暨南大学本科实验报告专用纸

课程名称 数值计算实验 成绩评定 _____

实验项目名称 Computing Problems 指导老师 Liangda Fang

实验项目编号 01 实验项目类型 验证型 实验地点 _____

学生姓名 _____ 学号 _____

学院 国际学院 系 _____ 专业 计算机科学与技术

实验时间 2023 年 10 月 13 日上午 10:30 ~ 12:10

I. Problem

Let A be the 1000×1000 matrix with entries $A(i,i) = i, A(i,i+1) = A(i+1,i) = \frac{1}{2}, A(i,i+2) = A(i+2,i) = \frac{1}{2}$ for all i that fit within the matrix.

- Solve the system with $Ax = [1, 1, \cdots, 1]^\top$ by the following methods in 15 steps:
  1. The Jacobi Method;
  2. The Gauss-Seidel Method;
  3. SOR with $\omega = 1.1$;
  4. The Conjugate Gradient Method;
  5. The Conjugate Gradient Method with Jacobi preconditioner.
- Report the errors of every step for each method.

II. Algorithm Summary

1. The Jacobi Method

Method Explanation: The Jacobi method is an iterative technique for solving a system of linear equations. It updates the solution vector by considering one equation at a time and uses the previous solution to estimate the next value.

Method Application: Jacobi can be applied to solve linear systems, particularly when the matrix is diagonally dominant or positive definite. It's also used in solving partial differential equations and eigenvalue problems.

Mathematical Steps and Formulas: The Jacobi iteration for solving $Ax = b$: Update x at each step using the formula, suppose that A is a matrix, D is the main diagonal of A, L is the lower triangle of A, U is the upper triangle of A.

$$
\begin{aligned}
Ax &= b \\
(D + L + U)x &= b \\
Dx &= b - (L + U)x \\
Dx_{k+1} &= b - (L + U)x_k \\
x_{k+1} &= D^{-1}(b - (L + U)x_k)
\end{aligned}
$$

Jacobi Method is as follows:

$$
\begin{aligned}
x_0 &= \text{initial vector} \\
x_{k+1} &= D^{-1}(b - (L+U)x_k) \\
x_{k+1,i} &= \frac{1}{a_{ii}}(b_i - \sum_{j \neq i} a_{ij}x_{k,j}) \text{ for } 1 \leq i \leq n
\end{aligned}
$$

2. The Gauss-Seidel Method

Method Explanation: The Gauss-Seidel method is another iterative approach for solving linear systems. It updates the solution vector, considering the most recent values as soon as they become available.

Method Application: Like Jacobi, Gauss-Seidel is used to solve linear systems and can be particularly effective when the matrix is diagonally dominant.

Mathematical Steps and Formulas: The Gauss-Seidel iteration for solving $Ax = b$: Update x at each step using the formula, suppose that A is a matrix, D is the main diagonal of A, L is the lower triangle of A, U is the upper triangle of A.

$$
\begin{aligned}
Ax &= b \\
(D+L+U)x &= b \\
(D+L)x &= b - Ux \\
(D+L)x_{k+1} &= b - Ux_k
\end{aligned}
$$

Gauss-Seidel Method is as follows:

$$
\begin{aligned}
x_0 &= \text{initial vector} \\
x_{k+1} &= D^{-1}(b - Ux_k - Lx_{k+1}) \\
x_{k+1,i} &= \frac{1}{a_{ii}}(b_i - \sum_{j>i} a_{ij}x_{k,j} - \sum_{j<i} a_{ij}x_{k+1,j}) \text{ for } 1 \leq i \leq n
\end{aligned}
$$

3. SOR with $\omega = 1.1$

Method Explanation: Successive Over-Relaxation (SOR) is an extension of the Gauss-Seidel method with an over-relaxation parameter (ω). SOR can accelerate convergence for the right choice of ω.

Method Application: SOR is used when convergence of the Gauss-Seidel method is slow. By tuning ω, it can be applied to a variety of linear systems.

