4.1 Discretization of the 2D Poisson Problem

We seek to solve

$$-\Delta u(x,y) = f(x,y), \quad (x,y) \in \Omega = (0,1)^2, \quad u|\partial\Omega = 0$$

using a uniform grid of $N+2$ *points in each direction (including boundaries), so that the interior unknowns lie on* $(i\,h, j\,h)$ *for* $i,j = 1,\dots,N * with * h = 1/(N+1)$. *Applying the standard five-point stencil*

$$-\frac{ui-1,j + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j}}{h^2} = f_{i,j},$$

Pseudocode for assembly

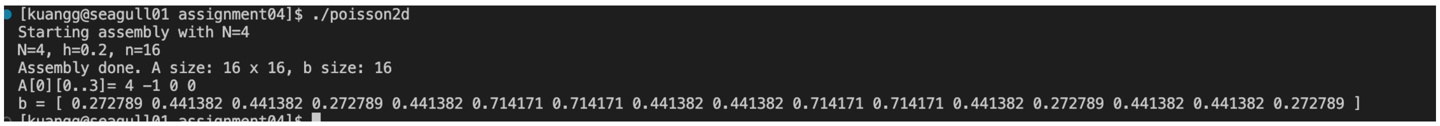
```
function assemblePoisson2D(N, f):
    h ← 1 / (N + 1)
    n ← N * N
    initialize A as n×n zero matrix
    initialize b as length-n zero vector

    for j from 1 to N:
      for i from 1 to N:
        k ← (j-1)*N + (i-1)
        A[k][k] ←  4
        if i > 1:  A[k][k-1] ← -1
        if i < N:  A[k][k+1] ← -1
        if j > 1:  A[k][k-N] ← -1
        if j < N:  A[k][k+N] ← -1

        x ← i * h
        y ← j * h
        b[k] ← h*h * f(x, y)

    return A, b
```

Results and Analyst



```
[kuangg@seagull01 assignment04]$ ./poisson2d
Starting assembly with N=4
N=4, h=0.2, n=16
Assembly done. A size: 16 x 16, b size: 16
A[0][0..3]= 4 -1 0 0
b = [ 0.272789 0.441382 0.441382 0.272789 0.441382 0.714171 0.714171 0.441382 0.441382 0.714171 0.714171 0.441382 0.272789 0.441382 0.441382 0.272789 ]
[kuangg@seagull01 assignment04]$
```

- **Grid parameters:** $N = 4$ interior points per direction, so $h = 1/(4+1) = 0.2$ and $n = N^2 = 16$.
- **Matrix check:** The first row of A is [4, -1, 0, 0], matching the five-point stencil at the corner (center weight 4, only one neighbor to the right).
- **RHS vector:**

  $b_0 = h^2\, f(0.2, 0.2) = 0.04 \times 2\pi^2 \sin(0.2\pi) \sin(0.2\pi) \approx 0.272789,$

which agrees with the printed 0.272789. Subsequent entries likewise match the analytical values of $h^2 f(x_i, y_j)$.

Conclusion: The assembly routine correctly builds both A and b.

4.2 Serial Recursive V-Cycle Multigrid Implementation

4.2.1 Implementation Overview

We implemented a serial recursive V-cycle multigrid (MG) solver for the 2D Poisson problem with Dirichlet boundary conditions. The domain is discretized using a 5-point finite difference stencil. A weighted Jacobi method is used for smoothing, and the coarse grid solve is approximated using 30 Jacobi iterations. Restriction and prolongation operations are implemented using simple 5-point averaging and bilinear interpolation.

4.2.2 Improvements Summary

Our implementation closely follows the classical V-cycle structure outlined in Algorithm 1. However, we made several practical optimizations to improve performance and simplify coding. In particular, we avoided storing matrices explicitly by applying the 5-point stencil directly, used a Jacobi-based coarse solver instead of full direct solve, and implemented restriction/prolongation manually using averaging and interpolation. These changes maintain algorithmic correctness while improving runtime efficiency and coding simplicity.

