求解离散泊松方程的三种迭代方法比较

摘要

本文研究了三种典型迭代方法（Jacobi、Gauss-Seidel 和 SOR）在求解一维离散泊松方程时的性能比较及其影响因素。通过数值实验，分析了迭代次数、最终残差和 CPU 运行时间等指标，验证了 SOR 方法在最优松弛参数下的显著优势。

实验结果表明，SOR 方法的收敛速度远超 Jacobi 和 Gauss-Seidel 方法，且计算效率对松弛参数（ω）和步长（h）高度敏感：当ω接近理论最优值时，SOR 的迭代次数和计算时间大幅减少；而步长减小虽不影响精度，但会显著增加计算成本。此外，研究还发现松弛参数超出合理范围（$0<\omega<2$）会导致迭代发散。

基于实验结果，本文建议在实际应用中需合理选择松弛参数和网格步长以平衡效率与精度。未来工作可扩展至高维泊松方程、自适应参数优化算法及其他迭代方法的对比研究。

关键词：泊松方程，迭代方法，收敛效率，SOR 迭代，松弛参数

Comparison of Three Iterative Methods for Solving Discrete Poisson Equations

Abstract

This paper compares the performance of three typical iterative methods (Jacobi, Gauss-Seidel, and SOR) for solving the one-dimensional discrete Poisson equation, analyzing influencing factors. Numerical experiments verify the significant advantages of the SOR method with optimal relaxation parameters, using metrics like iteration counts, final residuals, and CPU time.

Experimental results show SOR converges much faster than Jacobi and Gauss-Seidel. Computational efficiency is highly sensitive to relaxation parameter (ω) and step size (h): near-optimal ω reduces iterations and time drastically, while smaller h increases costs without affecting accuracy. Relaxation parameters outside a reasonable range ($0 < \omega < 2$) cause divergence.

Based on the experimental results, the study recommends balancing efficiency and accuracy by selecting appropriate ω and h. Future work may extend to high-dimensional Poisson equations, adaptive parameter optimization, and comparisons with other iterative methods.

Keywords: Poisson Equation, Iterative Methods, Convergence Efficiency, SOR Iteration, Relaxation Parameter

CONTENTS

1 INTRODUCTION OF THE PROBLEM

1.1 Background

The Poisson equation is one of the most fundamental partial differential equations in mathematical physics, with its standard for $-\nabla^2 u = f$. This equation finds wide applications across multiple domains, for example: in electrostatics we utilize it to describing electric potential distribution, in heat conduction we use it to characterizing steady-state temperature fields and in fluid mechanics it can be involved in streamfunction-vorticity formulations. And in engineering applications, the Poisson equation also make a difference in computer graphics, structural mechanics, etc.

Analytical solutions are typically only available for simple geometric configurations. For real-world engineering problems, numerical solutions become essential. For large-scale problems, direct methods (e.g., Gaussian elimination) become computationally prohibitive, which making iterative methods a crucial alternative.

Therefore today I'll choose to utilize three typical different iterative methods to solve the discrete Poisson equation and compare their advantages and disadvantages.

1.2 The Construction of Discrete Poisson Equations

1.2.1 One-Dimensional Case with Dirichlet Boundary Conditions

Consider the one-dimensional Poisson equation with Dirichlet boundary condition

$$\begin{cases} -\dfrac{d^2u}{dx^2} = f(x), \ \ 0 < x < 1 \\ u(0) = a, u(1) = b, \end{cases}$$

where $f(x)$ is given, $u(x)$ is unknown.

1.2.2 Finite Difference Discretization

Take step-size $h = \dfrac{1}{n+1}$, nodes $x_i = ih, i = 0, 1, 2, ..., n+1$.

We employ the centered difference approximation to obtain $(i = 1, 2, ..., n)$

$$-\frac{d^2u(x)}{dx^2}\bigg|_{x_i} = \frac{-u(x_{i-1}) + 2u(x_i) - u(x_{i+1})}{h^2} + O\left(h^2 \cdot \left\|\frac{d^4u}{dx^4}\right\|_\infty\right).$$

Substitute the approximation into Poisson equation, and drop the higher order term to obtain the discrete equations

$$-u_{i-1} + 2u_i - u_{i+1} = h^2 f_i,$$

where $f_i = f(x_i)$, u_i is the approximation of $u(x_i)$.

