Resolution of a 2D Poisson problem

Final Project for the course of Numerical Analysis

Nahid JAVADINARAB
Mohammad Amin Ghavami
Boaz Tamire Habte

Advisor: Dr. H. Beriot

University of Luxembourg

January 30, 2025

Contents

1 Introduction

Partial Differential Equations (PDEs) are fundamental to modeling a myriad of physical phenomena, ranging from fluid dynamics and electromagnetic fields to heat conduction and structural analysis. Among these, the **Poisson equation** stands out as a pivotal elliptic PDE, widely employed in diverse applications such as electrostatics, mechanical engineering, and gravitational studies. Formally, the Poisson equation in two dimensions is expressed as:

$$\Delta u(x, y) = f(x, y) \quad \text{in} \quad \Omega \tag{1}$$

where Δ denotes the Laplacian operator, $u(x, y)$ is the unknown function to be determined, $f(x, y)$ represents a source term, and Ω defines the spatial domain under consideration. Solving the Poisson equation accurately and efficiently is crucial for predicting system behaviors and informing design decisions across various scientific and engineering disciplines.

This project is dedicated to developing and analyzing numerical solvers for the two-dimensional Poisson equation using finite difference methods. Finite difference schemes are a cornerstone of numerical analysis, offering a straightforward approach to discretizing continuous PDEs by approximating derivatives with differences between function values at discrete grid points. The focus of this project encompasses both **second-order** and **fourth-order** finite difference schemes, each offering distinct advantages in terms of accuracy and computational complexity.

1.1 Objectives

The primary objectives of this project are as follows:

1. **Matrix Assembly:** Construct the discrete Laplacian matrix using second-order and fourth-order finite difference approximations. This involves translating the continuous PDE into a linear system of equations that can be solved numerically.

2. **Solver Implementation:** Implement a suite of solvers, including **direct methods** (e.g., Gaussian elimination) and **iterative methods** (e.g., Jacobi, Gauss-Seidel, Successive Over-Relaxation [SOR]). Emphasis is placed on leveraging **sparse matrix representations** to enhance computational efficiency and scalability.

3. **Convergence Analysis:** Conduct comprehensive convergence studies to evaluate the accuracy and performance of the implemented solvers. This includes assessing how the numerical solutions approach the exact analytical solutions as the grid is refined.

4. **Optimization of Iterative Methods:** Explore the impact of relaxation parameters in iterative solvers, particularly within the SOR method, to identify optimal configurations that minimize computational time while maintaining solution accuracy.

1.2 Significance

The Poisson equation serves as a benchmark problem in numerical analysis due to its ubiquity and the insights it provides into the behavior of more complex PDEs. By developing robust and efficient numerical solvers for the Poisson equation, this project contributes to the broader field of computational mathematics and engineering. The comparative analysis between different finite difference schemes and solver strategies offers valuable guidance for selecting appropriate numerical methods tailored to specific application requirements and computational resources.

2 Finite difference discretization

2.1 Designing the solver

2.1.1 Simplification with a Uniform Grid

When the grid is uniform ($h_x = h_y = h$), the discrete Laplacian:

$$\Delta_h u_{i,j} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2} \tag{2}$$

simplifies to:

$$\Delta_h u_{i,j} = \frac{(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}) + (u_{i,j+1} - 2u_{i,j} + u_{i,j-1})}{h^2} \tag{3}$$

which further simplifies to:

$$\Delta_h u_{i,j} = \frac{u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j}}{h^2}. \tag{4}$$

This leads to a symmetric stencil centered at $u_{i,j}$ with contributions from its four immediate neighbors.

2.1.2 Incorporating Boundary Conditions into the Linear System

The linear system can be expressed as:

$$Au = b + f \tag{5}$$

For Dirichlet boundary conditions, we enforce $u = g$ on the boundary, modifying the system as follows:

- If a boundary value ($u_{i,j}$) is known (i.e., it lies on $\partial\Omega$), we do not solve for it.
- Instead, its effect is moved to the right-hand side.

For an interior point adjacent to a boundary, say (i, j), if one of its neighbors is a boundary point, we replace (u_{boundary}) by its known value (g):

$$-4u_{i,j} + u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} = h^2 f_{i,j}. \tag{6}$$

If, for example, ($u_{i-1,j}$) is a boundary point with value ($g_{i-1,j}$), we move it to the right-hand side:

$$-4u_{i,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} = h^2 f_{i,j} - g_{i-1,j}. \tag{7}$$

Repeating this for all boundary neighbors, we define:

$$b_{i,j} = - \sum_{\text{boundary neighbors}} g_{\text{boundary}}. \tag{8}$$

Thus, the modified system of (5) becomes:

- (b) accounts for the known boundary values.
- (f) remains the source term.

This ensures that the boundary conditions are correctly incorporated into the numerical solution.