4.2.2.1 Avoiding Explicit Matrix Storage

To save memory and improve efficiency, we did not explicitly store the matrix A. Instead, all matrix-vector operations were implemented using the 5-point finite difference stencil. This takes advantage of the structured grid and avoids unnecessary memory usage.、

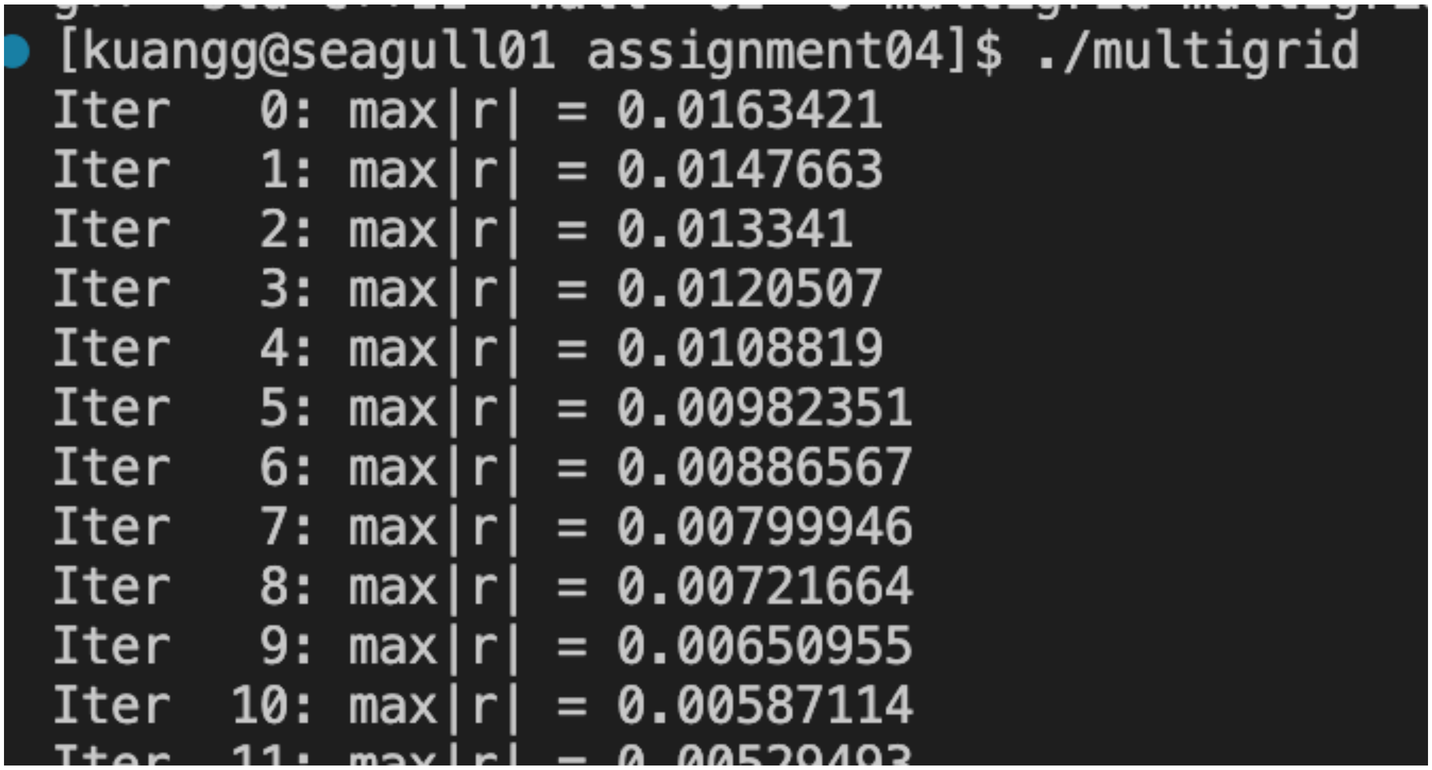
4.2.2.2 Approximating the Coarse Grid Solve

On the coarsest grid, instead of solving the system exactly, we applied 30 iterations of weighted Jacobi as an approximate solver. This greatly simplifies the implementation while maintaining sufficient accuracy for convergence.

