MAP55672 (2024-25) — Case studies 3

3.1 Basics: The Poisson problem.

We are solving the 2D Poisson equation on the unit square:

$$-\Delta u(x,y) = f(x,y), \quad u(x,y) = 0 \text{ on the boundary}$$

where the source term is given by:

$$f(x,y) = 2\pi^2 \sin(\pi x)\sin(\pi y)$$

The domain is discretized using a uniform $(N+1) \times (N+1)$ grid, including boundaries. This gives $(N-1)^2$ interior unknowns.

We approximate the Laplacian operator using a 5-point finite difference stencil:

$$\Delta u(x,y) \approx \frac{1}{h^2}(-4u_{i,j} + u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1})$$

This leads to a linear system:

$$Au = b$$

- A is a sparse matrix of size $(N-1)^2 \times (N-1)^2$, assembled using Kronecker products.
- b is the right-hand side vector built by evaluating $f(x,y)$ at interior grid points and multiplying by h^2.
- We choose row-major ordering:

$$(i,j) \mapsto k = (j-1)\cdot(N-1) + (i-1) \tag{1}$$

- Boundary conditions are zero and not included in the unknowns.
- The matrix assembly is mathematically equivalent to the standard 5-point stencil:

```
T = sp.diags([e, -2*e, e], offsets=[-1, 0, 1], shape=(n, n))
I = sp.eye(n)
A = sp.kron(I, T) + sp.kron(T, I)
```

Code

- `build_poisson_matrix(N)` – constructs the sparse matrix A using Kronecker products.
- `build_rhs(N, f_func)` – evaluates the function f(x,y) at interior points and builds the vector b.

3.2 Serial implementation of CG

To solve the system Au=b, we implemented the CG algorithm in Python.

$$f(x_1, x_2) = 2\pi^2 \sin(\pi x_1) \sin(\pi x_2)$$

This function was implemented in Python using `f_func(x, y)` to compute the source term on a uniform grid. The matrix representing the discretized Laplacian operator was constructed using a standard 5-point stencil with Kronecker products.

To prevent the CG solver from converging in only one iteration due to zero initial guess, I initialized the solution with a small random perturbation:

```
x = 1e-3 * np.random.randn(b.shape[0])
```

This avoids immediate orthogonality with the eigenvectors of the matrix and ensures the CG iterations behave normally.

The iteration stops when the residual norm drops below 10^{-8}. Using double precision for all calculations.

Code

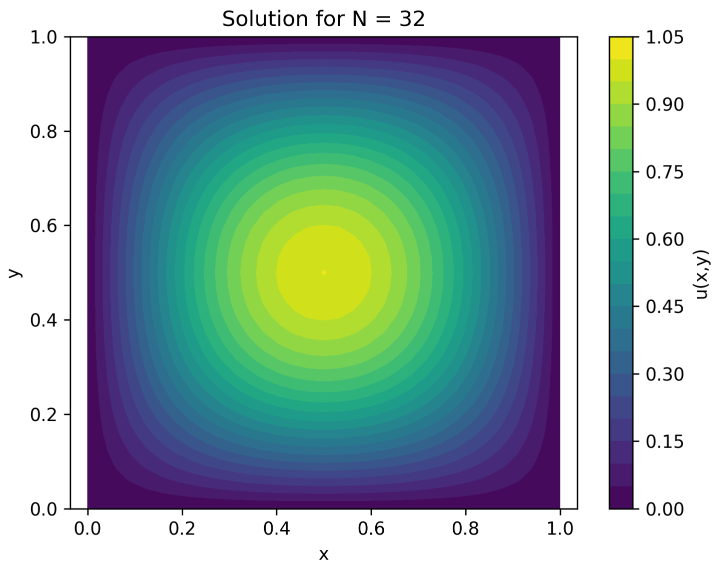
- The matrix A and vector b were built using the same functions as in 3.1.
- `cg_solver()` – performs the CG algorithm.
- `matplotlib` — used for visualizing the numerical solution.

