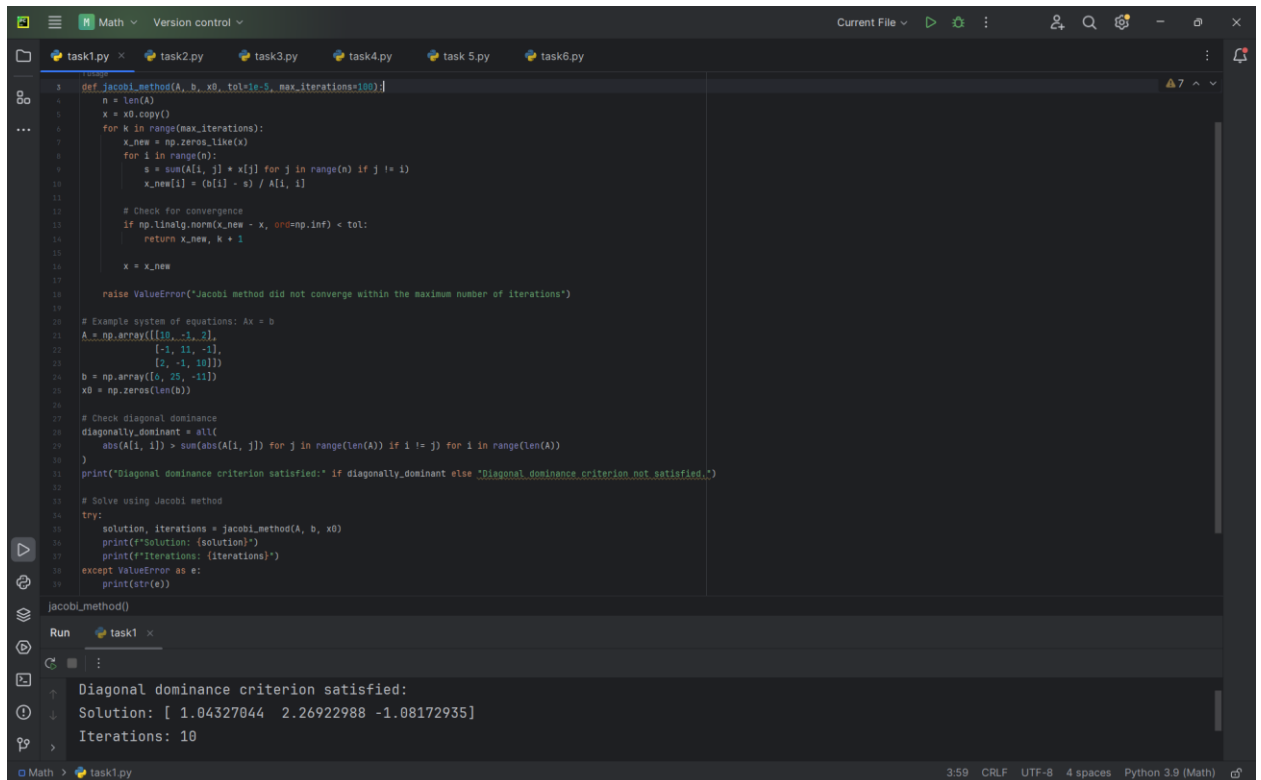


Task 1: Jacobi Method and Convergence Analysis



```
1 def jacobi_method(A, b, x0, tol=1e-5, max_iterations=100):
2     n = len(A)
3     x = x0.copy()
4     for k in range(max_iterations):
5         x_new = np.zeros_like(x)
6         for i in range(n):
7             s = sum(A[i, j] * x[j] for j in range(n) if j != i)
8             x_new[i] = (b[i] - s) / A[i, i]
9
10        # Check for convergence
11        if np.linalg.norm(x_new - x, ord=np.inf) < tol:
12            return x_new, k + 1
13
14        x = x_new
15
16    raise ValueError("Jacobi method did not converge within the maximum number of iterations")
17
18 # Example system of equations: Ax = b
19 A = np.array([[10, -1, 2],
20              [-1, 11, -1],
21              [2, -1, 10]])
22 b = np.array([0, 25, -11])
23 x0 = np.zeros(len(b))
24
25 # Check diagonal dominance
26 diagonally_dominant = all(
27     abs(A[i, i]) > sum(abs(A[i, j]) for j in range(len(A)) if i != j) for i in range(len(A))
28 )
29 print("Diagonal dominance criterion satisfied:" if diagonally_dominant else "Diagonal dominance criterion not satisfied.")
30
31 # Solve using Jacobi method
32 try:
33     solution, iterations = jacobi_method(A, b, x0)
34     print(f"Solution: {solution}")
35     print(f"Iterations: {iterations}")
36 except ValueError as e:
37     print(str(e))
38
39 jacobi_method()
```