2.1.3 Non-Zero Boundary Conditions

To verify if the exact solution is consistent with the boundary conditions, the following Python code is used:

```python
import numpy as np
X, Y = np.meshgrid(np.linspace(0, 1, 20), np.linspace(0, 1, 20))
u_exact = (np.sin(np.pi * X))**2 * (np.sin(np.pi * Y))**2
u_b = np.zeros_like(u_exact)
u_b[0, :], u_b[-1, :], u_b[:, 0], u_b[:, -1] = 0, 0, 0, 0  # Apply boundary condition
u_test = u_exact.copy()
u_test[1:-1, 1:-1] = 0
if np.max(abs(u_b - u_test)) < 1e-20:
    print('The exact solution is consistent with the boundary conditions')
else:
    print("The exact solution is not consistent with the boundary conditions!")
```

2.2 Validation of the implementation

To validate the implementation, we define an exact solution:

$$u_{\text{ex}}(x,y) = \sin^2(\pi x)\sin^2(\pi y). \tag{9}$$

2.3 Finding the Right-Hand Side $f(x,y)$

Taking the Laplacian of $u_{\text{ex}}(x,y)$:

$$\Delta u_{\text{ex}} = \frac{\partial^2 u_{\text{ex}}}{\partial x^2} + \frac{\partial^2 u_{\text{ex}}}{\partial y^2}. \tag{10}$$

Computing the partial derivatives:
1. First derivative with respect to x:

$$\frac{\partial u_{\text{ex}}}{\partial x} = 2\sin(\pi x)\cos(\pi x)\sin^2(\pi y). \tag{11}$$

2. Second derivative with respect to x:

$$\frac{\partial^2 u_{\text{ex}}}{\partial x^2} = 2\pi^2\cos(2\pi x)\sin^2(\pi y). \tag{12}$$

3. Similarly, for y:

$$\frac{\partial^2 u_{\text{ex}}}{\partial y^2} = 2\pi^2\cos(2\pi y)\sin^2(\pi x). \tag{13}$$

Thus, the right-hand side function is:

$$f(x,y) = -\Delta u_{\text{ex}} = -2\pi^2\left(\cos(2\pi x)\sin^2(\pi y) + \cos(2\pi y)\sin^2(\pi x)\right). \tag{14}$$

2.3.1 Consistency with Boundary Conditions

The solution $u_{\text{ex}}(x,y)$ satisfies the homogeneous Dirichlet boundary conditions:

$$u(x,0) = u(x,1) = u(0,y) = u(1,y) = 0. \tag{15}$$

This confirms that the exact solution is consistent with the boundary conditions.

2.4 Numerical Validation

2.4.1 Relative Error in the Maximum Norm

The relative error is computed as:

$$\text{Error} = \max_{i,j}\left|\frac{u_{\text{num}}(x_i,y_j) - u_{\text{ex}}(x_i,y_j)}{u_{\text{ex}}(x_i,y_j)+\epsilon}\right|, \tag{16}$$

where $\epsilon = 10^{-12}$ prevents division by zero. The numerical results confirm that the error systematically decreases as the grid is refined.

2.4.2 Expected Convergence Rate

For a second-order finite difference method, the expected convergence rate is **$O(h^2)$**, meaning the error should decrease proportionally to h^2 when refining the grid.

The log-log plot of the error vs. step size h confirms this, with the slope of the error curve being close to 2, validating the expected theoretical convergence rate.

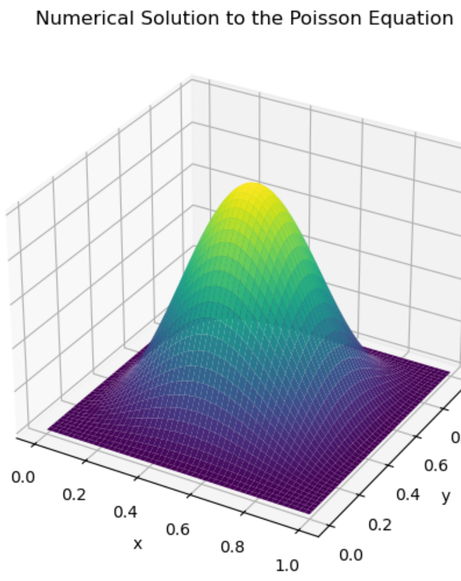
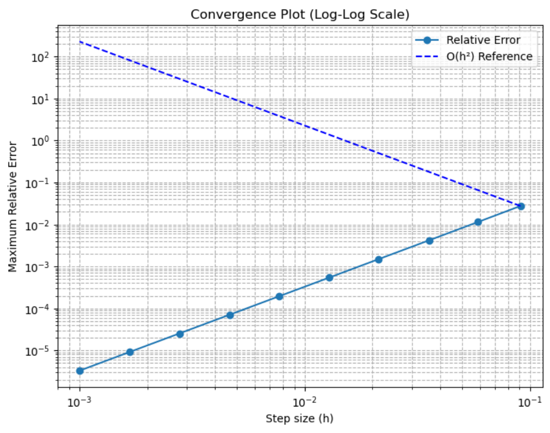
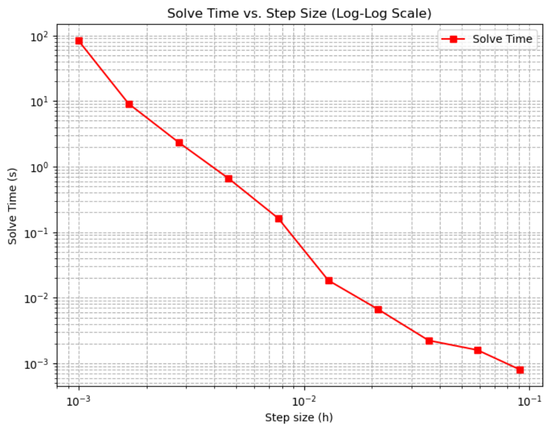


Figure 1: Numerical Solution to the Poisson Equation.