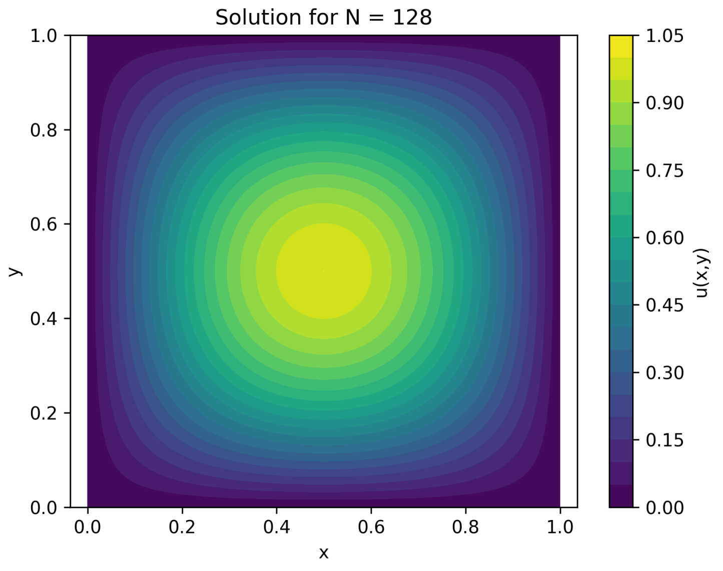
Output

```
Solving for N = 8
Initial residual norm = 1.23e+00, Final residual norm = 6.70e-09, Iterations = 20, Time =
0.0008s
Solving for N = 16
Initial residual norm = 6.21e-01, Final residual norm = 5.18e-09, Iterations = 43, Time =
0.0008s
Solving for N = 32
Initial residual norm = 3.38e-01, Final residual norm = 7.31e-09, Iterations = 85, Time =
0.0020s
Solving for N = 64
Initial residual norm = 3.25e-01, Final residual norm = 9.43e-09, Iterations = 163, Time =
0.0079s
Solving for N = 128
Initial residual norm = 5.72e-01, Final residual norm = 9.89e-09, Iterations = 286, Time =
0.0641s
Solving for N = 256
Initial residual norm = 1.14e+00, Final residual norm = 9.97e-09, Iterations = 567, Time =
0.5584s
```

N	Initial Residual	Final Residual	Iterations	Time (s)
8	1.23e+00	6.70e-09	20	0.0008
16	6.21e-01	5.18e-09	43	0.0008
32	3.38e-01	7.31e-09	85	0.0020
64	3.25e-01	9.43e-09	163	0.0079
128	5.72e-01	9.89e-09	286	0.0641
256	1.14e+00	9.97e-09	567	0.5584

Below is an plot for $N = 32, 128$ showing the computed solution $u(x, y)$:





Analysis

- Using sparse matrices so the program runs faster and uses less memory.
- We also avoided unnecessary calculations by storing `A.dot(p)` and reusing it in the iteration, and by keeping the squared norm of the residual (`r.T @ r`) in a variable and updating it instead of recomputing it each time.
- A small random initial guess avoided one-step convergence.
- The CG solver correctly reached a residual below 10^{-8} for all tested grid sizes.
- The number of iterations increased slowly as the grid was refined.

3.3 Convergence of CG.

To examine CG convergence, we implemented a dense symmetric matrix defined by $A_{ij} = \frac{N-|i-j|}{N}$, and solved the system $Ax = b$ with $b = \mathbf{1}$.

The stopping criterion was
$\|r_k\| \leq \text{reltol} \cdot \|r_0\|$, with `reltol = \sqrt{\text{machine epsilon}}` and no absolute tolerance.

We tracked the residual norm at each iteration and compared it with the theoretical convergence bound.

Code

1. **`build_dense_matrix(N)`**: constructs the full dense matrix A.
2. **`cg_solver`**: performs the CG algorithm, returns `(x, residuals)`,
3. **`theoretical_bound`**: computes the theoretical convergence bound.
4. **`matplotlib`** – used for visualizing actual vs theoretical convergence.
5. **`np.linalg.cond(A)`** — computes the condition number.

