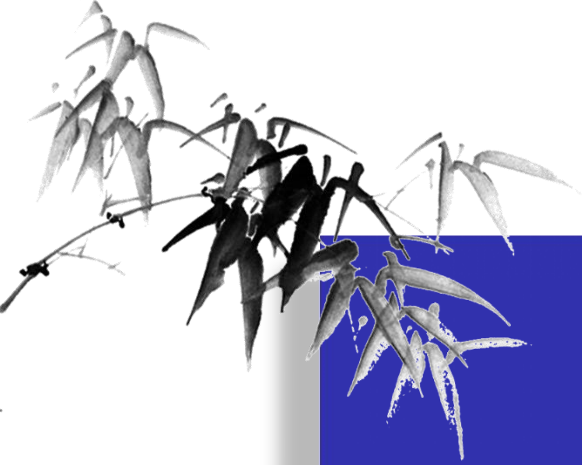
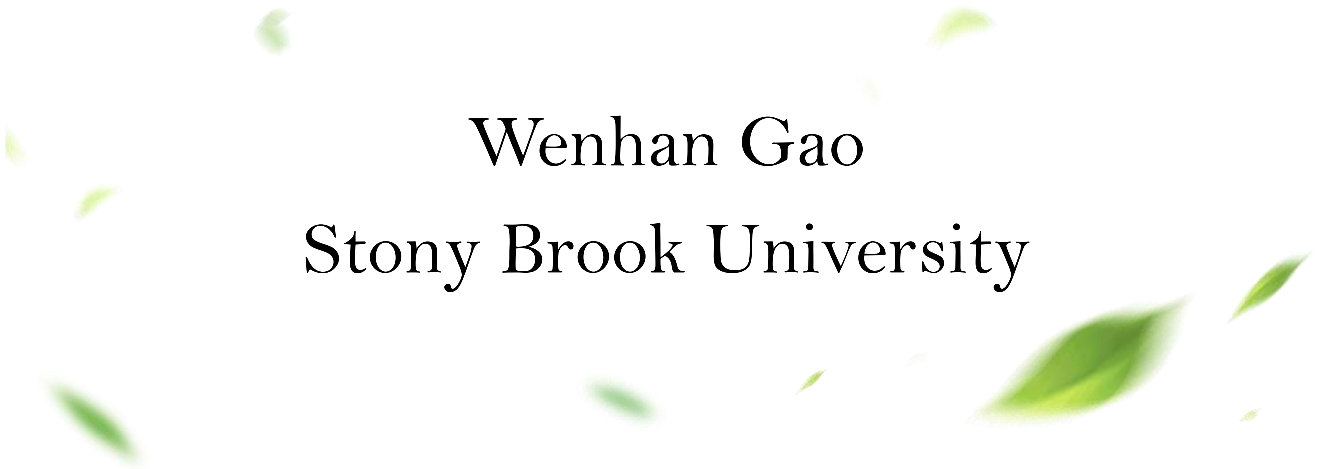
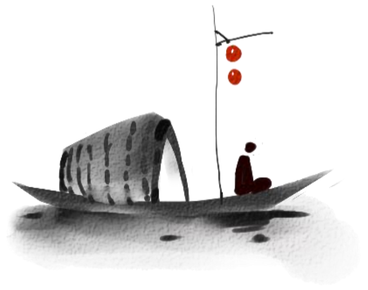


AMS 530 Final Project
Parallel 2D Poisson Equation Solver



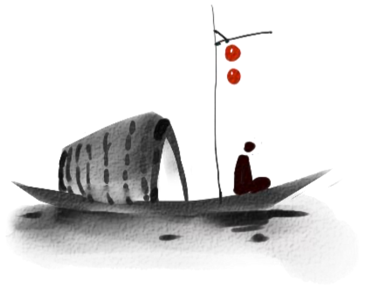
Wenhan Gao

Stony Brook University



Introduction and Motivation

- Physical laws often take the form of partial differential equations (PDEs), e.g., the Hamilton-Jacobi-Bellman (HJB) equation in control theory and the Schrodinger equation in quantum mechanics.
- There are various traditional numerical methods for solving PDEs; for example, the Finite Difference Method. However, when the grid size is large, it can be computationally intractable. Therefore, it is imperative to parallelize the algorithm to distribute the work to multiple processors.



