Final Project Report: Numerical Methods Tool

Alejandro Torres Muñoz

November 12, 2024

Numerical Methods Project
Developed using Django as a web application

Course: Numerical Analysis

Teacher: Alejandro Arenas Vasco

University: Universidad EAFIT

Project Description

This project involves the creation of a self-contained numerical tool that includes various methods learned in class. The tool is a website developed using Django and allows users to perform calculations for multiple numerical methods.

Instructions

To use the numerical tool:

1. Visit the website and enter the formula in LaTeX format in the input field provided.
2. Define the required parameters, such as the lower limit, tolerance, and maximum number of iterations.
3. Click "Calculate" to view the results for each method.

Each method page provides a description, instructions, and a detailed table of results.

Chapter 1

Single Variable Equations

1.1 Incremental Search

What is the Incremental Search Method?

The Incremental Search method is a root-finding technique used to locate roots in continuous functions by incrementally evaluating the function over an interval. This method works by detecting a change in sign between consecutive points, which indicates the presence of a root within that interval.

How it Works

The method begins at a starting point a and increments by a specified tolerance Δx. At each step, the function is evaluated at two points, and if a sign change is detected, it confirms the presence of a root within that interval. This approach is particularly useful for functions with multiple roots over an extended domain.

Implementation

The method was implemented by taking the function as input in LaTeX format, which is then converted to a symbolic expression using the SymPy library. The function is evaluated incrementally, and if a sign change occurs, the interval is stored as a potential root location.

Example

An example of using the Incremental Search method for finding the roots of the function $f(x) = x^2 - 4$ is shown below. Given the parameters:

- **Equation:** $x^2 - 4$
- **Lower Limit:** $a = 0$
- **Delta (Tolerance):** $\Delta x = 0.5$
- **Iterations:** $k = 10$

The result indicates that the method converged to a solution in 3 iterations. The interval containing the root is $[1.5, 2.0]$.

Interval with Roots
$[1.5, 2.0]$

The message displayed is:

The method converged to the solution in 3 iterations.

1.2 Bisection

What is the Bisection Method?

The Bisection method is a numerical technique used to find roots of continuous functions. It works by repeatedly dividing an interval in half and selecting subintervals that contain a sign change, narrowing down the location of the root.

How it Works

The method requires an initial interval $[a, b]$ where the function changes sign, ensuring that a root exists within the interval. The midpoint is calculated and evaluated; depending on the sign of the midpoint, the interval is halved to isolate the root further. This process continues until the interval reaches a specified tolerance.

Implementation

In the implementation, the function is input as LaTeX and converted to SymPy format. The method tracks iterations and error, with different error types (correct decimals and significant figures) handled according to user preference. Each iteration updates the interval based on sign changes at the midpoint.

Example

The following is an example of the Bisection method applied to the function $f(x) = x^2 - 4$ with the following parameters:

- **Equation:** $x^2 - 4$
- **Tolerance:** 1×10^{-6}
- **Iterations:** 20
- **Lower Bound (a):** 0
- **Upper Bound (b):** 3
- **Type of Error:** Correct Decimals

The method converged to the solution $x = 2.00000009536743164$ in 20 iterations, as shown in the table below.

Iterations	a	b	f(c)	Error
1	1.5	3.0	-1.75	1.5
2	1.5	2.25	1.0625	0.75
3	1.875	2.25	-0.484375	0.375
4	1.875	2.0625	0.25390625	0.1875
...	...	...	...	...
20	1.999998	2.000001	3.81469817151719e-6	2.86102294921875e-6

The message displayed is:

The method converged to the solution 2.00000009536743164.

1.3 False Position

What is the False Position Method?

The False Position method, or Regla Falsa, is a numerical technique for finding roots of continuous functions. Unlike the Bisection method, it uses a weighted average based on the function values at the endpoints of the interval, which helps converge faster for some types of functions.

