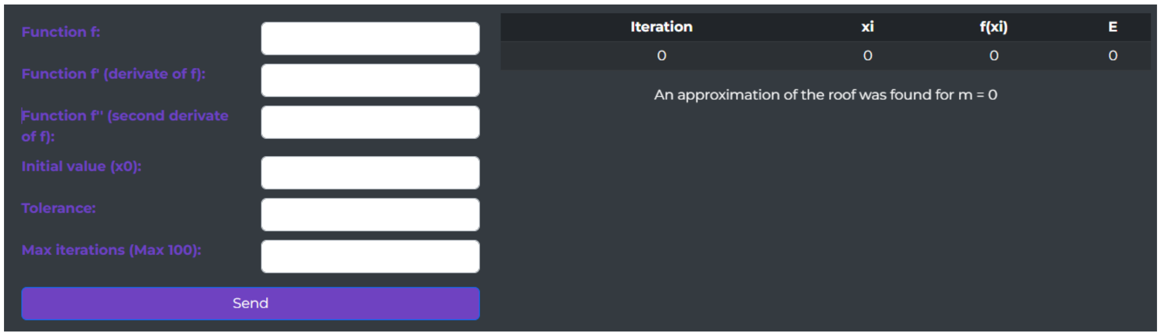
To execute the method, the boxes must be filled with the following information:

- ⋆ **Function f:** In this box, the user must input the target function $f(x)$ where the algorithm will search for roots.

- ⋆ **Function f' (Derivative of f):** In this box, the user must input the derivative of the function $f'(x)$.
- ⋆ **Initial value (x0):** Here, the user should input the initial value from where the iteration will start.
- ⋆ **Tolerance:** This is the level of tolerance to the error expected for root approximation.
- ⋆ **Max iterations:** This is the maximum number of times the algorithm will run to approximate the root. The entered value should not exceed 100.

On the right side of the screen, you can see the iterations of the Newton-Raphson method in execution with the iteration number, the current value xi, the function f(xi) evaluated at the current value, the derivative f'(xi) evaluated at the current value, and finally the absolute error E. This iterative process continues until the absolute error is within the specified tolerance or the maximum number of iterations is reached.

- **Multiple Roots:** This method is used in the Corolla application to find roots of a function that may occur multiple times. The following is an image of the interface:


Function f:
Function f' (derivate of f):
Function f'' (second derivate of f):
Initial value (x0):
Tolerance:
Max iterations (Max 100):
Send

Iteration	xi	f(xi)	E
0	0	0	0

An approximation of the roof was found for m = 0

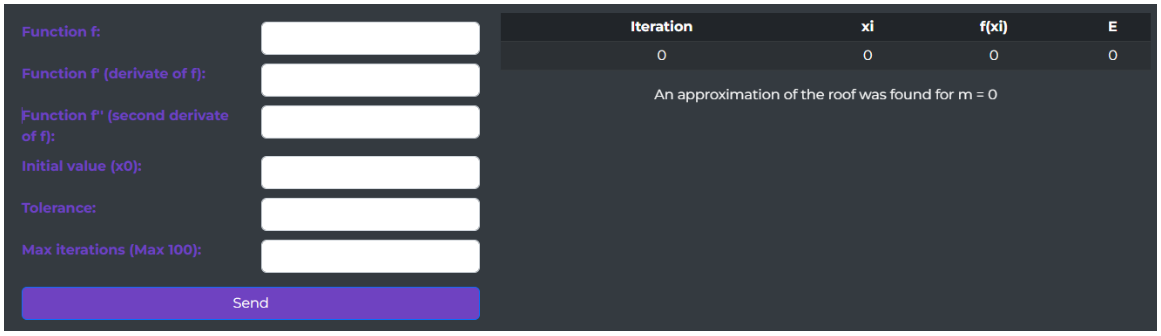
Figura 5: Image of the interface of the Multiple Roots method within the Corolla application.

To execute the method, the boxes must be filled with the following information:

- ⋆ **Function f:** In this box, the user must input the target function $f(x)$ where the algorithm will search for roots.
- ⋆ **Function f' (Derivative of f):** In this box, the user must input the derivative of the function $f'(x)$.
- ⋆ **Function f" (Second Derivative of f):** In this box, the user must input the second derivative of the function $f''(x)$.
- ⋆ **Initial value (x0):** Here, the user should input the initial value from where the iteration will start.
- ⋆ **Tolerance:** This is the level of tolerance to the error expected for root approximation.
- ⋆ **Max iterations:** This is the maximum number of times the algorithm will run to approximate the root. The entered value should not exceed 100.

On the right side of the screen, you can observe the iterations of the Multiple Roots method in action, displaying the iteration number, the current value (xi), the function (f(xi)) evaluated at (xi), the first derivative (f'(xi)) evaluated at (xi), the second derivative (f"(xi)) evaluated at (xi), and finally the absolute error (E). This iterative process is repeated until the absolute error falls below the specified tolerance level or the maximum number of iterations is reached.

- **Secant method** The Secant method is a root-finding algorithm that uses two initial approximations to iteratively converge to a root. The following is an image of the interface:



Function f:

Function f' (derivate of f):

Function f'' (second derivate of f):

Initial value (x0):

Tolerance:

Max iterations (Max 100):

Iteration	xi	f(xi)	E
0	0	0	0

An approximation of the roof was found for m = 0

Figura 6: Image of the interface of the Secant method within the Corolla application.

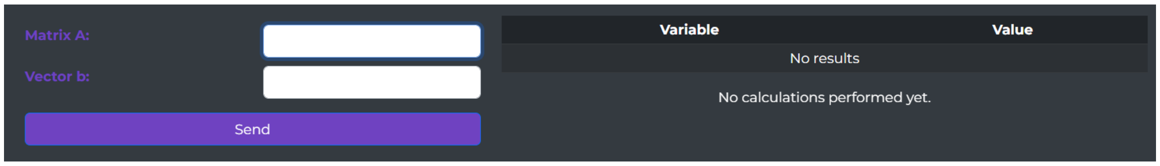
To execute the method, the boxes must be filled with the following information:

- ⋆ **Function f:** In this box, the user must input the target function $f(x)$ where the algorithm will search for roots.
- ⋆ **Initial value (x0):** In this box, the user should input the initial value from where the iteration will start.
- ⋆ **Initial value (x1):** In this box, the user should input another initial value close to $x0$. The algorithm uses both initial values to start the iteration process.
- ⋆ **Tolerance:** In this box, the user should input the level of tolerance to the error with which it is expected to approximate the root.
- ⋆ **Max iterations:** In this box, the user should input the maximum number of times the algorithm will run to approximate the root. The entered value should not exceed 100.

On the right side of the screen, you can see the iterations of the method in execution with the iteration number, the current value xi, the function evaluated at $f(xi)$, and finally the absolute error E.