(a) Convergence Plot (Log-Log Scale) of Maximum Relative Error vs. Step Size.



(b) Solve Time vs. Step Size (Log-Log Scale).

Figure 2: Comparison of convergence and solve time as a function of step size.

2.5 Computational Complexity and Solve Time Analysis

- The solve time increases exponentially as the grid is refined, indicating that computational cost grows significantly for larger grids.
- The log-log plot of solve time vs. step size follows a nearly linear relationship, confirming that solving the system becomes more expensive as the number of grid points increases.
- Since the problem requires solving a large sparse linear system, direct solvers become inefficient for fine grids. More efficient iterative solvers, such as Multigrid or Preconditioned Conjugate Gradient (PCG), are recommended for large-scale problems.

We can conclude:

- The finite difference method is correctly implemented and satisfies boundary conditions.
- The expected second-order convergence rate $O(h^2)$ is verified.
- The relative error decreases systematically as the grid resolution increases.
- The computational cost grows significantly for fine grids, necessitating more efficient solvers for large-scale problems.

This confirms that the method is both numerically accurate and computationally demanding for very fine grids.

3 Solving the linear system

The numerical solution of the Poisson equation via finite difference methods results in a system of linear equations of the form:

$$\mathbf{A}\mathbf{u} = \mathbf{b}, \tag{17}$$

where $\mathbf{A}$ is the discrete Laplacian matrix, $\mathbf{u}$ is the vector of unknown solution values at the interior grid points, and $\mathbf{b}$ is the source term vector incorporating both the discretized source function and boundary conditions. Efficiently solving this linear system is crucial for obtaining accurate and timely numerical solutions.

3.1 Direct methods

3.1.1 Dense vs. Sparse Matrix Representation

In the initial implementation, the discrete Laplacian matrix $\mathbf{A}$ was represented as a **dense matrix**, where all entries, including zeros, are explicitly stored in memory. While this approach is straightforward and simple to implement, it becomes computationally and memory-wise inefficient as the size of the grid increases. Specifically, for a grid with $N_x \times N_y$ interior points, the dense matrix $\mathbf{A}$ has dimensions $(N_xN_y) \times (N_xN_y)$, leading to a memory requirement that scales quadratically with the number of grid points.

However, the discrete Laplacian matrix inherently possesses a sparsity structure; each row contains a limited number of non-zero entries corresponding to the interactions between a grid point and its immediate neighbors. Exploiting this sparsity can lead to significant improvements in both memory usage and computational efficiency.

3.1.2 Sparse Matrix Representation

To capitalize on the sparsity of $\mathbf{A}$, we adopt a sparse matrix representation, which stores only the non-zero entries of the matrix. This approach drastically reduces memory consumption and accelerates matrix operations by ignoring the zero elements.

$$\Delta_h = \mathbf{I} \otimes \mathbf{T} + \mathbf{T} \otimes \mathbf{I}, \tag{18}$$

where:

- $\mathbf{I}$ is the identity matrix of size $N \times N$, corresponding to the number of interior points in one spatial dimension.
- $\mathbf{T}$ is a tridiagonal matrix representing the one-dimensional discrete Laplacian.

The Kronecker product formulation efficiently constructs the two-dimensional discrete Laplacian by combining the one-dimensional operators in each spatial direction.

3.1.3 Advantages of Sparse Representation

Adopting a sparse matrix representation offers several significant advantages:

1. **Memory Efficiency:** Only the non-zero elements of $\mathbf{A}$ are stored, leading to substantial memory savings. For large grids, this reduction is crucial, as storing a dense matrix becomes infeasible.
2. **Computational Efficiency:** Operations such as matrix-vector multiplications and linear system solvers are expedited since computations involving zero elements are omitted.
3. **Scalability:** Sparse representations enable the handling of much larger systems, making them suitable for high-resolution simulations and three-dimensional problems.

The operation counts are provided in Table 1.

Table 1: Comparison of Operation Counts between Dense and Tridiagonal Sparse Matrices

Method	**Dense Matrix**	**Tridiagonal Sparse Matrix**
Gaussian Elimination	$\mathcal{O}(N^3)$	$\mathcal{O}(N)$
Matrix-Vector Multiplication	$\mathcal{O}(N^2)$	$\mathcal{O}(N)$
Memory Usage	$\mathcal{O}(N^2)$	$\mathcal{O}(N)$

Table 1 illustrates the stark contrast in computational complexity between dense and tridiagonal sparse matrices. While dense matrices incur a cubic and quadratic scaling for Gaussian elimination and matrix-vector

multiplication respectively, tridiagonal sparse matrices benefit from linear scaling in both operations. This significant reduction in computational effort underscores the efficiency of sparse matrix representations, particularly for large-scale problems where N is substantial.