How it Works

The method starts with an interval $[a, b]$ where the function changes sign, ensuring a root within the interval. Instead of using the midpoint, the False Position method computes a point closer to the root based on the line joining the function values at the endpoints. This process continues, adjusting the interval based on the sign of the function at the computed point, until the tolerance is met.

Implementation

In this implementation, the function is input as LaTeX and converted to SymPy format. The method calculates the root estimate at each step and checks the error against a specified tolerance, with options for different error types.

Example

The following example demonstrates the False Position method for the function $f(x) = x^3 - x - 2$ with the following parameters:

- **Equation:** $x^3 - x - 2$
- **Upper Limit:** $b = 2$
- **Lower Limit:** $a = 1$
- **Tolerance:** 1×10^{-6}
- **Iterations:** 20
- **Type of Error:** Correct Decimals

The method converged to the solution $x = 1.52137946179016$ with an error less than 1×10^{-6} in 11 iterations, as shown in the table below.

Valor de a	Valor de b	Valor de xm	Error
2.0	1.33333333333333	1.33333333333333	1.000001
2.0	1.46268656716418	1.46268656716418	0.129353238380846
2.0	1.50401900394995	1.50401900394995	0.0413324367857701
2.0	1.51633056476026	1.51633056476026	0.0123115608103141
...	...	...	...
2.0	1.52137946179016	1.52137946179016	6.03386263176020e-7

The message displayed is:

The method converged to the solution 1.52137946179016 with an error less than 1×10^{-6}.

1.4 Fixed Point

What is the Fixed Point Method?

The Fixed Point method is an iterative technique for solving equations of the form $x = g(x)$. It transforms an equation $f(x) = 0$ into an equivalent form $x = g(x)$, where the solution of this form represents the root of $f(x)$. The convergence of the method depends on the choice of $g(x)$ and its derivative's properties.

How it Works

To apply the Fixed Point method, a function $g(x)$ is chosen so that the solution to $x = g(x)$ is the desired root. Starting from an initial approximation x_0, the method iteratively calculates $x_{n+1} = g(x_n)$ until the difference between x_{n+1} and x_n is below a specified tolerance, or until a maximum number of iterations is reached.

The steps for the Fixed Point method are as follows:

1. Choose a function $g(x)$ and an initial guess x_0.
2. Calculate $x_{n+1} = g(x_n)$.
3. Compute the error $|x_{n+1} - x_n|$.
4. Repeat steps 2 and 3 until the error is less than the specified tolerance or the maximum iterations are reached.

Implementation

The implementation of this method takes the function $g(x)$ as input in LaTeX format, which is then converted to a symbolic expression using the SymPy library. The function is evaluated iteratively, with each approximation being checked against the tolerance for convergence.

Example

The following example demonstrates the Fixed Point method applied to the function $g(x) = \sqrt{1+x}$ with the given parameters:

- **Equation:** $g(x) = \sqrt{1+x}$
- **Tolerance:** 1×10^{-6}
- **Iterations:** 30
- **Initial X:** $x_0 = 0.5$
- **Type of Error:** Correct Decimals

The method converged to the solution $x = 1.61803367285288$ in 13 iterations, as shown in the table below.

Valor de G(xi)	Valor de xi	Error
1.22474487139159	1.22474487139159	0.724744871391589
1.49155786726214	1.49155786726214	0.266812995870553
1.57846693575195	1.57846693575195	0.0869090684898055
1.60576054745156	1.60576054745156	0.0272936116996170
...	...	...
1.61803367285288	1.61803367285288	7.0637100811178e-7

The message displayed is:

The method converged to the solution 1.61803367285288 with an error less than 1×10^{-6}.

1.5 Newton-Raphson

What is the Newton-Raphson Method?

The Newton-Raphson method is an iterative technique for finding successively better approximations to the roots (or zeroes) of a real-valued function. It is particularly powerful for functions that are differentiable and is often faster than other root-finding methods when the initial guess is close to the actual root.