2.3. Methods for Solving Systems of Equations

- **Gaussian Elimination (all variants):** The Gaussian elimination variants are methods for solving systems of linear equations. The following is an image of the interface:



No results

No calculations performed yet.

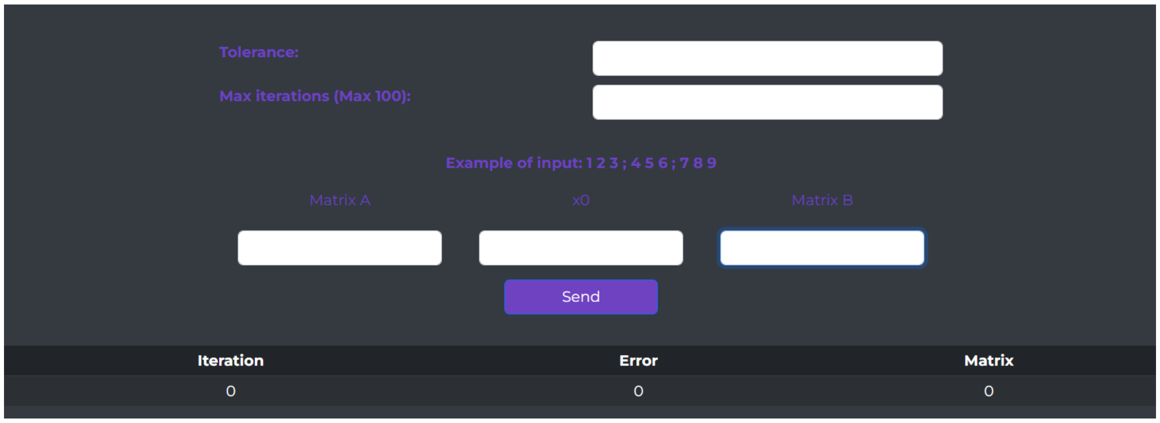
Figura 7: Image of the interface for the Gaussian elimination variants within the Corolla application.

To execute the method, the boxes must be filled with the following information:

- ⋆ **Matrix A:** In this box, the user must input the coefficients of the system of linear equations. The matrix should be written in bracket format, separating each row of coefficients with a semicolon (E.g. [[2, -1, 0], [-1, 2, -1], [0, -1, 2]]).
- ⋆ **Vector b:** In this box, the user should input the constants from each equation. The vector should be written in brackets, separating each value with a semicolon if there are multiple rows (E.g. [1, 0, 1]).

On the right side of the screen, you can see the results of the variables after executing the method with the 'Send' button.

- **Gauss-Seidel and Jacobi Methods** The Gauss-Seidel and Jacobi methods are iterative techniques used to solve systems of linear equations. The following is an image of the interface:


Tolerance:
Max iterations (Max 100):
Example of input: 1 2 3 ; 4 5 6 ; 7 8 9
Matrix A
x0
Matrix B
Send

Iteration	Error	Matrix
0	0	0

Figura 8: Image of the interface for the Gauss-Seidel and Jacobi methods within the Corolla application.

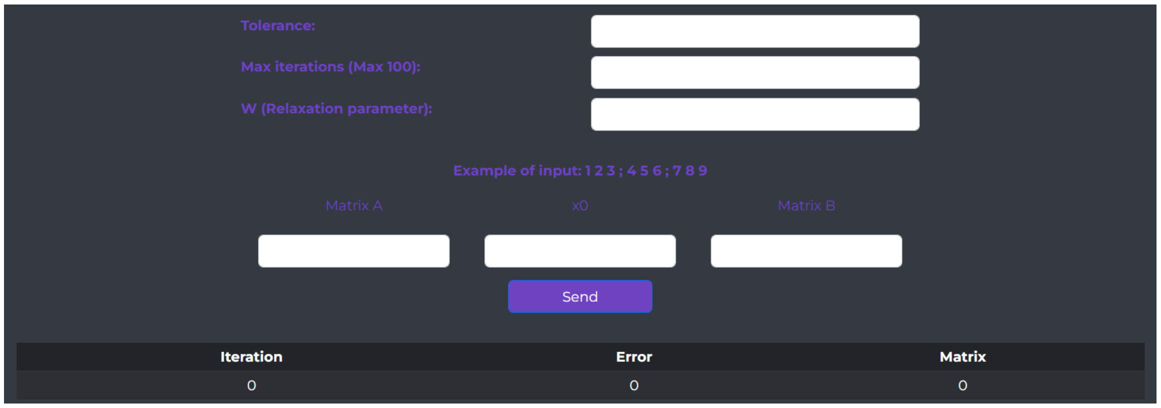
To execute either method, the boxes must be filled with the following information:

- ⋆ **Tolerance:** In this box, the user must input the level of tolerance for the error in the root approximation.
- ⋆ **Max Iterations:** In this box, the user should input the maximum number of iterations the algorithm will perform. The entered value should not exceed 100.
- ⋆ **Matrix A:** In this box, the user must input the coefficients of the system of linear equations. The matrix should be written in the format shown: row1; row2; ... (E.g. 8 4 ; 1 -4).

- ⋆ **Matrix B:** In this box, the user should input the constants from each equation. The vector should be written in the format shown: value1; value2; ... (E.g. 1 ; 1)
- ⋆ **Initial value (x0):** In this box, the user should input the initial guess for the solution vector. The vector should be written in the format shown: value1; value2; ... (E.g. 0 ; 0)

After entering the data and pressing the 'Send' button, the bottom of the interface will display the iteration number, the error value, and the current state of the matrix. The process continues until the error is within the specified tolerance or the maximum number of iterations is reached.

- **SOR Method** he Successive Over-Relaxation (SOR) method is an iterative technique used to solve systems of linear equations. It is a variant of the Gauss-Seidel method that accelerates convergence by using a relaxation parameter ω. The following is an image of the interface:


Tolerance:
Max iterations (Max 100):
W (Relaxation parameter):
Example of input: 1 2 3 ; 4 5 6 ; 7 8 9
Matrix A
x0
Matrix B
Send

Iteration	Error	Matrix
0	0	0

Figura 9: Image of the interface for the SOR method within the Corolla application.

To execute the SOR method, the boxes must be filled with the following information:

- ⋆ **Tolerance:** In this box, the user must input the level of tolerance for the error in the root approximation.
- ⋆ **Max Iterations:** In this box, the user should input the maximum number of iterations the algorithm will perform. The entered value should not exceed 100.
- ⋆ **W (Relaxation parameter):** In this box, the user must input the relaxation parameter ω, which is typically between 0 and 2. This parameter helps to improve the rate of convergence.
- ⋆ **Matrix A:** In this box, the user must input the coefficients of the system of linear equations. The matrix should be written in the format: row1; row2; ... (E.g. 8 4 ; 1 -4).
- ⋆ **Matrix B:** In this box, the user should input the constants from each equation. The vector should be written in the format: value1; value2; ...
- ⋆ **Initial value (x0):** In this box, the user should input the initial guess for the solution vector. The vector should be written in the format: value1; value2; ...