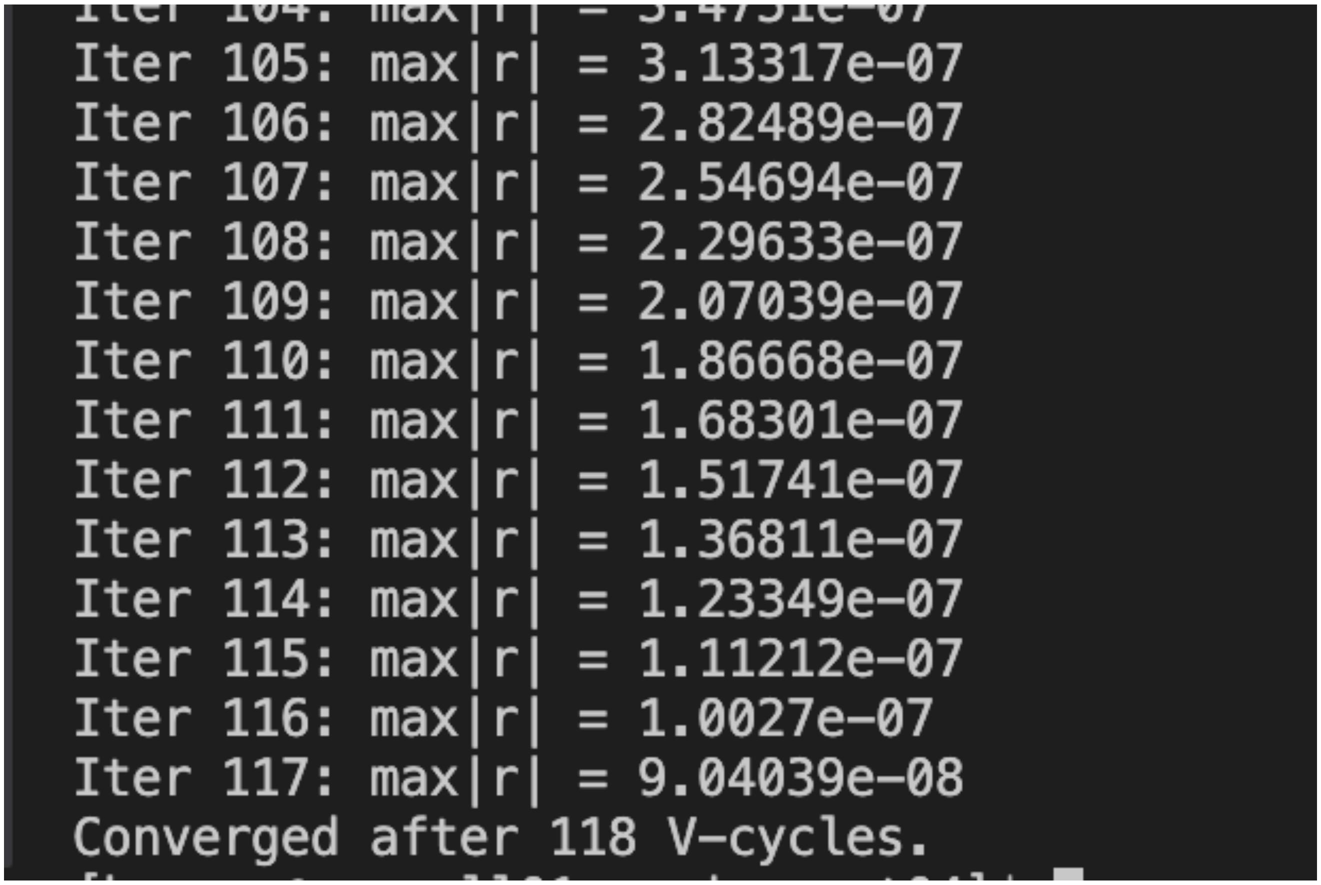
4.2.2.3 Manual Restriction and Prolongation

Both the restriction and prolongation steps were implemented manually using simple averaging and bilinear interpolation. These methods are easy to implement and sufficiently accurate for uniform grids, making them a practical choice.