How it Works

Starting from an initial guess x_0, the method uses the function $f(x)$ and its derivative $f'(x)$ to estimate the root with the following formula:

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

The process is repeated until the difference between successive approximations is within a specified tolerance or until a maximum number of iterations is reached.

Implementation

In the implementation, the function is entered in LaTeX format and converted to a symbolic expression using the SymPy library. The method then evaluates the function and its derivative at each step to update the estimate for the root.

Example

For the function $f(x) = x^2 - 4$, the Newton-Raphson method was applied with the following parameters:

- **Equation:** $x^2 - 4$
- **Initial Value:** $x = 1$
- **Tolerance:** 1×10^{-6}
- **Iterations:** 20
- **Type of Error:** Correct Decimals

The method converged to the solution $x = 2.00000009292229$ in 5 iterations, as shown in the table below.

x	f(x)	Error
1.0	-3.00000000000000	1.50000000000000
2.5	2.25000000000000	0.45000000000000
2.05	0.20250000000000	0.0493902439024390
2.00060975609756	0.00243939619274158	0.000609663175266117
2.00000009292229	3.71689187872448e-7	9.2922925272200e-8

The message displayed is:

The method converged to the solution in 5 iterations.

1.6 Secante

What is the Secant Method?

The Secant method is an iterative root-finding algorithm that approximates the root of a function by using secant lines instead of tangents. It is particularly useful for cases where the derivative of the function is difficult to compute, as it only requires two initial approximations to calculate the root.

How it Works

The method begins with two initial points, x_0 and x_1, and generates successive approximations by intersecting the secant line of the function at those points. The next approximation is computed iteratively using the formula:

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

This process continues until the error between successive approximations is less than a predefined tolerance or the maximum number of iterations is reached.

Implementation

In the implementation, the function is input in LaTeX format and converted to SymPy format for evaluation. The initial points are specified by the user, along with the tolerance and the number of iterations. The method checks the error at each step and stops if the desired accuracy is achieved.

Example

The following is an example of the Secant method applied to the function $f(x) = x^3 + x - 1$ with the following parameters:

- **Equation:** $x^3 + x - 1$
- **Tolerance:** 1×10^{-6}
- **Iterations:** 15
- **Initial Points:** $x_1 = 4$, $x_0 = 2$
- **Type of Error:** Correct Decimals

The method converged to the solution $x = 0.682327803940224$ with an error less than 1×10^{-6} in 10 iterations, as shown in the table below.

Iterations	Xi	f(xi)	Error
0	2.0	9.00000000000000	-
1	4.0	67.0000000000000	2.31034482758621
2	1.68965517241379	5.51351018901964	0.207169246223104
3	1.48248592619069	3.74064089817842	0.437113869178267
...	...	...	...
10	0.682327803940224	2.68921551693779e-10	7.56483601649371e-7

The message displayed is:

The method converged to the solution 0.682327803940224 in 10 iterations.

1.7 Multiple Roots

What is the Multiple Roots Method?

The Multiple Roots method is used to find roots in functions that have multiplicity, meaning they touch the x-axis but do not cross it. This method is an extension of Newton-Raphson, modified to handle such cases by including second derivatives in the calculation.

How it Works

In this method, an initial approximation x_0 is required. At each iteration, the method evaluates the function and its first and second derivatives. Using these values, a new approximation for the root is calculated. This process continues until the error falls below a specified tolerance or the maximum number of iterations is reached.

Implementation

The implementation involves taking the function in LaTeX format, converting it to SymPy for symbolic manipulation, and using the function's derivatives. The method updates the approximation iteratively and stops based on the tolerance or iteration limit.

Example

An example of using the Multiple Roots method for finding the root of the function $f(x) = (x-1)^2$ is given below. The parameters are as follows:

- **Equation:** $(x-1)^2$
- **Tolerance:** 1×10^{-6}
- **Iterations:** 20
- **Initial Value (x):** 1.5
- **Type of Error:** Correct Decimals

The method converged to the solution $x = 1.00000000000000$ with an error less than 1×10^{-6} in 2 iterations, as shown in the table below.