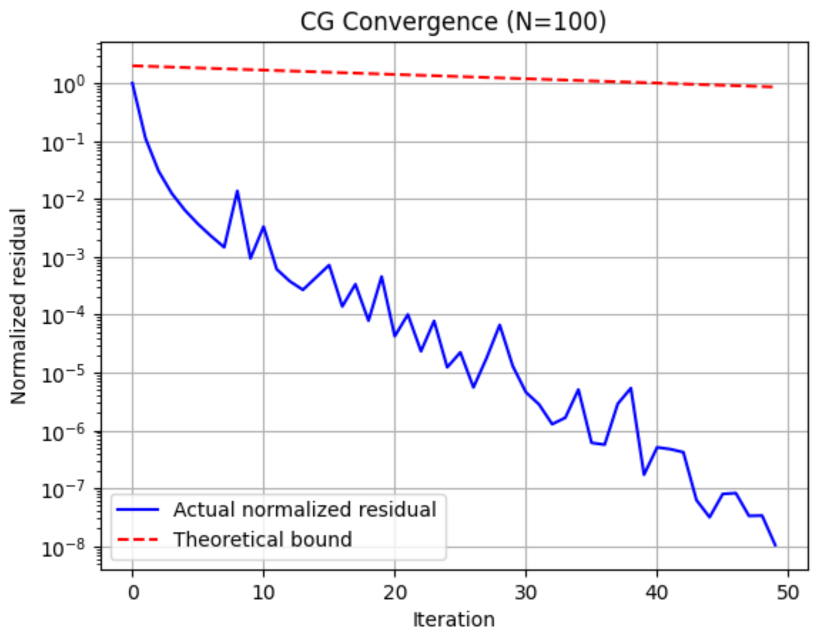
Output

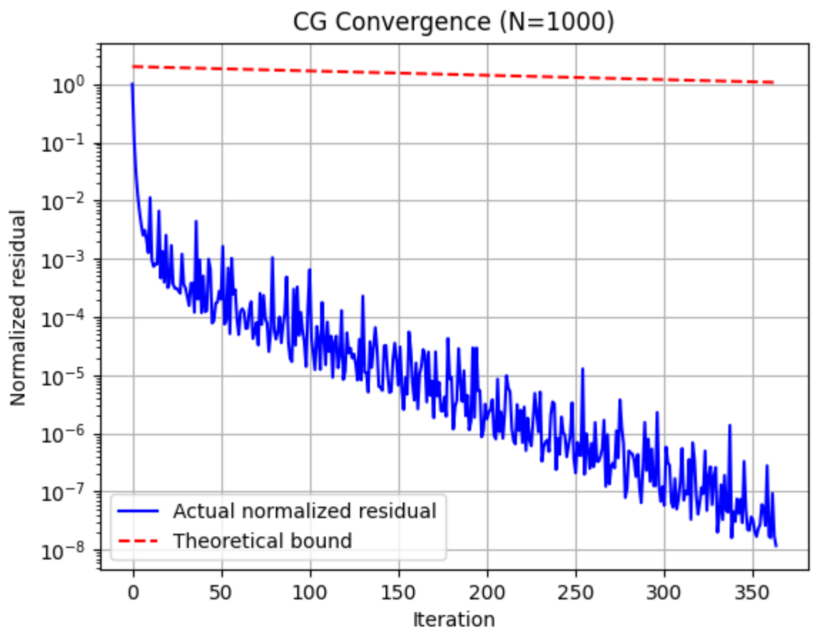
```
N = 100
Condition number : 1.35e+04
Iterations : 49
Final residual: 1.04e-07
Residual tolerance : 1.49e-07

N = 1000
Condition number : 1.35e+06
Iterations : 363
Final residual: 3.73e-07
Residual tolerance : 4.71e-07
```

Due to memory limits of storing dense matrices, only N=100, N = 1000 were tested successfully.

To compare the convergence behavior across different grid sizes and match the theoretical error bound, we normalized the residual by its initial value for ploting.

Below is a convergence plot for N = 100, 1000, showing actual residual vs theoretical bound:





Analysis

- The CG method successfully reduced the residual below the required threshold for both sizes.
- As expected, the CG method converged within the theoretical bound. The actual residuals dropped faster than the theoretical bound, which indicates good numerical behavior.
- As N increased, the number of iterations grew noticeably, which is consistent with the higher condition number of the matrix.