After entering the data and pressing the 'Send' button, the bottom of the interface will display the iteration number, the error value, and the current state of the matrix. The process continues until the error is within the specified tolerance or the maximum number of iterations is reached.

2.4. Matrix Factorization Methods

Matrix factorization methods are techniques used to decompose a matrix into simpler matrix products. The following is an image of the interface:

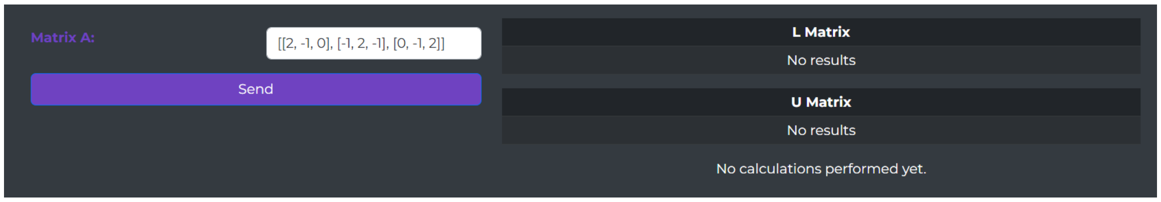


Figura 10: Image of the interface for matrix factorization methods within the Corolla application.

To execute a factorization method, the boxes must be filled with the following information:

- ⋆ **Matrix A:** In this box, the user must input the matrix to be factorized. The matrix should be written as row1; row2; ... (E.g. [[2, -1, 0], [-1, 2, -1], [0, -1, 2]])

After entering the data and pressing the 'Send' button, the bottom of the interface will display the resulting L and U matrices from the factorization. The factorization process decomposes matrix A into a lower triangular matrix L and an upper triangular matrix U. The exact form of the L and U matrices depends on the specific factorization method used.

2.5. Vandermonde Polynomial Interpolation Method

The Vandermonde method is a technique used to construct a polynomial that passes through a given set of points. The following is an image of the interface:

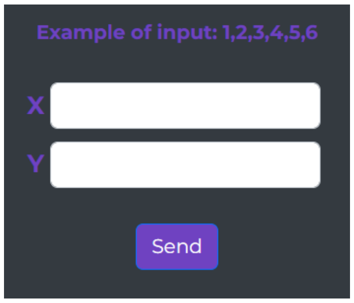


Figura 11: Image of the interface for the Vandermonde method within the Corolla application.

To use the Vandermonde method, the boxes must be filled with the following information:

- ⋆ **X:** In this box, the user must input the x-values of the data points, separated by commas.
- ⋆ **Y:** In this box, the user should input the corresponding y-values of the data points, separated by commas.

After entering the data and pressing the 'Send' button, the application will calculate the interpolating polynomial using the Vandermonde method. The results will be displayed below the input fields, showing the polynomial coefficients.

3. Implementation and Pseudo-code

The `manage.py` file is responsible for running the project through the Django library. Once the project is executed, it uses the HTML templates located within the `Corolla/application/templates` folder to run the graphical interface. The details of the interface for each method are explained in the User Manual for more details.. Finally, each part of the application executes some numerical method contained in Python code format in the `Corolla/application` folder. The subsections will be pseudocodes of each of the methods to explain their respective operation.

3.1. Incremental Search

The Incremental Search method starts with an initial value and incrementally increases this value by a predefined step size. This process continues until it finds at least one root or reaches a defined maximum number of iterations. The algorithm checks for a root between the current point and the previous one using Bolzano's theorem. This theorem states that if the product of the function values at two points is less than zero, then there is at least one root in the interval between these points. Corolla includes a version of the method that does not stop upon finding the first root, but instead completes the maximum number of iterations and returns all approximations of roots found.
The following is a pseudocode of the implementation used in the Corolla application.

```
Algorithm IncrementalSearch(f, xinit, dx, n, tolerance):
Input:
    f         : A function f(x)
    xinit     : Initial value of x
    dx        : Increment step
    n         : Number of iterations
    tolerance : Tolerance to consider a value as a root

Output:
    results   : List of intervals where roots are found
    message   : Status message

// Initialize variables and convert inputs to float type
x0 <- float(xinit)
dx <- float(dx)
tolerance <- float(tolerance)
results <- []
iteration <- 1

// Evaluate the function at the initial point
fx0 <- Evaluate f at x0

// Check if the initial value is a root
If |fx0| < tolerance Then
    Return results, "An approximation of the root was found
                                            at x = " + x0

// Perform the incremental search
For i from 1 to n Do:
        x1 <- x0 + dx
        fx1 <- Evaluate f at x1

        // Check if the current value is a root
        If |fx1| < tolerance Then
            Add "There is a root for the function in [" + x0 + ", " + x1 + "]"
                                                                    to results
        Else If fx0 * fx1 < 0 Then
            Add "There is a root for the function in [" + x0 + ", " + x1 + "]"
                                                                    to results

        // Update x0 and fx0 for the next iteration
        x0 <- x1
        fx0 <- fx1

    Return results, "Incremental search completed."
```

3.2. Bisection

The bisection method is a root-finding algorithm that applies to any continuous function for which one knows two values with opposite signs. The method consists of repeatedly bisecting the interval defined by these values and then selecting the subinterval in which the function changes sign, and therefore must contain a root. The process is continued until the interval has been narrowed down to a region where the function's root is known to a specified precision.
The Corolla app includes an implementation for this method described by the following pseudocode.

```
Algorithm Bisection(f, a, b, tol, max_iterations):
Input:
    f               : A function f(x)
    a               : Lower bound of the interval [a, b]
    b               : Upper bound of the interval [a, b]
    tol             : Tolerance to consider a value as a root
    max_iterations  : Maximum number of iterations allowed. Must be a
                      positive integer.

Output:
    results         : List of lists with the results of each iteration, where each
                      list contains:
                      - Iteration number
                      - Lower bound of the interval
                      - Midpoint of the interval
                      - Upper bound of the interval
                      - Value of f(x) at the midpoint
                      - Absolute error between successive approximations
    message         : Status message indicating if a root was found or if the
                      maximum number of iterations was reached.

// Initialize variables
a <- float(a)
```

```
b <- float(b)
results <- []

// Evaluate the function at the endpoints of the interval
fa <- Evaluate f at a
fb <- Evaluate f at b

// Check if the initial values are roots
If fa == 0 Then
    Return [], "An approximation of the root was found for x = " + a
Else If fb == 0 Then
    Return [], "An approximation of the root was found for x = " + b
Else If fa * fb < 0 Then
    // There is a root in the interval
    iteration <- 1
    error <- tol + 1 // Initialize error to be larger than tolerance

    // Perform the bisection method
    While iteration < max_iterations And error > tol And fx != 0 Do:
        x <- (a + b) / 2 // Calculate the midpoint
        fx <- Evaluate f at x

        // Append current state to results
        results.append([iteration, a, x, b, fx, "NaN" if iteration == 1 Else error])

        // Check if the root is found
        If fx == 0 Then
            Return results, "An approximation of the root was found for x = " + x
        Else If fa * fx < 0 Then
            // Update the interval to [a, x]
            b <- x
            fb <- fx
        Else
            // Update the interval to [x, b]
            a <- x
            fa <- fx

        // Update error and iteration
        error <- abs(x - x_prev)
        x_prev <- x
        iteration <- iteration + 1

    // Check final state
    If fx == 0 Then
        Return results, "An approximation of the root was found for x = " + x
    Else If error <= tol Then
        Return results, "An approximation of the root was found for x = " + x
    Else
```

```
            Return results, "Given the number of iterations and the
            tolerance, it was impossible to find a satisfying root"
    Else
        Return [], "Error: No root was found in the interval for the function"
```