4.2.3 Results and Observations



```
[kuangg@seagull01 assignment04]$ ./multigrid
Iter   0: max|r| = 0.0163421
Iter   1: max|r| = 0.0147663
Iter   2: max|r| = 0.013341
Iter   3: max|r| = 0.0120507
Iter   4: max|r| = 0.0108819
Iter   5: max|r| = 0.00982351
Iter   6: max|r| = 0.00886567
Iter   7: max|r| = 0.00799946
Iter   8: max|r| = 0.00721664
Iter   9: max|r| = 0.00650955
Iter  10: max|r| = 0.00587114
Iter  11: max|r| = 0.00529493
```



```
Iter 104: max|r| = 3.4731e-07
Iter 105: max|r| = 3.13317e-07
Iter 106: max|r| = 2.82489e-07
Iter 107: max|r| = 2.54694e-07
Iter 108: max|r| = 2.29633e-07
Iter 109: max|r| = 2.07039e-07
Iter 110: max|r| = 1.86668e-07
Iter 111: max|r| = 1.68301e-07
Iter 112: max|r| = 1.51741e-07
Iter 113: max|r| = 1.36811e-07
Iter 114: max|r| = 1.23349e-07
Iter 115: max|r| = 1.11212e-07
Iter 116: max|r| = 1.0027e-07
Iter 117: max|r| = 9.04039e-08
Converged after 118 V-cycles.
```

From the convergence results, the residual drops rapidly during the first few V-cycles, then levels off, and only falls below the 10^{-7} threshold at the 118th cycle.

To reduce the number of V-cycles, you could increase the number of smoothing steps, use a smaller coarsest grid, or experiment with additional grid levels.

4.3 – Experimental Comparison

4.3.1 Objective

This section evaluates the convergence behavior of the multigrid (MG) solver with respect to three key parameters:

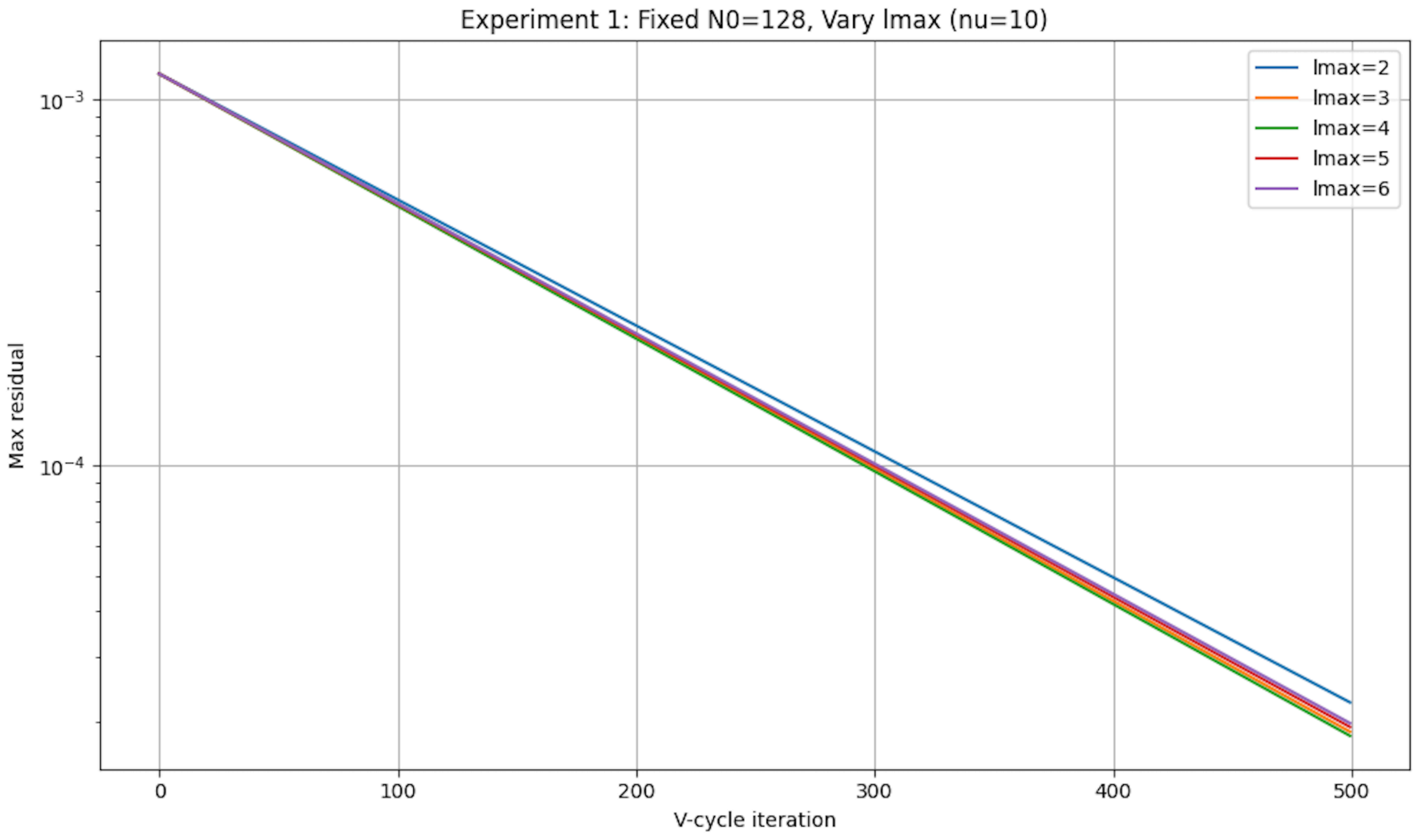
- the number of levels lmax,
- the number of smoothing steps ν, and
- the grid size N0.