Let $i = 1, 2, ..., n$, then we have n linear equations, and the matrix form is given by

$$T_n \boldsymbol{u} = \boldsymbol{f},$$

where

$$T_n = \begin{bmatrix} 2 & -1 & & \\ -1 & \ddots & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 2 \end{bmatrix}, \boldsymbol{u} = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n-1} \\ u_n \end{bmatrix}, \boldsymbol{f} = \begin{bmatrix} h^2 f_1 + u_0 \\ h^2 f_2 \\ \vdots \\ h^2 f_{n-1} \\ h^2 f_n + u_{n+1} \end{bmatrix}.$$

1.3 Restatement of the Problem

Therefore, the problem has been turned to compare three typical iterative methods (Jacobi, Gauss-Seidel, and SOR) for solving the n-dimensional linear system, and analyzing their convergence and computational efficiency, and implementation difficulty.

2 DESCRIPTION OF THE METHODS AND THEIR THEORIES

2.1 Matrix Splitting Iterative Method

Matrix splitting methods constitute a class of stationary iterative techniques for solving linear systems of the form:

$$A\boldsymbol{x} = \boldsymbol{b},$$

where $A \in \mathbb{R}^{n\times n}$ is nonsingular. The core idea involves decomposing the matrix A into:

$$A = M - N,$$

where M is nonsingular and easily invertible. This yields the iterative scheme:

$$\boldsymbol{x}^{(k+1)} = M^{-1}N\boldsymbol{x}^{(k)} + M^{-1}\boldsymbol{b}.$$

The iteration matrix $G = M^{-1}N$ determines convergence properties, with the method being convergent if and only if $\rho(G) < 1$, where ρ denotes the spectral radius.

2.2 Jacobi Iterative Method

2.2.1 Fundamental Formulation

The Jacobi method represents one of the simplest matrix splitting techniques for solving linear systems:

$$A\boldsymbol{x} = \boldsymbol{b},$$

where $A \in \mathbb{R}^{n\times n}$ is nonsingular. The method employs the following matrix decomposition:

$$A = D - (L + U),$$

where $D = \operatorname{diag}(A)$ is the diagonal component, $-L$ and $-U$ are the strictly lower and upper triangular parts of A, respectively.

2.2.2 Iterative Scheme

According to the matrix splitting iterative method, we denote $M = D$, $N = L + U$ and then obtain the resulting iterative process:

$$\boldsymbol{x}^{(k+1)} = D^{-1}(L + U)\boldsymbol{x}^{(k)} + D^{-1}\boldsymbol{b},$$

with the iteration matrix $G_J = D^{-1}(L + U)$.

The Jacobi iteration can be written into the following equivalent form:

$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + D^{-1}(\boldsymbol{b} - A\boldsymbol{x}^{(k)}) = \boldsymbol{x}^{(k)} + D^{-1}\boldsymbol{r}_k,$$

where $\boldsymbol{r}_k = \boldsymbol{b} - A\boldsymbol{x}^{(k)}$ is the residual vector after k iterations, $k = 0, 1, \ldots$. And the entry-wise form of Jacobi iteration is:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1, j\neq i}^{n} a_{ij} x_j^{(k)}\right), \ i = 1, 2, \ldots, n.$$

2.3 Gauss-Seidel Iterative Method

2.3.1 Fundamental Formulation

The Gauss-Seidel (G-S) method represents an enhanced version of the Jacobi approach for solving linear systems:

$$A\boldsymbol{x} = \boldsymbol{b},$$

where $A \in \mathbb{R}^{n \times n}$ is nonsingular. The method employs the following matrix decomposition:

$$A = (D - L) - U,$$

where $D = \text{diag}(A)$ is the diagonal component, $-L$ and $-U$ are the strictly lower and upper triangular parts of A, respectively.

