Solving the Two-Dimensional Poisson Equation by the Finite-Difference Method (A comparative study of some simple methods in Python)

Altankhuu Bayarsaikhan[*1], **Gantumur Tsogtgerel**[1,2]
[1]*Department of Physics, School of Arts and Sciences, National University of Mongolia*
[2]*Department of Mathematics and Statistics, McGill University*

Abstract

To model the motion of an incompressible fluid in a lid-driven cavity, one needs to solve the Navier–Stokes equations. When these equations are approximately solved for the primary variables (the velocity field and the pressure field), solving a Poisson equation becomes the most time-consuming step. Therefore, it is important to use a method that is very easy to implement, sufficiently fast, and able to produce results with an appropriate accuracy. In this work, we approximately solve the Poisson equation on a two-dimensional square grid using finite-difference schemes with homogeneous Neumann boundary conditions. We compare representative direct methods (dense-matrix LU factorization, sparse-matrix LU factorization, and the discrete cosine transform (DCT)) and the simplest iterative methods (Jacobi, Gauss–Seidel, and successive over-relaxation). The computational programs were written in Python 3.12 using NumPy, SciPy, and Numba in the Jupyter Notebook/Lab environment under Windows. The processor used for the computations was an Intel i7 Quad-Core (4 CPUs, 8 threads, 2.3 GHz base) with 16 GB RAM. According to the computational results, among the direct methods the DCT is the fastest and most memory-efficient, followed by the sparse-matrix method. Among the iterative methods, the successive over-relaxation method compiled with Numba JIT is the fastest when the relaxation parameter is chosen near its optimal value, but it is slower and less accurate than the direct methods mentioned here. The novelty of this work is that, to solve the Poisson equation, we selected the simplest well-studied methods and compared them experimentally (computationally) using specific programming tools on a personal computer with clearly stated technical specifications, and identified the fastest method among them (DCT). We believe that the concrete data reported in this work (computer specifications, programming tools used, implementation choices, and measured runtimes) can serve as a useful reference for similar comparative studies in the future.
In this study we focus only on selecting a Poisson solver that is as easy to use as possible while still being sufficiently fast. Using the selected solver in simulations of incompressible lid-driven cavity flow is left for future work.
Keywords: Poisson equation; computational fluid dynamics; lid-driven cavity; Neumann boundary condition; Discrete Cosine Transform; LU factorization; sparse matrices; successive over-relaxation; Numba.

I. INTRODUCTION

A classical benchmark problem in computational fluid dynamics is the incompressible lid-driven cavity flow. Its numerical study requires solving the Navier–Stokes equations. In many pressure–velocity coupling schemes, solving for the velocity field requires solving an associated Poisson equation for the pressure [1, 2].
In the present paper, however, the object of study is not the full cavity-flow solution itself but the model two-dimensional Poisson problem defined by Eqs. (1)–(3) on square Cartesian grids with homogeneous Neumann boundary conditions. The goal is to compare, on the same finite-difference linear-system family, the runtime, memory behavior, and practical usability of several standard solvers implemented in Python. Some other representative problem classes of this kind include insulated heat-conduction and steady-state temperature-distribution problems in heat transfer, electrostatic or gravitational potential problems

* E-mail: altankhuu@num.edu.mn

with homogeneous Neumann boundaries in mathematical physics, groundwater-flow and diffusion-type model equations on rectangular domains in geophysics and environmental modeling, and other separable elliptic boundary-value problems posed on regular grids.

All methods considered below use the same second-order five-point finite-difference discretization on the same collocated grid and the same ghost-point treatment of the boundary conditions. The methods differ only in the strategy used to solve the resulting discrete system. The remainder of the paper is organized as follows. Section 2 summarizes the computational methods and software tools. Sections 3–5 describe the direct, DCT-based, and iterative solvers. Section 6 presents the implementation and benchmarking procedure. Section 7 discusses the measured results, and Section 8 gives the final conclusions. With this scope established, we next define the model Poisson problem considered throughout the paper, together with its boundary conditions and discrete formulation.

For an unknown scalar function $p(x, y)$ (pressure) and a given source function $r(x, y)$, the two-dimensional Poisson equation reads

$$\Delta p(x, y) = r(x, y), \tag{1}$$

where $\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$ is the Laplacian, $x \in [0, L_x]$, $y \in [0, L_y]$, and L_x and L_y are the side lengths of the cavity (square domain). For impermeable cavity walls, homogeneous Neumann boundary conditions apply on the domain boundary:

$$\begin{cases} \dfrac{\partial p(0,y)}{\partial x} = 0 & \text{(left boundary)} \\ \dfrac{\partial p(L_x,y)}{\partial x} = 0 & \text{(right boundary)} \\ \dfrac{\partial p(x,0)}{\partial y} = 0 & \text{(top boundary)} \\ \dfrac{\partial p(x,L_y)}{\partial y} = 0 & \text{(bottom boundary).} \end{cases} \tag{2}$$

