3. Conjugate Gradient (CG) Algorithm

The objective of this case study is to implement the Conjugate Gradient algorithm to solve a square linear system Ax = b, where A is a symmetric, positive definite, and regular matrix, and the right-hand side vector b is nonzero.

3.1 Fundamentals: The Poisson Problem

The Poisson problem is a common type of partial differential equation. Its standard form is

$-\Delta u(x) = f(x)$, with Dirichlet boundary conditions.

Intuitively, the problem can be viewed as follows: in a square domain, determine the value at each interior point based on the “instructions” provided by the function f(x), while the function values on the edges of the square are fixed at 0.

3.1.1 Step01: Discretize with Finite Difference Method

Computers cannot directly work with continuous functions or differential equations, so the continuous problem must be converted into a discrete one. In this case, we discretize the entire square region by constructing a grid within the domain, with each grid point representing a discrete location. By dividing the domain into a grid and restricting the function values to the grid points, we approximate the original differential equation with algebraic equations.

1. **Interior and Boundary Nodes:**
   The boundary nodes (i.e., the points where i = 0 or i = N, or j = 0 or j = N) have $u(x) = 0$, which are known. Thus, we only need to solve for the values at the interior nodes (those not on the boundary).
2. **Determining the Grid:**

   On the unit square Ω, we discretize the problem into a regular two-dimensional grid. Divide the unit square into N+1 nodes (including the boundary nodes), where the spacing between points is given by $h = 1/N$.

For example, when N = 8, you divide the square into 9 equally spaced points (from 0 to 1).

3.1.2 Step02: Approximate the Differential Operator Using the Finite Difference Method

Since the continuous second derivatives cannot be computed directly on a computer, we approximate them using the finite difference method. For two-dimensional problems, the standard five-point finite difference scheme is used to approximate the Laplace operator. The five-point difference formula provides a good approximation of the two-dimensional Laplacian, and its simplicity makes it easy to implement in code.

1. **Five-Point Difference Formula:**

   At each interior node (i, j), the values of the function at the four neighboring nodes are used to approximate the Laplace operator. The formula is:

   $$\Delta u(x_{(i,j)}) \approx (u_(i-1,j) + u_(i+1,j) + u_(i,j-1) + u_(i,j+1) - 4u_(i,j))/h^2.$$

   This formula tells us that the second derivative (or “curvature”) at a point can be approximated by the values at that point and its immediate neighbors in the up, down, left, and right directions.

2. **Substituting into the Original Equation:**

   The original partial differential equation is

   $$-\Delta u(x) = f(x).$$

   Multiplying by h², it can be rewritten as:

   $$-u_(i-1,j) - u_(i+1,j) - u_(i,j-1) - u_(i,j+1) + 4u_(i,j) = h^2 f_(i,j).$$

   Here,$f_(i,j)$ represents the value of f(x) at the node(x_i, x_j). In this way, each interior node provides an algebraic equation.

3.1.3 Step03: Construct the Linear System Ay = b

1. **Defining the Unknown Vector y:**
   - The vector y contains the values of u(x) at all the interior nodes, since the

boundary values are already known to be 0 and do not need to be solved. Only the interior nodes need to be solved for, and the boundary values are simply “filled in” as zeros.

- If there are N+1 grid points per dimension, then the number of interior nodes is $(N-1)^2$. That is, y is a vector of length $(N-1)^2$.

2. **Determining the Grid Point Ordering :**

   - Mapping the **Two-Dimensional Points** to a One-Dimensional Vector.
   Common Method: Lexicographic (Row-Wise) Ordering

   For example, map the 2D point (i, j) to a one-dimensional index k using:

   $k = i + (j-1) * (N-1), \qquad \text{for } i, j = 1, \ldots, N-1.$

   With this ordering, we can store the values at each interior node sequentially in the vector y.

3. **Constructing the Matrix A:**

   For each interior node (i, j), the discretized equation includes:

   - A diagonal entry (corresponding to $u(i, j)$) with a coefficient of 4,

   - Off-diagonal entries (corresponding to the neighboring nodes $u(i \pm 1, j)$ and $u(i, j \pm 1)$) with coefficients of -1.

   Since many elements of A are zero (each equation involves only the current node and its neighbors), A is a sparse matrix—a significant computational advantage. This efficiency is crucial for later applying iterative methods like the Conjugate Gradient algorithm to solve the system.

4. **Constructing the Right-Hand Side Vector b:**

   For each interior node (i, j), the right-hand side in the finite difference equation is $h^2 f(i, j)$.
   Thus, each component of the vector b is
   h^2 multiplied by the value of $f(x)$ at the corresponding node.

   If an interior node is adjacent to a boundary, even though the finite difference formula may include values corresponding to boundary nodes, those values are 0 (and thus known) and can be neglected.