Run task1

```
Diagonal dominance criterion satisfied:
Solution: [ 1.04327844  2.26922988 -1.08172935]
Iterations: 10
```

Description:

Objective: Solve a linear system using the Jacobi method and analyze convergence.

Implementation:

The matrix A and vector b define the system $Ax=b$.

The Jacobi method iteratively updates each variable using only the values from the previous iteration.

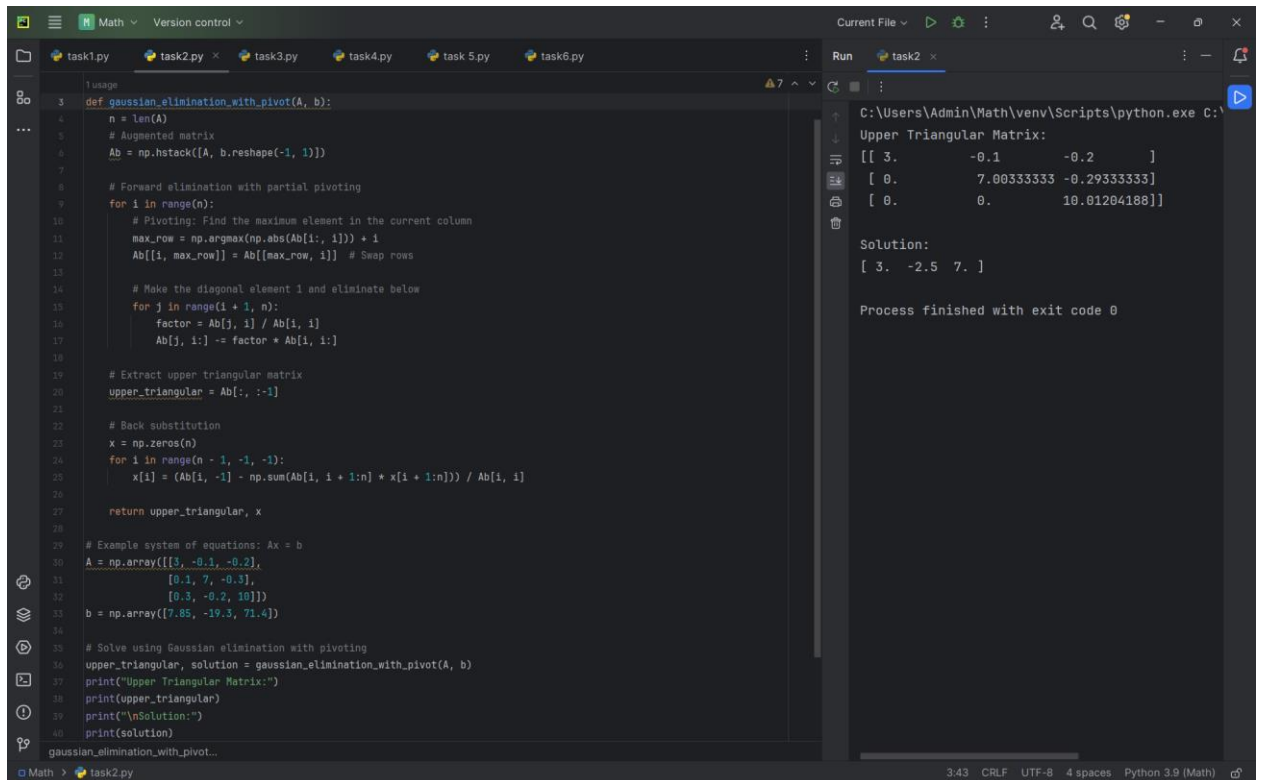
The diagonal dominance of the matrix is checked using the criterion $|a_{ii}| \geq \sum_{j \neq i} |a_{ij}|$, which is critical for ensuring convergence.