Valor de F(xi)	Valor de xi	Error
0.250000000000000000	1.00000000000000	0.500000000000000
0	1.00000000000000	0

The message displayed is:

The method converged to the solution $x = 1.00000000000000$ *with an error less than* 1×10^{-6}.

Chapter 2

System of Equations

2.1 Simple Gaussian Elimination

What is Simple Gaussian Elimination?

Simple Gaussian elimination is a method used to solve systems of linear equations. This method transforms the original system into an upper triangular system, which can then be solved by back-substitution. The process involves performing elementary row operations on the augmented matrix to achieve zeros below the main diagonal.

How it Works

The simple Gaussian elimination method is carried out in two stages:

1. **Transformation to Upper Triangular Form:** The matrix is converted to an upper triangular form by applying row operations to make zeros in the positions below the main diagonal.
2. **Back-Substitution:** Once the matrix is in upper triangular form, the system is solved by back-substitution, starting from the last equation.

Implementation

In this implementation, the system of equations is entered by specifying matrix A and vector b. The function performs row operations to convert A into an upper triangular matrix and then applies back-substitution to find the solution.

Example

An example of using simple Gaussian elimination to solve the following system with matrix A and vector b:

$$A = \begin{bmatrix} 4 & 3 & -2 & -7 \\ 3 & 12 & 8 & -3 \\ 2 & 3 & -9 & 3 \\ 1 & -2 & -5 & 6 \end{bmatrix}, \quad b = \begin{bmatrix} 20 \\ 18 \\ 31 \\ 12 \end{bmatrix}$$

After applying the Gaussian elimination steps, the resulting matrices are:

- **Lower Triangular Matrix L:**

$$L = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.75 & 1.0 & 0.0 & 0.0 \\ 0.5 & 0.15384615384615385 & 1.0 & 0.0 \\ 0.25 & -0.28205128205128205 & 0.1924119241192412 & 1.0 \end{bmatrix}$$

- **Upper Triangular Matrix U:**

$$U = \begin{bmatrix} 4.0 & 3.0 & -2.0 & -7.0 \\ 0.0 & 9.75 & 9.5 & 2.25 \\ 0.0 & 0.0 & -9.461538461538462 & 6.153846153846154 \\ 0.0 & 0.0 & 0.0 & 7.200542005420054 \end{bmatrix}$$

Using back-substitution on the upper triangular matrix, the solution vector x is:

$$x = \begin{bmatrix} 3.5705683101242 \\ 1.9552126458411745 \\ -1.8189687617631852 \\ 0.5408355287918704 \end{bmatrix}$$

The message displayed is:

The method successfully obtained the solution vector x using Simple Gaussian Elimination.

2.2 Gaussian Elimination with Partial Pivoting

What is Gaussian Elimination with Partial Pivoting?

Gaussian Elimination with Partial Pivoting is a variation of the Gaussian Elimination method for solving systems of linear equations. It includes the step of pivoting, which helps to reduce numerical errors by rearranging the rows of the matrix based on the largest pivot element (in absolute value) in each column. This pivoting improves the stability and accuracy of the results, especially when the matrix has very small or zero values that could lead to division errors.

How it Works

The steps of Gaussian Elimination with Partial Pivoting are as follows:

1. For each column in the matrix, identify the row with the largest absolute value in that column (the pivot element).
2. Swap the current row with the row containing the pivot element.
3. Perform Gaussian elimination by using the pivot element to eliminate all elements below it in the column.
4. Continue to the next column and repeat the process until the matrix is in an upper triangular form.
5. Perform back-substitution to solve for the values of the variables.

Implementation

The implementation of Gaussian Elimination with Partial Pivoting takes matrix A and vector b as inputs. The algorithm proceeds through the matrix, identifying pivot elements, performing row swaps, and applying Gaussian elimination. Once the matrix is in an upper triangular form, back-substitution is used to find the solution vector x.