3.2 Serial implementation of CG

3.2.1 Implementation of the Conjugate Gradient method

In this serial implementation of the Conjugate Gradient method for solving the 2D Poisson equation, I adopt a matrix-free approach using a 5-point finite difference stencil. To avoid unnecessary computation:

- **Sparse matrix-vector multiplication without storing the full matrix:** Instead of explicitly storing the full matrix A, which would be both memory-intensive and unnecessary due to its sparse structure, the program applies the five-point finite difference stencil directly in the apply_A() function. This approach leverages the known sparsity pattern of the Laplacian operator, significantly reducing both memory usage and computation time.
- **Convergence check based on the residual norm:** To avoid unnecessary iterations, the algorithm checks for convergence by monitoring the norm of the residual vector. As soon as the residual drops below a given tolerance, the iteration stops immediately. This ensures computational efficiency by preventing redundant calculations once an acceptable solution is reached.
- **Small random perturbation for initial guess:** The initial solution vector is not set to zero, but instead initialized with a small random perturbation. This is done to avoid pathological cases where starting from zero may lead to slow convergence or degenerate search directions, especially when the system has certain symmetries. The random perturbation helps the algorithm explore more robust directions from the very beginning.

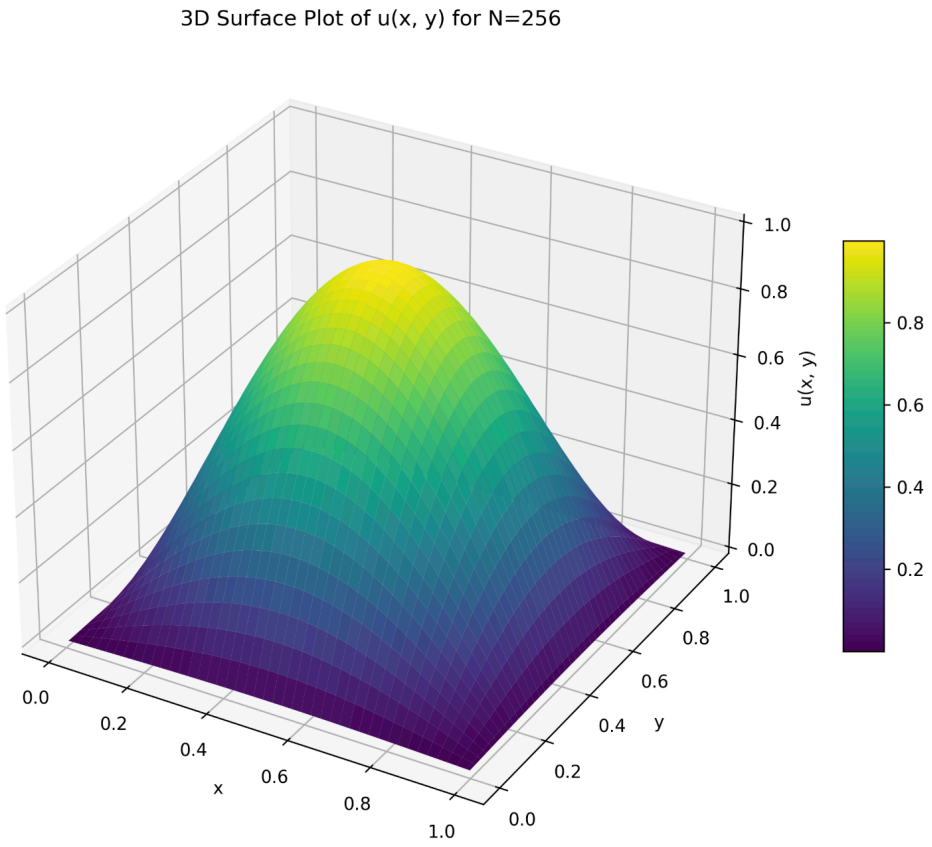
3.2.2 Iterations and runtime table

As shown in the table, the number of iterations increases with grid resolution N. This is expected since a finer grid results in a larger linear system, making it more challenging for the Conjugate Gradient (CG) method to converge. However, the increase in iteration count is relatively moderate, demonstrating the algorithm's scalability. Notably, the runtime remains impressively low even for $N = 256$, with the entire CG solve completing in under 0.07 seconds, highlighting the efficiency of the implementation and the benefits of using sparse matrix-vector multiplication.

N	Iterations	Time (s)
8	14	0.0000
16	24	0.0000
32	45	0.0002
64	77	0.0011
128	135	0.0175
256	220	0.0677

3.2.3 Plot of function $u(x)$

The 3D surface plot of $u(x, y)$ at $N = 256$ illustrates the expected shape of the solution to the Poisson problem with homogeneous Dirichlet boundary conditions. The solution reaches its maximum near the center of the domain and smoothly decays to zero at the boundaries. The symmetry and smoothness of the plot also confirm the correctness of the numerical scheme and the CG solver implementation. The shape matches the analytical form $\sin(\pi x) \sin(\pi y)$, validating both the discretization and the numerical accuracy.