Table 2: Comparison of Computational Time and Relative Error between Dense and Sparse Direct Solvers

Grid Size	Dense Solver Time(s)	Sparse Solver Time(s)	Dense Error	Sparse Error
10x10	0.0033	0.0005	$1.00849 \times 10^{+00}$	$1.00849 \times 10^{+00}$
20x20	0.0061	0.0018	$1.00228 \times 10^{+00}$	$1.00228 \times 10^{+00}$
40x40	0.0767	0.0074	$1.00060 \times 10^{+00}$	$1.00060 \times 10^{+00}$
80x80	3.7698	0.0413	$1.00015 \times 10^{+00}$	$1.00015 \times 10^{+00}$

Table 2 presents the computational time and relative error for both dense and sparse direct solvers across different grid sizes. The results demonstrate that the sparse solver consistently outperforms the dense solver in terms of computational time, while maintaining identical relative errors. Specifically, for a grid size of 80×80, the dense solver requires approximately 3.77 seconds, whereas the sparse solver completes the task in just 0.0413 seconds. This stark contrast highlights the efficiency gains achieved through sparse matrix representations.

The transition from dense to sparse matrix representations in direct solvers for the Poisson equation yields substantial benefits in both computational time and memory usage. Sparse matrices not only facilitate the handling of larger systems by drastically reducing memory consumption but also accelerate computations through reduced operation counts. This efficiency is particularly evident in operations like Gaussian elimination and matrix-vector multiplications, where sparse representations transform computational complexities from cubic and quadratic to linear scaling. Consequently, adopting sparse matrix techniques is indispensable for achieving computational efficiency and scalability in numerical PDE solving, especially for high-resolution and large-scale applications.

3.2 Iterative Methods

Eigenvalues of the Discrete Laplacian in 2D

Understanding the eigenvalues of the discrete Laplacian matrix is crucial for analyzing the convergence properties of iterative solvers. The spectral characteristics of the matrix influence the convergence rates and stability of methods such as Jacobi, Gauss-Seidel, and SOR.

Eigenvalues of Matrix T in 1D Consider the one-dimensional discrete Laplacian matrix $\mathbf{T}$ of size $N \times N$ with Dirichlet boundary conditions, defined as:

$$\mathbf{T} = \begin{bmatrix} 4 & -1 & 0 & \cdots & 0 \\ -1 & 4 & -1 & \ddots & 0 \\ 0 & -1 & 4 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & 0 & -1 & 4 \end{bmatrix} \tag{19}$$

To determine the eigenvalues of $\mathbf{T}$, we seek solutions to the eigenvalue equation $\mathbf{Tv} = \lambda \mathbf{v}$. Assuming eigenvectors of the form:

$$v_k(i) = \sin\left(\frac{\pi k i}{N+1}\right), \quad k = 1, 2, \ldots, N \tag{20}$$

where $i = 1, 2, \ldots, N$, we substitute into the eigenvalue equation and utilize recurrence relations and properties of Chebyshev polynomials to derive the eigenvalues:

$$\lambda_k = 2 - 2\cos\left(\frac{\pi k}{N+1}\right), \quad k = 1, 2, \ldots, N \tag{21}$$

This expression demonstrates that all eigenvalues of $\mathbf{T}$ are positive, confirming that $\mathbf{T}$ is a symmetric positive definite matrix.

Eigenvalues of the 2D Discrete Laplacian Using Kronecker Products Leveraging the properties of the Kronecker product, the eigenvalues of Δ_h are the sums of the eigenvalues of the individual Kronecker factors. If λ_i and λ_j are the eigenvalues of $\mathbf{T}$, then the eigenvalues $\lambda_{i,j}$ of Δ_h are given by:

$$\lambda_{i,j} = \lambda_i + \lambda_j = \left[2 - 2\cos\left(\frac{\pi i}{N_x+1}\right)\right] + \left[2 - 2\cos\left(\frac{\pi j}{N_y+1}\right)\right], \quad i = 1, 2, \ldots, N_x; \quad j = 1, 2, \ldots, N_y \tag{22}$$

3.2.1 Jacobi and Gauss-Seidel Methods

This subsection details the implementation and validation of the Jacobi and Gauss-Seidel iterative methods for solving the linear system $\mathbf{Au} = \mathbf{b}$, where $\mathbf{A}$ is the discrete Laplacian matrix.

