

22011082	مايكل مجدى نجيب عجبان قلّس
22011042	كيرلس سامى ابراهيم بشاى بخيت
22011029	كارولين حنا وهبه ميخائيل ساويرس
22010701	جون وليم فوزى فهمى يوسف
22010691	جنا محمد فتحي احمد محمد

1. Pseudocode and Complexity:

I. Gauss elimination:

##Function: GaussElimination(size,A, B, sig_figs)

1. *Input*:

- size: Integer, size of the matrix A.
- A: 2D array, a symmetric positive definite matrix.
- B: 1D array, the right-hand side vector.
- sig_figs: Integer, number of significant figures for rounding.

2. *Output*:

- X: Solution vector (1D array).
- execution_time: Time taken for the computation.

3. *Steps*:

1. *Start Timer*

2.Convert A and B to NumPy arrays.

3.For each row k from 0 to size - 1 (forward elimination loop):

- 1.Call partial_pivoting(A, B, size, k).
- 2.If $A[k][k] == 0$ or $\text{abs}(A[k][k]) < 1e-10$:
Raise Error "Pivot is too small or zero".
- 3.For each row i from k+1 to size-1:
 - 1.Calculate factor = RoundOff($A[i][k] / A[k][k]$, sig_figs).
 - 2.Set $A[i][k] = 0$ (make the element below the pivot zero).
 - 3.For each column j from k+1 to size-1:
 - 1.Update $A[i][j] -= \text{RoundOff}(\text{factor} * A[k][j], \text{sig_figs})$.
 - 4.Update $B[i] -= \text{RoundOff}(\text{factor} * B[k], \text{sig_figs})$.

4.Back Substitution:

- 1.Initialize solutions = [0] * size.
- 2.For each row k from size-1 down to 0 (back substitution loop):
 - 1.Calculate the sum: sum_of_terms = B[k].
 - 2.For each column i from k+1 to size-1:
 - 1.Update sum_of_terms -= $A[k][i] * \text{solutions}[i]$.
 - 3.Update $\text{solutions}[k] = \text{RoundOff}(\text{sum_of_terms} / A[k][k], \text{sig_figs})$.

5.End Timer and calculate execution time.

6.Return: solutions, execution_time.

*Helper Function*: RoundOff(value, sig_figs)

- *Input*: value (float), sig_figs (int).

- *Output*: Rounded value to sig_figs.

*Helper Function*: partial_pivoting(A, B, size, k):

- *Input*: A: Coefficient matrix, B: Constants vector, size: Size of the matrix, k: Current row ind.

Output: None (modifies A and B in place).

Description: Finds the row with the largest absolute value in column k and swaps it with row k.

If the pivot element is 0, raises an error indicating that the matrix is singular.

Time complexity:

- a. **Forward Elimination: $O(n^3)O(n^3)$ for converting the matrix into an upper triangular form.**
- b. **Backward Substitution: $O(n^2)O(n^2)$ for solving the equations from the triangular form.**
- c. **Total Complexity: $O(n^3)O(n^3)$, as the $O(n^3)O(n^3)$ term dominates**

Gauss Jordan:

##Function: GaussJordan(size,A, B, sig_figs)

1.*Input*:

- size: Integer, size of the matrix A.
- A: 2D array, a symmetric positive definite matrix.
- B: 1D array, the right-hand side vector.
- sig_figs: Integer, number of significant figures for rounding.

2. *Output*:

- X: Solution vector (1D array).
- execution_time: Time taken for the computation.

3. *Steps*:

1.*Start Timer*

2.Convert A and B to NumPy arrays.

3.For each row k from 0 to size - 1 (**forward elimination loop**):

1.Call partial_pivoting(A, B, size, k)

2. Set pivot = A[k][k]

3. For each column j from 0 to size-1:

- Normalize row k by dividing each element by the pivot
- $A[k][j] = \text{Round_off}(A[k][j] / \text{pivot}, \text{sig_figs})$

4. Set $B[k] = \text{Round_off}(B[k] / \text{pivot}, \text{sig_figs})$

5.For each row l from k+1 to size-1 (this loop eliminates elements below the pivot):

1.Set factor = A[l][k]

2.Set A[l][k] = 0

Set the element below the pivot to zero (this eliminates the coefficient below the pivot).

3. For each column j from k+1 to size-1:

$A[l][j] = \text{Round_off}(A[l][j] - \text{factor} * A[k][j], \text{sig_figs})$

4. Set $B[l] = \text{Round_off}(B[l] - \text{factor} * B[k], \text{sig_figs})$.

4. *Back Elimination:*

1. For each row i from k-1 down to 0 (this loop eliminates elements above the pivot):

1. **Set** factor = A[i][k]

2. **Set** A[i][k] = 0 (this eliminates the coefficient above the pivot).

3. **For each column j from k+1 to size-1:**

A[i][j] = Round_off(A[i][j] - factor * A[k][j], sig_figs)

4. **Set** B[i] = Round_off(B[i] - factor * B[k], sig_figs)

5. End Timer and calculate execution time.

6. Return: B, execution_time.

***Helper Function*: RoundOff(value, sig_figs)**

- ***Input*:** value (float), sig_figs (int).

- ***Output*:** Rounded value to sig_figs.

***Helper Function*: partial_pivoting(A, B, size, k):**

- ***Input*:** A: Coefficient matrix, B: Constants vector, size: Size of the matrix, k: Current row ind.

Output: None (modifies A and B in place).

Time complexity:

Total Complexity: Still $O(n^3)$, but with a higher constant factor than Gaussian Elimination.

II. LU decomposition:

1-Doolittle:

Pseudocode for LU Decomposition with Pivoting

***Main Function*: solve_using_lu(n, A, B, sig_figs)**

1. *Input*:

- n: Integer representing the size of matrix A.
- A: 2D list or matrix representing the coefficient matrix.
- B: 1D list representing the right-hand side vector.
- sig_figs: Integer representing the number of significant figures for rounding.

2. *Output*:

- x: 1D list representing the solution vector.
- execution_time: Time taken for the entire computation.

3. *Steps*:

1. Start Timer.
2. Convert A and B to numpy arrays (of type float).
3. Call lu_decomposition_with_pivoting(n, A, B, sig_figs) to decompose matrix A into P, L, and U matrices.
4. Compute Pb = P * B (matrix multiplication of P and B).
5. Call forward_substitution(L, Pb, sig_figs) to solve the equation Ly = Pb for y.

6. Call `backward_substitution(U, y, sig_figs)` to solve the equation $Ux = y$ for x .
7. End Timer and compute `execution_time`.
8. Return the solution vector x and `execution_time`.

Helper Function: `lu_decomposition_with_pivoting(n, A, B, sig_figs)`

1. *Input*:

- n : Integer representing the size of matrix A .
- A : 2D list representing the coefficient matrix.
- B : 1D list representing the right-hand side vector.
- sig_figs : Integer representing the number of significant figures for rounding.

2. *Output*:

- P : 2D list representing the permutation matrix.
- L : 2D list representing the lower triangular matrix.
- U : 2D list representing the upper triangular matrix.

3. *Steps*:

1. Initialize matrix L as a zero matrix of size $n \times n$.
2. Initialize matrix U as a zero matrix of size $n \times n$.
3. Initialize matrix P as an identity matrix of size $n \times n$.
4. Loop through each row i from 0 to $n-1$:
 1. *Partial Pivoting*: Find the row with the largest element in column i and swap it with the current row.
 2. Decompose the matrix into upper (U) and lower (L) matrices by:
 - For each element $U[i][j]$, subtract the dot product of $L[i, :i]$ and $U[:i, j]$ and round off.
 - For each element $L[j, i]$, subtract the dot product of $L[j, :i]$ and $U[:i, i]$, divide by $U[i, i]$, and round off.
5. Return matrices P , L , and U .