For all benchmark runs reported below, we set $L_x = L_y = 1$ and use the smooth model right-hand side

$$r(x, y) = \frac{\cos(\pi x) + \cos(\pi y)}{\pi}. \tag{3}$$

(We chose this source term deliberately as a fixed smooth forcing in order to benchmark the solver families themselves, rather than the details of any particular transient pressure-correction right-hand side arising at one time step of a Navier–Stokes simulation. A one-step Navier–Stokes right-hand side would depend additionally on the instantaneous intermediate-velocity field, the Reynolds number, the time-step size, and other flow-specific choices, thereby mixing solver performance with case-specific CFD effects [1, 2]. By contrast, the present explicit model source isolates the cost, memory demand, and numerical behavior of the Poisson solvers on the same discrete operator and boundary treatment. The lid-driven cavity problem remains the target application motivating the study, but the benchmark is intentionally posed as a single, explicitly defined model Poisson problem that can later be embedded into that broader CFD setting. Moreover, this scope distinction is also important because the present benchmark is formulated for a collocated-grid Poisson operator, whereas the planned lid-driven cavity solver will use a staggered-grid pressure equation; consequently, even a pressure-correction right-hand side extracted from a single Navier–Stokes time step would not eliminate the need for a later solver adaptation.)

On a square grid with $N_x \times N_y$ points and uniform spacings $h_x = \frac{L_x}{N_x - 1}$ and $h_y = \frac{L_y}{N_y - 1}$, the discrete form $(\Delta p)_{ji} = r_{ji}$ of (1) using the 5-point-stencil finite-difference scheme is

$$\frac{(p_{j-1,i} - 2p_{j,i} + p_{j+1,i})}{{h_y}^2} + \frac{(p_{j,i-1} - 2p_{j,i} + p_{j,i+1})}{{h_x}^2} = r_{j,i}, \tag{4}$$

where $p_{j,i} = p(x_i, y_j)$ and $r_{j,i} = r(x_i, y_j)$, with $1 \le i \le N_x$ and $1 \le j \le N_y$. Applying this equation at every grid point yields a linear system. To enforce the boundary conditions, we use a ghost-point method [3]. By representing the boundary conditions (2) using second-order central differences, the ghost points can be eliminated in terms of interior points (for example, on the left boundary $p_{j,0} = p_{j,2}$ and for the top boundary $p_{0,i} = p_{2,i}$, and similarly on the other boundaries.) [3]. Figure 1 shows the $(N_x \times N_y)$ grid (dark dots), the 5-point stencil, and the ghost points (light gray dots).

If the matrices p_{ji} and r_{ji} are reshaped into column vectors P and R of length $(N_y \cdot N_x)$, then the system (4) can be written in matrix form as

$$A \cdot P = R, \tag{5}$$

where A (the Neumann Laplacian) is a sparse block

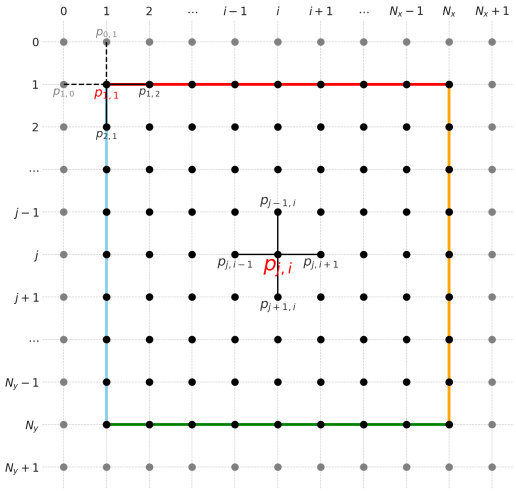

Figure 1. Computational grid; matrix p_{ji}; 5-point stencil; ghost points

matrix of size $(N_y \cdot N_x) \times (N_y \cdot N_x)$ with a tridiagonal block structure. Due to the boundary condition (2), the sum of the elements in any row of this matrix is zero. Therefore, multiplying it by any constant column vector

$$C = \overbrace{(c, c, c, \dots, c)}^{N_y \cdot N_x}{}^T = c\,\overbrace{(1, 1, 1, \dots, 1)}^{N_y \cdot N_x}{}^T$$

yields zero:

$$A \cdot C = 0, \tag{6}$$

where c is an arbitrary real number. Hence there exist infinitely many solution vectors of (5) with the same right-hand side that differ only by a constant c [4, 5]. To select a unique solution from these equivalent solutions, we impose the *zero-mean solution* condition. For this solution to exist, the right-hand side must satisfy the *zero-mean RHS* condition [4]:

$$\begin{cases} \overline{p_{ji}} = 0 \\ \overline{r_{ji}} = 0. \end{cases} \tag{7}$$

For the benchmark source term (3), this compatibility condition is satisfied naturally, since its domain average over the unit square is zero. In practice, we enforce (7) as

$$\begin{cases} p_{ji} = p_{ji} - \overline{p_{ji}} \\ r_{ji} = r_{ji} - \overline{r_{ji}} \end{cases} \tag{8}$$

[4, 3].

II. COMPUTATIONAL METHODS AND TOOLS

For our purposes, a method for solving the linear system (5) on a computer should be simple, fast, and sufficiently accurate. Therefore, we compare the following methods and choose among them.

- Direct methods: dense-matrix LU factorization and sparse-matrix LU factorization;
- Spectral method: discrete cosine transform (DCT);
- Iterative methods: Jacobi method, Gauss–Seidel method, and successive over-relaxation method.

We use the following tools for computations.

- Computer: laptop (personal computer) with Intel i7 Quad-Core, 4 CPUs, 8 threads, 2.3 GHz (base), 16 GB RAM;
- Software: Windows/Linux environment, Jupyter Notebook/Lab system;
- Coding/numerical tools: Python 3.12, NumPy, SciPy, and Numba modules.