Jacobi Method The Jacobi method updates each component of the solution vector based on the values from the previous iteration. The update rule for the i-th component is:

$$u_i^{(k+1)} = \frac{1}{A_{ii}}\left(b_i - \sum_{j\neq i} A_{ij} u_j^{(k)}\right), \quad \forall i = 1, 2, \ldots, N \tag{23}$$

Gauss-Seidel Method The Gauss-Seidel method improves upon the Jacobi method by utilizing the most recent updates within the same iteration. The update rule is:

$$u_i^{(k+1)} = \frac{1}{A_{ii}}\left(b_i - \sum_{j<i} A_{ij} u_j^{(k+1)} - \sum_{j>i} A_{ij} u_j^{(k)}\right), \quad \forall i = 1, 2, \ldots, N \tag{24}$$

Validation of Implementation Both the Jacobi and Gauss-Seidel methods were implemented and tested on a small example. The obtained results are:

- Jacobi Method: 63 iterations, relative error = 1.04571
- Gauss-Seidel Method: 33 iterations, relative error = 1.04571
- When increasing the complexity, both methods failed to converge within 1000 iterations, with final relative errors of approximately 1.00057 (Jacobi) and 1.00059 (Gauss-Seidel).

To validate the implementation, solutions were compared against the exact solution using contour plots. The results indicate:

- The Jacobi method converges more slowly, with larger residuals.
- The Gauss-Seidel method converges faster but still has some errors.

These observations align with theoretical expectations, where Gauss-Seidel generally converges faster due to its sequential update mechanism.

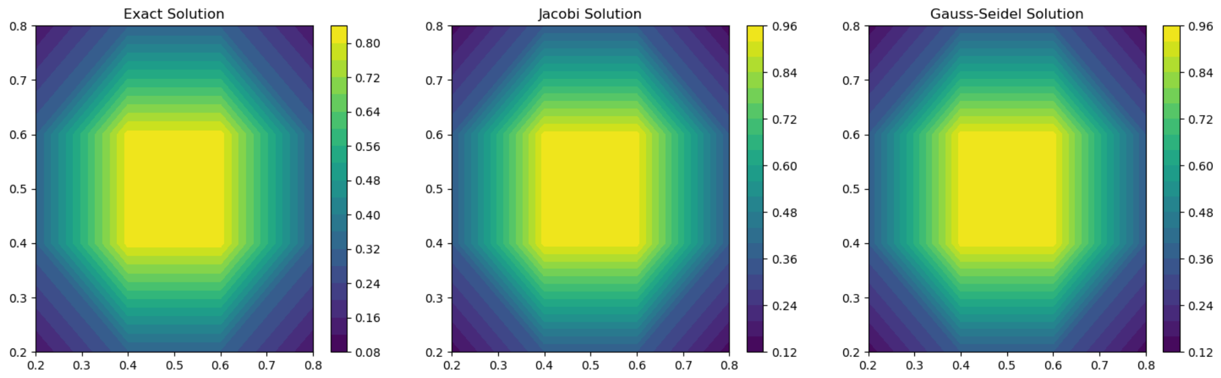


Figure 3: Comparison of Exact, Jacobi, and Gauss-Seidel solutions.

Figure 3 compares the exact solution with the numerical solutions obtained from both methods, showing that while both approximate the exact solution, the Gauss-Seidel method provides a closer match due to its faster convergence.

Convergence Criteria The iteration matrices for Jacobi and Gauss-Seidel methods are:

$$T_J = D^{-1}(L+U) \quad \text{(Jacobi)} \tag{25}$$
$$T_{GS} = (D+L)^{-1}U \quad \text{(Gauss-Seidel)} \tag{26}$$

where D is the diagonal, L is the lower, and U is the upper triangular part of A. Convergence occurs if the spectral radius satisfies:

$$\rho(T) < 1. \tag{27}$$

For the Poisson problem:

- Jacobi method has a larger $\rho(T)$, leading to slower convergence.
- Gauss-Seidel method has a smaller $\rho(T)$, explaining its faster convergence.

Effect of Grid Refinement As the number of grid points increases:

- The matrix size increases.
- The condition number of A worsens.
- The spectral radius $\rho(T)$ increases, slowing convergence.

For large grids, iterative solvers like Jacobi and Gauss-Seidel become inefficient, and methods such as Successive Over-Relaxation (SOR) or Multigrid are preferred.

Measuring the Cost of an Iterative Solver The cost of an iterative solver is determined by:

Number of Iterations

- More iterations are required if $\rho(T)$ is close to 1.
- Jacobi takes more iterations than Gauss-Seidel due to its higher $\rho(T)$.

Cost per Iteration Each iteration requires:

- Matrix-vector multiplication, with complexity $O(n^2)$ for dense matrices but $O(n)$ for sparse Poisson matrices.
- Memory access: Jacobi allows parallelization, but Gauss-Seidel has sequential dependencies.

Thus, Jacobi is computationally cheaper per iteration, but requires more iterations.

3.2.2 Successive Over-Relaxation (SOR) Method

The Successive Over-Relaxation (SOR) method enhances the Gauss-Seidel approach by introducing a relaxation parameter ω to potentially accelerate convergence.

The SOR formula is given by:

$$u_i^{(k+1)} = (1-\omega)u_i^{(k)} + \frac{\omega}{A_{ii}}\left(b_i - \sum_{j<i} A_{ij}u_j^{(k+1)} - \sum_{j>i} A_{ij}u_j^{(k)}\right) \tag{28}$$

Relaxation Parameter ω

- **Range:** $0 < \omega < 2$ ensures convergence.
- **Optimal Value:** The value of ω that minimizes the number of iterations required for convergence.
- **Effects:**
  - $\omega = 1$: Reduces to the Gauss-Seidel method.
  - $\omega > 1$: Over-relaxation; can lead to faster convergence.
  - $\omega < 1$: Under-relaxation; can dampen oscillations but may slow convergence.