Mathematical Steps and Formulas: SOR iteration for solving $Ax = b$: Update x at each step using the formula, ω is the relaxation factor, suppose that A is a matrix, D is the main diagonal of A, L is the lower triangle of A, U is the upper triangle of A.

$$
\begin{aligned}
Ax &= b \\
\omega Ax &= \omega b \\
(\omega D + \omega L + \omega U)x &= \omega b \\
(D + \omega L)x &= \omega b - \omega U x + (1-\omega)Dx \\
(D + \omega L)x_{k+1} &= \omega b - \omega U x_k + (1-\omega)Dx_k \\
Dx_{k+1} &= \omega b + (1-\omega)Dx_k - \omega U x_k - \omega L x_{k+1}) \\
x_{k+1} &= (1-\omega)x_k + D^{-1}(\omega b - \omega U x_k - \omega L x_{k+1})
\end{aligned}
$$

Successive Over-Relaxation is as follows:

$$
\begin{aligned}
x_0 &= \text{initial vector} \\
x_{k+1} &= (1-\omega)x_k + D^{-1}(\omega b - \omega U x_k - \omega L x_{k+1}) \\
x_{k+1,i} &= (1-\omega)x_{k,i} + \frac{\omega}{a_{ii}}(b_i - \sum_{j>i} a_{ij}x_{k,j} - \sum_{j<i} a_{ij}x_{k+1,j}) \text{ for } 1 \le i \le n
\end{aligned}
$$

4. The Conjugate Gradient Method

Method Explanation: The Conjugate Gradient method is an iterative technique to solve symmetric positive-definite linear systems. It minimizes the error by considering conjugate directions in the solution space.

Method Application: Conjugate Gradient is used for solving large systems of linear equations, especially when A is symmetric and positive definite, e.g., in finite element analysis and optimization problems.

Mathematical Steps and Formulas: Conjugate Gradient iteration: $x_0 = \text{initial guess}$, $d_0 = r_0 = b - Ax_0$, update x, r, and d at each step using the formulas:

$$\alpha = \frac{r^\top \cdot r}{d^\top \cdot A \cdot d}$$

$$x_{k+1} = x_k + \alpha \cdot d$$

$$r_{k+1} = r_k - \alpha \cdot A \cdot d$$

$$\beta = \frac{r_{k+1}^\top \cdot r_{k+1}}{r_k^\top \cdot r_k}$$

$$d_{k+1} = r_{k+1} + \beta \cdot d_k$$

Repeat the above steps when r_k is sufficiently small, then return x_k.

5. The Conjugate Gradient Method with Jacobi Preconditioner

Method Explanation: The Conjugate Gradient (CG) method with the Jacobi preconditioner is an iterative technique used to solve symmetric positive-definite linear systems. It combines the Conjugate Gradient method with a preconditioning step using the Jacobi preconditioner, which is the diagonal matrix D of matrix A. The Jacobi preconditioner improves the conditioning of the linear system, which can speed up convergence, particularly for ill-conditioned matrices.
Method Application: The Conjugate Gradient method with Jacobi preconditioner is valuable for solving large systems of linear equations when the matrix A is symmetric and positive definite, and when the system's convergence is slow or numerically unstable. This method finds applications in various fields, including finite element analysis, structural analysis, and optimization problems.
Mathematical Steps and Formulas:

a) Preconditioning Step:

Apply the Jacobi preconditioner to the residual vector r before each iteration. This step involves multiplying r by the inverse of the preconditioned submatrix between the left and right sides of $Ax = b$:

$$z = D^{-1} \cdot r$$

Here, D is the diagonal matrix of A, and D^{-1} is its inverse.

b) Conjugate Gradient Iteration with Preconditioning:

Initialization: Start with initial values $x_0 = 0$ and $r_0 = b$.
Iteration: For each iteration k:
Compute the value of d_k using the formula:

$$d_{k+1} = z_{k+1} + \beta_{k+1} \cdot d_k$$

Calculate the value of α_k as:

$$\alpha_k = \frac{r_k^\top \cdot z_k}{d_k^\top \cdot A \cdot d_k}$$

Update the solution vector x_k, the residual r_k and z_k as follows:

$$x_{k+1} = x_k + \alpha_k \cdot d_k$$

$$r_{k+1} = r_k + \alpha_k \cdot A \cdot d_k$$

$$z_{k+1} = D^{-1} r_{k+1}$$

Calculate the value of β_k using:

$$\beta_k = \frac{r_{k+1}^\top \cdot z_{k+1}}{r_k^\top \cdot z_k}$$

c) Convergence:

Continue the iterations until a convergence criterion is met, such as a specified number of iterations or a tolerance level for the residual error.

The Jacobi preconditioner reduces the number of iterations needed for convergence compared to the standard Conjugate Gradient method when dealing with ill-conditioned matrices. It enhances the stability and efficiency of the solution process.
This modified Conjugate Gradient method with Jacobi preconditioner is particularly useful when dealing with linear systems in various engineering and scientific applications, where rapid and stable convergence is essential.

III. Experimental procedures

Step1: Define the matrix A, initial solution x and the right side of the equation, we denote it as b;
Matrix A:

$$A = \begin{bmatrix} 1.0 & 0.5 & 0.5 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 \\ 0.5 & 2.0 & 0.5 & 0.5 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 \\ 0.5 & 0.5 & 3.0 & 0.5 & 0.5 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.5 & 0.5 & 4.0 & 0.5 & 0.5 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.5 & 0.5 & 5.0 & 0.5 & 0.5 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.5 & 0.5 & 6.0 & 0.5 & 0.5 & \cdots & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.5 & 0.5 & 7.0 & 0.5 & \cdots & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.5 & 0.5 & 8.0 & \cdots & 0.0 & 0.0 & 0.0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 998.0 & 0.5 & 0.5 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.5 & 999.0 & 0.5 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.5 & 0.5 & 1000.0 \end{bmatrix}$$

The initial x we define it as follows:

$$x = \begin{bmatrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & \cdots & 0.0 & 0.0 & 0.0 \end{bmatrix}^\top$$

Right side of the equation, we denote it as b:

$$b = \begin{bmatrix} 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & \cdots & 1.0 & 1.0 & 1.0 \end{bmatrix}^\top$$

Step2: Implenment various methods(Jacobi's Method, Gauss-Seidel Method, etc.) for finding solutions to systems of linear equations;
Step3: Define function to computes the error in every iteration step, our error function definition is as follows:

$$E(x, y) = \sum_{i=0}^{n} (x_i^2 - y_i^2)$$

Where x denotes the predict solution, y denotes the exact solution, we square each element in the vector, subtract it from the value at the corresponding position, and finally sum it;
Step4: Using the various implementation of above methods to find the solution to the given system of linear equation;
Step5: Print the solution to the equation $Ax = b$ after a given number of iterations(15);
Step6: Plot the errors for the above iterative algorithm.

IV. Result Analysis

1. Result:

The exact solution to system of linear equation $Ax = b$ we use $A^{-1}b$ to find out is as follows:

$$x_{\text{exact solution}} = \begin{bmatrix} 0.83821553 & 0.22085007 & 0.10271887 & \cdots & 0.00100000 & 0.00099950 & 0.00099900 \end{bmatrix}^\top$$

The results we obtained after 15 iterations using different methods are as follows:
The Jacobi Method:

$$x = \begin{bmatrix} 0.83884825 & 0.22130780 & 0.10307445 & \cdots & 0.00100000 & 0.00099950 & 0.00099900 \end{bmatrix}^\top$$

The Gauss-Seidel Method:

$$x = \begin{bmatrix} 0.83821553 & 0.22085007 & 0.10271887 & \cdots & 0.00100000 & 0.00099950 & 0.00099900 \end{bmatrix}^\top$$

SOR Method:

$$x = \begin{bmatrix} 0.83821553 & 0.22085007 & 0.10271887 & \cdots & 0.00100000 & 0.00099950 & 0.00099900 \end{bmatrix}^\top$$