Helper Function: `forward_substitution(L, B, sig_figs)`

1. *Input*:

- L : 2D list representing the lower triangular matrix.
- B : 1D list representing the right-hand side vector.
- sig_figs : Integer representing the number of significant figures for rounding.

2. *Output*:

- y : 1D list representing the solution vector for the equation $Ly = B$.

3. *Steps*:

1. Initialize y as a zero vector of size n .
2. Loop through each row i from 0 to $n-1$:
 1. Compute the value of $y[i]$ by subtracting the dot product of $L[i, :i]$ and $y[:i]$ from $B[i]$, then round off to `sig_figs`.
3. Return the solution vector y .

Helper Function: `backward_substitution(U, y, sig_figs)`

1. *Input*:

- U : 2D list representing the upper triangular matrix.
- y : 1D list representing the right-hand side vector (solution from forward substitution).
- `sig_figs`: Integer representing the number of significant figures for rounding.

2. *Output*:

- x : 1D list representing the solution vector for the equation $Ux = y$.

3. *Steps*:

1. Initialize x as a zero vector of size n .
2. Loop through each row i from $n-1$ to 0 (reverse order):
 1. Compute the value of $x[i]$ by subtracting the dot product of $U[i, i+1:]$ and $x[i+1:]$ from $y[i]$, then dividing by $U[i, i]$ and rounding off to `sig_figs`.
3. Return the solution vector x .

Helper Function: `RoundOff(value, sig_figs)`

1. *Input*:

- `value`: Float representing the number to be rounded.
- `sig_figs`: Integer representing the number of significant figures for rounding.

2. *Output*:

- `rounded_value`: Rounded value of `value` to the specified number of significant figures.

3. *Steps*:

1. Perform standard rounding to the specified number of significant figures and return the result.

Helper Function: `IsSingular(matrix)`

1. *Input*:

- matrix: 2D list or matrix.

2. *Output*:

- Boolean: True if the matrix is singular (determinant is 0), False otherwise.

3. *Steps*:

1. Compute the determinant of the matrix and return whether it is zero.

Test Cases

Case 1: Valid Input

Input:

```
```python
```

```
A = [[0, 2, 5], [2, 1, 1], [3, 1, 0]]
```

```
B = [1, 1, 2]
```

```
sig_figs = 5
```

## 2-Crout

## ### Pseudocode for LU Decomposition with Pivoting

### #### \*Main Function\*: solve\_using\_lu(n, A, B, sig\_figs)

#### 1. \*Input\*:

- n: Integer representing the size of matrix A.
- A: 2D list or matrix representing the coefficient matrix.
- B: 1D list representing the right-hand side vector.
- sig\_figs: Integer representing the number of significant figures for rounding.

#### 2. \*Output\*:

- x: 1D list representing the solution vector.
- execution\_time: Time taken for the entire computation.

**3. \*Steps\*:**

1. Start Timer.
2. Convert A and B to numpy arrays (of type float).
3. Call `lu_decomposition_with_pivoting(n, A, B, sig_figs)` to decompose matrix A into P, L, and U matrices.
4. Compute  $Pb = P * B$  (matrix multiplication of P and B).
5. Call `forward_substitution(L, Pb, sig_figs)` to solve the equation  $Ly = Pb$  for y.
6. Call `backward_substitution(U, y, sig_figs)` to solve the equation  $Ux = y$  for x.
7. End Timer and compute `execution_time`.
8. Return the solution vector x and `execution_time`.

---

**#### \*Helper Function\*: `lu_decomposition_with_pivoting(n, A, B, sig_figs)`****1. \*Input\*:**

- n: Integer representing the size of matrix A.
- A: 2D list representing the coefficient matrix.
- B: 1D list representing the right-hand side vector.
- sig\_figs: Integer representing the number of significant figures for rounding.

**2. \*Output\*:**

- P: 2D list representing the permutation matrix.
- L: 2D list representing the lower triangular matrix.
- U: 2D list representing the upper triangular matrix.

**3. \*Steps\*:**



1. Initialize matrix L as a zero matrix of size  $n \times n$ .
2. Initialize matrix U as a zero matrix of size  $n \times n$ .
3. Initialize matrix P as an identity matrix of size  $n \times n$ .
4. Loop through each row  $i$  from 0 to  $n-1$ :
  1. **\*Partial Pivoting\***: Find the row with the largest element in column  $i$  and swap it with the current row.
  2. Decompose the matrix into upper (U) and lower (L) matrices by:
    - For each element  $U[i][j]$ , subtract the dot product of  $L[i, :i]$  and  $U[:i, j]$  and round off.
    - For each element  $L[j, i]$ , subtract the dot product of  $L[j, :i]$  and  $U[:i, i]$ , divide by  $U[i, i]$ , and round off.
5. Return matrices P, L, and U.

---

#### #### **\*Helper Function\*: forward\_substitution(L, B, sig\_figs)**

##### **1. \*Input\*:**

- L: 2D list representing the lower triangular matrix.
- B: 1D list representing the right-hand side vector.
- sig\_figs: Integer representing the number of significant figures for rounding.

##### **2. \*Output\*:**

- y: 1D list representing the solution vector for the equation  $Ly = B$ .

##### **3. \*Steps\*:**

1. Initialize y as a zero vector of size  $n$ .
2. Loop through each row  $i$  from 0 to  $n-1$ :
  1. Compute the value of  $y[i]$  by subtracting the dot product of  $L[i, :i]$  and  $y[:i]$  from  $B[i]$ , then round off to sig\_figs.

3. Return the solution vector  $y$ .

---

**#### \*Helper Function\*: backward\_substitution(U, y, sig\_figs)**

**1. \*Input\*:**

- U: 2D list representing the upper triangular matrix.
- y: 1D list representing the right-hand side vector (solution from forward substitution).
- **sig\_figs: Integer representing the number of significant figures for rounding.**

**2. \*Output\*:**

- x: 1D list representing the solution vector for the equation  $Ux = y$ .

**3. \*Steps\*:**

1. Initialize x as a zero vector of size n.
2. Loop through each row i from n-1 to 0 (reverse order):
  1. Compute the value of  $x[i]$  by subtracting the dot product of  $U[i, i+1:]$  and  $x[i+1:]$  from  $y[i]$ , then dividing by  $U[i, i]$  and rounding off to sig\_figs.
3. Return the solution vector x.

---

**#### \*Helper Function\*: RoundOff(value, sig\_figs)**

**1. \*Input\*:**

- value: Float representing the number to be rounded.
- **sig\_figs: Integer representing the number of significant figures for rounding.**

**2. \*Output\*:**

- rounded\_value: Rounded value of value to the specified number of significant figures.

**3. \*Steps\*:**

1. Perform standard rounding to the specified number of significant figures and return the result.

---

**#### \*Helper Function\*: IsSingular(matrix)****1. \*Input\*:**

- matrix: 2D list or matrix.

**2. \*Output\*:**

- Boolean: True if the matrix is singular (determinant is 0), False otherwise.

**3. \*Steps\*:**

1. Compute the determinant of the matrix and return whether it is zero.

---

**### \*Test Cases\*****#### Case 1: Valid Input****\*Input\*:**

```
```python
```

```
A = [[0, 2, 5], [2, 1, 1], [3, 1, 0]]
```

B = [1, 1, 2]

sig_figs = 5

[23:11, 01/12/2024] John William: ### *Pseudocode*

Main Function: croutDecomposition(size, A, B, sig_figs)

1. *Input*:

- size: Integer, size of the matrix A.
- A: 2D array, matrix to be decomposed.
- B: 1D array, right-hand side vector.
- sig_figs: Integer, significant figures for rounding.

2. *Output*:

- X: Solution vector.
- execution_time: Time taken for the decomposition and solving.