Optimal Relaxation Parameter for SOR The Successive Over-Relaxation (SOR) method was implemented to solve the discrete Poisson problem. The goal was to determine the optimal relaxation parameter ω that minimizes the number of iterations required to reach a given residual threshold.

- For small values of ω, particularly $\omega \leq 1.2$, the SOR method does not converge within 1000 iterations.
- The optimal value of ω was found to be 1.85, requiring only 100 iterations to converge.
- Beyond $\omega = 1.85$, the number of iterations begins to increase, indicating that excessive relaxation can lead to instability.

Theoretically, the optimal ω for SOR can be predicted using spectral analysis of the iteration matrix. For a discretized Laplacian, an approximation for the optimal ω is given by:

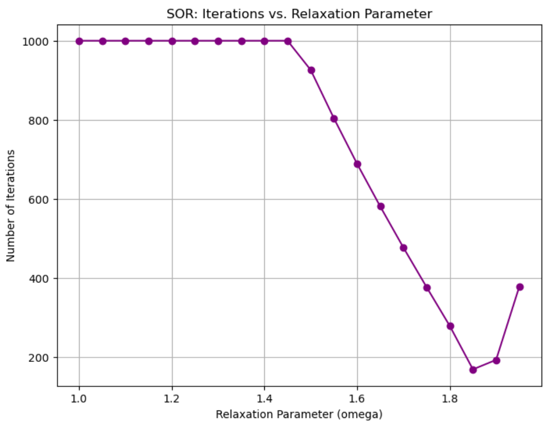
$$\omega_{opt} \approx \frac{2}{1 + \sqrt{1 - \rho(T_{GS})^2}} \tag{29}$$

where $\rho(T_{GS})$ is the spectral radius of the Gauss-Seidel iteration matrix. This explains why an optimal ω exists and provides an estimate for its value.

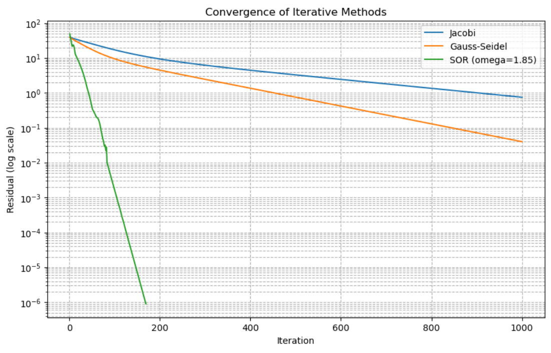
3.2.3 Comparison of Iterative Methods

The computational performance of Jacobi, Gauss-Seidel, and SOR methods was analyzed by plotting the decrease in residual over iterations. The key observations are:

- **Jacobi** requires the highest number of iterations, making it the slowest method.
- **Gauss-Seidel** improves convergence speed but remains slower than SOR.
- **SOR** with $\omega = 1.85$ significantly accelerates convergence, requiring fewer iterations than both Jacobi and Gauss-Seidel.



(a) Number of iterations vs. relaxation parameter ω for SOR.



(b) Convergency of Jacobi, Gauss-Seidel, and SOR methods.

Figure 4a illustrates the number of iterations required for the SOR method to converge as a function of the relaxation parameter ω. It shows that for small values of ω, particularly $\omega \leq 1.2$, the method does not converge