Convergence is determined by comparing the change in the solution vector across iterations against a defined tolerance 10^{-5} .

Explaining Convergence:

Convergence depends on the matrix structure. Diagonal dominance or a spectral radius of the iteration matrix less than 1 ensures that errors diminish over iterations

Task 2: Gaussian Method with Leading Element Selection



```
1 usage
2
3 def gaussian_elimination_with_pivot(A, b):
4     n = len(A)
5     # Augmented matrix
6     Ab = np.hstack([A, b.reshape(-1, 1)])
7
8     # Forward elimination with partial pivoting
9     for i in range(n):
10         # Pivoting: Find the maximum element in the current column
11         max_row = np.argmax(np.abs(Ab[i:, i])) + i
12         Ab[[i, max_row]] = Ab[[max_row, i]] # Swap rows
13
14         # Make the diagonal element 1 and eliminate below
15         for j in range(i + 1, n):
16             factor = Ab[j, i] / Ab[i, i]
17             Ab[j, i:] -= factor * Ab[i, i:]
18
19     # Extract upper triangular matrix
20     upper_triangular = Ab[:, :-1]
21
22     # Back substitution
23     x = np.zeros(n)
24     for i in range(n - 1, -1, -1):
25         x[i] = (Ab[i, -1] - np.sum(Ab[i, i + 1:n] * x[i + 1:n])) / Ab[i, i]
26
27     return upper_triangular, x
28
29 # Example system of equations: Ax = b
30 A = np.array([[3, -0.1, -0.2],
31               [0.1, 7, -0.3],
32               [0.3, -0.2, 10]])
33 b = np.array([7.85, -19.3, 71.4])
34
35 # Solve using Gaussian elimination with pivoting
36 upper_triangular, solution = gaussian_elimination_with_pivot(A, b)
37 print("Upper Triangular Matrix:")
38 print(upper_triangular)
39 print("\nSolution:")
40 print(solution)
41
42 gaussian_elimination_with_pivot...
```

Current File: task2.py

Upper Triangular Matrix:

```
[[ 3.         -0.1        -0.2         ]
 [ 0.         7.00333333 -0.29333333]
 [ 0.          0.         10.01204188]]
```

Solution:

```
[ 3.  -2.5  7. ]
```

Process finished with exit code 0

Description:

Objective: Solve a linear system using Gaussian elimination with partial pivoting to improve numerical stability.

Implementation:

Pivot selection identifies the row with the largest absolute value in the current column, minimizing numerical errors during division.

Forward elimination transforms the matrix into an upper triangular form by eliminating variables step-by-step.

Back substitution starts from the last variable and moves upward, solving for each variable sequentially.

Explaining Pivot Importance:

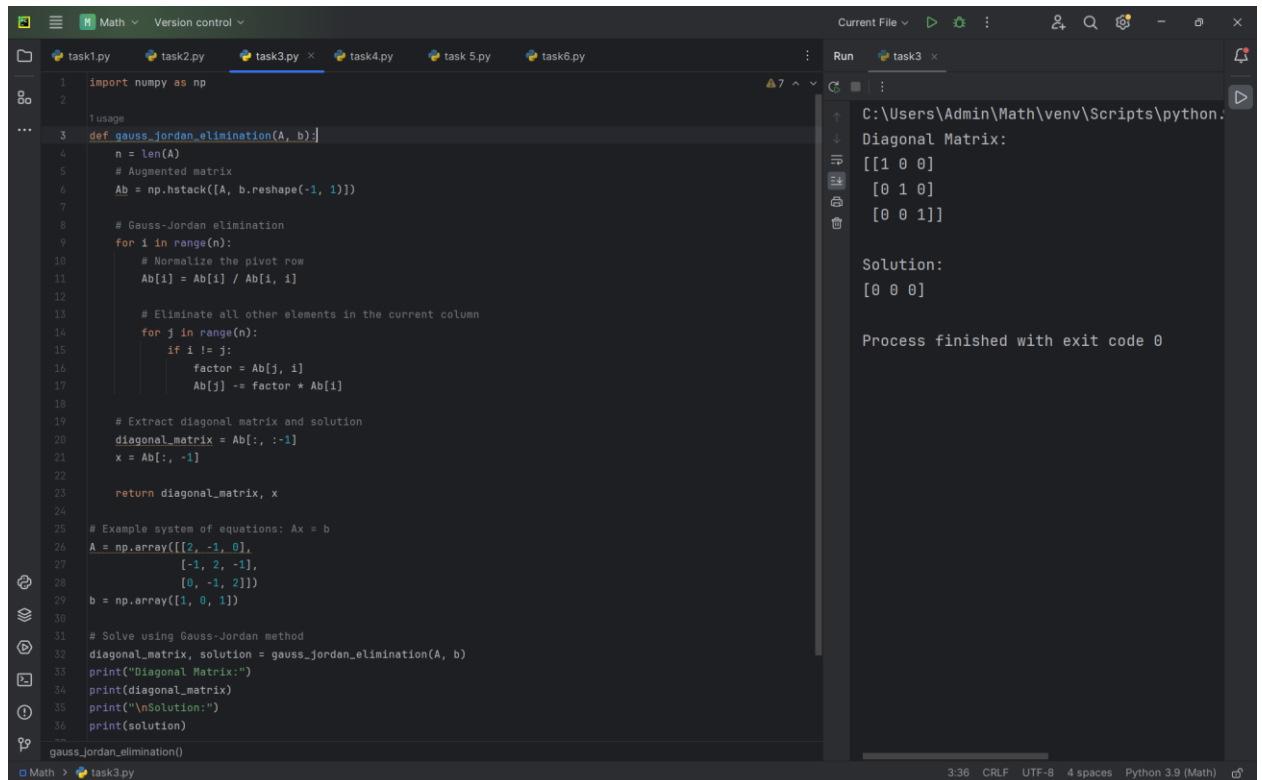
Proper pivoting avoids division by small numbers, which can cause rounding errors and instability in floating-point arithmetic. It ensures a more accurate and stable solution.

Output:

Upper triangular matrix after forward elimination.

Final solution vector x.

Task 3: Gauss-Jordan Method



```
1 import numpy as np
2
3 def gauss_jordan_elimination(A, b):
4     n = len(A)
5     # Augmented matrix
6     Ab = np.hstack([A, b.reshape(-1, 1)])
7
8     # Gauss-Jordan elimination
9     for i in range(n):
10         # Normalize the pivot row
11         Ab[i] = Ab[i] / Ab[i, i]
12
13         # Eliminate all other elements in the current column
14         for j in range(n):
15             if i != j:
16                 factor = Ab[j, i]
17                 Ab[j] -= factor * Ab[i]
18
19     # Extract diagonal matrix and solution
20     diagonal_matrix = Ab[:, :-1]
21     x = Ab[:, -1]
22
23     return diagonal_matrix, x
24
25 # Example system of equations: Ax = b
26 A = np.array([[2, -1, 0],
27               [-1, 2, -1],
28               [0, -1, 2]])
29 b = np.array([1, 0, 1])
30
31 # Solve using Gauss-Jordan method
32 diagonal_matrix, solution = gauss_jordan_elimination(A, b)
33 print("Diagonal Matrix:")
34 print(diagonal_matrix)
35 print("\nSolution:")
36 print(solution)
37
38 gauss_jordan_elimination()
```

Output:

```
C:\Users\Admin\Math\venv\Scripts\python.
Diagonal Matrix:
[[1 0 0]
 [0 1 0]
 [0 0 1]]

Solution:
[0 0 0]

Process finished with exit code 0
```

Description:

Objective: Solve a linear system using the Gauss-Jordan method, transforming the matrix to diagonal form.

Implementation:

Each pivot row is normalized by dividing it by the pivot element, converting the pivot element to 1.

Other rows are adjusted to zero out elements in the pivot column, leading to a diagonal matrix.

Advantages Over Gauss Method:

The Gauss-Jordan method eliminates the need for back substitution, providing the solution directly.