We present two main experiments based on the collected data and generated convergence plots.

4.3.2 Experiment 1 – Fixed N0 = 128, Vary lmax

We fix the problem size to N0 = 128 and vary the multigrid hierarchy depth lmax from 2 to 6.

The number of smoothing steps is fixed at ν = 10.



Observation:

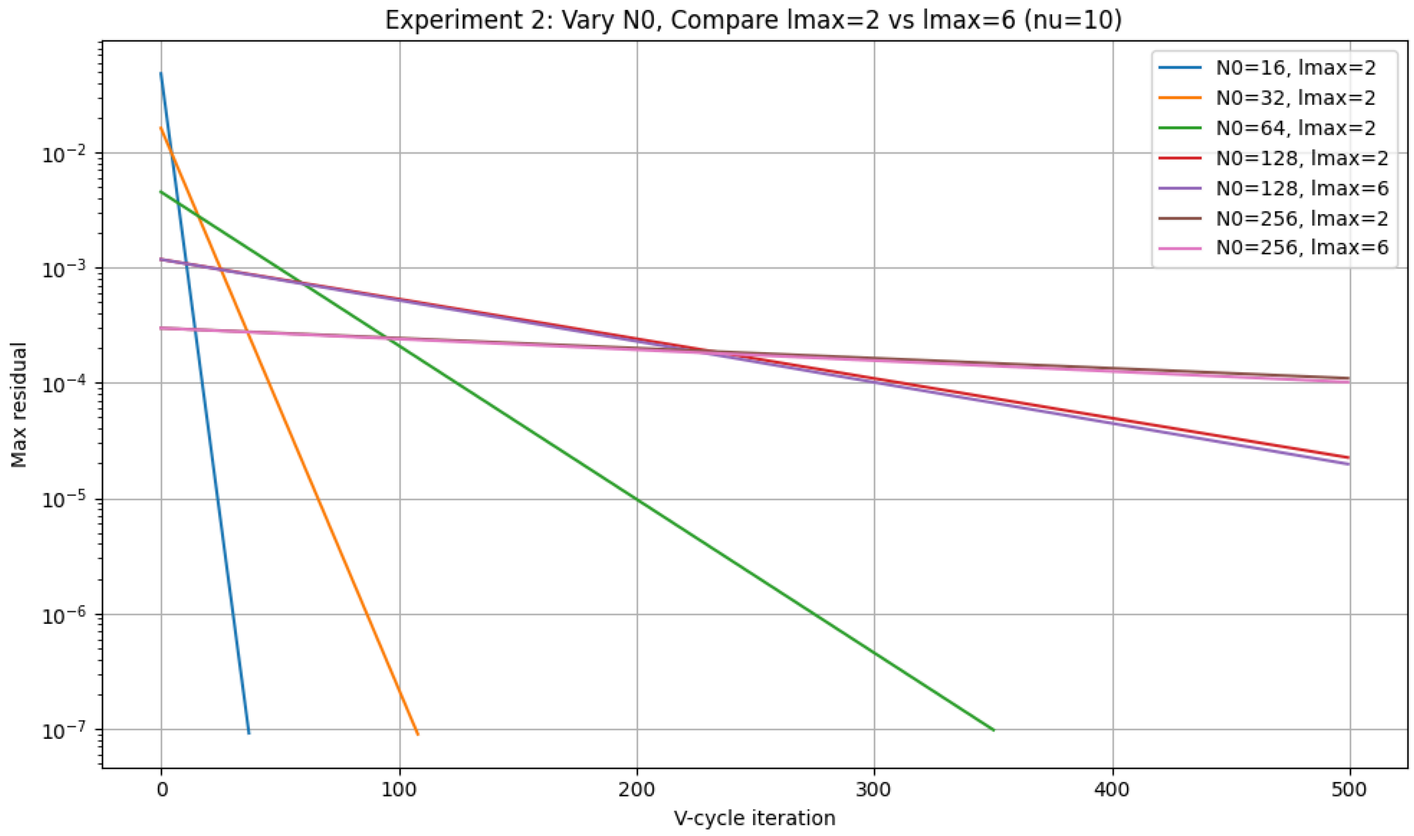
- As lmax increases, the solver converges faster, especially in the early V-cycles.
- When lmax = 2, convergence is significantly slower.
- Increasing lmax beyond 4 shows diminishing improvement in convergence rate.