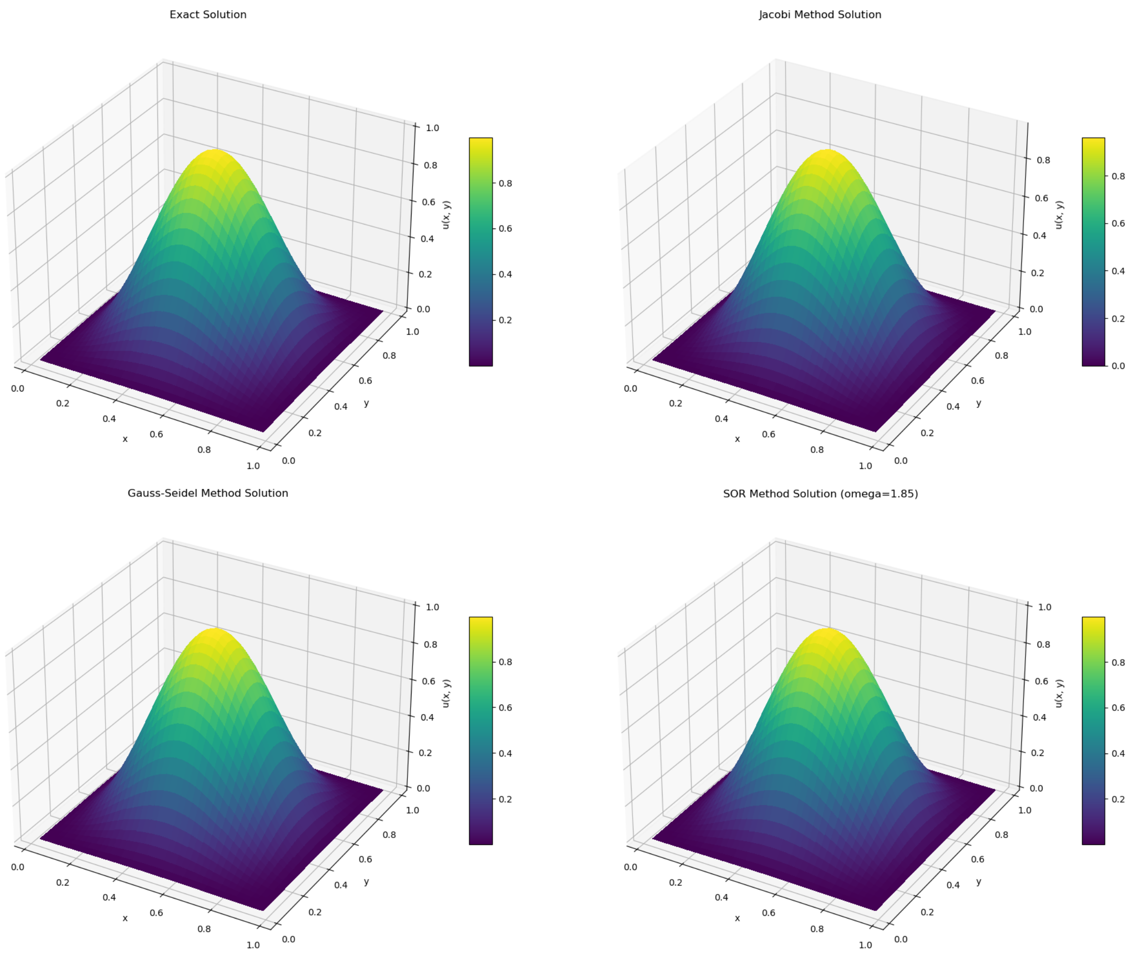


Figure 5: Comparison of Exact, Jacobi, Gauss-Seidel, and SOR solutions.

within the maximum allowed iterations. The optimal value is found at $\omega = 1.85$, where the number of iterations is minimized, and beyond this value, instability leads to an increasing number of iterations.

Figure 4b provides a direct comparison of residual reduction across the three iterative methods. It highlights that while Jacobi and Gauss-Seidel exhibit gradual convergence, SOR achieves an exponential-like decrease in residual, making it the most efficient method among the three The analysis of the SOR method demonstrates the importance of selecting an optimal relaxation parameter.
The results confirm that:

- SOR is the most efficient method among the three, provided an optimal ω is chosen.
- Choosing an optimal ω can significantly reduce computation time and iterations.
- Excessively high values of ω can lead to instability and increased iteration counts.

Figure 6 compares the exact solution of the discrete Poisson problem with numerical solutions obtained using Jacobi, Gauss-Seidel, and SOR methods. The results indicate that while all three methods approximate the exact solution, SOR provides the most accurate representation in the fewest iterations due to its accelerated convergence.

4 Extensions to the solver

4.1 Motivation for Higher-Order Schemes

The second-order finite difference scheme approximates derivatives with a truncation error of $O(h^2)$ While sufficiently accurate for many applications, certain problems may demand higher precision, especially when dealing with smooth solutions or when minimizing numerical artifacts is crucial. Implementing a higher-order finite difference, in this case, fourth-order, will reduce the error (in fourth-order $O(h4)$) and provide more accurate derivative approximations. In addition, a higher-order formula better captures rapid changes in the solution without requiring excessive grids. However, increasing stencil width leads to more complex matrix structures and slightly higher computational costs per iteration.

4.2 Implementing a higher order finite difference formula

The standard second-order central difference for the second derivative is:

$$\frac{d^2u}{dx^2} \approx \frac{u_{i-1} - 2u_i + u_{i+1}}{h^2} + \mathcal{O}(h^2)$$

To achieve fourth-order accuracy, incorporate additional points:

$$\frac{d^2u}{dx^2} \approx \frac{-u_{i-2} + 16u_{i-1} - 30u_i + 16u_{i+1} - u_{i+2}}{12h^2} + \mathcal{O}(h^4)$$

Applying the fourth-order central difference in both x and y directions:

$$\Delta_h u_{i,j} = \frac{-u_{i-2,j} + 16u_{i-1,j} - 30u_{i,j} + 16u_{i+1,j} - u_{i+2,j}}{12h_x^2} + \frac{-u_{i,j-2} + 16u_{i,j-1} - 30u_{i,j} + 16u_{i,j+1} - u_{i,j+2}}{12h_y^2}$$

In this project we assumed $h_x = h_y = h$, the discrete Laplacian simplifies to:

$$\Delta_h u_{i,j} = \frac{-u_{i-2,j} + 16u_{i-1,j} - 30u_{i,j} + 16u_{i+1,j} - u_{i+2,j}}{12h^2} + \frac{-u_{i,j-2} + 16u_{i,j-1} - 30u_{i,j} + 16u_{i,j+1} - u_{i,j+2}}{12h^2}$$

To reproduce the previous algorithms, we should consider that higher-order schemes require additional grid points to compute derivatives. Specifically, a fourth-order Laplacian stencil extends two points in each direction, necessitating a wider boundary layer. Also, the extended stencil makes implementing boundary conditions more intricate.