All benchmarked methods are applied to the same underlying discrete problem and the same grid-point set; this includes the DCT method, which acts on the same nodal unknowns as the direct and iterative solvers and differs only in the solution strategy used for the resulting linear system. Python 3.12 was selected as the implementation language because it is widely used in scientific computing and is supported by mature numerical libraries. NumPy and SciPy provide well-established array, sparse-linear-algebra, and transform routines, while Numba offers JIT acceleration for selected iterative kernels [6, 7, 8]. Compared with proprietary numerical environments such as Matlab, the Jupyter Notebook/Lab + Python + NumPy/SciPy workflow also offers an open and reproducible software stack for benchmarking.

III. DIRECT METHODS: DENSE AND SPARSE LU FACTORIZATION

Direct methods solve the system (5) in a finite number of steps and, in theory, produce the exact solution. In practice, when computations are performed using floating-point arithmetic with limited precision, the computed solution differs from the exact one by machine precision; for example, for double precision the error is on the order of 10^{-16} [9].

Algorithmically, the simplest direct method to implement is Gaussian elimination. It applies transformations to the rows of the matrix A (multiplying by a constant coefficient, adding or subtracting, and sometimes swapping rows) to eliminate unknowns

and form an upper triangular matrix, and then solves the resulting triangular system by starting from the last remaining single-variable equation and proceeding upward (back-substitution) to obtain the solution vector [10].

LU factorization is an extended form of Gaussian elimination. It represents A as a product of a lower and an upper triangular matrix:

$$A = L \cdot U, \tag{9}$$

where L is the lower triangular matrix containing the multipliers used in Gaussian elimination, and U is the upper triangular matrix produced by the elimination [10]. However, we do not need to program LU factorization step by step. For dense matrices we use tools from the **linalg** submodule of SciPy (based on the high-performance linear-algebra library LAPACK), and for sparse matrices we use tools from **sparse.linalg** (based on the SuperLU library for solving linear systems) by calling them directly in our Python programs [7].

A. Dense-matrix LU

The method consists of the following steps.

1. Create and allocate the matrix A in memory: **A=numpy.zeros((Ny*Nx),(Ny*Nx))** or **A=numpy.ones((Ny*Nx),(Ny*Nx))**;
2. Fill A programmatically with the coefficients of the Neumann Laplacian;
3. Compute the LU factorization of A (factorize into the product of a lower and an upper triangular matrix by Gaussian elimination): **lu_piv = scipy.linalg.lu_factor(A)**;
4. Solve for the solution: **scipy.linalg.lu_solve(lu_piv, R)**, where R is the right-hand side of (5).

Advantage: Steps 1–3 are performed only once before the main computations, and when R changes only Step 4 needs to be repeated [10]. As noted above, the computed solution differs from the exact one by machine precision (assuming no discretization error) [9].

Disadvantages: 1) Memory intensive. Working with the matrix A requires, on average, $O(n^2)$ memory, where $n = N_y \cdot N_x$. For a matrix of real numbers of type **float64**, this corresponds to $8n^2$ bytes [11, 10]. For example, when $N_y = N_x = 513$, about 550 GB of memory would be required. 2) Since Gaussian elimination is applied n times to at most $O(n^2)$ data, the total work is on the order of $O(n^3)$ floating-point operations (FLOPs) [9, 10].

B. Sparse-matrix LU

The matrix A itself is sparse: the block matrices outside the block tridiagonal are zero matrices, and within each diagonal block the entries are zero except along its diagonals [10]. Therefore:

1. Instead of storing the full matrix A, we can store only its nonzero entries in memory, which is done programmatically;
2. The collected nonzero entries are stored in a special sparse-matrix format: **scipy.sparse.diags()** and **scipy.tocsc()**;
3. Compute the LU factorization of the sparse matrix: **scipy.sparse.linalg.factorized()**;
4. Solve for the solution: **solve(R)** [7].

Advantage: Steps 1–3 are performed only once before the main computations, and when R changes only Step 4 needs to be repeated [10]. The computed solution differs from the exact one by machine precision (assuming no discretization error) [9]. Compared to the dense factorization, on average about $O(n^{3/2})$ floating-point operations and $O(n \log n)$ memory are required [10].

Disadvantage: As the grid becomes larger, during LU factorization the number of nonzero entries written into the L and U matrices can increase; eventually they may become denser than the original matrix A [10, 12].

IV. DISCRETE COSINE TRANSFORM (DCT)

Since homogeneous Neumann boundary conditions (2) hold on the boundaries of the square grid, the eigenvectors of the matrix A are

$$v_{ji}^{(k_y,k_x)} = \cos\left(\frac{\pi k_y (j-1)}{N_y - 1}\right) \times \cos\left(\frac{\pi k_x (i-1)}{N_x - 1}\right) \tag{10}$$

and the corresponding eigenvalues are

$$\lambda_{k_y,k_x} = \frac{2}{{h_y}^2}\left(1 - \cos\left(\frac{\pi k_y}{N_y - 1}\right)\right) + \frac{2}{{h_x}^2}\left(1 - \cos\left(\frac{\pi k_x}{N_x - 1}\right)\right), \tag{11}$$