3. *Steps*:

1. Start Timer.
2. Convert A and B to NumPy arrays.
3. Call createLUCrout to decompose A into L and U.
4. Solve $LY = B$ using forward substitution.
5. Solve $UX = Y$ using backward substitution.
6. End Timer and compute execution_time.
7. Return X and execution_time.

Helper Function: createLUCrout(size, A, sig_figs)

1. *Input*:

- size: Integer, size of the matrix A.
- A: 2D array, matrix to be decomposed.
- sig_figs: Integer, significant figures for rounding.

2. *Output*:

- L: Lower triangular matrix.
- U: Upper triangular matrix.

3. *Steps*:

1. Initialize L as a zero matrix and U as an identity matrix.
2. For each column j:
 - Check if $A[j, j]$ is close to zero and perform row swap if necessary.
 - Compute elements of L (below and on diagonal).
 - Compute elements of U (above diagonal).
3. Return L and U.

Helper Function: RoundOff(value, sig_figs)

- Round the value to the specified number of significant figures.

*Time Complexity*

- *LU Decomposition* (createLUCrout):
 - Outer loop for columns: $\backslash (O(n) \backslash)$

- Inner loops for computing L and U: $\mathcal{O}(n)$ for each, leading to $\mathcal{O}(n^3)$ total.
- ***Forward Substitution***: $\mathcal{O}(n^2)$.
- ***Backward Substitution***: $\mathcal{O}(n^2)$.
- ***Overall Time Complexity***: $\mathcal{O}(n^3)$.

Space Complexity

- ***Matrix Storage*** (A, L, U): $\mathcal{O}(n^2)$.
- ***Vector Storage*** (B, X, Y): $\mathcal{O}(n)$.
- ***Overall Space Complexity***: $\mathcal{O}(n^2)$.

3-Choleskey:

Pseudocode

Main Function: CholeskyDecomposition(size, A, B, sig_figs)

1. *Input*:

- size: Integer, size of the matrix A.
- A: 2D array, a symmetric positive definite matrix.
- B: 1D array, the right-hand side vector.
- sig_figs: Integer, number of significant figures for rounding.

2. *Output*:

- X: Solution vector.
- execution_time: Time taken for the decomposition and solving.

3. *Steps*:

1. *Start Timer*.

2. Convert A and B to NumPy arrays.
3. Initialize L (lower triangular matrix) as a zero matrix of the same size as A.
4. Loop over rows i:
 1. Loop over columns j (up to i):
 - *If i == j (Diagonal Elements)*:
 1. Compute $\text{sum_diagonal} = \text{Sum of squares of } L[i][k] \text{ for } k \text{ from } 0 \text{ to } j-1.$
 2. Compute $L[i][j] = \text{RoundOff}(\sqrt{A[i][i] - \text{sum_diagonal}}, \text{sig_figs}).$
 - *Else (Off-Diagonal Elements)*:
 1. Compute $\text{sum_off_diagonal} = \text{Sum of products } L[i][k] * L[j][k] \text{ for } k \text{ from } 0 \text{ to } j-1.$
 2. Compute $L[i][j] = \text{RoundOff}((A[i][j] - \text{sum_off_diagonal}) / L[j][j], \text{sig_figs}).$
5. Compute LT (transpose of L).
6. Solve $LY = B$ using forward substitution: $Y = \text{ForwardSubstitution}(\text{size}, L, B, \text{sig_figs}).$
7. Solve $LTX = Y$ using backward substitution: $X = \text{BackwardSubstitution}(\text{size}, LT, Y, \text{sig_figs}).$
8. *End Timer* and calculate `execution_time`.
9. Return X and `execution_time`.

Helper Function: RoundOff(value, sig_figs)

- *Input*: value (float), sig_figs (int).
- *Output*: Rounded value to sig_figs.
- Use standard rounding logic.

*Helper Function*: ForwardSubstitution(size, L, B, sig_figs)

- Solve lower triangular system $LY = B$.
- *Input*: size, L (lower triangular matrix), B (vector), sig_figs.
- ***Output*: Solution vector Y.**
- Use iterative substitution formula.

*Helper Function*: BackwardSubstitution(size, LT, Y, sig_figs)

- Solve upper triangular system $LTX = Y$.
- *Input*: size, LT (upper triangular matrix), Y (vector), sig_figs.
- *Output*: Solution vector X.
- Use iterative substitution formula.

*Helper Function*: IsSymmetric(matrix)

- *Input*: Matrix.
- *Output*: Boolean indicating if the matrix is symmetric.
- Compare matrix with its transpose.

Test Cases

Case 1: Valid Input

- *Input*:

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

$B = [1, 2, 3]$

sig_figs = 4

- *Expected Output*:

X = Solution vector

execution_time = Time taken

Case 2: Non-Symmetric Matrix

- *Input*:

$A = \begin{bmatrix} 4 & 12 & -16 \\ 12 & 37 & -43 \\ 10 & -43 & 98 \end{bmatrix}$ (not symmetric)

- *Expected Output*: Error message ("Matrix must be symmetric").

Case 3: Positive Definite Violation

- *Input*:

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

- *Expected Output*: Error during square root computation.

Case 4: Singular Matrix

- *Input*:

$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$

- *Expected Output*: Error due to division by zero.

Case 5: Small Matrix

- *Input*:

$A = \begin{bmatrix} 4 \end{bmatrix}$

$B = \begin{bmatrix} 2 \end{bmatrix}$

$\text{sig_figs} = 3$

- *Expected Output*: $X = \begin{bmatrix} 0.5 \end{bmatrix}$

Time Complexity

- *Matrix Decomposition*:
 - Outer loop over i: $\mathcal{O}(n)$.
 - Inner loop over j: $\mathcal{O}(n)$.
 - Computation inside loop: $\mathcal{O}(n)$ for summation.
 - *Total*: $\mathcal{O}(n^3)$.
- *Forward and Backward Substitution*:
 - Each requires $\mathcal{O}(n^2)$.
- *Overall*: $\mathcal{O}(n^3)$.

Space Complexity

- Storage for matrices A, L, B, LT: $\mathcal{O}(n^2)$.
- Storage for solution vectors X, Y: $\mathcal{O}(n)$.
- *Overall*: $\mathcal{O}(n^2)$.

III. Jcobi:

Pseudocode

Function: JacobiMethod(A, B, initial_guess, max_iterations, tolerance, sig_figs)

1. *Input*:

- A: 2D array (coefficient matrix).
- B: 1D array (right-hand side vector).
- initial_guess: 1D array (initial guess for the solution).
- max_iterations: Maximum number of iterations allowed (default: 100).
- tolerance: Acceptable error for convergence (default: $\backslash(1 \backslash \text{times } 10^{\{-8\}}\backslash)$).
- sig_figs: Number of significant figures for rounding (default: 8).

2. *Output*:

- x_new: Approximate solution vector.
- num_iterations: Number of iterations taken to converge.
- execution_time: Time taken for computation.

3. *Steps*:

1. *Start Timer*.

2. Initialize:

- n = len(A) (number of equations).
- x = initial guess (convert to NumPy array).
- x_new = x.copy() (array for the updated solution).

3. Convert A and B to NumPy arrays for efficient computation.

4. Loop over k from 0 to max_iterations:

1. Loop over each row i in matrix A:

- Compute partial sums:
 - sum1 = DotProduct(A[i, :i], x[:i]).
 - sum2 = DotProduct(A[i, i+1:], x[i+1:]).
- Update solution:
 - x_new[i] = (B[i] - sum1 - sum2) / A[i, i].
 - Apply rounding: x_new[i] = RoundOff(x_new[i], sig_figs).

2. Compute the error:

- error = MaxNorm(x_new - x).
- 3. If error < tolerance:
 - Stop and return x_new, k+1, and execution_time.
- 4. Update x[:] = x_new.
- 5. If maximum iterations are reached:
 - Return x, execution_time, and max_iterations.