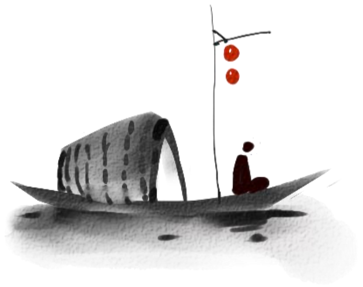
Problem Description

Consider a 2D Poisson's equation with Dirichlet boundary conditions:

$$\begin{aligned} \Delta u(x,y) &= f(x,y), && \text{in } \Omega := (0,1)\times(0,1), \\ u(x,y) &= g(x,y), && \text{on } \partial\Omega, \end{aligned}$$

where Δ is the Laplacian operator, $f(x) = \sin(x-y)$, and $g(x,y) = 1$.

- In this report, the PDE above will be solved by finite difference in parallel on three different grids, 100 × 100, 200 × 200, and 400 × 400, using P = 2, 10, 20 processors.



Sequential FDM: Discretization

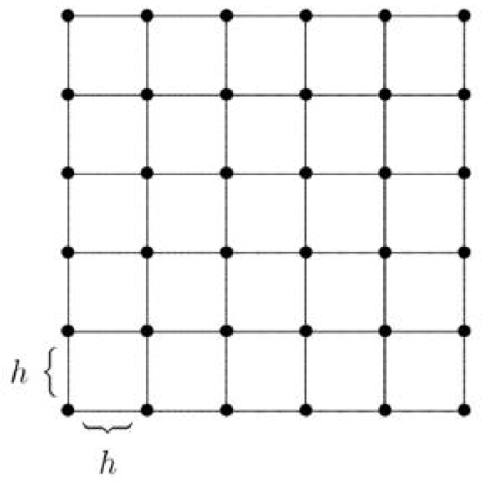
- Discretization:

The unit square domain can be discretized as an $N \times N$ matrix:

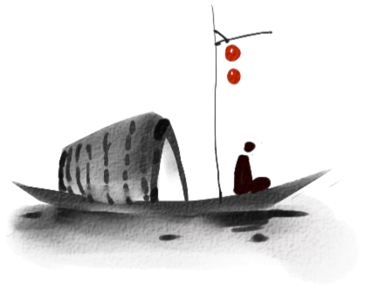
$$x_i = i \cdot h,\ y_j = j \cdot h,\ i, j \in [0, N],$$

where the step size $h = dx = dy = \frac{1}{N}$, and

$$u_{i,j} = u\left(x_i, y_j\right),\ f_{i,j} = f\left(x_i, y_j\right) = sin(x_i - y_j).$$



- We wish to find the values of u(x, y) on this $N \times N$ grid.
- Values on the boundary, i.e., 4 edges of the square is known. Therefore, there are $(N-2) \times (N-2)$ unkown grid values.



Sequential FDM: Discretization

- By central difference approximation, for an interior grid point:

$$\frac{u_{i+1,j}+u_{i-1,j}-2u_{i,j}}{h^2}+\frac{u_{i,j+1}+u_{i,j-1}-2u_{i,j}}{h^2}=\sin\left(x_i-y_j\right)$$

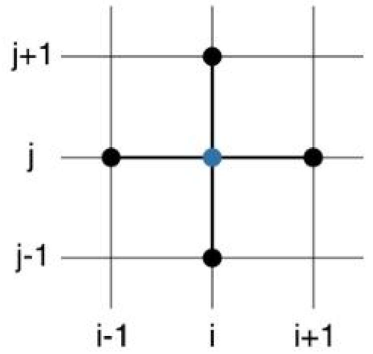
Overall, we have a linear system of $(\mathrm{N}-2)^2$ equations with $(\mathrm{N}-2)^2$ unknown variables, which can be written in the form of $\mathrm{Au}=\mathrm{b}$ as shown on the right.