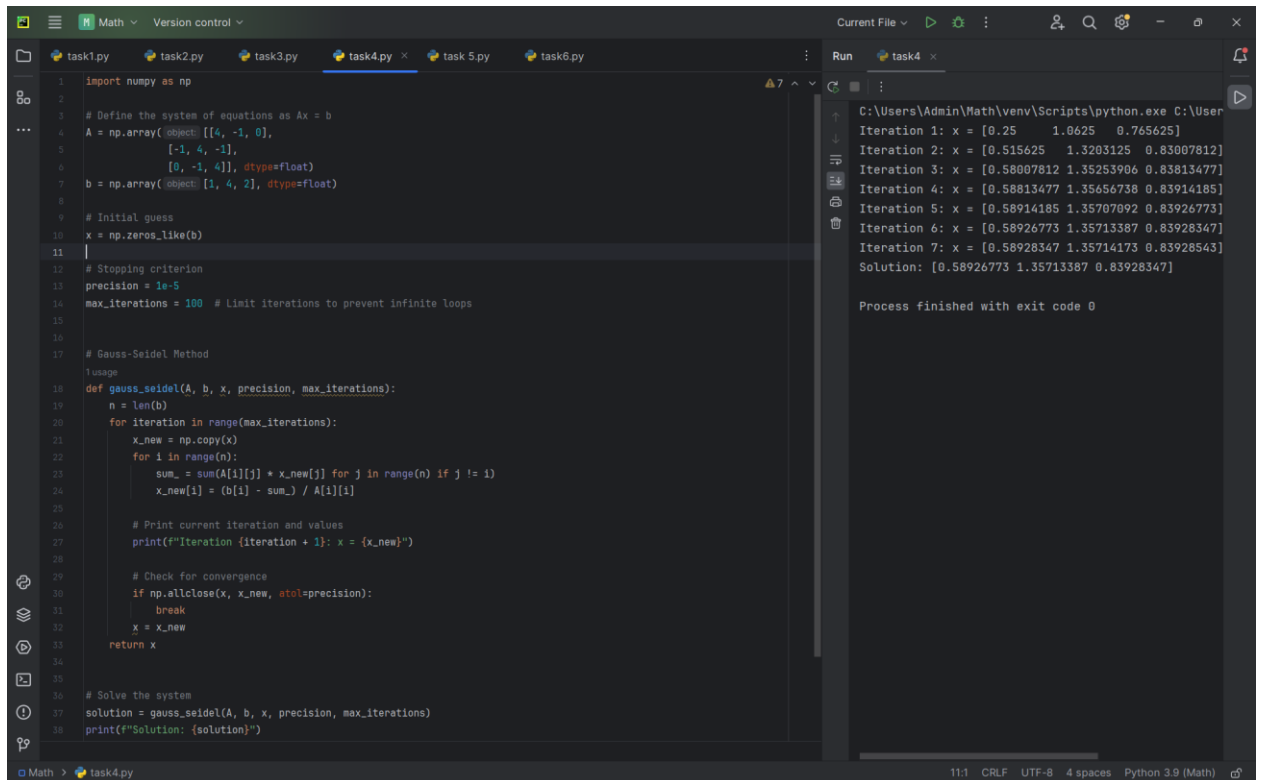
It simultaneously calculates the inverse matrix (if required) and is more efficient for certain systems, especially when solving multiple systems with the same coefficient matrix.

Output:

Diagonal matrix with the solution vector directly visible.

Final solution vector x.

Task 4: Gauss-Seidel Method



```
1 import numpy as np
2
3 # Define the system of equations as Ax = b
4 A = np.array([[4, -1, 0],
5              [-1, 4, -1],
6              [0, -1, 4]], dtype=float)
7 b = np.array([1, 4, 2], dtype=float)
8
9 # Initial guess
10 x = np.zeros_like(b)
11
12 # Stopping criterion
13 precision = 1e-5
14 max_iterations = 100 # Limit iterations to prevent infinite loops
15
16 # Gauss-Seidel Method
17
18 def gauss_seidel(A, b, x, precision, max_iterations):
19     n = len(b)
20     for iteration in range(max_iterations):
21         x_new = np.copy(x)
22         for i in range(n):
23             sum_ = sum(A[i][j] * x_new[j] for j in range(n) if j != i)
24             x_new[i] = (b[i] - sum_) / A[i][i]
25
26         # Print current iteration and values
27         print(f"Iteration {iteration + 1}: x = {x_new}")
28
29         # Check for convergence
30         if np.allclose(x, x_new, atol=precision):
31             break
32         x = x_new
33     return x
34
35 # Solve the system
36 solution = gauss_seidel(A, b, x, precision, max_iterations)
37 print(f"Solution: {solution}")
```