Helper Functions

RoundOff(value, sig_figs)

- Rounds a floating-point number to the specified number of significant figures.

DotProduct(v1, v2)

- Computes the dot product of two vectors.

MaxNorm(vector)

- Computes the maximum absolute value in a vector.

Test Cases

Case 1: Simple Convergent System

- *Input*:

python

A = [[4, 1], [2, 3]]

B = [1, 2]

initial_guess = [0, 0]

sig_figs = 5

- *Expected Output*:

- Approximate solution x.
- Number of iterations.
- Execution time.

Case 2: Singular Matrix (No Solution)

- *Input*:

python

A = [[1, 2], [2, 4]] # Singular matrix

B = [3, 6]

initial_guess = [0, 0]

sig_figs = 5

- *Expected Output*:

- Maximum iterations reached.

- No convergence.

Case 3: Ill-Conditioned Matrix

- *Input*:

python

A = [[1, 1], [1.001, 1]]

B = [2, 2.001]

initial_guess = [0, 0]

sig_figs = 5

- *Expected Output*:

- Slow convergence or failure to converge.

Case 4: Larger System

- *Input*:

python

A = [[10, -1, 2, 0], [-1, 11, -1, 3], [2, -1, 10, -1], [0, 3, -1, 8]]

B = [6, 25, -11, 15]

initial_guess = [0, 0, 0, 0]

sig_figs = 8

```
max_iterations = 500
tolerance = 1e-10
```

- *Expected Output*:
- Accurate solution x .

```
---
```

Complexity Analysis

Time Complexity

- *Outer Loop* (iterations): Runs up to max_iterations in the worst case.
- *Inner Loop* (updating variables): Iterates over n rows of A .
- *Dot Product*: $\mathcal{O}(n)$ per row.
- *Total Time Complexity*: $\mathcal{O}(\text{max_iterations} \cdot n^2)$.

Space Complexity

- *Storage for matrices A and B *: $\mathcal{O}(n^2 + n)$.
- *Storage for solution vectors x and x_{new} *: $\mathcal{O}(n)$.
- *Total Space Complexity*: $\mathcal{O}(n^2)$

IV. Gauss Seidel:

*Pseudocode*

*Function*: GaussSeidel(A, B, initial_guess, sig_figs, max_iterations, tolerance)

1. *Input*:

- A: 2D array (coefficient matrix).
- B: 1D array (right-hand side vector).
- initial_guess: 1D array (initial solution guess).
- sig_figs: Integer, number of significant figures for rounding.
- max_iterations: Integer, maximum number of iterations allowed (default: 100).
- tolerance: Float, acceptable error for convergence (default: (1×10^{-8})).

2. *Output*:

- x: Approximate solution vector.
- num_iterations: Number of iterations taken to converge.
- execution_time: Time taken for the computation.

3. *Steps*:

1. *Start Timer*.
2. Initialize:

- $n = \text{len}(A)$ (number of equations).
 - $x = \text{initial_guess}$ (convert to NumPy array).
3. Convert A and B to NumPy arrays for efficient computations.
 4. Loop over k from 0 to max_iterations:
 1. Save the current solution: $x_{\text{old}} = x.\text{copy}()$.
 2. Loop over rows i in A:
 - Compute partial sums:
 - $\text{sum1} = \text{DotProduct}(A[i, :i], x[:i])$.
 - $\text{sum2} = \text{DotProduct}(A[i, i+1:], x_{\text{old}}[i+1:])$.
 - Update the solution:
 - $x[i] = (B[i] - \text{sum1} - \text{sum2}) / A[i, i]$.
 - Apply rounding: $x[i] = \text{RoundOff}(x[i], \text{sig_figs})$.
 3. Compute the error:
 - $\text{error} = \text{MaxNorm}(x - x_{\text{old}})$.
 4. If error < tolerance:
 - Stop and return x, k+1, and execution_time.
 5. If maximum iterations are reached, return x, max_iterations, and execution_time.

*Helper Functions*

*RoundOff(value, sig_figs)*

- Rounds a floating-point number to the specified number of significant figures.

DotProduct(v1, v2)

- Computes the dot product of two vectors.

MaxNorm(vector)

- Computes the maximum absolute value in a vector.

*Test Cases*

*Case 1: Valid Input*

- ***Input*:**

python

A = [[4, 1], [2, 3]]

B = [1, 2]

initial_guess = [0, 0]

sig_figs = 5

- ***Expected Output*:**

- Approximate solution x.

- Iterations taken (depends on convergence).
- Execution time.

Case 2: Non-Convergent System

- *Input*:

python

A = [[1, 2], [2, 4]] # Singular matrix

B = [3, 6]

initial_guess = [0, 0]

sig_figs = 5

- *Expected Output*:

- No convergence within max_iterations.

Case 3: High Precision Requirement

- *Input*:

python

A = [[4, -1, 0], [-1, 4, -1], [0, -1, 4]]

```
B = [15, 10, 10]
initial_guess = [0, 0, 0]
sig_figs = 10
tolerance = 1e-12
```

- ***Expected Output*:**

- Solution vector with high precision.

Case 4: Large System

- ***Input*:**

```
python
A = RandomPositiveDefiniteMatrix(100)
B = RandomVector(100)
initial_guess = [0]*100
sig_figs = 5
max_iterations = 1000
```

- ***Expected Output*:**

- Solution vector (may take many iterations).

Complexity Analysis

Time Complexity

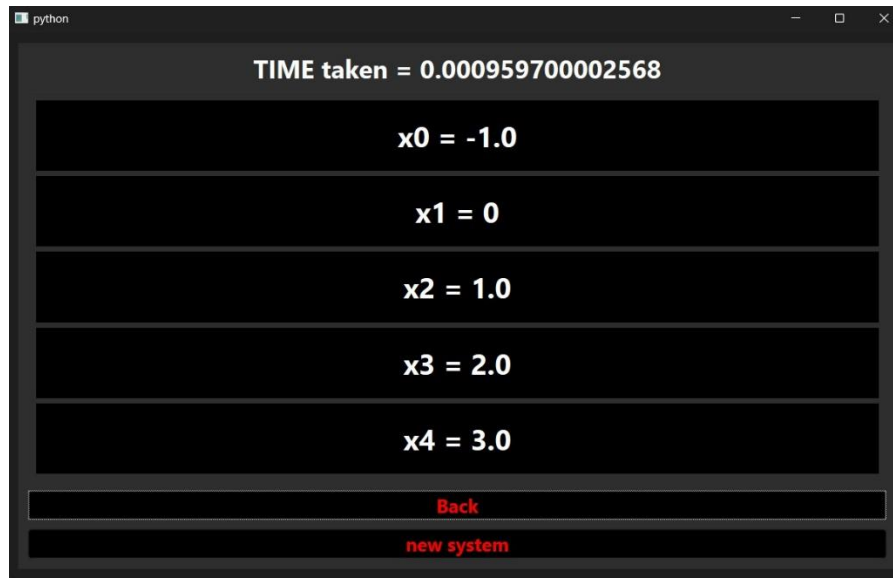
- *Outer Loop* (iterations): Runs for max_iterations in the worst case.
- *Inner Loop* (updating variables): Iterates over n rows for each equation.
- *Dot Product*: $O(n)$ per row.
- *Total Time Complexity*: $O(\text{max_iterations} \cdot n^2)$.