The Conjugate Gradient Method:

$$x = \begin{bmatrix} 0.17520578 & 0.16089050 & 0.14741100 & \cdots & 0.00136011 & 0.00142141 & 0.00148697 \end{bmatrix}^\top$$

The Conjugate Gradient Method with Jacobi preconditioner:

$$x = \begin{bmatrix} 0.83821553 & 0.22085007 & 0.10271887 & \cdots & 0.00100000 & 0.00099950 & 0.00099900 \end{bmatrix}^\top$$

2. The errors of every step for each method:

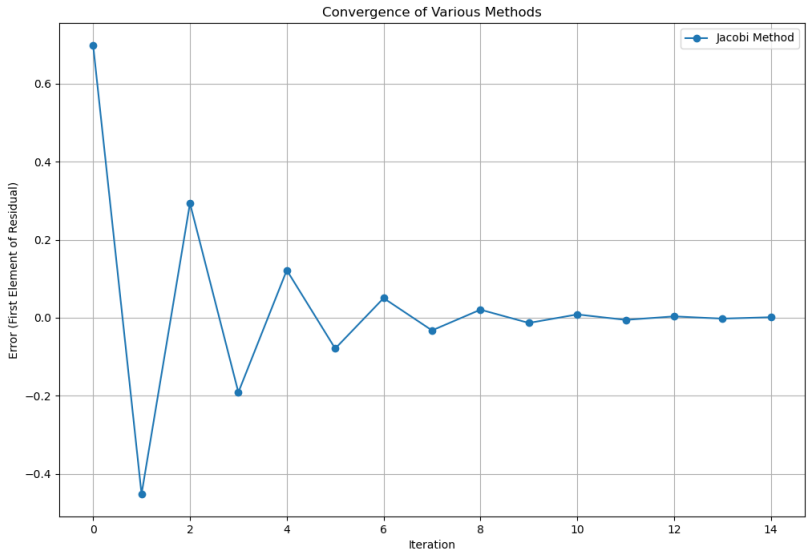

Figure 1: The Jacobi Method

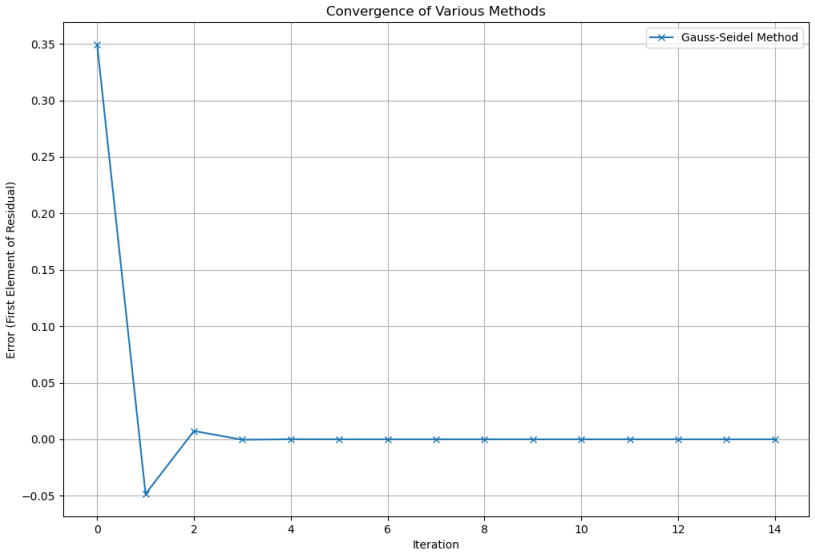

Figure 2: The Gauss-Seidel Method

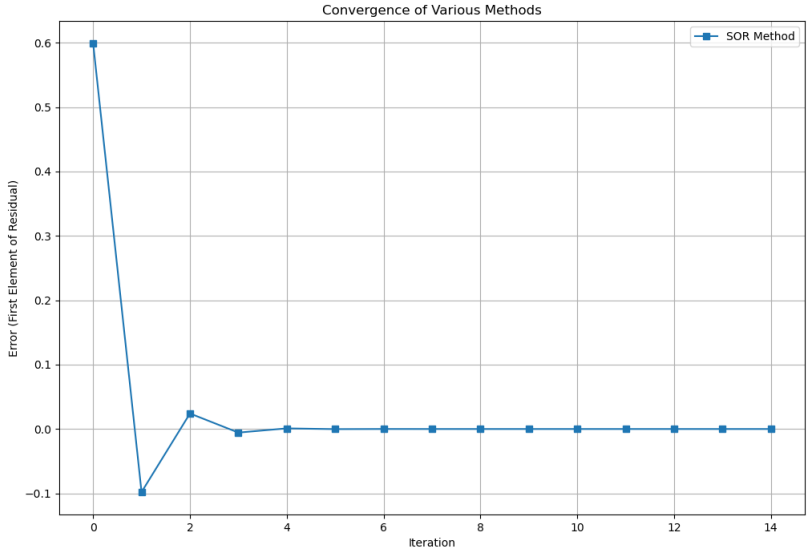

Figure 3: SOR Method with $\omega = 1.1$

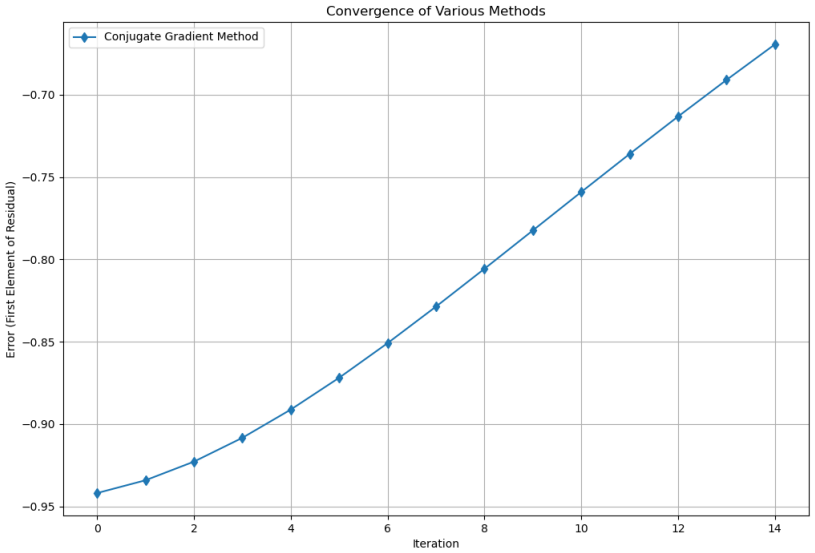

Figure 4: The Conjugate Gradient Method

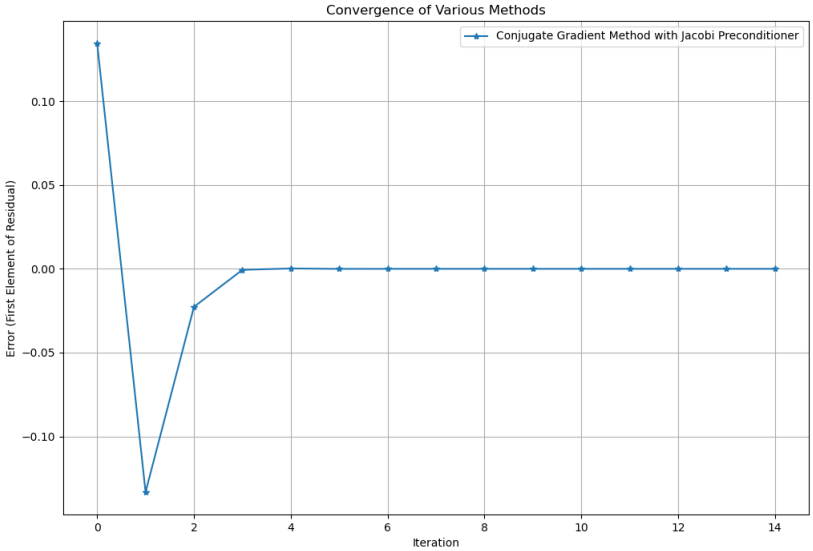

Figure 5: The Conjugate Gradient Method with Jacobi preconditioner

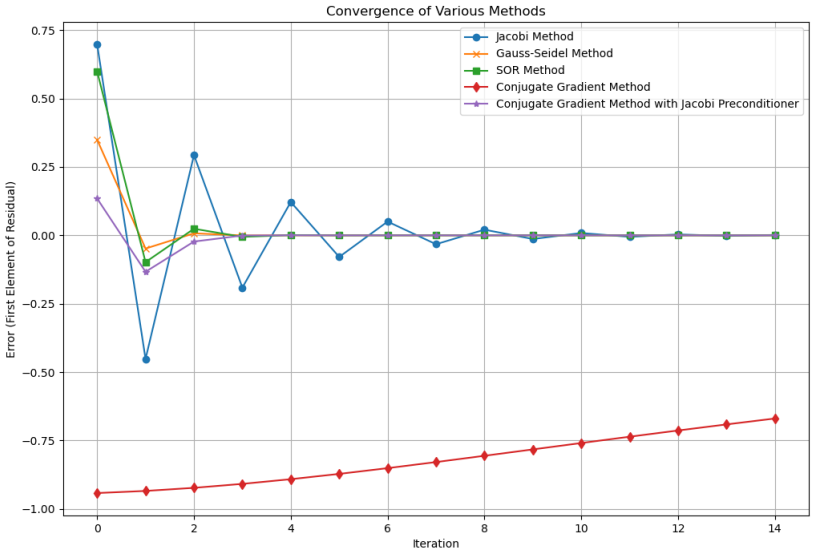

Figure 6: All Result

3. Analysis:

We can observe that the fastest convergence speed is the Gauss-Seidel method, followed by the SOR method, then the conjugate gradient method with Jacobi preconditioner, then