Example

Consider the following system of equations represented by matrix A and vector b:

$$A = \begin{bmatrix} 0 & 2 & 1 & 2 \\ 1 & 0 & 1 & 3 \\ 3 & 1 & -4 & 2 \\ -4 & 0 & 1 & 1 \end{bmatrix}, \quad b = \begin{bmatrix} 5 \\ 5 \\ 2 \\ -2 \end{bmatrix}$$

The parameters for this example are:

- **Matrix Size:** 4×4

After applying Gaussian Elimination with Partial Pivoting, the solution vector x is:

$$x = \begin{bmatrix} 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \end{bmatrix}$$

Matrix A:			
0.0	2.0	1.0	2.0
1.0	0.0	1.0	3.0
3.0	1.0	-4.0	2.0
-4.0	0.0	1.0	1.0

Vector b:
5.0
5.0
2.0
-2.0

Solution x:			
1.0	0.9999999999999996	1.0000000000000002	1.0000000000000002

The message displayed is:

Gaussian Elimination with Partial Pivoting completed, yielding the solution vector x.

2.3 Doolittle

What is the Doolittle Method?

The Doolittle method is a type of LU decomposition used to solve systems of linear equations, invert matrices, or calculate determinants. It decomposes a given matrix A into a product of a lower triangular matrix L and an upper triangular matrix U, where L has ones on its diagonal.

How it Works

The algorithm starts with a matrix A and iteratively computes the entries of L and U such that $A = LU$. The diagonal entries of L are set to 1, and the remaining entries of L and U are calculated by solving the equations that arise from the decomposition.

Implementation

In this implementation, the matrix A is received as input. The method then proceeds to fill in the values of L and U based on the decomposition formulas. The final result includes the original matrix A, as well as the computed matrices L and U.

Example

Below is an example using the Doolittle method on the following 4×4 matrix:

$$A = \begin{bmatrix} 2 & -1 & 1 & 3 \\ 4 & 2 & -2 & 1 \\ -2 & -1 & 4 & -2 \\ 1 & 3 & -3 & 1 \end{bmatrix}$$

The Doolittle decomposition yielded the following results:

- **Matrix L:**

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ -1 & -0.5 & 1 & 0 \\ 0.5 & 0.875 & 0 & 1 \end{bmatrix}$$

- **Matrix U:**

$$U = \begin{bmatrix} 2 & -1 & 1 & 3 \\ 0 & 4 & -4 & -5 \\ 0 & 0 & 3 & -1.5 \\ 0 & 0 & 0 & 3.875 \end{bmatrix}$$

The message displayed is:

Doolittle decomposition completed.

2.4 Crout

What is the Crout Method?

The Crout method is a variant of the LU decomposition technique for solving systems of linear equations. It decomposes a given matrix A into a lower triangular matrix L and an upper triangular matrix U where the diagonal elements of U are all equal to one.

How it Works

The Crout decomposition starts with an assumption that U has ones on its diagonal. The algorithm then proceeds row by row to calculate the elements of L and U. This approach reduces the computational complexity and provides stability when solving linear systems.

Implementation

In the implementation, the input matrix is taken in LaTeX format and converted for computation. The algorithm calculates the values of matrices L and U iteratively, ensuring that $A = LU$ holds.

Example

Consider the following example matrix A to be decomposed using the Crout method:

$$A = \begin{bmatrix} 2 & -2 & 2 & -2 \\ 1 & -3 & 3 & -1 \\ 0 & -1 & 3 & 2 \\ 1 & -2 & -1 & -4 \end{bmatrix}$$

Given the parameters:

- **Matrix Size:** 4×4

The Crout method decomposes the matrix into:

$$L = \begin{bmatrix} 2.0 & 0.0 & 0.0 & 0.0 \\ 1.0 & -2.0 & 0.0 & 0.0 \\ 0.0 & -1.0 & 2.0 & 0.0 \\ 1.0 & -1.0 & -3.0 & 0.0 \end{bmatrix}$$

$$U = \begin{bmatrix} 1.0 & -1.0 & 1.0 & -1.0 \\ 0.0 & 1.0 & -1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 1.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

The message displayed is:

The Crout decomposition successfully produced matrices L and U that satisfy $A = LU$.