Space Complexity

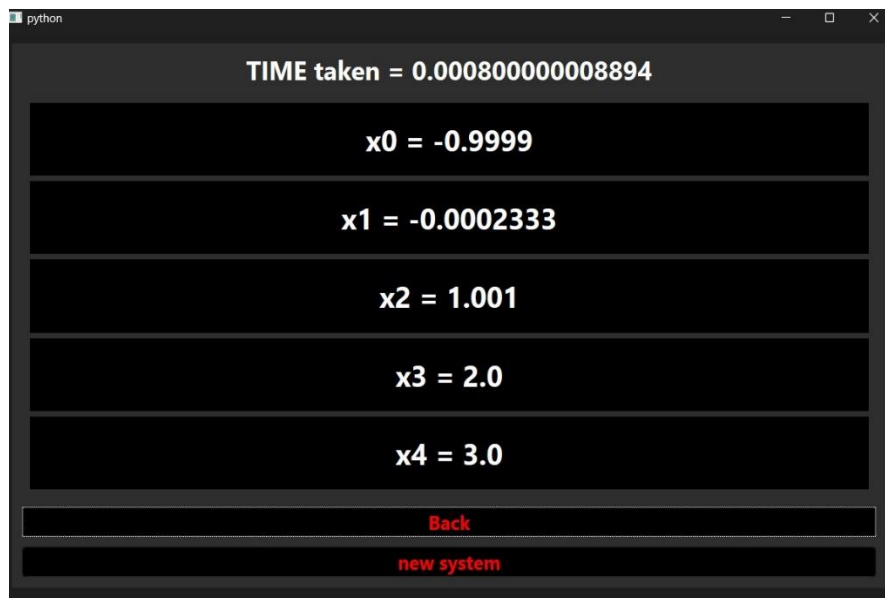
- *Storage for matrices A and B*: $O(n^2 + n)$.
 - *Storage for solution vector x *: $O(n)$.
 - *Total Space Complexity*: $O(n^2)$.
-

2.Test Cases:

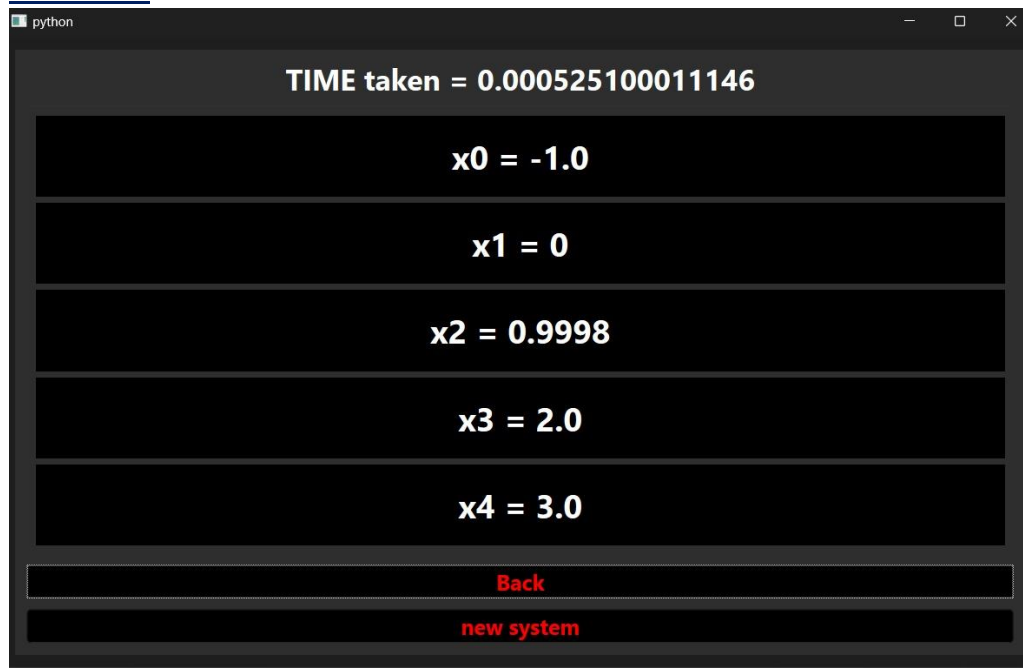
Gauss elimination



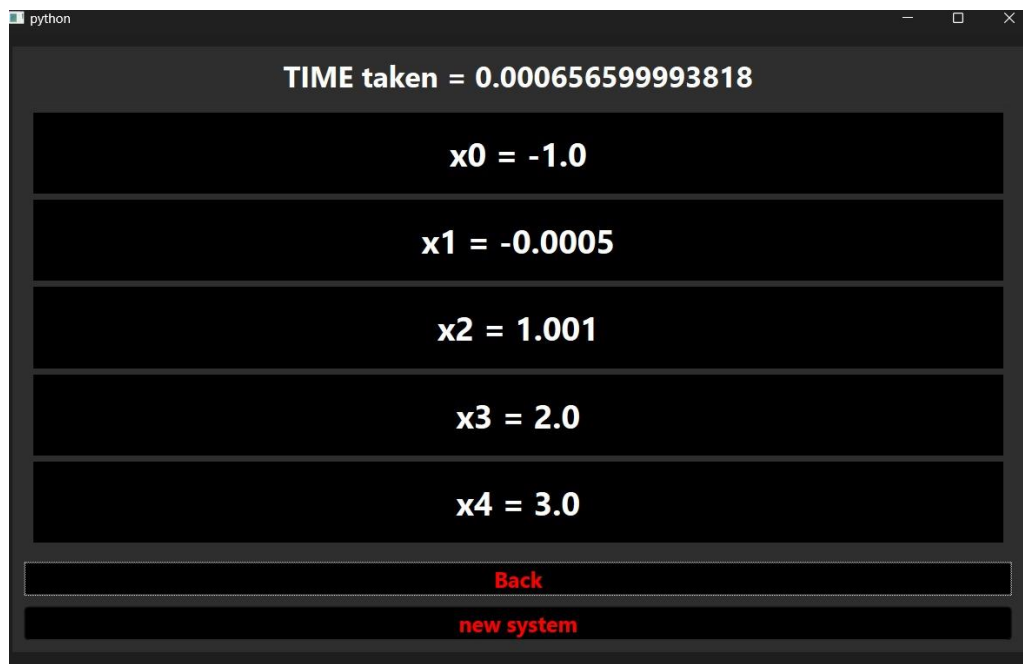
Gauss Jordan



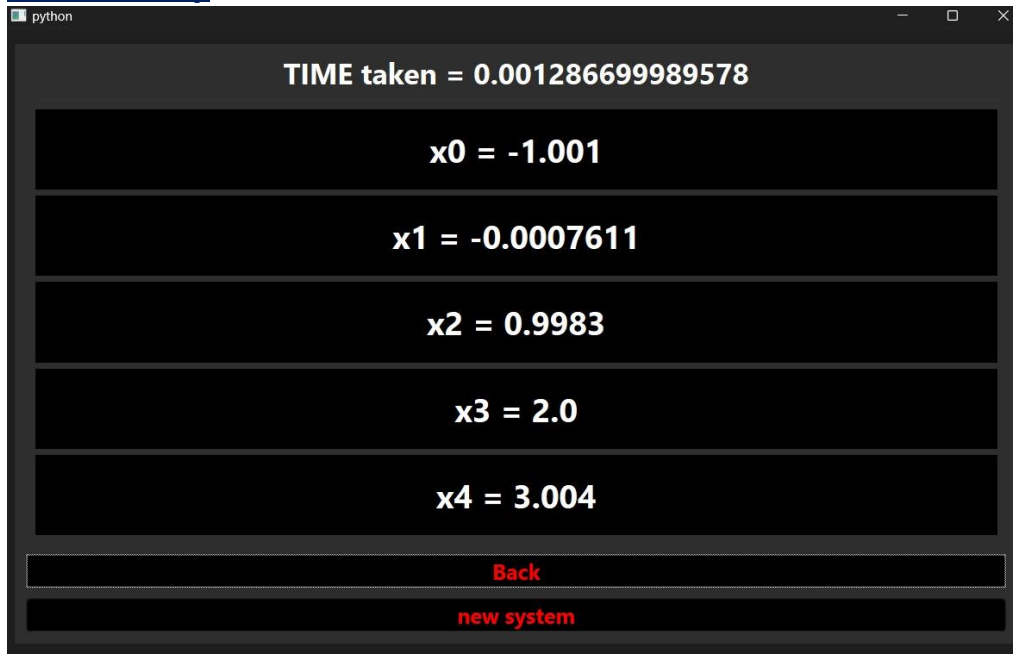
Doolittle



Crout

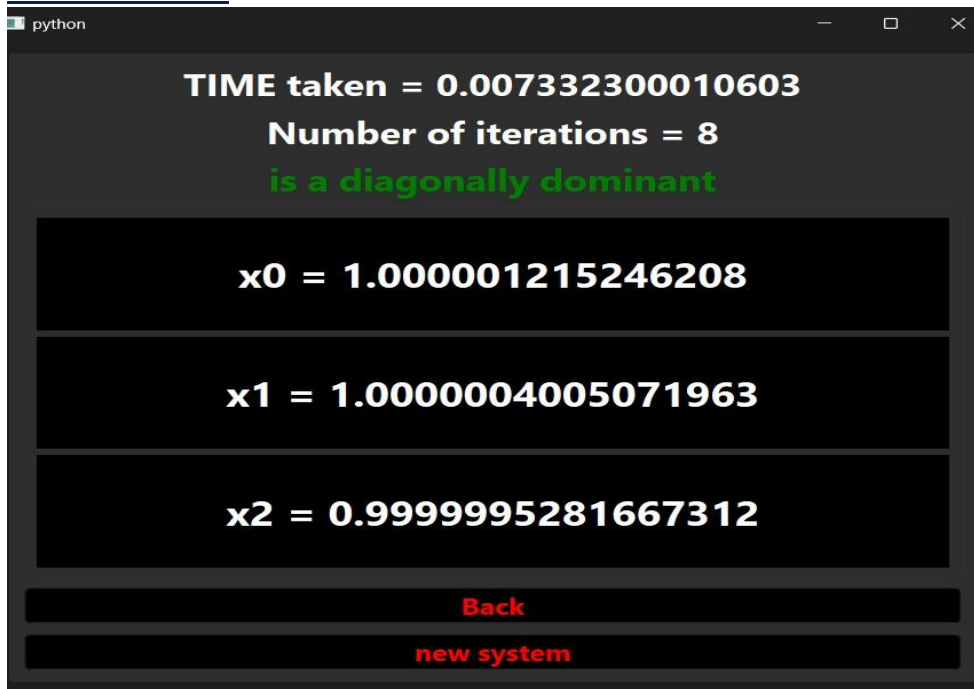


Chooleskey

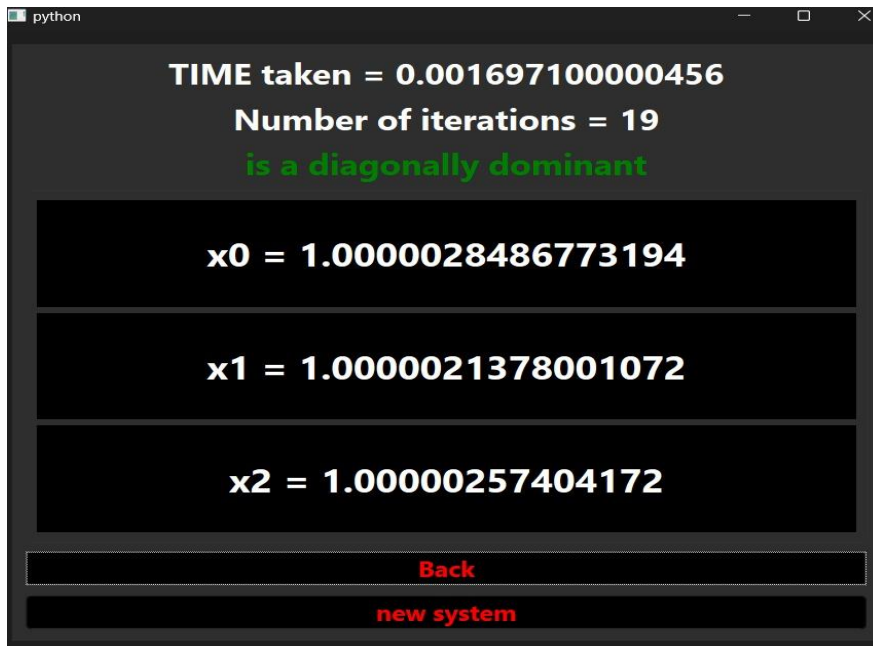


Test 2:

Gauss Seidel



Jacobi:



A Python window titled 'python' with a dark background. It displays the following text:

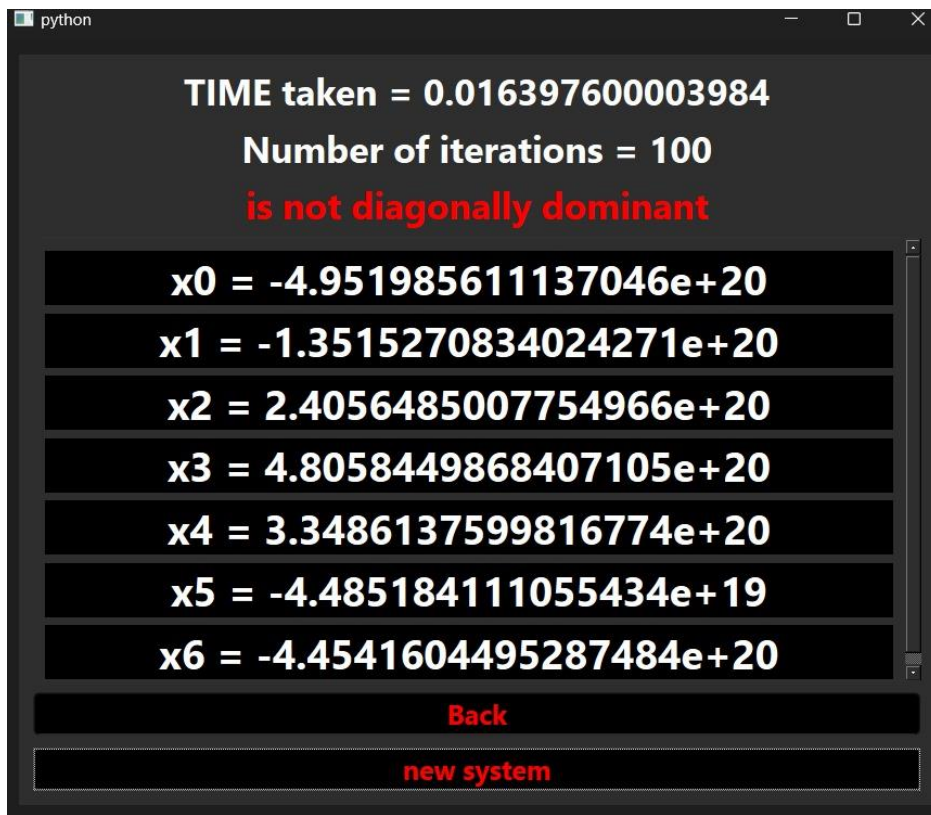
```
TIME taken = 0.001697100000456
Number of iterations = 19
is a diagonally dominant
```

Below this, three values are shown in separate boxes:

```
x0 = 1.0000028486773194
x1 = 1.0000021378001072
x2 = 1.00000257404172
```

At the bottom, there are two buttons: 'Back' and 'new system'.