2.3.2 Iterative Scheme

According to the matrix splitting iterative method, we denote $M = D - L$, $N = U$ and then obtain the resulting iterative process:

$$\boldsymbol{x}^{(k+1)} = (D-L)^{-1}U\boldsymbol{x}^{(k)} + (D-L)^{-1}\boldsymbol{b},$$

with the iteration matrix $G_{G-S} = (D-L)^{-1}U$.

The G-S iteration can be written into the following equivalent form:

$$D\boldsymbol{x}^{(k+1)} = L\boldsymbol{x}^{(k+1)} + U\boldsymbol{x}^{(k)} + \boldsymbol{b},$$

then we have the entry-wise form:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right),\ i = 1, 2, \ldots, n.$$

2.4 Successive Over-Relaxation Iterative Method

2.4.1 Fundamental Formulation

The Successive Over-Relaxation (SOR) method represents an accelerated variant of the Gauss-Seidel technique for solving linear systems:

$$A\boldsymbol{x} = \boldsymbol{b},$$

where $A \in \mathbb{R}^{n\times n}$ is nonsingular. The method introduces a relaxation parameter ω to enhance convergence:

$$A = \left(\frac{D}{\omega} - L\right) - \left(\frac{(1-\omega)D}{\omega} + U\right),$$

where $D = \text{diag}(A)$ is the diagonal component, $-L$ and $-U$ are the strictly lower and upper triangular parts of A respectively, $\omega \in (0,2)$ is the relaxation parameter.

2.4.2 Iterative Scheme

The motivation of SOR iteration (Successive Overrelaxation) is to improve the G-S iteration by taking an appropriate weighted average of $\boldsymbol{x}^{(k+1)}$ and $\boldsymbol{x}^{(k)}$. We denote $M = \frac{1}{\omega}D - L$, $N = \frac{1-\omega}{\omega}D + U$ then can obtain

$$\boldsymbol{x}^{(k+1)} = (D-\omega L)^{-1}((1-\omega)D + \omega U)\boldsymbol{x}^{(k)} + \omega(D-\omega L)^{-1}\boldsymbol{b},$$

with the iteration matrix $G_{SOR} = (D-\omega L)^{-1}((1-\omega)D+\omega U)$, or equivalently

$$\boldsymbol{x}^{(k+1)} = (1-\omega)\boldsymbol{x}^{(k)} + \omega\big(D^{-1}(L\boldsymbol{x}^{k+1} + U\boldsymbol{x}^{(k)}) + D^{-1}\boldsymbol{b}\big).$$

Due to the matrix form, we have the entry-wise form of SOR iteration

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

2.4.3 Relaxation Mechanism

We distinguish three cases depending on the values of ω:

Case 1: when$\omega = 1$,it is equivalent to G-S iteration;

Case 2: when$\omega < 1$,it is called underrelaxation;

Case 3: when$\omega > 1$,it is called overrelaxation.

In many cases, $\omega > 1$ will be a better choice.

2.5 Convergence Analysis

The convergence analysis of stationary iterative methods is established through the following fundamental result:

Theorem(Convergence of Stationary Iterations)

For any initial guess $\boldsymbol{x}^{(0)}$, the iteration $\boldsymbol{x}^{(k+1)} = G\boldsymbol{x}^{(k)} + \boldsymbol{b}$ converges to the unique solution if and only if $\rho(G) < 1$, where $\rho(G)$ denotes the spectral radius of the iteration matrix.

3 NUMERICAL RESULTS AND COMMENTS

In this section, we will introduce three performance indicators—number of iterations, final residual, and CPU running time—to compare three different iterative methods. Then, the SOR (Successive Over-Relaxation) iterative method will be selected for a controlled variable test to explore the effects of different relaxation parameters and step sizes on these three performance metrics.

In this section, the one-dimensional discrete Poisson equation and its Dirichlet boundary conditions that we investigate are as follows:

$$\begin{cases} -\dfrac{d^2u}{dx^2} = \sin(\pi x), \ \ 0 < x < 1 \\ u(0) = a, u(1) = b \end{cases},$$

where constant a and b will be explicitly given before each test.