where the grid indices satisfy $1 \le j \le N_y$ and $1 \le i \le N_x$, and the eigenvector (eigenvalue) indices satisfy $0 \le k_y \le N_y - 1$ and $0 \le k_x \le N_x - 1$ [13]. The DCT applies the $(N_y \cdot N_x) \times (N_y \cdot N_x)$ matrix Q constructed from the vectors (10) to P, R, and A

to obtain the spectral form of (5),

$$\Lambda \cdot \hat{P} = \hat{R}, \tag{12}$$

where $\Lambda = Q \cdot A \cdot Q^{-1}$ is a diagonal matrix containing the eigenvalues (11) (known), $\hat{P} = Q \cdot P$ is the vector P in spectral space (unknown), $\hat{R} = Q \cdot R$ is the vector R in spectral space (known), Q is the DCT matrix, and Q^{-1} is the inverse DCT matrix [13].
From (12), the unknown vector $\hat{P}$ is

$$\hat{P} = \Lambda^{-1} \cdot \hat{R}, \tag{13}$$

and therefore, if $\hat{P}$ and $\hat{R}$ are the vectorized forms of the matrices $\hat{p}_{k_y k_x}$ and $\hat{r}_{k_y k_x}$, then for any element with $(k_y, k_x) \neq (0, 0)$,

$$\hat{p}_{k_y,k_x} = \frac{\hat{r}_{k_y,k_x}}{\lambda_{k_y,k_x}}, \tag{14}$$

i.e., we solve a simple algebraic expression, where $\hat{p}_{k_y,k_x}$ is unknown, $\hat{r}_{k_y,k_x}$ is known, and λ_{k_y,k_x} is the diagonal element of Λ (known). Meanwhile, $\hat{p}_{0,0} = 0$ and $\hat{r}_{0,0} = 0$ (condition (7) in spectral space) [3, 4]. Applying the inverse DCT to the computed $\hat{P}$ gives the solution vector P:

$$P = Q^{-1} \cdot \hat{P}, \tag{15}$$

We perform the DCT using tools from the SciPy's **fft** submodule [7, 13]:

1. Evaluate (11) to construct the eigenvalue matrix **lambda**;
2. Apply the DCT to the matrix r_{ji} to obtain $\hat{r}_{ji}$: **r_hat = scipy.fft.dctn(r, type=1)**;
3. Compute $\hat{p}_{ji}$ using (14): **p_hat = r_hat / lambda**;
4. Enforce (7): **p_hat[0,0] = 0**;
5. Apply the inverse DCT (15) to obtain p_{ji}: **p = scipy.fft.idctn(p_hat, type=1)**.

Advantage: As with direct methods, it produces an answer that differs from the exact solution by machine precision in a finite number of steps (assuming no discretization error) [9, 13]. It requires $O(N_y \cdot N_x)$ memory. Although the matrix Q has size $(N_y \cdot N_x) \times (N_y \cdot N_x)$, the two-dimensional DCT can be viewed as a sequence of one-dimensional DCTs applied first along y and then along x, so it is sufficient to store only a single matrix of size $(N_y \cdot N_x)$. If $N_y = N_x = N$, each one-dimensional DCT requires $O(N \log N)$ operations, so the total work is $O(N^2 \log N)$ [14, 15].

Disadvantage: In the present implementation, the DCT method is restricted to the structured Cartesian Neumann benchmark considered here. For irregular geometries, non-separable operators, or more general boundary conditions, sparse direct or iterative methods may be more appropriate.

V. ITERATIVE METHODS

From the discrete Poisson equation (4), solving for the point (j, i) gives

$$p_{j,i} = \frac{1}{2(1+\delta)}\big(p_{j+1,i} + p_{j-1,i} + \delta\left(p_{j,i+1} + p_{j,i-1}\right) - {h_y}^2 r_{j,i}\big), \tag{16}$$

where $\delta = \dfrac{{h_y}^2}{{h_x}^2}$, $1 \le i \le N_x$, and $1 \le j \le N_y$.
Notice that, to compute the value at each point (j, i), neighboring points are used. On the grid boundary, the boundary conditions (2) are handled using ghost points [3].
The iterative methods considered here are based on (16) and repeatedly compute $p_{j,i}$. At each new iteration step, the computed value should move closer to the true solution. Let the iteration index be k. Then:

- Jacobi method (the k-th iterate): the average of the neighboring values from the previous iterate ($O(N_y N_x)$ operations):

$$p_{j,i}^{(k)} = \frac{1}{2(1+\delta)}\big(p_{j+1,i}^{(k-1)} + p_{j-1,i}^{(k-1)} + \delta\left(p_{j,i+1}^{(k-1)} + p_{j,i-1}^{(k-1)}\right) - {h_y}^2 r_{j,i}\big), \tag{17}$$

- Gauss–Seidel method (the k-th iterate): use neighboring values as soon as they are updated ($O(N_y N_x)$ operations):

$$p_{j,i}^{(k)} = \frac{1}{2(1+\delta)}\big(p_{j+1,i}^{(k)} + p_{j-1,i}^{(k)} + \delta\left(p_{j,i+1}^{(k)} + p_{j,i-1}^{(k)}\right) - {h_y}^2 r_{j,i}\big), \tag{18}$$

- Successive over-relaxation (SOR) method (the k-th iterate): an improved form of Gauss–Seidel. It uses both the previous value at the point and the newly updated neighboring values ($O(N_y N_x)$ operations):

$$p_{j,i}^{(k)} = p_{j,i}^{(k-1)} + \omega\left[\begin{array}{r} \frac{1}{2(1+\delta)}\big(p_{j+1,i}^{(k)} + p_{j-1,i}^{(k)} + \\ \delta\left(p_{j,i+1}^{(k)} + p_{j,i-1}^{(k)}\right) \\ -h_y^2 r_{j,i}\big) \\ -p_{j,i}^{(k-1)} \end{array}\right], \tag{19}$$

where $1 < \omega < 2$ is the relaxation parameter (used to accelerate convergence) [16, 17].