$$\left[\begin{array}{cccc|cccc|cccc|cccc}
4 & -1 & & & -1 & & & & & & & & & & & \\
-1 & 4 & -1 & & & -1 & & & & & & & & & & \\
& -1 & 4 & -1 & & & -1 & & & & & & & & & \\
& & -1 & 4 & & & & -1 & & & & & & & & \\
\hline
-1 & & & & 4 & -1 & & & -1 & & & & & & & \\
& -1 & & & -1 & 4 & -1 & & & -1 & & & & & & \\
& & -1 & & & -1 & 4 & -1 & & & -1 & & & & & \\
& & & -1 & & & -1 & 4 & & & & -1 & & & & \\
\hline
& & & & -1 & & & & 4 & -1 & & & -1 & & & \\
& & & & & -1 & & & -1 & 4 & -1 & & & -1 & & \\
& & & & & & -1 & & & -1 & 4 & -1 & & & -1 & \\
& & & & & & & -1 & & & -1 & 4 & & & & -1 \\
\hline
& & & & & & & & -1 & & & & 4 & -1 & & \\
& & & & & & & & & -1 & & & -1 & 4 & -1 & \\
& & & & & & & & & & -1 & & & -1 & 4 & -1 \\
& & & & & & & & & & & -1 & & & -1 & 4
\end{array}\right] * \begin{bmatrix} U(1,1)\\ U(2,1)\\ U(3,1)\\ U(4,1)\\ U(1,2)\\ U(2,2)\\ U(3,2)\\ U(4,2)\\ U(1,3)\\ U(2,3)\\ U(3,3)\\ U(4,3)\\ U(1,4)\\ U(2,4)\\ U(3,4)\\ U(4,4) \end{bmatrix} = \begin{bmatrix} b(1,1)\\ b(2,1)\\ b(3,1)\\ b(4,1)\\ b(1,2)\\ b(2,2)\\ b(3,2)\\ b(4,2)\\ b(1,3)\\ b(2,3)\\ b(3,3)\\ b(4,3)\\ b(1,4)\\ b(2,4)\\ b(3,4)\\ b(4,4) \end{bmatrix}$$

Sequential FDM: Five-point Stencil

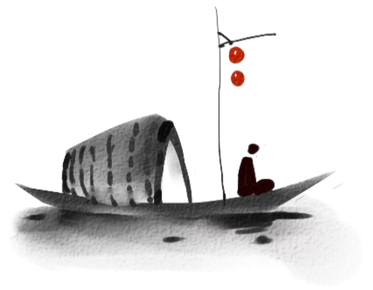
Since this system of linear equations is sparse and strictly diagonally dominant, Jacobi method [4] can be applied to solve this system of equations. The updates for interior grid points ($u_{i,j}, i, j \in [1, N-1]$) are:

$$u_{i,j}^{n+1} = \frac{1}{4}(u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n - f_{i,j} \cdot h^2). \tag{2.1}$$