3.3. False Position

The False Position method, also known as the Regula Falsi method, is a root-finding algorithm that combines aspects of the bisection method and the secant method. It starts with an interval where the function changes sign (like the bisection method) and produces a sequence of shrinking intervals, but instead of picking the midpoint of the interval, it draws a straight line through the points defined by the ends of the interval and takes the x-intercept of this line as the new guess. The next is a pseudocode that describes the implementation of the method used in the Corolla app.

```
Algorithm FalsePosition(f, a, b, tol, max_iterations):
    Input:
        f               : A function f(x)
        a               : Lower bound of the interval [a, b]
        b               : Upper bound of the interval [a, b]
        tol             : Tolerance to consider a value as a root
        max_iterations  : Maximum number of iterations allowed.
                          Must be a positive integer.

    Output:
        results         : List of lists with the results of each iteration, where
                          each list contains:
                          - Iteration number
                          - Lower bound of the interval
                          - Upper bound of the interval
                          - Value of x where f(x) is evaluated
                          - Value of f(x) at x
                          - Relative or absolute error
        message         : Status message indicating whether a root
                            was found or if the maximum number of iterations was reached.

    // Initialize variables
    a <- float(a)
    b <- float(b)
    results <- []

    // Replace '^' with '**' for correct exponentiation in f
    f <- Replace '^' with '**' in f

    // Evaluate the function at the endpoints of the interval
    fa <- Evaluate f at a
    fb <- Evaluate f at b

    // Check if the initial values are roots
    If fa == 0 Then
    Return [], "An approximation of the root was found for x = " + a
Else If fb == 0 Then
    Return [], "An approximation of the root was found for x = " + b
Else If fa * fb < 0 Then
    // There is a root in the interval
    x <- (a * fb - b * fa) / (fb - fa)
    fx <- Evaluate f at x

    iteration <- 1
    error <- tol + 1 // Initialize error to be larger than tolerance

    // Append initial state to results
    results.append([iteration, '{:.10f}'.format(a), '{:.10f}'.format(x),
    '{:.10f}'.format(b), '{:.1e}'.format(fx).replace('e-0', 'e-'), 'NaN'])

    // Perform the false position method
    While fx != 0 and error > tol and iteration < max_iterations:
        If fa * fx < 0 Then
            b <- x
            fb <- fx
        Else
            a <- x
            fa <- fx

        x_prev <- x
        x <- (a * fb - b * fa) / (fb - fa)
        fx <- Evaluate f at x

        error <- abs(x - x_prev)
        iteration <- iteration + 1

        // Append current state to results
        results.append([iteration, '{:.10f}'.format(a), '{:.10f}'.format(x),
        '{:.10f}'.format(b), '{:.1e}'.format(fx).replace('e-0', 'e-'),
        '{:.1e}'.format(error).replace('e-0', 'e-')])

    // Check final state
    If fx == 0 Then
        Return results, "An approximation of the root was found for x = " + x
    Else If error <= tol Then
        Return results, "An approximation of the root was found for x = " + x
    Else
        Return results, "Given the number of iterations and the tolerance,
        it was impossible to find a satisfying root"
Else
    Return results, "Error: No root was found in the interval for the function"
```

3.4. Fixed Point

The Fixed Point Iteration method starts with an initial guess for the root of the function. This guess is then plugged into a function, which is derived from the original function, to produce a new guess. This process is repeated until the difference between successive guesses is below a certain threshold, indicating that the method has converged to the root.

The function used in Fixed Point Iteration is typically a rearrangement or transformation of the original function such that the root of the original function is a fixed point of the transformed function. In other words, if x is a root of the original function, then x is a fixed point of the transformed function, meaning that plugging x into the transformed function yields x.

Here's a pseudocode that describes the implementation of the Fixed Point Iteration method:

```
Algorithm FixedPoint(f, g, x0, tol, max_iterations):
Input:
    f               : A function f(x)
    g               : Iteration function g(x)
    x0              : Initial value for starting the iteration
    tol             : Tolerance to consider a value as a root
    max_iterations  : Maximum number of iterations allowed.
                      Must be a positive integer.

Output:
    results         : List of lists with the results of each
                      iteration, where each list contains:
                      - Iteration number
                      - Value of x in the current iteration
                      - Value of g(x) in the current iteration
                      - Value of f(x) in the current iteration
                      - Absolute error between successive approximations
    message         : Status message indicating whether a root was found or
                      if the maximum number of iterations was reached.

// Initialize variables
x <- float(x0)
results <- []

// Replace '^' with '**' for correct exponentiation in f and g
f <- Replace '^' with '**' in f
g <- Replace '^' with '**' in g

// Evaluate the functions at the initial value
fx <- Evaluate f at x
gx <- Evaluate g at x

iteration <- 0
error <- tol + 1 // Initialize error to be larger than tolerance

// Append initial state to results
results.append([iteration, '{:.10f}'.format(x), '{:.10f}'.format(gx),
'{:.1e}'.format(fx).replace('e-0', 'e-'), 'NaN'])

// Perform the fixed point iteration
While fx != 0 and error > tol and iteration < max_iterations:
    x_prev <- x
    x <- gx

    // Evaluate the functions at the new value of x
    fx <- Evaluate f at x
    gx <- Evaluate g at x

    error <- abs(x - x_prev)
    iteration <- iteration + 1

    // Append current state to results
    results.append([iteration, '{:.10f}'.format(x), '{:.10f}'.format(gx),
    '{:.1e}'.format(fx).replace('e-0', 'e-'), '{:.10f}'.format(error)])

// Check final state
If fx == 0 Then
    Return results, "An approximation of the root was found for x = " + x
Else If error <= tol Then
    Return results, "An approximation of the root was found for x = " + x
Else
    Return results, "Given the number of iterations and the tolerance,
    it was impossible to find a satisfying root"
```