3.3 Convergence of CG

In this section, we need to find the convergence behavior of the Conjugate Gradient (CG) method when solving a dense symmetric positive definite linear system of the form:

$$A_{ij} = \frac{N-|i-j|}{N}, \quad b_j = 1$$

The CG implementation is tested for increasing problem sizes: $N \in 10^2, 10^3, 10^4, 10^5$. This matrix is symmetric and positive definite by construction, ensuring the applicability of the CG algorithm. The implementation collects the residual norm at each iteration and compares the convergence behavior with the theoretical error bound

3.3.1 Perfromance and Observations

N	Iterations	Time (s)	Final Residual	Condition Number κ
100	56	0.0003	1.45e-07	1.35e+04
1000	368	0.3261	3.55e-07	1.35e+06
10000	2634	249.75	1.36e-06	1.35e+08

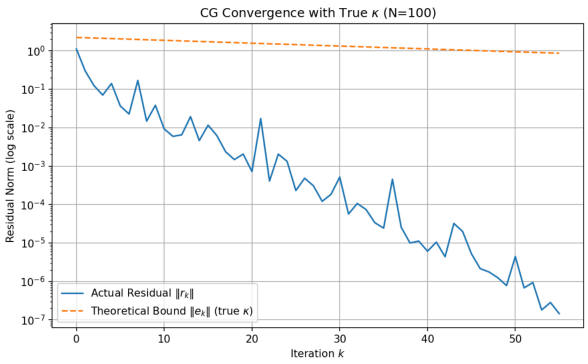
We can see that:

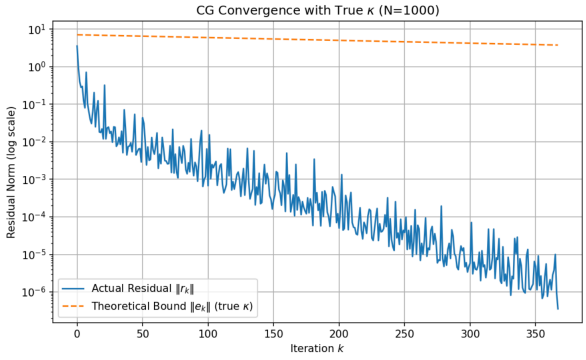
- The **iteration count increases roughly proportionally to** $\sqrt{\kappa}$, as predicted by CG theory.
- The condition number κ increases quadratically with N, leading to significantly slower convergence for larger systems.
- The final residuals consistently fall below the tolerance, confirming the correctness of the implementation.
- Despite the dense nature of the matrix, the method remains stable for N \leq 10^4, though computational cost becomes noticeable.

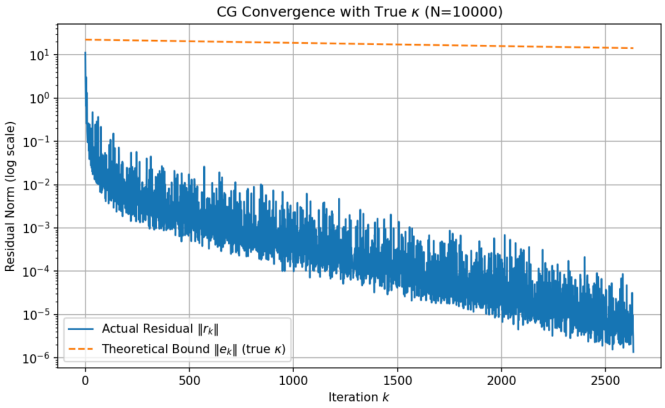
3.3.2 CG Residual vs Theoretical Bound

The residual convergence plots for N = 100, 1000, and 10000 reveal a consistent pattern: as the problem size increases, the condition number of the system grows, and the CG method requires significantly more iterations to reach convergence.

For smaller systems (N = 100), the residual decreases smoothly and far outpaces the theoretical error bound, indicating rapid convergence. As N increases, the **residual curves become more oscillatory and converge more slowly**, yet still remain well below the theoretical bound throughout. This demonstrates that the CG method remains effective even for large systems, although the theoretical bound tends to be overly conservative, especially in the early iterations.







3.4 Conclusion

While the CG method performed well for problem sizes up to $N = 10{,}000$, I encountered a **segmentation fault** when attempting to run the same solver with $N = 100{,}000$. This is due to the excessive memory usage required to store the dense matrix and associated vectors. In a matrix of size $100{,}000 \times 100{,}000$, even a single double-precision array would require over 74 GB of memory, which exceeds the capacity of typical desktop machines. This highlights the limitations of dense representations for large-scale problems and motivates the use of matrix-free or sparse implementations in practical applications.

In summary, we investigated the convergence behavior of the Conjugate Gradient method when applied to dense symmetric positive definite systems with varying condition numbers. The empirical results demonstrate that CG remains numerically stable and converges reliably, even for large systems. However, as the

matrix size increases, the number of iterations grows significantly, consistent with the theoretical dependence on the condition number \kappa. Moreover, while the theoretical error bound provides a useful reference, it consistently overestimates the actual error, particularly in early iterations. These findings confirm both the practical robustness of CG and the importance of preconditioning or other acceleration strategies for large, ill-conditioned systems.