2.5 Cholesky

What is the Cholesky Method?

The Cholesky decomposition is a method used to solve systems of linear equations, where the matrix is decomposed into a product of a lower triangular matrix and its transpose. It is especially useful for solving positive definite matrices and is commonly used in various numerical and scientific computations.

How it Works

The Cholesky method factors a symmetric, positive-definite matrix A into the product of a lower triangular matrix L and its transpose, $A = LL^T$. This decomposition simplifies solving linear equations by substituting back, making it computationally efficient for certain types of matrices.

Implementation

The input matrix is entered by the user, which is then checked for compatibility with the Cholesky decomposition criteria (symmetric and positive-definite). The decomposition is computed, and the resulting matrix L is presented to the user. In this implementation, only the matrix L is displayed, as the transpose is implicitly understood.

Example

An example of using the Cholesky method with the following matrix:

$$A = \begin{pmatrix} 4 & 12 & -16 \\ 12 & 37 & -43 \\ -16 & -43 & 98 \end{pmatrix}$$

The Cholesky decomposition yields:

- **Matrix A:**

$$\begin{pmatrix} 4.0 & 12.0 & -16.0 \\ 12.0 & 37.0 & -43.0 \\ -16.0 & -43.0 & 98.0 \end{pmatrix}$$

- **Result L:**

$$\begin{pmatrix} 2.0 & 0.0 & 0.0 \\ 6.0 & 1.0 & 0.0 \\ -8.0 & 5.0 & 3.0 \end{pmatrix}$$

The message displayed is:

Cholesky decomposition was successful, and matrix L was obtained as shown.

2.6 Jacobi

What is the Jacobi Method?

The Jacobi method is an iterative algorithm used to solve systems of linear equations. It relies on the idea of decomposing the matrix into diagonal, lower, and upper parts and iteratively solving for each variable. This method is especially useful for systems where the matrix is diagonally dominant.

How it Works

The Jacobi method starts with an initial guess for each variable in the system of equations. For each iteration, the value of each variable is updated based on the values from the previous iteration. The process continues until the solution converges within a specified tolerance or the maximum number of iterations is reached.

Steps of the Jacobi Method:

1. Start with an initial guess vector $\mathbf{x}^{(0)}$.
2. For each iteration, compute each x_i using:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right)$$

   where a_{ij} are the elements of matrix A, b_i are the elements of vector B, and $x_j^{(k)}$ are the values from the previous iteration.

3. Calculate the error between the current and previous iterations.
4. Repeat until the error is below the tolerance or the maximum iterations are reached.

Implementation

The implementation of the Jacobi method involves receiving matrix A, vector B, and initial guess $\mathbf{x}^{(0)}$ as inputs. The algorithm then iteratively updates the values of x_i for each variable in the system.

Example

An example of using the Jacobi method to solve the system of equations represented by the following matrix A and vector B:

$$A = \begin{pmatrix} 10 & -1 & 2 \\ -1 & 11 & -1 \\ 2 & -1 & 10 \end{pmatrix}$$

$$B = \begin{pmatrix} 6 \\ 25 \\ -11 \end{pmatrix}$$

Given the parameters:

- **Matrix Size:** 3×3
- **Tolerance:** 1×10^{-6}
- **Iterations:** 20
- **Initial Guess:** $\mathbf{x}^{(0)} = (0, 0, 0)$

The Jacobi method converged to the solution:

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1.04326920462214 \\ 2.269230786387123 \\ -1.0817307936369862 \end{pmatrix}$$

after 13 iterations, as shown in the table below.