3.1 The Comparison of Three Iterative Methods

Under the same conditions, the three iterative methods were tested respectively, and the following results were obtained.

Firstly, we set $a = 0$, $b = 0$, $n = 80$, $\omega = 1.8$, which we can easily calculate the true solution is $\dfrac{\sin(\pi x)}{\pi^2}$ and execute the corresponding program to get data1:

Table 1 Test data1

Method	Iterations	Final Residual	CPU Time (s)
JACOBI	5761	1.00e-06	0.2574
G-S	3342	1.00e-06	0.1543
SOR	524	9.94e-07	0.0321

Secondly, we set $a=0$, $b=0$, $n=100$, $\omega=1.8$ and execute the corresponding program to get data2:

Table 2 Test data2

Method	Iterations	Final Residual	CPU Time (s)
JACOBI	8046	1.00e-06	0.4516
G-S	4740	1.00e-06	0.2738
SOR	771	9.91e-07	0.0606

We have visualized the results after iterative solving, as shown in the following figure:

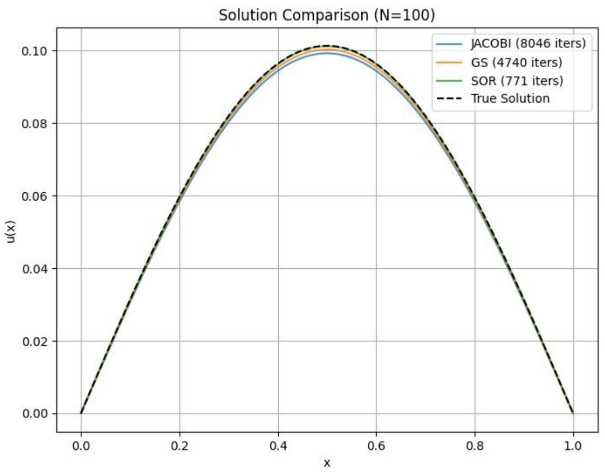

Figure 1 The results after iteration

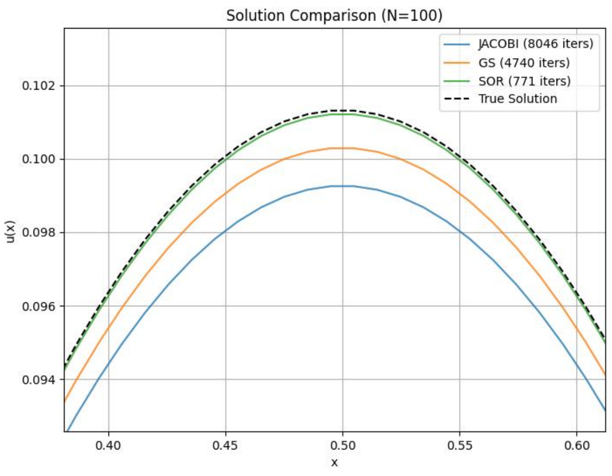

Figure 2 The detail of figure1

Thirdly, we set $a=0$, $b=0$, $n=100$, $\omega=1.5$ and execute the corresponding program to get data3:

Table 3 Test data3

Method	Iterations	Final Residual	CPU Time (s)
JACOBI	8046	1.00e-06	0.4630
G-S	4740	1.00e-06	0.2945
SOR	1957	9.99e-07	0.1614

According to the three test data, the test results clearly show that SOR is the fastest method while keeping the accuracy. Gauss-Seidel performs moderately better than Jacobi but remains much slower than SOR. Jacobi, though simple to use, is the slowest option.

The key finding is that SOR works best when using the right relaxation parameter. But when adjust the relaxation parameter, we find that the performance of SOR start to go bad. Thus we will explore the effect of the relaxation parameters and step sizes to the performance of SOR in the next subsection.

3.2 The Influencing Factors of SOR

3.2.1 Relaxation Parameters

Through abundant experiments, we find the convergence rate of SOR is critically dependent on the relaxation parameter ω. So we assume $a=0$, $b=0$, $n=100$, and execute the corresponding program to get the variation of the result based on different ω.