Starting from an initial guess at $k = 0$ (for example, $p_{ji}^{(0)} = 0$), we substitute into (17)–(19) to compute $p_{ji}^{(1)}$. We enforce (7) on the obtained solution. We then check the stopping condition (whether the error has reached the prescribed tolerance). If it has, the computation stops; if not, we repeat (17)–(19) for $k = 2, 3, \ldots$ until the condition is satisfied [18]. The relative error of the k-th iterate is

$$residual_k = \frac{\left\| r_{ji} - (\Delta p)_{ji}^{(k)} \right\|_2}{\left\| r_{ji} \right\|_2}, \tag{20}$$

the tolerance (linked to the grid size) is

$$tol \approx \frac{1}{{h_x}^2 + {h_y}^2},$$

and the stopping condition is $residual_k < tol$ [18, 19, 20]. On the other hand, sometimes the solution may not converge at all, or convergence may require too many iterations. In that case, an additional termination condition is needed; for example, we can impose an upper limit on the number of iterations, k_{max}. If the tolerance is not reached and $k = k_{max}$, we terminate the algorithm and mark the computation as "unsuccessful" [18].

Advantage: There is no need to store a very large matrix in memory. These methods can work not only on square grids but also on irregular grids with arbitrary boundary conditions [12, 21].

Disadvantage: Since these are approximate methods, they do not produce machine-precision answers like direct methods; even when the solution converges, the accuracy is limited by the prescribed tolerance. As the grid is refined, the computation time increases [12, 16, 21].

VI. COMPUTATION (IMPLEMENTATION)

The numerical methods were tested on square grids with $N_x = N_y$ for the following cases: small (17×17, 33×33, 65×65), medium (129×129, 257×257), large (513×513, 1025×1025), and very large (2049×2049). We focused on the asymptotic memory requirement and the elapsed time (number of operations).

The benchmark therefore concerns progressively larger instances of one and the same model problem rather than different physical case studies. In all timing experiments we used the unit-square problem defined by Eqs. (1), (2), and (4) together with the explicit source term (3). Peak memory was not instrumented and logged during the original runs; consequently, Table 1 reports measured runtimes, while the memory discussion below combines theoretical storage complexity with the observed out-of-memory thresholds.

To evaluate speed, for each method on a given grid we ran the solver at least three times and measured the mean runtime to produce a solution, using tools from Python's **time** module and NumPy:

- Start time: **start = time.perf_counter()**;
- End time: **stop = time.perf_counter()**;
- Elapsed time (s): **duration = stop - start**;
- Repeat 3–5 times and average (s): **duration_mean = numpy.mean(duration1, duration2, duration3, ...)** or **duration_mean = numpy.median(duration1, duration2, duration3, ...)**.

For the DCT, the SOR, and its Numba-accelerated variant, we performed 5–10 warm-up runs before the main timed runs. The runtime of these warm-up executions was not included in the reported measurements. Timing was started only after this preliminary stage in order to reduce startup, caching, and JIT-compilation effects and thereby obtain more representative runtimes.

For the Gauss–Seidel and SOR methods, we traversed the grid indices using a checkerboard (Red–Black) ordering. First the "white" cells are updated using the old values of the "black" cells; then the "black" cells are updated using the new values of the "white" cells. This ordering is standard for Red–Black smoothers and, in our implementation, is also convenient for parallel execution and NumPy vectorization [22, 21].

A. Numba JIT (Just-In-Time compiler)

Python is interpreted, whereas C/C++ and Fortran are compiled languages. In performance-critical kernels, compiled execution is typically faster than pure interpreted execution.

With the help of the Numba module, which has been developed since 2012, Python code can be compiled into machine code in a compiler-like manner. This makes it possible to accelerate parts of a Python program to the speed of C/C++ programs [8]. On modern personal computers with multi-core processors, Numba can run code in parallel using threads. In this work, we used 8 threads or 6 threads on a 2019 Intel i7 quad-core processor. For this purpose, we accelerated the

SOR method with Numba JIT. In doing so, we tested two variants of grid traversal:

- checkerboard (Red–Black) ordering;
- naive ordering.

[8, 23]. We also tested two ways of running the iterations: copying the entire result of the previous iterate, and partially copying element by element (in-place).

B. Successive over-relaxation method

For the SOR method, it is important to find an appropriate (optimal) value of the relaxation parameter ω. This is the ω for which the solution converges the fastest. When $\omega = 1$, (19) reduces to the Gauss–Seidel method, and when $\omega = 2$, the computation often diverges. Therefore, the appropriate value lies in $\omega_{opt} \in [1.0, 2.0]$ [16, 17]. For the Poisson equation on a square grid, approximately

$$\omega_{opt} \sim \frac{2.0}{\left(1.0 + \sin\left(\frac{\pi}{N}\right)\right)}, \tag{21}$$

where $N = N_x = N_y$ [16, 17]. However, in our case, empirically searched values of ω_{opt} in the interval $[1.8, 2.0[$ led to faster convergence.