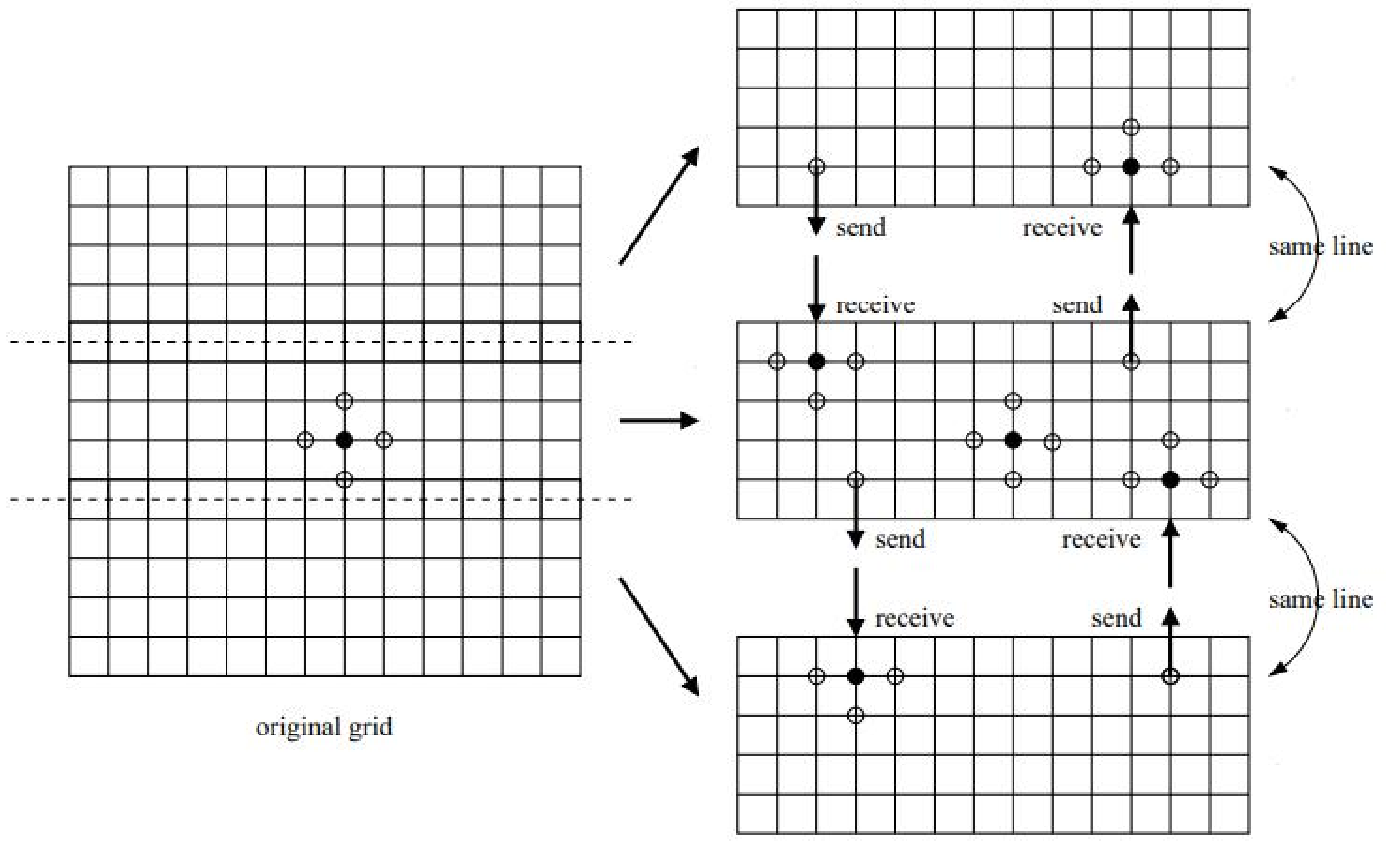
Start with an initial guess and keep applying Jacobi updates, the grid values will eventually converge to the solution of the linear system.

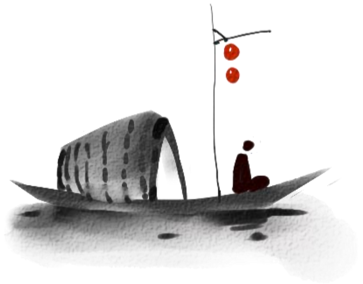
1. For the update of an interior grid point, we need the values from its 4 neighbours.
2. Each grid point's update is independent from other points' updates.



Parallelization: Domain Decomposition

- In this report, we are going to parallelize the Jacobi method using so called "domain decomposition". The domain is decomposed into P "stripes".