Note: For simplicity, assume that boundary conditions are homogeneous Dirichlet ($u = 0$).

4.3 results

We've investigated the accuracy and convergence rate of the fourth-order finite difference solver by solving the Poisson equation on grids of varying sizes and computing the relative error in the maximum norm.

4.3.1 Accuraccy compare to second-order

Based on the presented files (where Nx, Ny = 100, 100), the second-order solver had a relative error = 3.22567×10^{-04} while the fourth-order solver had a relative error = 7.08125×10^{-05}, which means the error improved 4.555 times and the fourth-order solver is more than 4 times accurate It's important to consider the fact that the second-order solver is more than five times faster than the fourth-order solver (in the mentioned condition)

- Second-Order Sparse Solver: Time = 0.0492s
- Fourth-Order Sparse Solver: Time = 0.2806s

4.3.2 Dense and Sparse Methods Comparison

Dense solver took 10.7813s while **Sparse** solver in 5.8592s calculated the answer. Both methods had the same relative error equal to 7.30943×10^{-02}

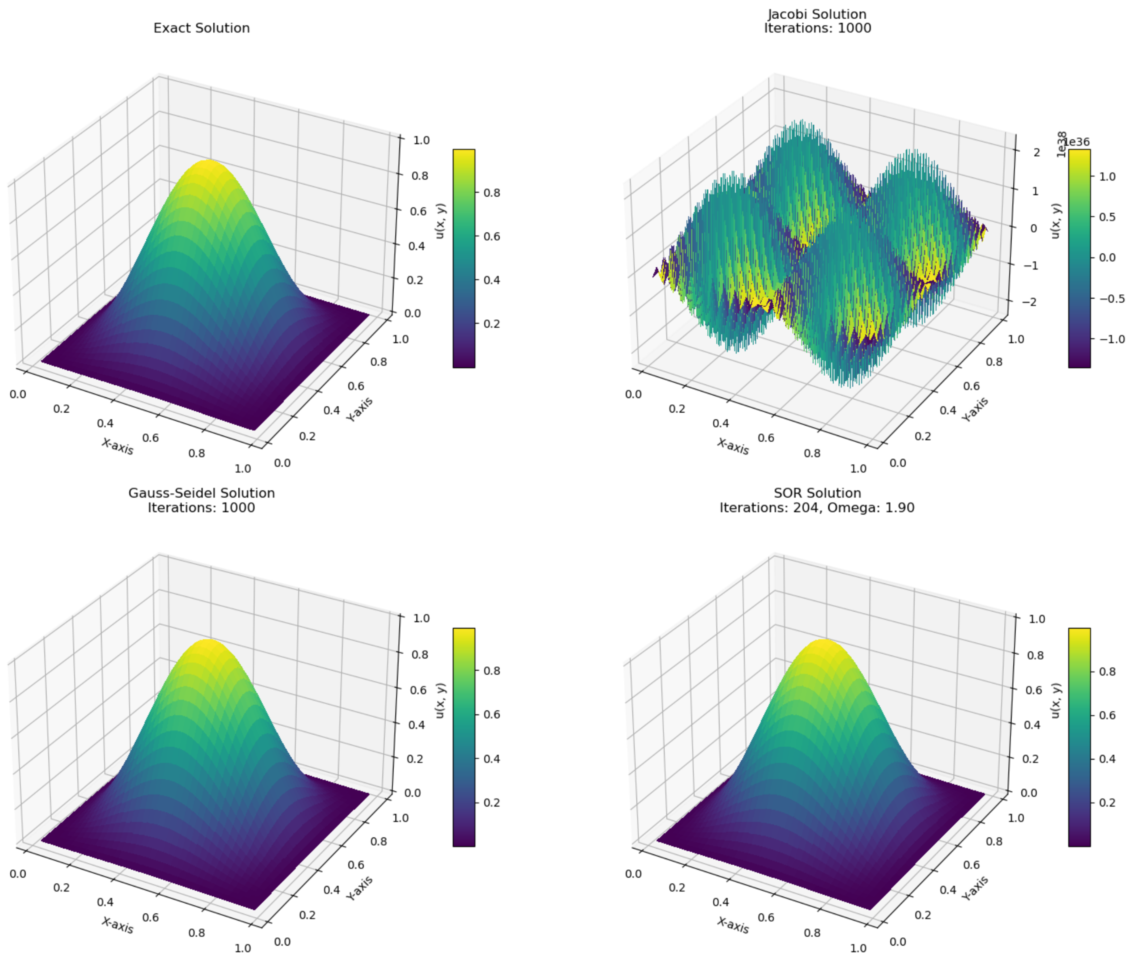


Figure 6: Exact, Jacobi, Gauss-Seidel, and SOR solutions according to fourth-order formula.