We performed the search in two stages. In the first "coarse" stage, we varied ω over the given interval using a relatively large step and recorded the cases that converged successfully. In the second "fine" stage, we selected one or two ω values that gave the shortest runtime among the successful cases, and then searched around them using a much smaller step size. The value with the minimum runtime is taken as $\omega = \omega_{opt}$.

In our observations, empirical factors influencing whether an appropriate value is found include:

- The cadence of computing the residual: computing the relative error at every iteration is accurate but can significantly increase the measured runtime when searching ω on a fine scale. Instead, computing the error every 20th, 30th, 50th, or 100th iteration (when the error decreases smoothly over time) can save time.
- The maximum iteration limit (iter_max): values such as 5000, 10000, 20000, …, 100000. If it is too small, the method may stop before reaching the prescribed tolerance (unsuccessful computation). If it is too large, time may be wasted by running all iterations when the tolerance is not reachable (also unsuccessful). In any case, if the relative error reaches the prescribed tolerance before this limit, the computation is considered successful; otherwise, the computation is considered unsuccessful and is stopped immediately.
- For Numba JIT SOR, the **cache** option of the **@njit** decorator:
  @njit(Cache=True) or **@njit(cache=False)**.

For example, when implementing SOR with Numba JIT using naive ordering on a (1025×1025) grid,

1. with cadence = 100 and $iter\_max = 20000$, the coarse search did not find a reference ω value (the iteration limit was reached without satisfying the stopping condition);
2. with cadence = 100 and $iter\_max = 30000$, the coarse search converged at $\omega_i = 1.995$, and all other values led to an unsuccessful computation;
3. performing a fine search around $\omega_i = 1.995$ yielded many values where the solution converged. Among them, we selected the value with the minimum runtime as ω_{opt} and used it in the comparative study.

VII. RESULTS AND DISCUSSION

This section reports the benchmark results for the model problem defined by Eqs. (1)–(3), namely the two-dimensional Poisson equation with homogeneous Neumann boundary conditions on square Cartesian grids. The reported measurements therefore compare how different solvers behave on the same discretized problem family as the grid is refined.

The main goal of this study is a comparative analysis of the performance of numerical methods; therefore, we present the measured results together with their interpretation.

The mean runtimes measured for each method are shown in Table 1. The values are given in seconds. Log–log plots constructed from the measured values are shown in Figures 2 and 3 for the direct and iterative methods, respectively.

From the results for the direct methods, DLU and SLU are the fastest on small grids, and sufficiently fast on medium grids ($t < 10^{-1}$ s). However, as the grid size increases, DLU quickly runs out of memory (Table 1) and its runtime grows sharply (Figure 2). In contrast, SLU runs out of memory only on very large grids (Table 1), and its runtime

N_y x N_x	DLU	SLU	DCT	Jacobi	GS (RB)	SOR (RB)	SOR Numba	SOR Numba (RB)
17 x 17	1.48E-04	5.99E-05	x	5.00E-02	3.95E-02	4.98E-03	x	x
33 x 33	4.32E-04	1.37E-04	x	2.59E-01	2.08E-01	2.01E-02	3.38E-02	9.51E-02
65 x 65	1.15E-02	1.08E-03	2.98E-04	1.72E+00	1.28E+00	8.65E-02	6.74E-02	2.56E-01
129 x 129	1.70E-01	3.91E-03	1.26E-03	1.96E+01	1.37E+01	7.38E-01	1.42E-01	7.78E-01
257 x 257	OOM	1.77E-02	4.90E-03	x	x	7.50E+00	5.55E-01	2.99E+00
513 x 513	OOM	7.99E-02	2.03E-02	x	x	1.14E+02	3.00E+00	1.91E+01
1025 x 1025	OOM	2.90E-01	1.15E-01	x	x	x	2.79E+01	x
2049 x 2049	OOM	OOM	4.84E-01	x	x	x	x	x

Table 1. Mean runtime (s) of each numerical method on each grid. Methods: DLU — dense LU; SLU — sparse LU; DCT — discrete cosine transform; Jacobi — Jacobi method; GS (RB) — Gauss–Seidel (Red–Black order); SOR (RB) — SOR (Red–Black order); SOR Numba — Numba-accelerated SOR (naive order); SOR Numba (RB) — Numba-accelerated SOR (Red–Black order). Values: OOM — out of memory; x — not computed or unsuccessful computation.