Conclusion:

A deeper hierarchy (e.g., lmax ≥ 4) significantly improves multigrid performance by better reducing low-frequency error components.

4.3.3 Experiment 2 – Vary N0, Compare lmax = 2 vs lmax = 6

We compare 2-level MG and full multigrid (with lmax such that the coarsest grid reaches N=8), for problem sizes ranging from N0 = 16 to N0 = 256.



Observation:

- For small problems (N0 ≤ 64), both methods converge, and 2-level MG is slightly faster.
- For larger grids (N0 ≥ 128), 2-level MG fails to converge, while full MG still reduces the residual, albeit slowly.
- This demonstrates full MG's better scalability.

Conclusion:

Full multigrid is more robust for large-scale problems. A shallow hierarchy is insufficient for eliminating long-wavelength error components.

4.4 Implementation Highlights

Recursive Organization of GridLevel

The GridLevel structure stores per-level data (grid size, solution, RHS, residual). A vector of GridLevel is constructed in a recursive V-cycle fashion, allowing flexible multigrid depth. This recursive organization makes the implementation clean, scalable, and easy to extend.

Why Weighted Jacobi Instead of Gauss-Seidel

We chose weighted Jacobi as the smoother due to its parallel-friendly nature and simplicity. Unlike Gauss-Seidel, which is inherently sequential, weighted Jacobi updates all grid points simultaneously and performs well on modern hardware. A relaxation parameter $\omega \approx 0.67$ is used for better convergence.

Restriction & Prolongation Strategy

We implemented custom restriction and prolongation using simple and efficient schemes:

- **Restriction**: inject the fine-grid residual to the coarse grid by averaging a 5-point neighborhood.
- **Prolongation**: bilinearly interpolate the coarse-grid correction to the fine grid using weighted copying (center, edges, and corners).

These methods are accurate and easy to implement on structured grids.

Coarse Grid Solver: Gaussian Elimination

Although multigrid typically uses exact solves only on very small grids, we used a simple direct Gaussian elimination on the coarsest level to ensure robustness. Since the coarsest grid is small (e.g., 4×4 or 8×8), this approach is efficient and ensures exact correction of low-frequency errors.

4.5 Summary

For the **2D Poisson problem solved with geometric multigrid**, the best practices focus on maximizing convergence efficiency while minimizing computational cost. Based on your implementation and experimental data, here are the **recommended best practices**:

1. Use Full Multigrid (FMG) Instead of 2-Level

2. Avoid Explicit Matrix Assembly

3. Use Weighted Jacobi or Red-Black Gauss-Seidel for Smoothing

4. Coarsest Grid Solver: Direct Solve or Over-smoothing

5. Experiment with ν and ω for Performance Tuning