3.5. Newton-Raphson

The Newton-Raphson method uses an approximation of the root of the function through finite differences and the iterative structure of the fixed point method to arrive at the value of x for which the function takes the value 0. The approximation to the root is made using the following formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{1}$$

The pseudocode that describes the implementation used is:

```
Algorithm NewtonRaphson(f, df, x0, tol, max_iterations):
Input:
    f               : A string representing the function f(x) to evaluate.
    df              : A string representing the derivative of the function f(x).
    x0              : Initial value to start the iteration.
    tol             : Tolerance to consider a value as a root
    max_iterations  : Maximum number of iterations allowed. Must be a positive integer.

Output:
    results         : List of lists with the results of each iteration, where each
                      list contains:
```

```
                      - Iteration number
                      - Value of x in the current iteration
                      - Value of f(x) in the current iteration
                      - Absolute error between successive approximations
    message         : Status message indicating whether a root was found or
                      if the maximum number of iterations was reached.

// Define a function to evaluate f at a given x
Function eval_f(x):
    Return Evaluate f at x

// Define a function to evaluate df at a given x
Function eval_df(x):
    Return Evaluate df at x

// Initialize variables
x <- x0
fx <- eval_f(x)
dfx <- eval_df(x)
iteration <- 0
error <- tol + 1 // Initialize error to be larger than tolerance
results <- []

// Append initial state to results
results.append([iteration, '{:.10f}'.format(x), '{:.1e}'.format(fx).replace('e-0', 'e-'),
'NaN'])

// Perform the Newton-Raphson iteration
While fx != 0 and dfx != 0 and error > tol and iteration < max_iterations:
    x_prev <- x
    x <- x - fx / dfx

    // Evaluate the functions at the new value of x
    fx <- eval_f(x)
    dfx <- eval_df(x)

    error <- abs(x_prev - x)
    iteration <- iteration + 1

    // Append current state to results
    results.append([iteration, '{:.10f}'.format(x), '{:.1e}'.format(fx).replace('e-0', 'e-'),
    '{:.1e}'.format(error).replace('e-0', 'e-')])

// Check final state
If fx == 0 Then
    Return results, "An approximation of the root was found for x = " + x
Else If error <= tol Then
    Return results, "An approximation of the root was found for x = " + x
```

```
Else
    Return results, "Given the number of iterations and the tolerance,
    it was impossible to find a satisfying root"
```

3.6. Multiple Roots

The method is an adaptation of the Newton-Raphson method that adapts the formula to be applicable in the search for roots with multiplicity greater than 1. To achieve this, the applied formula looks as follows:

$$x_{n+1} = x_n - \frac{f(x_n)f'(x_n)}{(f'(x_n))^2 - f(x_n)f''(x_n)}$$

In the Corolla application, the method was implemented following the description below:

```
Algorithm MultipleRoots(f, df, df2, x0, tol, max_iterations):
Input:
    f               : A string representing the function f(x) to evaluate.
    df              : A string representing the first derivative of the function f(x).
    df2             : A string representing the second derivative of the function f(x).
    x0              : Initial value to start the iteration.
    tol             : Tolerance to consider a value as a root
    max_iterations  : Maximum number of iterations allowed. Must
    be a positive integer.

Output:
    results         : List of lists with the results of each iteration, where
    each list contains:
                      - Iteration number
                      - Value of x in the current iteration
                      - Value of f(x) in the current iteration
                      - Absolute error between successive approximations
    message         : Status message indicating whether a root was found or if the maximum
    number of iterations was reached.

// Initialize variables
x <- x0
fx <- Evaluate f at x
dfx <- Evaluate df at x
df2x <- Evaluate df2 at x
iteration <- 0
error <- tol + 1 // Initialize error to be larger than tolerance
results <- []

// Append initial state to results
results.append([iteration, '{:.10f}'.format(x), '{:.1e}'.format(fx).replace('e-0', 'e-'), 'NaN'])

// Perform the Multiple Roots iteration
While fx != 0 and dfx != 0 and error > tol and iteration < max_iterations:
        numerator <- fx * dfx
        denominator <- dfx^2 - fx * df2x
        x1 <- x - numerator / denominator

        // Evaluate the functions at the new value of x
        fx <- Evaluate f at x1
        dfx <- Evaluate df at x1
        df2x <- Evaluate df2 at x1

        error <- abs(x1 - x)
        iteration <- iteration + 1

        // Append current state to results
        results.append([iteration, '{:.10f}'.format(x1), '{:.1e}'.format(fx).replace('e-0', 'e-'),
        '{:.1e}'.format(error).replace('e-0', 'e-')])

        // Update x for the next iteration
        x <- x1

    // Check final state
    If fx == 0 Then
        Return results, "An approximation of the root was found for x = " + x
    Else If error <= tol Then
        Return results, "An approximation of the root was found for x = " + x
    Else
        Return results, "Given the number of iterations and the tolerance,
        it was impossible to find a satisfying root"
```

3.7. Secant method

The **Secant Method** is a root-finding algorithm used in numerical analysis. It is a recursive method that uses a series of roots of secant lines to better approximate a root of a function. The secant method can be thought of as a finite-difference approximation of Newton's method. The method is defined by the recurrence relation:

$$x_{n+1} = x_n - f(x_n) \cdot \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$$

The method was implemented following the next pseudocode:

```
    Algorithm Secant(f, x0, x1, tol, max_iterations):
    Input:
        f               : A string representing the function f(x) to evaluate.
        x0              : First initial value to start the iteration.
        x1              : Second initial value to start the iteration.
        tol             : Tolerance for the absolute error between successive approximations.
                          Must be a positive number.
        max_iterations  : Maximum number of iterations allowed. Must be a positive integer.
```

```
Output:
    results         : List of lists with the results of each iteration, where each
                      list contains:
                      - Iteration number
                      - Value of x in the current iteration
                      - Value of f(x) in the current iteration
                      - Absolute error between successive approximations
    message         : Status message indicating whether a root was found or if the maximum
                      number of iterations was reached.

// Define a function to evaluate the mathematical expression represented by the string
Function eval_f(x):
    Return evaluate f at x as a float

// Initialize variables
fx0 <- eval_f(x0)
fx1 <- eval_f(x1)
iteration <- 0
error <- tol + 1 // Initialize error to be larger than tolerance
results <- []

// Append initial state to results
results.append([iteration, '{:.10f}'.format(x0),
'{:.1e}'.format(fx0).replace('e-0', 'e-'), 'NaN'])
iteration <- iteration + 1
results.append([iteration, '{:.10f}'.format(x1), '{:.1e}'.format(fx1).replace('e-0', 'e-'),
'{:.1e}'.format(abs(x1 - x0)).replace('e-0', 'e-')])

// Perform the Secant iteration
While fx1 != 0 and error > tol and iteration < max_iterations:
    x2 <- x1 - fx1 * (x1 - x0) / (fx1 - fx0)
    fx2 <- eval_f(x2)
    error <- abs(x2 - x1)
    iteration <- iteration + 1
    results.append([iteration, '{:.10f}'.format(x2), '{:.1e}'.format(fx2).replace('e-0', 'e-'),
    '{:.1e}'.format(error).replace('e-0', 'e-')])

    // Update values for the next iteration
    x0 <- x1
    fx0 <- fx1
    x1 <- x2
    fx1 <- fx2

// Check final state
If fx1 == 0 Then
    Return results, "An approximation of the root was found for x = " + x1
Else If error <= tol Then
```

```
        Return results, "An approximation of the root was found for x = " + x1
    Else
        Return results, "Given the number of iterations and the tolerance,
        it was impossible to find a satisfying root"
```