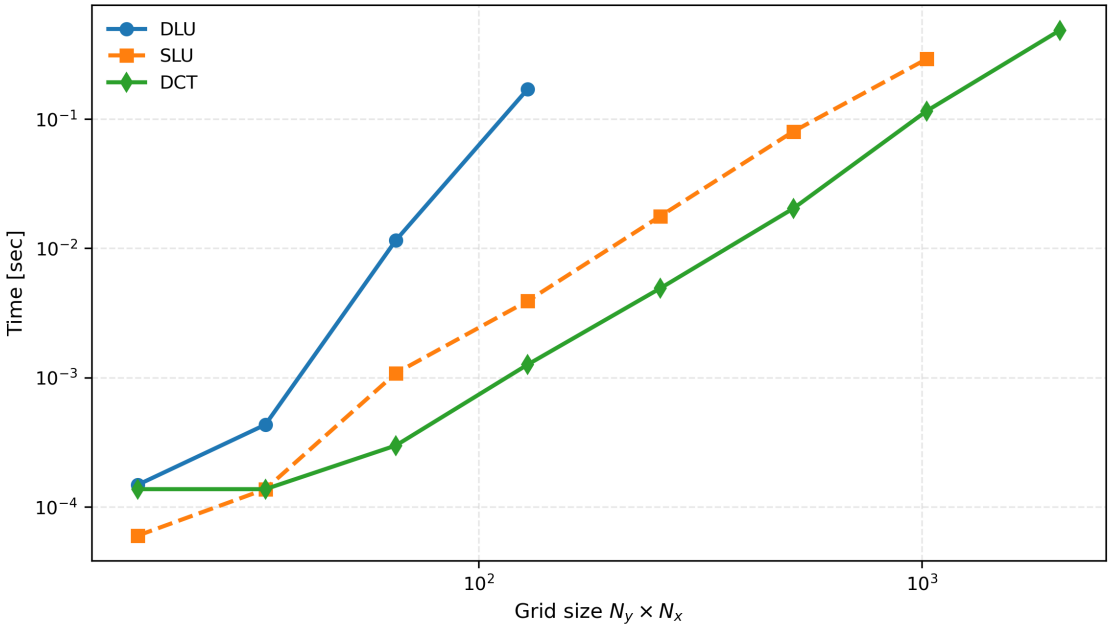

Figure 2. Runtime versus grid size (direct methods)

grows more slowly than DLU (Figure 2). For medium, large, and very large grids, the DCT method does not operate with the Laplacian matrix and is therefore memory efficient; its growth trend is similar to SLU but it is about 2–3 times faster on average, as seen in Table 1 and Figure 2. Even on a 2049 × 2049 grid it requires on average $t < 10^{-1}$ s. We did not test DCT on the smallest grids.

From the results for the iterative methods, the Jacobi and Gauss–Seidel methods can be used only on small and moderately sized grids (Table 1). Beyond that, the solution does not converge or convergence requires an excessively long time, so the methods lose practical significance; this is reflected in the runtime growth trend (Figure 3). The SOR method is faster than the previous two and can be used on somewhat larger grids, but its growth trend is similar (Figure 3); here computations were performed using the optimal relaxation parameter ω_{opt}. In particular, on the smallest 17 × 17 grid it is the fastest among all iterative methods. We did not test the Numba-accelerated SOR variants on the 17 × 17 grid (Table 1). On the 33 × 33 grid, the Numba SOR (naive order) is similar to SOR, but starting from the 65 × 65 grid its growth trend differs significantly and it becomes faster (Figure 3). Numba SOR (RB) has a similar trend to Numba SOR but is slower on average. As the grid becomes large (513 × 513 and above), both variants lose practical significance (Table 1).

These measured tendencies are consistent with the method-level strengths and limitations discussed earlier. Dense LU is penalized mainly by memory growth, sparse LU benefits from exploiting the matrix sparsity pattern, and the DCT benefits from diagonalization of the structured Neumann prob-

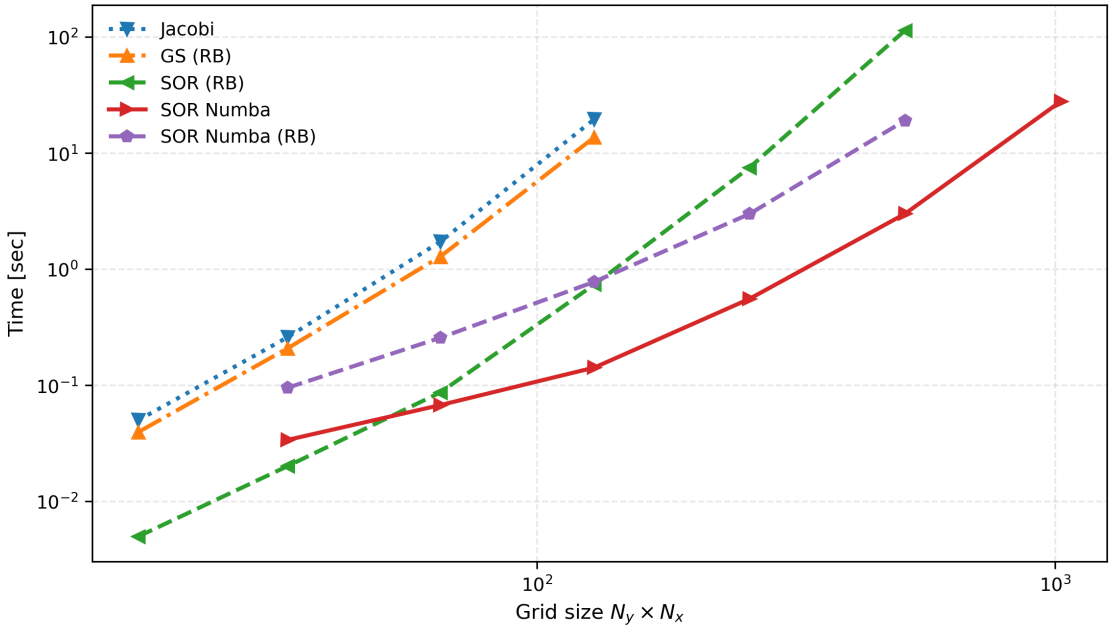

Figure 3. Runtime versus grid size (iterative methods)

lem. By contrast, the iterative methods avoid large matrix storage and remain conceptually flexible, but their iteration counts grow unfavorably with grid refinement, which is directly reflected in the timings. This same flexibility is also the reason why iterative methods may still be preferable on irregular grids or under more general boundary conditions, even though they are not the fastest option for the structured benchmark considered here.