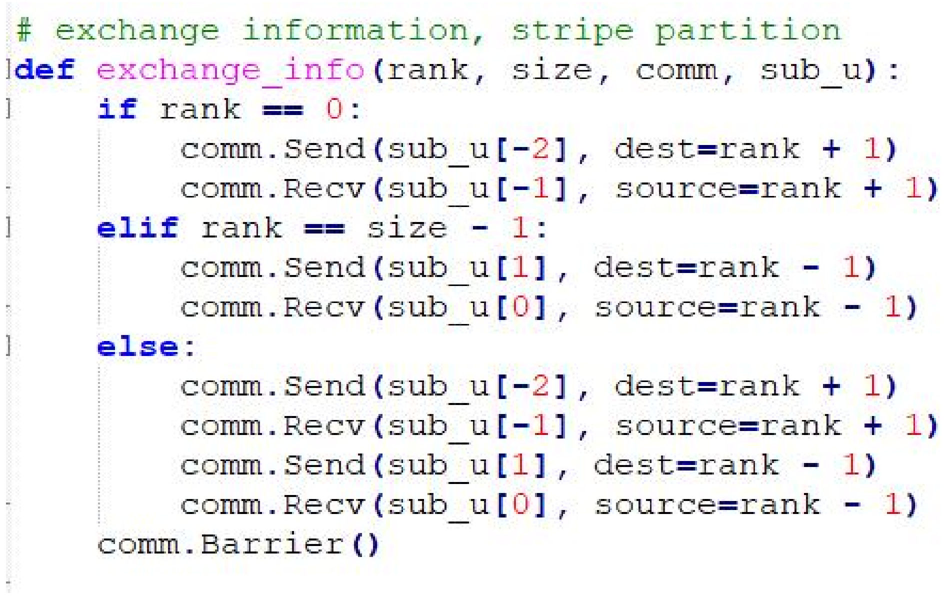




Parallelization: Implementation

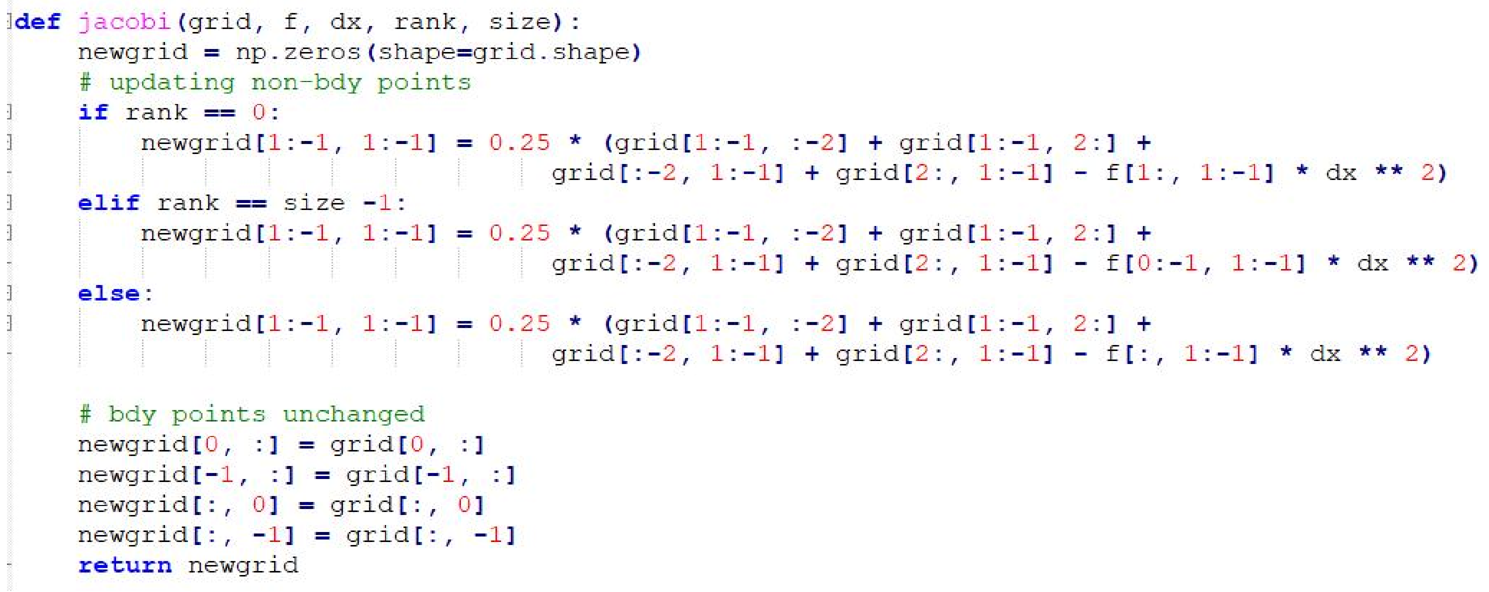
- In a nutshell, in the parallel Jacobi algorithm, we first scatter the solution matrix u(x, y), and then there are two steps in each iteration: Information Exchange + Local Jacobi Updates. Finally, when the convergence is reached, we gather the solution matrix to the root processor.