Table 4 Results on different ω

ω	Iterations	Final Residual	CPU Time (s)
1.00	4740	1.00e-06	0.3224
1.09	4141	1.00e-06	0.2818
1.17	3606	9.99e-07	0.2459
1.26	3122	9.99e-07	0.2138
1.34	2680	1.00e-06	0.1867
1.43	2274	9.99e-07	0.1583
1.51	1896	9.98e-07	0.1358
1.60	1541	9.97e-07	0.1078
1.69	1203	1.00e-06	0.0848
1.77	878	9.95e-07	0.0612
1.86	554	9.97e-07	0.0390
1.94	159	9.87e-07	0.0110
2.03	10000	1.26e+120	0.7013
2.11	3657	0.00e+00	0.2627
2.20	2044	0.00e+00	0.1453

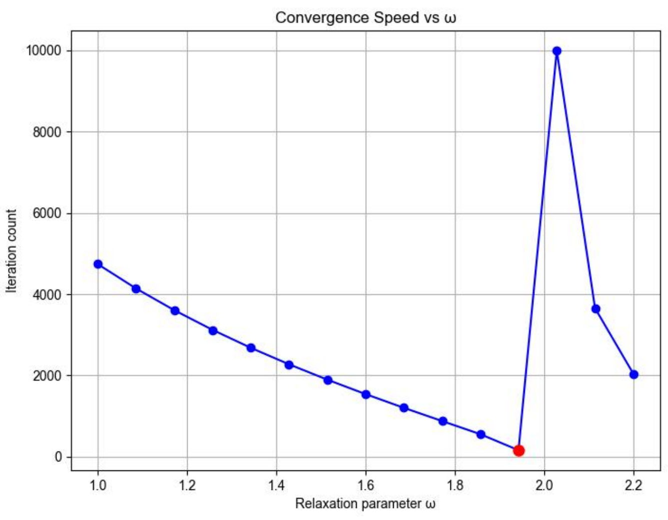

Figure 3 Iterations to different ω

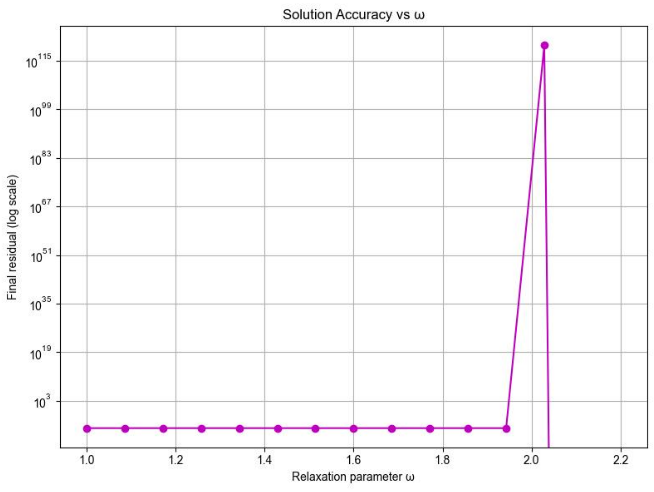

Figure 4 Final Residual to different ω

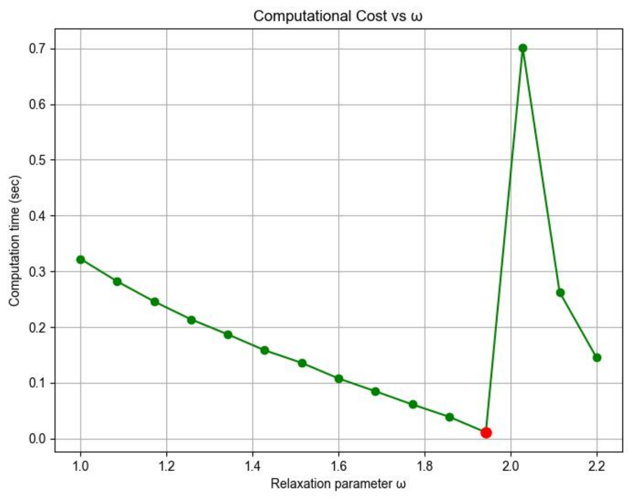

Figure 5 CPU Time to different ω

We can more intuitively discover through visualization that when ω is around 1.9, the optimal relaxation parameter can be obtained, but when it exceeds 2, non-convergence or errors will occur, which is also consistent with the theory that the iteration converges when ω is between 0 and 2. Experiments have shown that the gap between different relaxation parameters is relatively obvious, which inspires us to pay attention to selecting appropriate relaxation parameters when using the SOR iteration method for solving in the future, otherwise undesirable results will be produced.

3.2.2 Step Sizes

Through abundant experiments, we find the convergence rate of SOR is related to the step sizes. So we assume $a = 0$, $b = 0$, $n = 100$, $\omega = 1.8$, and execute the corresponding program to get the variation of the result based on different n.

Table 5 Test data3

n	h	Iterations	Final Residual	CPU Time (s)
60	0.01639	315	9.92e-07	0.0131
70	0.01408	415	9.85e-07	0.0210