Test 3:



A Python window titled 'python' with a dark background. It displays the following text:

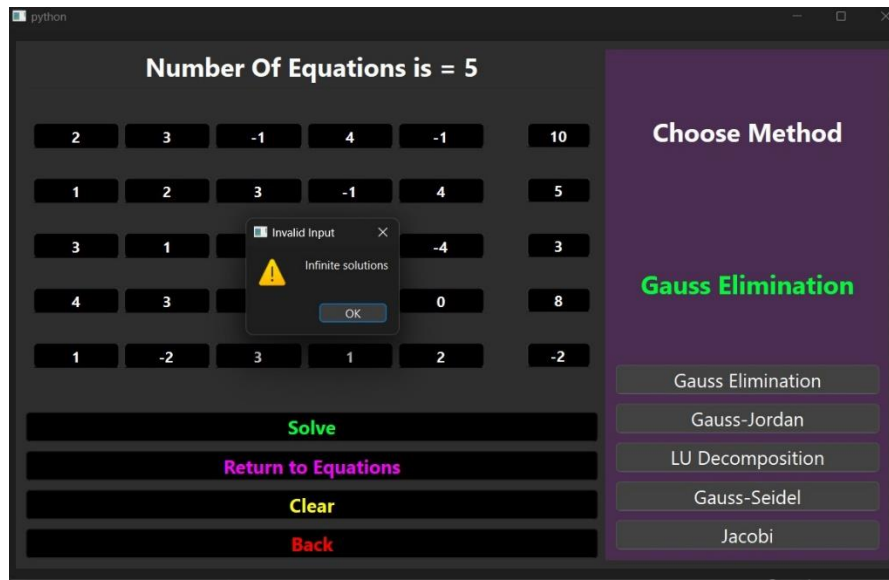
```
TIME taken = 0.016397600003984
Number of iterations = 100
is not diagonally dominant
```

Below this, seven values are shown in separate boxes:

```
x0 = -4.951985611137046e+20
x1 = -1.3515270834024271e+20
x2 = 2.4056485007754966e+20
x3 = 4.8058449868407105e+20
x4 = 3.3486137599816774e+20
x5 = -4.485184111055434e+19
x6 = -4.4541604495287484e+20
```

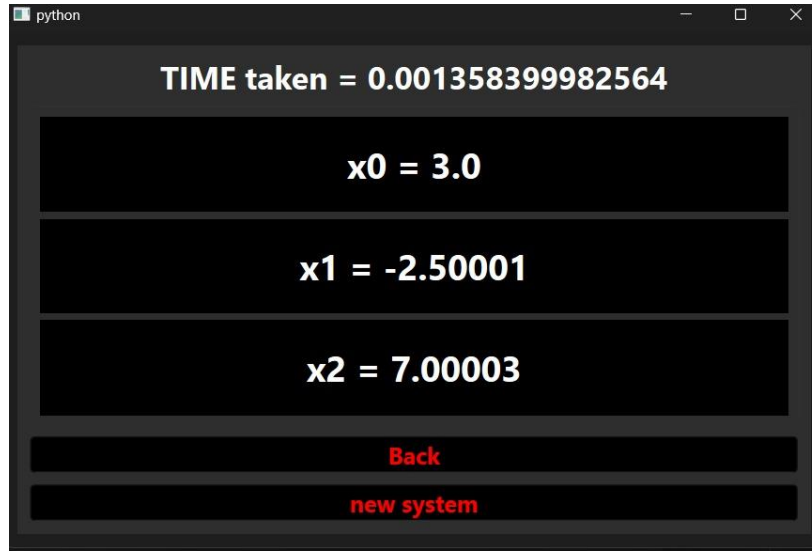
At the bottom, there are two buttons: 'Back' and 'new system'.

Test 4:

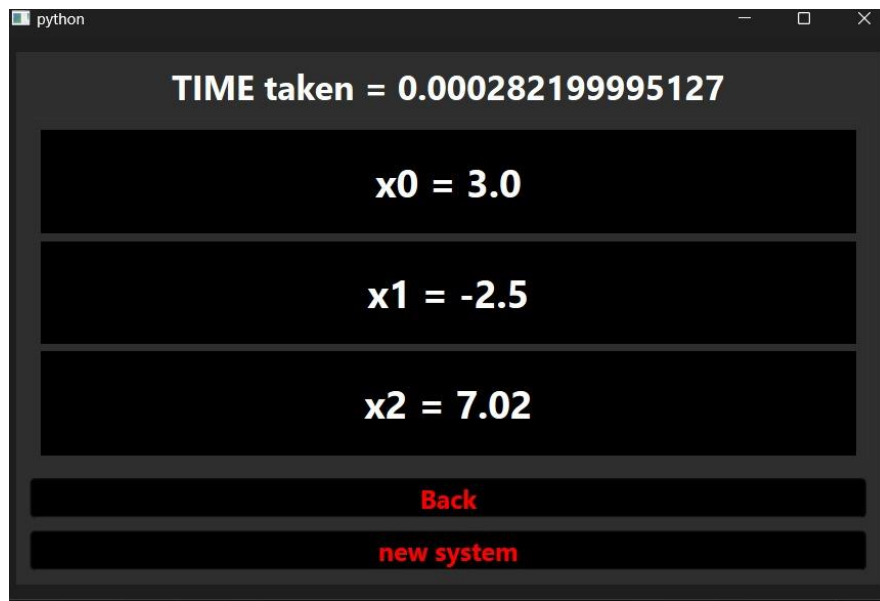


Test 5:

Sig 6

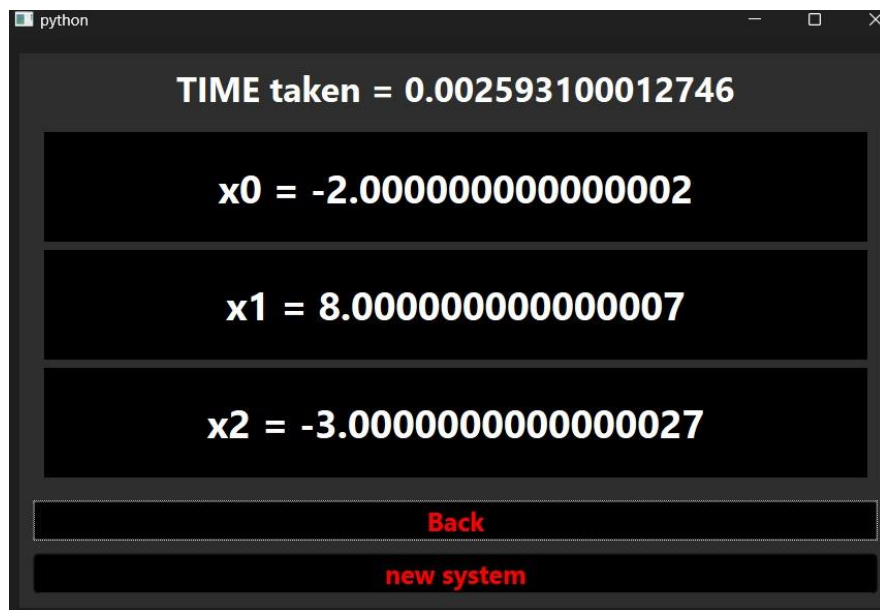


Sig 3

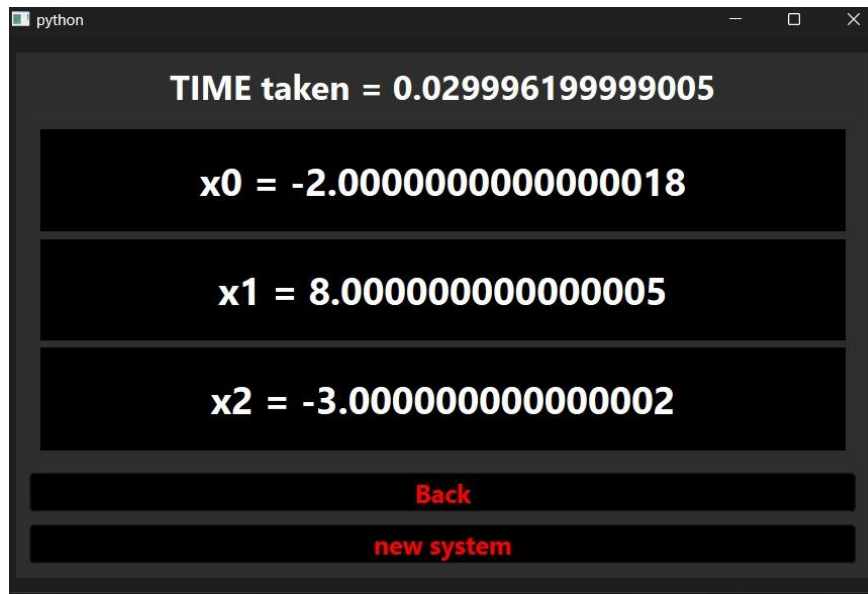


Test 6:

choleskey

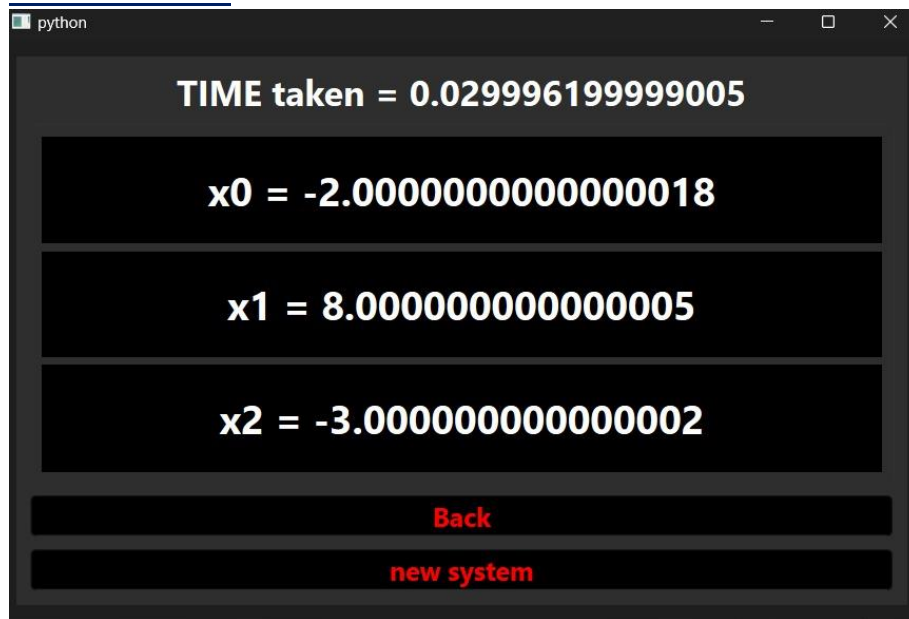


Gauss Jordan

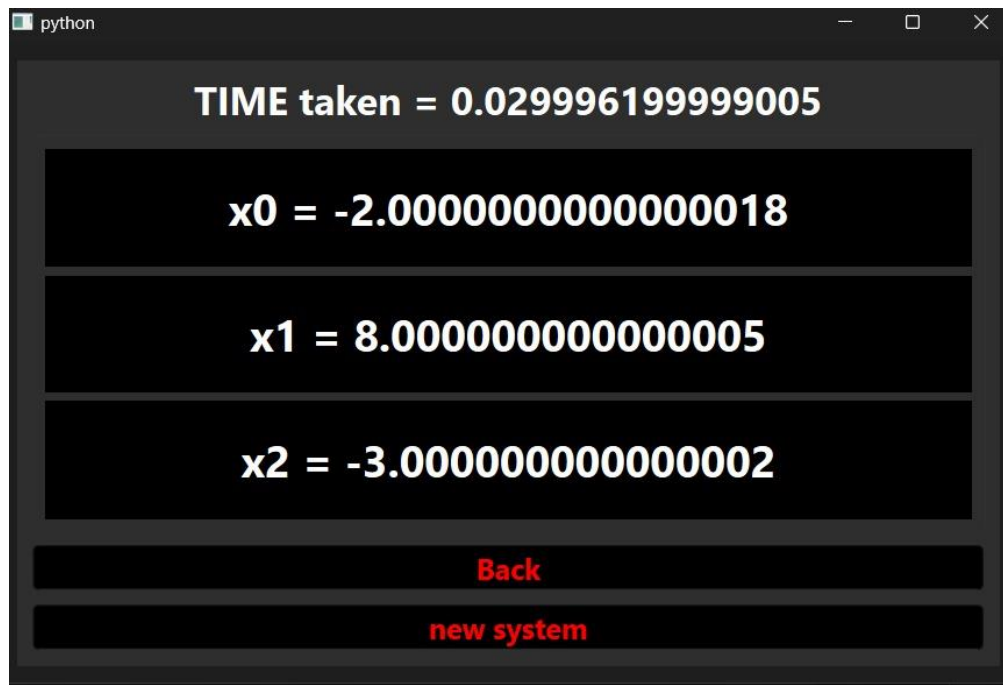


Test 7:

Gauss Seidel



Jacobi



The Gui:

In our GUI we have created 5 pages using scaffold widget :

These 5 pages navigate through them across the (Main) Page which we run from it it contains the navigators functions for the scaffold:

`go_to_first_page()`

`go_to_second_page()`

`go_to_third_page()`

`go_to_fourt_page()`

`go_to_fifth_page()` and a special `go_to_fifth_page_from_fourth_page()`

the explanation of these pages roles are:

Page 1:

The MENU page:

structure:

two buttons wrapped by a static widget either the user chooses to start the program or exits it .

Page 2:

The Input size page

structure:

A] 2 input fields one for the number of equations and the other for the significant figures

B] 2 buttons one to submit the choices and navigate to the third page and the other for back step to the previous page (Menu)

Page 3:

The Matrix Page:

Structure: according to $AX=B$

A] the Matrix (A) [THE GRID WIDGET]

which its number of rows and columns generated according to the the number of equations entered where $n*(n+1)$ with one more column generated for the solutions of equations (B) wrapped by a scrollable widget where to add the space for upto a limit $n=50$ equations

B] 4 buttons

1-[Solve] on clicking on it at first it doesn't navigate and gives an alert that the user should choose the method of solving assuming you have entered the coefficients inputs maintain any validation programming rules ,**By then:**

→We handled a control function page that contains switch of choosing the method of solving and if any of the first three methods chosen

→go_to_fifth_page() which renders the solution after sending to the 7 functions of solving (OR)

→go_to_fourth_page() if the chosen methods are either (Jacobi or Gauss Seidel)

2-[Choose Solving Method] on clicking a toggling sidebar widget that consists of 5 buttons for each method of solution appears

,**By then:**

1-Gauss Elimination button doesn't get us anywhere as there are no conditions for it

2-Gauss Jordan : same approach [No Conditions]

3-LU Decomposition : on clicking a small menu appears down below the buttons for choosing which submethod of LU to be chosen

4- Gauss Seidel : just choosing this method will sent us to the

→go_to_fourth_page() on clicking on the solve bottom afterwards [it's a condition in gui on choosing this method and the Jacobi]

5-Jacobi:same approach as gauss seidel

Page 4:

THE CONDITIONS FOR ITERATING METHODS :

We sent to it as parameters the number of equations saved from second page to show the scaffold of input boxes ready with the number of inputs for the initial guess

A]THE INITIAL GUESS:

B]THE STOPPING CONDITIONS

1- Relative Appoximate Error

2- Number of Iterations

→ On clicking on submit go_to_fifth_page()

Page 5:

The Final answer page:

The Display of the Solution:

We send to it according to the control condition for the 7 functions ->if sent from fourth page we send the conditions for the iterating method input in fourth page in addition to the basic conditions

->else we just send the basic conditions

1-the Matrix ($n*n$)

2-the solution vector (B)

3-(n) the size of the matrix

4-the number of significant figures chosen