3.8. Gaussian elimination

Gaussian Elimination is a method used to solve linear systems of equations. It transforms the system into an upper triangular matrix, from which the unknowns are derived using backward substitution. On the other hand, pivoting is a strategy employed during Gaussian Elimination to improve numerical stability and prevent division by zero. This is achieved by rearranging the rows (in the case of partial pivoting) or both the rows and columns (in the case of total pivoting) to ensure the pivot element is the largest possible in absolute value. All three variants are included in the Corolla application. The pseudocode for these methods is provided in the following items of the document.

- **No pivoting:**

```
    Algorithm SimpleElimination(A, b):
Input:
    A : Matrix of coefficients (list of lists)
    b : Vector of independent terms (list)

Output:
    x : List of solutions for the system of equations

// Step 1: Forward Elimination
n <- length of A
For i from 0 to n-1 do:
    // Step 1.1: Check for division by zero
    If A[i][i] == 0 then:
        Raise error "Division by zero"

    // Step 1.2: Eliminate variable i from all rows below row i
    For j from i+1 to n-1 do:
        ratio <- A[j][i] / A[i][i]
        For k from i to n-1 do:
            A[j][k] <- A[j][k] - ratio * A[i][k]
        b[j] <- b[j] - ratio * b[i]

// Step 2: Back Substitution
x <- list of zeros with length n
For i from n-1 to 0 do:
    x[i] <- b[i]
    For j from i+1 to n-1 do:
        x[i] <- x[i] - A[i][j] * x[j]

    // Step 2.1: Check for division by zero again
    If A[i][i] == 0 then:
        Raise error "Division by zero"

    x[i] <- x[i] / A[i][i]

Return x
```

- **Partial Pivoting:**

```
    Algorithm GaussianEliminationPP(A, b):
Input:
    A : Matrix of coefficients (list of lists).
    b : Vector of independent terms (list).

Output:
    x : List with the values of the variables of the system of equations.

// Get the number of rows in matrix A
n <- length(A)

// Perform Gaussian elimination with partial pivoting
for i <- 0 to n-1 do:
    // Partial pivoting
    max_row <- index of the row with the maximum absolute value
               in column i from row i to n-1
    if i != max_row then:
        Swap row i and max_row in A
        Swap b[i] and b[max_row]

    // Eliminate entries below the pivot
    for j <- i+1 to n-1 do:
        if A[i][i] == 0 then:
            Raise an error "Division by zero"
        ratio <- A[j][i] / A[i][i]
        for k <- i to n-1 do:
            A[j][k] <- A[j][k] - ratio * A[i][k]
        b[j] <- b[j] - ratio * b[i]

// Perform back substitution
x <- list of n zeros
for i <- n-1 to 0 do:
    x[i] <- b[i]
    for j <- i+1 to n-1 do:
        x[i] <- x[i] - A[i][j] * x[j]
    if A[i][i] == 0 then:
        Raise an error "Division by zero"
    x[i] <- x[i] / A[i][i]
```

```
Return x
```

- **Total pivoting:**

```
   Algorithm EliminationWithTotalPivoting(A, b):
Input:
    A : Matrix of coefficients (list of lists)
    b : Vector of independent terms (list)

Output:
    x : List of solutions for the system of equations

n <- length of A
column_changes <- list of range(n)

// Step 1: Forward Elimination with Total Pivoting
For i from 0 to n-1 do:
    // Step 1.1: Find the pivot with the largest absolute value
    max_row, max_col <- (i, i)
    For r from i to n-1 do:
        For c from i to n-1 do:
            If abs(A[r][c]) > abs(A[max_row][max_col]) then:
                max_row, max_col <- r, c

    // Step 1.2: Check for a singular matrix
    If A[max_row][max_col] == 0 then:
        Raise error "The matrix is singular and cannot be solved"

    // Step 1.3: Swap rows
    If i != max_row then:
        Swap A[i] with A[max_row]
        Swap b[i] with b[max_row]

    // Step 1.4: Swap columns
    If i != max_col then:
        For each row in A do:
            Swap row[i] with row[max_col]
        Swap column_changes[i] with column_changes[max_col]

    // Step 1.5: Eliminate variables below the pivot
    For j from i+1 to n-1 do:
        If A[i][i] == 0 then:
            Raise error "Division by zero"
        ratio <- A[j][i] / A[i][i]
        For k from i to n-1 do:
            A[j][k] <- A[j][k] - ratio * A[i][k]
        b[j] <- b[j] - ratio * b[i]

// Step 2: Back Substitution
x <- list of zeros with length n
For i from n-1 to 0 do:
    x[i] <- b[i]
    For j from i+1 to n-1 do:
        x[i] <- x[i] - A[i][j] * x[j]
    If A[i][i] == 0 then:
        Raise error "Division by zero"
    x[i] <- x[i] / A[i][i]

// Step 3: Revert column changes
final_result <- list of zeros with length n
For i from 0 to n-1 do:
    final_result[column_changes[i]] <- x[i]

Return final_result
```

3.9. LU Factorization

LU factorization, also known as LU decomposition, is a method in linear algebra where a given square matrix is decomposed into the product of a lower triangular matrix (L) and an upper triangular matrix (U). It is commonly done using the following three methods, all included in the Corolla app with implementation described following the pseudocodes.