Run task4

```
C:\Users\Admin\Math\venv\Scripts\python.exe C:\User
Iteration 1: x = [0.25  1.0625  0.765625]
Iteration 2: x = [0.515625  1.3203125  0.83007812]
Iteration 3: x = [0.58007812  1.35253906  0.83813477]
Iteration 4: x = [0.58813477  1.35456738  0.83914185]
Iteration 5: x = [0.58914185  1.35707092  0.83926773]
Iteration 6: x = [0.58926773  1.35713387  0.83928347]
Iteration 7: x = [0.58928347  1.35714173  0.83928543]
Solution: [0.58926773  1.35713387  0.83928347]

Process finished with exit code 0
```

Description:

Objective: Solve a linear system using the Gauss-Seidel method with a specified stopping criterion.

Implementation:

Each variable is updated immediately after it is computed, using the most recent values to accelerate convergence.

Convergence is assessed using the infinity norm of the difference between successive iterations, stopping when it is below 10^{-5} .

Explaining Stopping Criterion:

A stricter stopping criterion (smaller tolerance) ensures higher accuracy but requires more iterations and increases execution time.

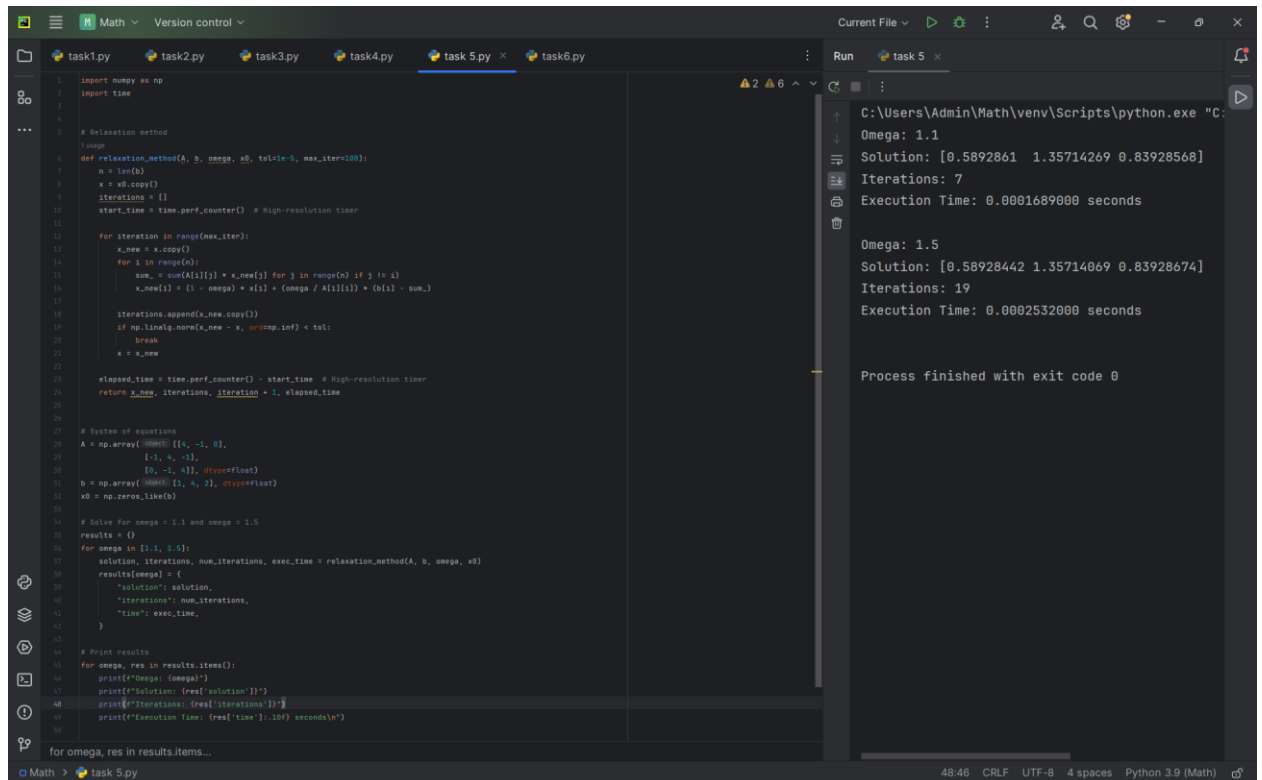
A looser criterion reduces iterations and execution time but may compromise accuracy.