Iterations	**X1**	**X2**	**X3**
1	0.600000	2.272727	-1.100000
2	1.047273	2.227273	-0.992727
3	1.021273	2.277686	-1.086727
4	1.045114	2.266777	-1.076486
...	...	...	...
13	1.043269	2.269231	-1.081731

The message displayed is:

The method converged to the solution in 13 iterations.

2.7 Gauss-Seidel

What is the Gauss-Seidel Method?

The Gauss-Seidel method is an iterative technique used to solve systems of linear equations. This method is an improvement over the Jacobi method, where each equation is solved for one variable at a time, using the updated values of the previous variables within the same iteration.

How it Works

The method starts with an initial guess for the solution vector. For each iteration, the method updates each component of the solution vector based on the most recent values. This process continues until the error between iterations falls below a given tolerance or the maximum number of iterations is reached.

The steps for the Gauss-Seidel method are as follows:

1. Choose an initial guess vector X_0.
2. For each variable x_i, update it using the equation from the system, utilizing the latest values available.
3. Calculate the error $|x_{i,new} - x_{i,old}|$ for each component.
4. Repeat until the error is within the specified tolerance or the maximum iterations are reached.

Implementation

In this implementation, the system of equations is entered by specifying matrix A, vector B, and the initial vector X_0. The algorithm iterates through each element, updating it with the latest computed values to accelerate convergence.

Example

The following example demonstrates the Gauss-Seidel method for the system:

$$A = \begin{bmatrix} 10 & -1 & 2 \\ -1 & 11 & -1 \\ 2 & -1 & 10 \end{bmatrix}, \quad B = \begin{bmatrix} 6 \\ 25 \\ -11 \end{bmatrix}$$

with the parameters:

- **Matrix Size:** 3×3
- **Tolerance:** 1×10^{-6}
- **Iterations:** 20
- **Initial Vector** X_0**:** $\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$

The Gauss-Seidel method converged to the solution with the following values:

- $X_1 = 1.043269261330706$
- $X_2 = 2.269230742891677$
- $X_3 = -1.0817307779769734$

Iteration	X_1	X_2	X_3
1	1.030182	2.276628	-1.078374
2	1.030182	2.276628	-1.078374
3	1.043338	2.269542	-1.081713
4	1.043297	2.269235	-1.081736
5	1.043271	2.269230	-1.081731
6	1.043269	2.269231	-1.081731

The message displayed is:

The method converged to the solution in 6 iterations.

Chapter 3

Conclusions

Project Conclusions

This project provided a comprehensive exploration of several numerical methods, implemented as a web tool through Django. Each method was analyzed and developed in a structured way, providing users with a clear and interactive interface to understand and apply these mathematical techniques. Here are the key conclusions drawn from the project:

1. **Practical Application of Numerical Methods:** The tool allows users to input custom equations and parameters, applying methods such as Bisection, Newton-Raphson, Jacobi, and Gauss-Seidel, among others. This flexibility demonstrates the power of numerical methods in solving real-world problems, especially where analytical solutions are impractical.

2. **Iteration and Convergence:** One of the main insights gained from this project is the importance of iteration and convergence in numerical methods. Techniques like Gauss-Seidel and Jacobi highlight how iterative approaches can effectively reach approximate solutions, especially in large systems of equations. Convergence criteria, such as tolerance and maximum iterations, are crucial in controlling the precision of these methods.

3. **Challenges in Implementation:** Implementing each method revealed the inherent challenges of numerical computation. Issues like matrix decomposition in LU methods (e.g., Doolittle and Crout) or handling errors in iterative methods required careful planning and coding. Each method posed unique challenges, which reinforced the understanding of their theoretical foundations and limitations.

4. **Usability and Interactivity with Django:** Developing the tool as a Django web application made the project accessible and interactive, allowing users to experiment with different methods without needing advanced programming knowledge. This reinforces the importance of creating user-friendly tools in education, enhancing learning by making complex mathematical concepts approachable.

Overall, this project demonstrates the practicality of numerical methods and the effectiveness of web-based tools in teaching and applying complex mathematical concepts.