- **Dolittle:** For this method, all diagonal elements in the L matrix are taken as ones (1). The next is the pseudocode that describes the implementation of the method for the application.

```
Algorithm DoolittleFactorization(A):
Input:
    A : Matrix of coefficients (list of lists)

Output:
    L : Lower triangular matrix (list of lists)
    U : Upper triangular matrix (list of lists)

n <- length of A
Initialize L as an n x n matrix of zeros
Initialize U as an n x n matrix of zeros

For i from 0 to n-1 do:
    // Compute the U matrix
    For k from i to n-1 do:
        sum_ <- 0
        For j from 0 to i-1 do:
            sum_ <- sum_ + L[i][j] * U[j][k]
        U[i][k] <- A[i][k] - sum_
        // Set the diagonal elements of L to 1
        L[i][i] <- 1.0

        // Compute the L matrix
        For k from i+1 to n-1 do:
            sum_ <- 0
            For j from 0 to i-1 do:
                sum_ <- sum_ + L[k][j] * U[j][i]
            If U[i][i] == 0 then:
                Raise error "Division by zero"
            L[k][i] <- (A[k][i] - sum_) / U[i][i]

    Return L, U
```

- **Crout:** the Crout's method is similar to the Doolittle's method, but it assumes the diagonal elements of the U matrix to be 1 instead of the L matrix. This assumption is used to compute the elements of both matrices. Corolla finds the elements using an implementation described by the pseudocode shown:

```
    Algorithm CroutDecomposition(A):
    Input:
        A : Matrix of coefficients (list of lists)

    Output:
        L : Lower triangular matrix (list of lists)
        U : Upper triangular matrix (list of lists)

    n <- length of A
    Initialize L as an n x n matrix of zeros
    Initialize U as an n x n matrix of zeros

    For i from 0 to n-1 do:
        // Compute the L matrix
        For j from i to n-1 do:
            sum_ <- sum of L[j][k] * U[k][i] for k from 0 to i-1
            L[j][i] <- A[j][i] - sum_

        // Set the diagonal element of U to 1
        U[i][i] <- 1.0

        // Compute the U matrix
        For j from i+1 to n-1 do:
            sum_ <- sum of L[i][k] * U[k][j] for k from 0 to i
            If L[i][i] == 0 then:
                Raise error "Division by zero"
            U[i][j] <- (A[i][j] - sum_) / L[i][i]

    Return L, U
```

- **Cholesky:** for this method U is taken as L^T. In this way, it is only necessary to calculate one matrix L instead of two, such that $A = L \cdot L^T$. The following pseudocode describes the implementation of the method within the Corolla app.

```
Algorithm CholeskyFactorization(A):
Input:
    A : Matrix of coefficients (list of lists)

Output:
    L : Lower triangular matrix (list of lists)

n <- length of A
Initialize L as an n x n matrix of zeros

For i from 0 to n-1 do:
    For j from 0 to i do:
        sum_ <- sum of L[i][k] * L[j][k] for k from 0 to j-1

        If i equals j then:
            // Diagonal elements
            If A[i][i] - sum_ <= 0 then:
                Raise error "La matriz no es definida positiva."
            L[i][j] <- Square root of (A[i][i] - sum_)
        Else:
            L[i][j] <- (A[i][j] - sum_) / L[j][j]

Return L
```

3.10. Jacobi

The Jacobi method, also known as the Jacobi iteration method, is an iterative algorithm used to solve a strictly diagonally dominant system of linear equations. The method uses an initial guess to approximate the values of the variables iteratively, using the approximations from the previous iteration of the other terms to clear the approximation of each term in the current iteration. The following is the pseudocode of the implementation used in the Corolla app:

```
Algorithm JacobiMethod(a, b, x0, n, tol):
Input:
    a : Coefficient matrix (numpy array)
    b : Right-hand side vector (numpy array)
    x0: Initial guess (numpy array)
    n : Maximum number of iterations (integer)
    tol: Tolerance for convergence (float)

Output:
    tableListData: List of dictionaries containing iteration data
    message: Convergence message

Initialize tableListData as an empty list
    Initialize tempMapIterData as an empty dictionary

    // Decompose coefficient matrix a into lower triangular matrix l,
    upper triangular matrix u, and diagonal matrix d
    l = -np.tril(a, -1)
    u = -np.triu(a, 1)
    d = a + l + u

    // Calculate matrix t and vector c for the iteration formula
    t = inv(d) @ (l + u)
    c = inv(d) @ b

    // Check for convergence criteria based on the spectral radius of t
    If the maximum absolute value of eigenvalues of t is greater than 1, then:
        Return tableListData, "The function doesn't converge"

    // Check if the determinant of a is 0, which would prevent the method from being executed
    If the determinant of a is 0, then:
        Return tableListData, "det(A) is 0. The method cannot be executed."

    // Compute the first iteration
    xn = t @ x0 + c
    cont = 1
    error = max(abs(x0 - xn))

    // Record iteration data
    tempMapIterData['iteracion'] = str(cont)
    tempMapIterData['x0'] = str(x0)
    tempMapIterData['xn'] = str(xn)
    tempMapIterData['E'] = '{:.1e}'.format(error).replace('error-0', 'error-')
    tableListData.append(tempMapIterData.copy())
    tempMapIterData.clear()

    // Perform subsequent iterations until convergence or maximum iterations reached
    While (x0 != xn).all() and cont < n and error > tol do:
        x0 = xn
        xn = t @ x0 + c
        cont += 1
        error = max(abs(x0 - xn))

        // Round xn to avoid floating point precision errors
        xn = np.around(xn, decimals=10)

        // Record iteration data
        tempMapIterData['iteracion'] = str(cont)
        tempMapIterData['x0'] = str(x0)
        tempMapIterData['xn'] = str(xn)
        tempMapIterData['E'] = '{:.1e}'.format(error).replace('error-0', 'error-')
```

```
        tableListData.append(tempMapIterData.copy())
        tempMapIterData.clear()

    // Return iteration data and convergence message
    Return tableListData, "It converges at point = " + str(xn)
```