Output:

Iterative results with the current values of variables.

Total number of iterations to achieve the specified accuracy.

Task 5: Relaxation Method



```
1 import numpy as np
2 import time
3
4 # Relaxation method
5 def relaxation_method(A, b, omega, x0, tol=1e-6, max_iter=100):
6     n = len(b)
7     x = x0.copy()
8     iterations = 0
9     start_time = time.perf_counter() # High-resolution timer
10
11     for iteration in range(max_iter):
12         x_new = x.copy()
13         for i in range(n):
14             sum_ = sum(A[i][j] * x[j] for j in range(n) if j != i)
15             x_new[i] = (1 - omega) * x[i] + (omega / A[i][i]) * (b[i] - sum_)
16
17             iterations.append(x_new.copy())
18
19             if np.linalg.norm(x_new - x, ord=np.inf) < tol:
20                 break
21             x = x_new
22
23     elapsed_time = time.perf_counter() - start_time # High-resolution timer
24     return x_new, iterations, iteration + 1, elapsed_time
25
26 # System of equations
27 A = np.array([[4, -1, 0],
28              [-1, 4, -1],
29              [0, -1, 4]], dtype=float)
30 b = np.array([1, 4, 2], dtype=float)
31 x0 = np.zeros_like(b)
32
33 # Solve for omega = 1.1 and omega = 1.5
34 results = {}
35 for omega in [1.1, 1.5]:
36     solution, iterations, num_iterations, exec_time = relaxation_method(A, b, omega, x0)
37     results[omega] = {
38         "solution": solution,
39         "iterations": num_iterations,
40         "time": exec_time,
41     }
42
43 # Print results
44 for omega, res in results.items():
45     print(f"Omega: {omega}")
46     print(f"Solution: {res['solution']}")
47     print(f"Iterations: {res['iterations']}")
48     print(f"Execution Time: {res['time']:.10f} seconds")
49
50 for omega, res in results.items...
```

Run task 5

```
C:\Users\Admin\Math\venv\Scripts\python.exe "C:\Users\Admin\Math\venv\Scripts\python.exe"
Omega: 1.1
Solution: [0.5892861 1.35714269 0.83928568]
Iterations: 7
Execution Time: 0.0001689000 seconds

Omega: 1.5
Solution: [0.58928442 1.35714069 0.83928674]
Iterations: 19
Execution Time: 0.0002532000 seconds

Process finished with exit code 0
```

Description:

Objective: Solve a linear system using the relaxation method with different relaxation parameters ω .

Implementation:

The parameter ω modifies the update formula to speed up or slow down convergence.

For $\omega > 1$ (over-relaxation), the method accelerates updates, reducing iterations if stable.

For $\omega < 1$ (under-relaxation), the method converges more slowly but is more stable.

Explaining the Effect of ω :