80	0.01235	524	9.94e-07	0.0288
90	0.01099	643	9.93e-07	0.0395
100	0.00990	771	9.91e-07	0.0537
110	0.00901	907	9.94e-07	0.0687

120	0.00826	1051	9.96e-07	0.0862
150	0.00662	1528	9.98e-07	0.1678

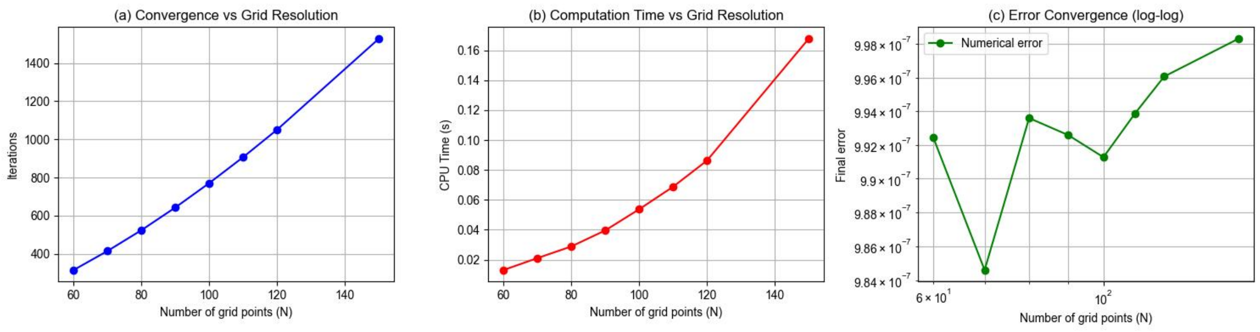

Figure 6 Iterations, Final Residual, CPU Time to different n

The test results show that as the grid is added (h decreases), the number of iterations and calculation time of the SOR method increase significantly, but the accuracy remains stable. Specifically, when h decreases from 0.016 to 0.006, the number of iterations increases by about 5 times, and the calculation time increases by about 13 times. This indicates that under the premise of ensuring calculation accuracy, it is necessary to reasonably select the grid step size to balance the calculation efficiency. It is recommended to choose $h \approx 0.01$ (n=100) to achieve a good balance between accuracy and efficiency.

4 CONCLUSIONS AND FUTURE WORK

4.1 Conclusions

4.1.1 Method Comparison

The SOR method demonstrates superior performance, requiring significantly fewer iterations and computation time than Jacobi and Gauss-Seidel while maintaining comparable accuracy. Gauss-Seidel shows moderate improvement over Jacobi but remains substantially slower than optimally tuned SOR.

4.1.2 Parameter Sensitivity

SOR performance critically depends on proper parameter selection:

The optimal relaxation parameter $\omega \approx 1.9$ delivers 30x speedup over suboptimal choices. Step size reduction increases iterations and computation time while preserving accuracy. Practical use requires balancing $h \approx 0.01$ for efficiency/accuracy trade-offs.