3.11. Gauss-Seidel

The Gauss-Seidel method is a method for solving systems of linear equations similar to Jacobi's, but with the difference that for the estimates of each value, the latest calculated estimates of the other values are used, even if these estimates come from the current iteration. The pseudocode that describes the used implementation is as follows.

```
Algorithm GaussSeidelMethod(a, b, x0, n, tol):
    Input:
        a : Coefficient matrix (numpy array)
        b : Right-hand side vector (numpy array)
        x0: Initial guess (numpy array)
        n : Maximum number of iterations (integer)
        tol: Tolerance for convergence (float)

    Output:
        tableListData: List of dictionaries containing iteration data
        message: Convergence message

    Initialize tableListData as an empty list
    Initialize tempMapIterData as an empty dictionary

    // Decompose coefficient matrix a into lower triangular matrix l, upper triangular matrix u, and dia
    l = -np.tril(a, -1)
    u = -np.triu(a, 1)
    d = a + l + u

    // Calculate matrix t and vector c for the iteration formula
    t = inv(d - l) @ u
    c = inv(d - l) @ b

    // Check for convergence criteria based on the spectral radius of t
    If the maximum absolute value of eigenvalues of t is greater than 1, then:
        Return tableListData, "The function doesn't converge"

    // Check if the determinant of a is 0, which would prevent the method from being executed
    If the determinant of a is 0, then:
        Return tableListData, "det(A) is 0. The method cannot be executed."

    // Compute the first iteration
    xn = t @ x0 + c
    cont = 1
        error = max(abs(x0 - xn))

        // Record iteration data
        tempMapIterData['iteracion'] = str(cont)
        tempMapIterData['x0'] = str(x0)
        tempMapIterData['xn'] = str(xn)
        tempMapIterData['E'] = '{:.1e}'.format(error).replace('error-0', 'error-')
        tableListData.append(tempMapIterData.copy())
        tempMapIterData.clear()

        // Perform subsequent iterations until convergence or maximum iterations reached
        While (x0 != xn).all() and cont < n and error > tol do:
            x0 = xn
            xn = t @ x0 + c
            cont += 1
            error = max(abs(x0 - xn))

            // Round xn to avoid floating point precision errors
            xn = np.around(xn, decimals=10)

            // Record iteration data
            tempMapIterData['iteracion'] = str(cont)
            tempMapIterData['x0'] = str(x0)
            tempMapIterData['xn'] = str(xn)
            tempMapIterData['E'] = '{:.1e}'.format(error).replace('error-0', 'error-')
            tableListData.append(tempMapIterData.copy())
            tempMapIterData.clear()

        // Return iteration data and convergence message
        Return tableListData, "It converges at point = " + str(xn)
```

3.12. SOR

SOR is an adaptation of the Gauss-Seidel method that uses a relaxation parameter that seeks to accelerate the convergence of the method. The expression that allows calculating the values for the iteration given by:

$$\phi_i^{(k+1)} = \frac{b_i - \sum_{j\neq i} a_{ij}\phi_j^{(k)}}{a_{ii}}$$

becomes

$$\phi_i^{(k+1)} = (1-w)\phi_i^{(k)} + \frac{w}{a_{ii}}(b_i - \sum_{j\neq i} a_{ij}\phi_j^{(k)})$$

by adding the relaxation parameter w. The implemented pseudocode is as follows.

```
    Algorithm SORMethod(a, b, x0, w, tol, n):
    Input:
        a : Coefficient matrix (list of lists or numpy array)
        b : Right-hand side vector (list or numpy array)
        x0: Initial guess (list or numpy array)
    w : Relaxation parameter (float)
    tol: Tolerance for convergence (float)
    n : Maximum number of iterations (integer)

Output:
    tableListData: List of dictionaries containing iteration data
    message: Convergence message

// Convert input matrices and vectors to numpy arrays
a = np.array(a)
b = np.array(b)

// Initialize tableListData as an empty list
Initialize tableListData as an empty list

// Initialize tempMapIterData as an empty dictionary
Initialize tempMapIterData as an empty dictionary

// Initialize cont to 0
Initialize cont to 0

// Initialize phi as a copy of x0
Initialize phi as a copy of x0

// Compute the initial error using the difference between Ax and b
Compute the initial error using the difference between Ax and b
error = np.max(np.abs(np.matmul(a, phi) - b))

// While loop to iterate until convergence criteria are met or maximum iterations reached
While error > tol and cont < n do:
    // Iterate over each row of the coefficient matrix a
    Iterate over each row of the coefficient matrix a:
        // Initialize sigma to 0
        Initialize sigma to 0

        // Iterate over each column of the coefficient matrix a
        Iterate over each column of the coefficient matrix a:
            // If the column index is not equal to the row index, compute the sum of products
            If the column index is not equal to the row index:
                Add the product of the element a[i, j] and phi[j] to sigma

        // Update the value of phi[i] using the SOR formula
        Update the value of phi[i] using the SOR formula
        phi[i] = (1 - w) * phi[i] + (w / a[i, i]) * (b[i] - sigma)

    // Recompute the error using the updated phi
    Recompute the error using the updated phi
    error = np.max(np.abs(np.matmul(a, phi) - b))
```

```
        // Increment the iteration counter
        Increment the iteration counter
        cont += 1

        // Round the elements of phi to avoid floating-point precision errors
        Round the elements of phi to avoid floating-point precision errors
        phi = np.around(phi, decimals=10)

        // Record iteration data
        Record iteration data:
            tempMapIterData['iteracion'] = str(cont)
            tempMapIterData['E'] = '{:.1e}'.format(error).replace('e-0', 'e-')
            tempMapIterData['xn'] = str(phi)
            Append a copy of tempMapIterData to tableListData and then clear tempMapIterData

    // Return iteration data and convergence message
    Return tableListData and a convergence message stating the final point
    return tableListData, f'It converges at point = {str(phi)}'
```

3.13. Vandermonde

The Vandermonde method is a polynomial interpolation method that takes advantage of the fact that the interpolation polynomial is of degree at most n-1, where n is the number of points. The method consists of converting the problem into a system of linear equations with a coefficient matrix, known as the Vandermonde matrix of size n*n. To solve the system, we use the inverse of the Vandermonde matrix and a vector b that contains the y coordinates of the points. This allows us to solve for the n-1 coefficients and the independent term of the interpolation polynomial. Please verify that this is correct. The following is the pseudocode which describes the algorithm used.

```
    Algorithm Vandermonde(x, y):
    Input:
        x : Array of x values (list or numpy array)
        y : Array of y values (list or numpy array)

    Output:
        data : Dictionary containing Vandermonde matrix and polynomial coefficients

    // Convert input arrays to numpy arrays
    xn = np.array(x)
    yn = np.array([y]).T

    // Create the Vandermonde matrix A
    A = np.vander(xn)

    // Compute the inverse of the Vandermonde matrix
    Ainv = np.linalg.inv(A)

    // Compute the coefficients of the polynomial
```

```
a = np.dot(Ainv, yn)

// Initialize tableListData with the Vandermonde matrix A
tableListData = A

// Initialize an empty list to store the coefficients
coeficientes = []

// Format and store the coefficients in the list
for i in a:
    num = '{:.10f}'.format(round(i[0], 10))
    coeficientes.append(num)

// Build the polynomial expression from the coefficients
fun = ''
size = len(a) - 1

for i in coeficientes:
    // Append the term with its coefficient and power of x to the polynomial expression
    if i[0] != '-':
        fun += f' + {i}*x^{size} '
    else:
        fun += f'{i}*x^{size} '

    // Decrement the power of x for the next term
    size -= 1

// Replace unnecessary terms to simplify the polynomial expression
fun = fun.replace('*x^1', '*x').replace('*x^0', '')

// Store the Vandermonde matrix and polynomial coefficients in a dictionary
data = {
    'resultados' : tableListData,
    'coeficientes' : fun
}

// Return the dictionary containing Vandermonde matrix and polynomial coefficients
return data
```