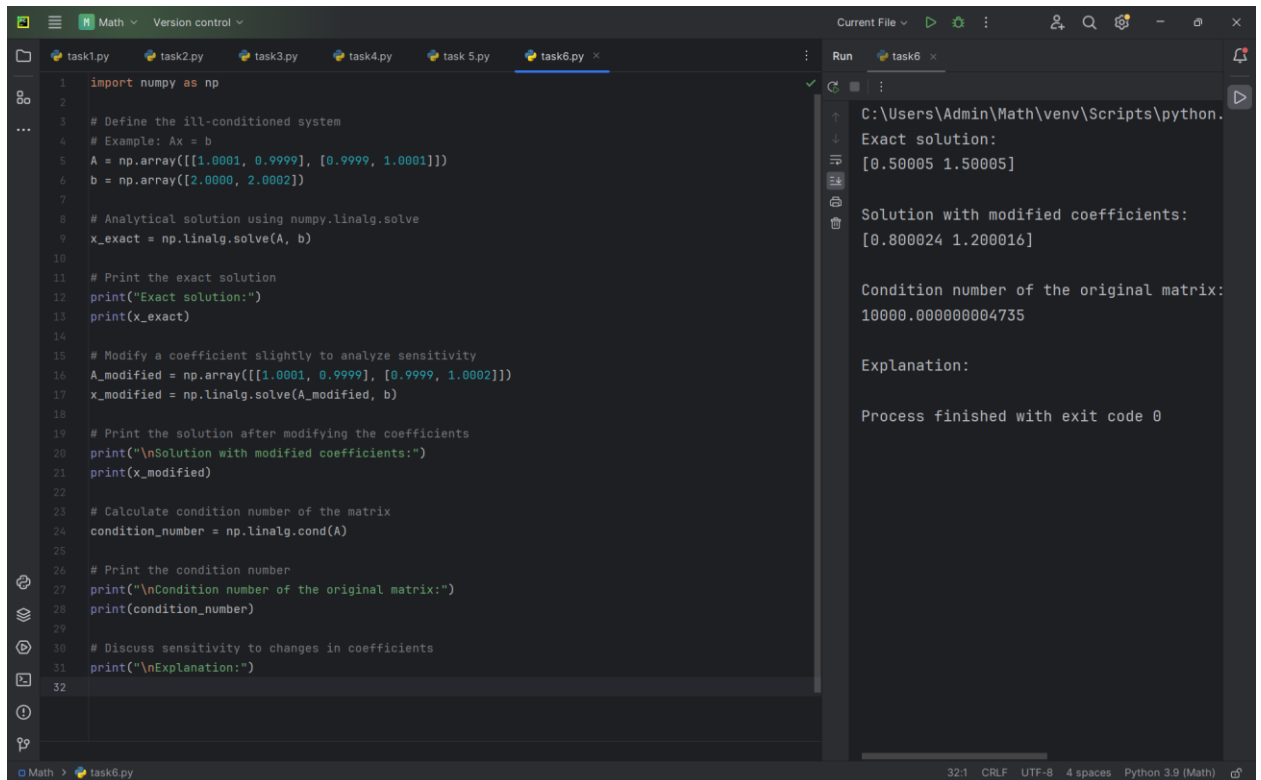
The relaxation parameter significantly impacts the speed of convergence. Over-relaxation is efficient but can lead to divergence if ω is too large.

Output:

Solutions for both $\omega=1.1$ and $\omega=1.5$.

Comparison of iterations and execution times for each parameter.

Task 6: Ill-Conditioned Systems



```
1 import numpy as np
2
3 # Define the ill-conditioned system
4 # Example: Ax = b
5 A = np.array([[1.0001, 0.9999], [0.9999, 1.0001]])
6 b = np.array([2.0000, 2.0002])
7
8 # Analytical solution using numpy.linalg.solve
9 x_exact = np.linalg.solve(A, b)
10
11 # Print the exact solution
12 print("Exact solution:")
13 print(x_exact)
14
15 # Modify a coefficient slightly to analyze sensitivity
16 A_modified = np.array([[1.0001, 0.9999], [0.9999, 1.0002]])
17 x_modified = np.linalg.solve(A_modified, b)
18
19 # Print the solution after modifying the coefficients
20 print("\nSolution with modified coefficients:")
21 print(x_modified)
22
23 # Calculate condition number of the matrix
24 condition_number = np.linalg.cond(A)
25
26 # Print the condition number
27 print("\nCondition number of the original matrix:")
28 print(condition_number)
29
30 # Discuss sensitivity to changes in coefficients
31 print("\nExplanation:")
32
```

Output:

```
C:\Users\Admin\Math\venv\Scripts\python.
Exact solution:
[0.50005 1.50005]

Solution with modified coefficients:
[0.800024 1.200016]

Condition number of the original matrix:
10000.000000004735

Explanation:

Process finished with exit code 0
```

Description:

Objective: Solve an ill-conditioned system analytically and numerically, demonstrating sensitivity to coefficient changes.

Implementation:

The original system is solved to find the baseline solution.

A small perturbation is introduced to one coefficient in the matrix, and the system is resolved to observe the change in the solution.

Explaining Sensitivity:

Ill-conditioned systems have high condition numbers, meaning small changes in the coefficients lead to large changes in the solution.

This occurs because the rows or columns of the matrix are nearly linearly dependent, amplifying errors.

Output:

Analytical solution for the unperturbed system.

Numerical solution for the unperturbed and perturbed systems.

Comparison of the solutions to demonstrate sensitivity.