the Jacobi method, and finally the simple conjugate gradient method. At the same time, we find that the conjugate gradient has a large error within 15 steps.

V. Experimental Summary

In this experiment, five different iterative methods were used to solve the linear equations $Ax = b$, and the convergence rates of the five methods were compared under the same number of iterations. Let me know that sometimes we don't need to get the exact solution, we just want to get an approximate solution in a short time. It reflects the trade between time and quality. It is also noted that A is not strictly diagonally dominant, but it still applies to Jacobi, G-S, and SOR. In short, these five methods have their own advantages, only in accordance with local conditions to use, in order to obtain satisfactory results.

VI. Appendix: Source Code

1. methods.py

```python
import numpy as np


def jacobi(A, b, exact_sol, iter_num):
    n = len(b)
    x = np.zeros(n)
    D_inverse = np.diag(1 / np.diag(A))
    Rest = A - np.diag(np.diag(A))  # Rest part = Lower triangle + Upper triangle
    errors = []

    for k in range(iter_num):
        x_new = D_inverse @ (b - Rest @ x)
        error = np.sum(x_new**2 - exact_sol**2)
        errors.append(error)
        x = x_new

    return x, errors


def gauss_seidel(A, b, exact_sol, iter_num):
    n = len(b)
    x = np.zeros(n)
    errors = []

    for k in range(iter_num):
        x_new = np.copy(x)
        for i in range(n):
            x_new[i] = (b[i] - A[i, :i] @ x_new[:i] - A[i, i + 1 :] @ x[i + 1 :]) / A[i, i]

        error = np.sum(x_new**2 - exact_sol**2)
        errors.append(error)
        x = x_new

    return x, errors


def sor(A, b, omega, exact_sol, iter_num):
    n = len(b)
    x = np.zeros(n)
    errors = []

    for k in range(iter_num):
        x_new = np.copy(x)
        for i in range(n):
            x_new[i] = (
                    (1 - omega) * x[i] + (omega / A[i, i]) * (b[i] - A[i, :i] @ x_new[:i] - A[i, i + 1 :] @ x[i + 1 :])
            )

        error = np.sum(x_new**2 - exact_sol**2)
        errors.append(error)
        x = x_new