These findings confirm SOR as the preferred method for one-dimensional Poisson problem with Dirichlet boundaries situation when parameters are properly configured.

4.2 Future Work

Further research could explore: (1) extending the analysis to 2D/3D Poisson equations with complex boundary conditions, (2) developing adaptive SOR algorithms that automatically optimize relaxation parameters during computation, and (3) comparing SSOR Iteration, AOR

Iteration, Richardson Iteration and Block Iteration, etc. These advancements would enhance the method's efficiency and applicability to large-scale scientific computing problems.

REFERENCES

[1]Jin, X., & Wei, Y. Numerical Linear Algebra and Its Applications. In Series in Information and Computational Science-Collector's Edition (Vol. 35). Science Press.

[2]Burden, R. L., & Faires, J. D. (2010). Numerical analysis (9th ed.). Cengage Learning.

[3]Golub, G. H., & Van Loan, C. F. (2013). Matrix computations (4th ed.). Johns Hopkins University Press.

[4]Kelley, C. T. (1995). Iterative methods for linear and nonlinear equations. SIAM.

[5]Morton, K. W., & Mayers, D. F. (2005). Numerical solution of partial differential equations (2nd ed.). Cambridge University Press.

APPENDICES

Program for Comparing Three Iterative Methods:

```
import numpy as np

import matplotlib.pyplot as plt

import time


def solve_poisson_1d(f, a, b, N, method='jacobi', omega=1.0,

max_iter=10000, tol=1e-6):

#  求解一维泊松方程  -u'' = f(x), u(0)=a, u(1)=b

    h = 1.0 / (N + 1)

    x = np.linspace(0, 1, N+2)

    u = np.zeros(N+2)

    u[0], u[-1] = a, b

    residuals = []


    rhs = h**2 * f(x[1:-1])

    rhs[0] += a

    rhs[-1] += b


    start_time = time.time()
```

```
for k in range(max_iter):
    u_old = u.copy()
    residual = 0.0

    for i in range(1, N+1):
        if method == 'jacobi':
            u[i] = 0.5 * (u_old[i-1] + u_old[i+1] + rhs[i-1])
        elif method == 'gs':
            u[i] = 0.5 * (u[i-1] + u_old[i+1] + rhs[i-1])
        elif method == 'sor':
            gs_update = 0.5 * (u[i-1] + u_old[i+1] + rhs[i-1])
            u[i] = u_old[i] + omega * (gs_update - u_old[i])

        residual = max(residual, abs(u[i] - u_old[i]))

    residuals.append(residual)
    if residual < tol:
        break

elapsed_time = time.time() - start_time
return x, u, residuals, elapsed_time
```

```
# 测试案例
def f(x):
    return np.sin(np.pi * x)

# 参数设置
a, b = 0, 0
N = 100
methods = ['jacobi', 'gs', 'sor']
omega = 1.8  # SOR 松弛因子

# 结果表格
print(f"{'Method':<10} | {'Iterations':>10} | {'Final Residual':>15} |
{'CPU Time (s)':>12}")
print("-" * 55)

# 求解并记录结果
results = {}
for method in methods:
    x, u, res, time_used = solve_poisson_1d(f, a, b, N, method=method,
omega=omega)
    results[method] = {
        'iterations': len(res),
            'residual': res[-1],
            'time': time_used
        }
        print(f"{method.upper():<10} | {len(res):>10} | {res[-1]:>15.2e} |
{time_used:>12.4f}")

# 可视化
plt.figure(figsize=(8, 6))
for method in methods:
    x, u, res, _ = solve_poisson_1d(f, a, b, N, method=method,
omega=omega)
    plt.plot(x, u, label=f'{method.upper()} ({len(res)} iters)', alpha=0.8)

# 真解
true_sol = np.sin(np.pi * x) / (np.pi**2)
plt.plot(x, true_sol, 'k--', label='True Solution')
plt.xlabel('x')
plt.ylabel('u(x)')
plt.title(f'Solution Comparison (N={N})')
plt.legend()
plt.grid(True)
plt.show()
```