Information Exchange



```python
# exchange information, stripe partition
def exchange_info(rank, size, comm, sub_u):
    if rank == 0:
        comm.Send(sub_u[-2], dest=rank + 1)
        comm.Recv(sub_u[-1], source=rank + 1)
    elif rank == size - 1:
        comm.Send(sub_u[1], dest=rank - 1)
        comm.Recv(sub_u[0], source=rank - 1)
    else:
        comm.Send(sub_u[-2], dest=rank + 1)
        comm.Recv(sub_u[-1], source=rank + 1)
        comm.Send(sub_u[1], dest=rank - 1)
        comm.Recv(sub_u[0], source=rank - 1)
    comm.Barrier()
```

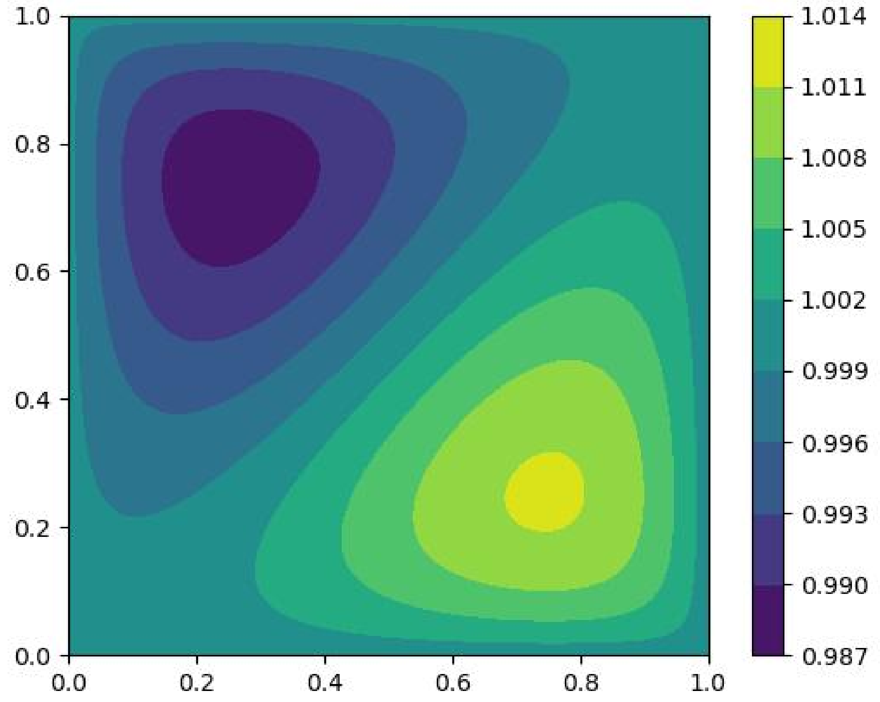
Jacobi Updates



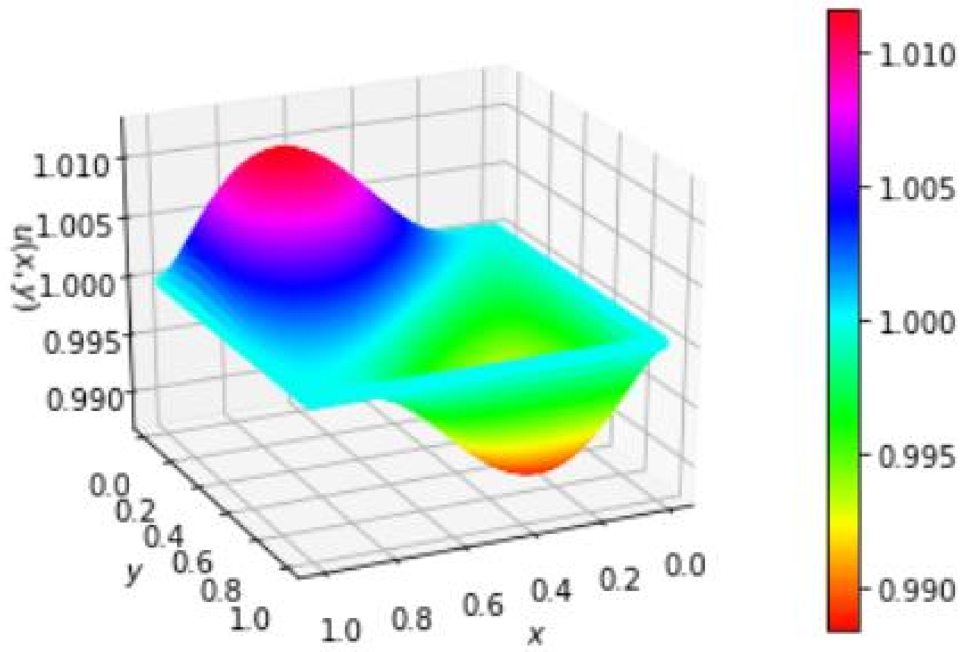
```python
def jacobi(grid, f, dx, rank, size):
    newgrid = np.zeros(shape=grid.shape)
    # updating non-bdy points
    if rank == 0:
        newgrid[1:-1, 1:-1] = 0.25 * (grid[1:-1, :-2] + grid[1:-1, 2:] +
                                       grid[:-2, 1:-1] + grid[2:, 1:-1] - f[1:, 1:-1] * dx ** 2)
    elif rank == size -1:
        newgrid[1:-1, 1:-1] = 0.25 * (grid[1:-1, :-2] + grid[1:-1, 2:] +
                                       grid[:-2, 1:-1] + grid[2:, 1:-1] - f[0:-1, 1:-1] * dx ** 2)
    else:
        newgrid[1:-1, 1:-1] = 0.25 * (grid[1:-1, :-2] + grid[1:-1, 2:] +
                                       grid[:-2, 1:-1] + grid[2:, 1:-1] - f[:, 1:-1] * dx ** 2)

    # bdy points unchanged
    newgrid[0, :] = grid[0, :]
    newgrid[-1, :] = grid[-1, :]
    newgrid[:, 0] = grid[:, 0]
    newgrid[:, -1] = grid[:, -1]
    return newgrid
```