```

```
    return x, errors


def conjugate_gradient(A, b, exact_sol, iter_num):
    n = len(b)
    x = np.zeros(n)
    r = b - A @ x
    p = r
    errors = []

    for k in range(iter_num):
        Ap = A @ p
        alpha = (r.T @ r) / (p.T @ Ap)
        x_new = x + alpha * p
        r_new = r - alpha * Ap

        error = np.sum(x_new**2 - exact_sol**2)
        errors.append(error)

        beta = (r_new.T @ r_new) / (r.T @ r)
        p = r_new + beta * p
        r, x = r_new, x_new

    return x, errors


def conjugate_gradient_preconditioned(A, b, exact_sol, iter_num):
    n = len(b)
    x = np.zeros(n)
    M_inv = np.diag(1 / np.diag(A))
    r = b - A @ x
    z = M_inv @ r
    p = z
    errors = []

    for k in range(iter_num):
        Ap = A @ p
        alpha = (r.T @ z) / (p.T @ Ap)
        x_new = x + alpha * p
        r_new = r - alpha * Ap

        error = np.sum(x_new**2 - exact_sol**2)
        errors.append(error)

        z_new = M_inv @ r_new
        beta = (r_new.T @ z_new) / (r.T @ z)
        p = z_new + beta * p
        z, r, x = z_new, r_new, x_new

    return x, errors
```

2. solutions.py

```python
import matplotlib.pyplot as plt
import numpy as np

from lab1_methods import jacobi, gauss_seidel, sor, conjugate_gradient, conjugate_gradient_preconditioned

n = 1000  # Matrix size
iteration_num = 15

# Define the matrix A
A = np.zeros((1000, 1000))
A += np.diag(0.5 * np.ones(999), 1)  # Diagonal values at 1, 2, -1, -2 offsets
A += np.diag(0.5 * np.ones(998), 2)
A += np.diag(0.5 * np.ones(999), -1)
A += np.diag(0.5 * np.ones(998), -2)
A += np.diag(np.arange(1, 1001))  # Main diagonal

# Define vector b
b = np.ones(n)

# Calculate the exact solution of Ax = b, y denotes the exact solution
y = np.array(np.matrix(A).I @ b)

# Run the methods and record the errors
sol_jacobi, errs_jacobi = jacobi(A, b, y, iteration_num)
sol_gs, errs_gs = gauss_seidel(A, b, y, iteration_num)
sol_sor, errs_sor = sor(A, b, 1.1, y, iteration_num)
sol_cg, errs_cg = conjugate_gradient(A, b, y, iteration_num)
sol_cg_preconditioned, errs_cg_preconditioned = conjugate_gradient_preconditioned(A, b, y, iteration_num)

# Print the final time solutions
print("The solution of A_inv @ b(exact solution):", y)
print("The solution of Jacobi Method:", sol_jacobi)
print("The solution of Gauss-Seidel Method:", sol_gs)
print("The solution of SOR Method:", sol_sor)
print("The solution of Conjugate Gradient Method:", sol_cg)
print("The solution of Conjugate Gradient Method with Jacobi Preconditioner:", sol_cg_preconditioned)

# Plot the error at each iteration
plt.figure(figsize=(12, 8))
plt.plot(errs_jacobi, marker="o", label="Jacobi Method")
plt.plot(errs_gs, marker="x", label="Gauss-Seidel Method")
plt.plot(errs_sor, marker="s", label="SOR Method")
plt.plot(errs_cg, marker="d", label="Conjugate Gradient Method")
plt.plot(errs_cg_preconditioned, marker="*", label="Conjugate Gradient Method with Jacobi Preconditioner")

plt.xlabel("Iteration")
plt.ylabel("Error (First Element of Residual)")
plt.title("Convergence of Various Methods")
plt.legend()
plt.grid(True)

plt.show()
```