In summary, the direct methods outperform the iterative methods on small, medium, and large grids for the structured benchmark studied here. Among them, the DCT is the strongest performer on moderate and larger grids, whereas sparse LU is the simplest robust high-performance option to deploy in Python. Dense LU is of limited practical significance because its memory requirements are manageable only on small grids. The fastest iterative method considered in this work cannot outperform sparse LU in either speed or accuracy for the present structured Neumann problem.

Nevertheless, the DCT method should not be interpreted as the universally best Poisson solver irrespective of the problem setting. Its advantage in Table 1 relies on the square Cartesian grid, homogeneous Neumann boundary conditions, and the separability of the discrete operator. For the next stage of this research, namely incompressible lid-driven cavity simulations in primitive variables, sparse LU and DCT remain the most suitable candidates among the methods tested here [1, 2]. It should also be noted that the present solver ranking is established for the collocated-grid finite-difference Poisson operator studied here. In the planned lid-driven cavity solver, the pressure equation will be posed on a staggered grid, so the preferred method identified in this work will require adaptation before being transferred directly to that setting.

The Numba-accelerated SOR method may still be a viable choice in the following cases:

- when the processor capability is high (e.g., 10 or more cores);
- when memory is insufficient;
- when considering an irregular grid or more general geometric settings.

[21].

VIII. CONCLUSIONS

This work benchmarked several simple solvers for the same second-order finite-difference discretization of the two-dimensional Poisson equation with homogeneous Neumann boundary conditions on square Cartesian grids. Within this benchmark setting, the DCT-based solver provided the best overall combination of runtime and memory efficiency, while sparse LU was the most practical general-purpose direct method to implement and use in Python. Dense LU was restricted to small grids by memory cost, and the stationary iterative

methods became progressively less competitive as the grid was refined.

The principal conclusion is therefore conditional rather than universal: for the structured Neumann problem considered here, DCT is the best choice among the tested methods, but this ranking should not be generalized automatically to all Poisson problems. A natural extension of the present work would be to add measured peak-memory profiling and to test the same methods on irregular grids and under more general boundary-condition settings.

REFERENCES

[1] J. D. Anderson. *Computational Fluid Dynamics: The Basics with Applications*. McGraw-Hill, 1995.

[2] J. H. Ferziger, M. Perić, and R. L. Street. *Computational Methods for Fluid Dynamics*. Springer, 4 edition, 2020.

[3] U. Schumann and R. A. Sweet. A direct method for the solution of poisson's equation with neumann boundary conditions on a staggered grid of arbitrary size. *Journal of Computational Physics*, 20(2):171–182, 1976.

[4] L. C. Evans. *Partial Differential Equations*. American Mathematical Society, 2 edition, 2010.

[5] W. A. Strauss. *Partial Differential Equations: An Introduction*. Wiley, 2 edition, 2007.

[6] C. R. Harris, K. J. Millman, S. J. van der Walt, et al. Array programming with NumPy. *Nature*, 585:357–362, 2020.

[7] P. Virtanen, R. Gommers, T. E. Oliphant, et al. Scipy 1.0: Fundamental algorithms for scientific computing in python. *Nature Methods*, 17(3):261–272, 2020.

[8] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A LLVM-based python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 2015.

[9] James W. Demmel. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, 1997.

[10] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 4 edition, 2013.

[11] NumPy Developers. Numpy reference manual (stable). https://numpy.org/doc/stable/, 2025.

[12] W. Hackbusch. *Iterative Solution of Large Sparse Systems of Equations*. Springer, 1994.

[13] Gilbert Strang. The discrete cosine transform. *SIAM Review*, 41(1):135–147, 1999.

[14] R. W. Hockney. A fast direct solution of poisson's equation using fourier analysis. *Journal of the ACM*, 12(1):95–113, 1965.

[15] P. N. Swarztrauber. Cyclic reduction, fourier analysis and FACR for poisson's equation. *SIAM Review*, 19(3):490–501, 1977.

[16] S. P. Frankel. Convergence rates of iterative treatments of PDEs. *Mathematics of Computation*, 4:65–75, 1950.

[17] D. M. Young. *Iterative Solution of Large Linear Systems*. Academic Press, 1971.

[18] Anne Greenbaum. *Iterative Methods for Solving Linear Systems*. Society for Industrial and Applied Mathematics, 1997.

[19] R. Barrett, M. Berry, T. F. Chan, et al. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Society for Industrial and Applied Mathematics, 2 edition, 1994.

[20] A. Iserles. *A First Course in the Numerical Analysis of Differential Equations*. Cambridge University Press, 1996.

[21] U. Trottenberg, C. W. Oosterlee, and A. Sch"uller. *Multigrid*. Academic Press, 2001.

[22] S. V. Parter. Estimates for multigrid based on red–black GS. *Numerische Mathematik*, 52:701–723, 1988.

[23] Numba Developers. Numba user guide (stable). https://numba.readthedocs.io/, 2025.