Results: Solution Plots

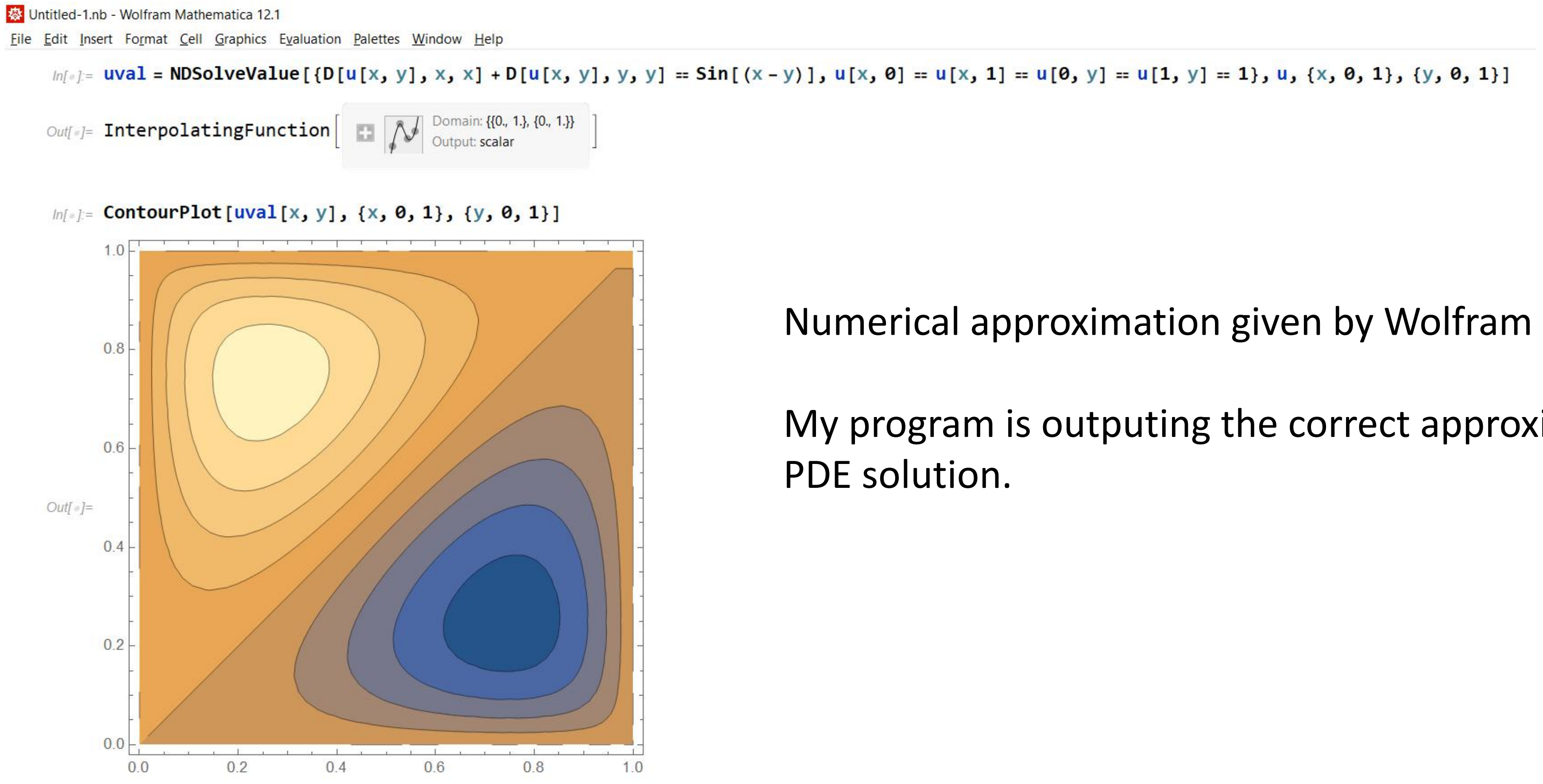
Solution Contour Plot



Solution 3D Plot



Results: Correctness Verification


Untitled-1.nb - Wolfram Mathematica 12.1
File Edit Insert Format Cell Graphics Evaluation Palettes Window Help

```
In[•]:= uval = NDSolveValue[{D[u[x, y], x, x] + D[u[x, y], y, y] == Sin[(x - y)], u[x, 0] == u[x, 1] == u[0, y] == u[1, y] == 1}, u, {x, 0, 1}, {y, 0, 1}]

Out[•]= InterpolatingFunction[ Domain: {{0., 1.}, {0., 1.}} Output: scalar ]

In[•]:= ContourPlot[uval[x, y], {x, 0, 1}, {y, 0, 1}]

Out[•]=
```

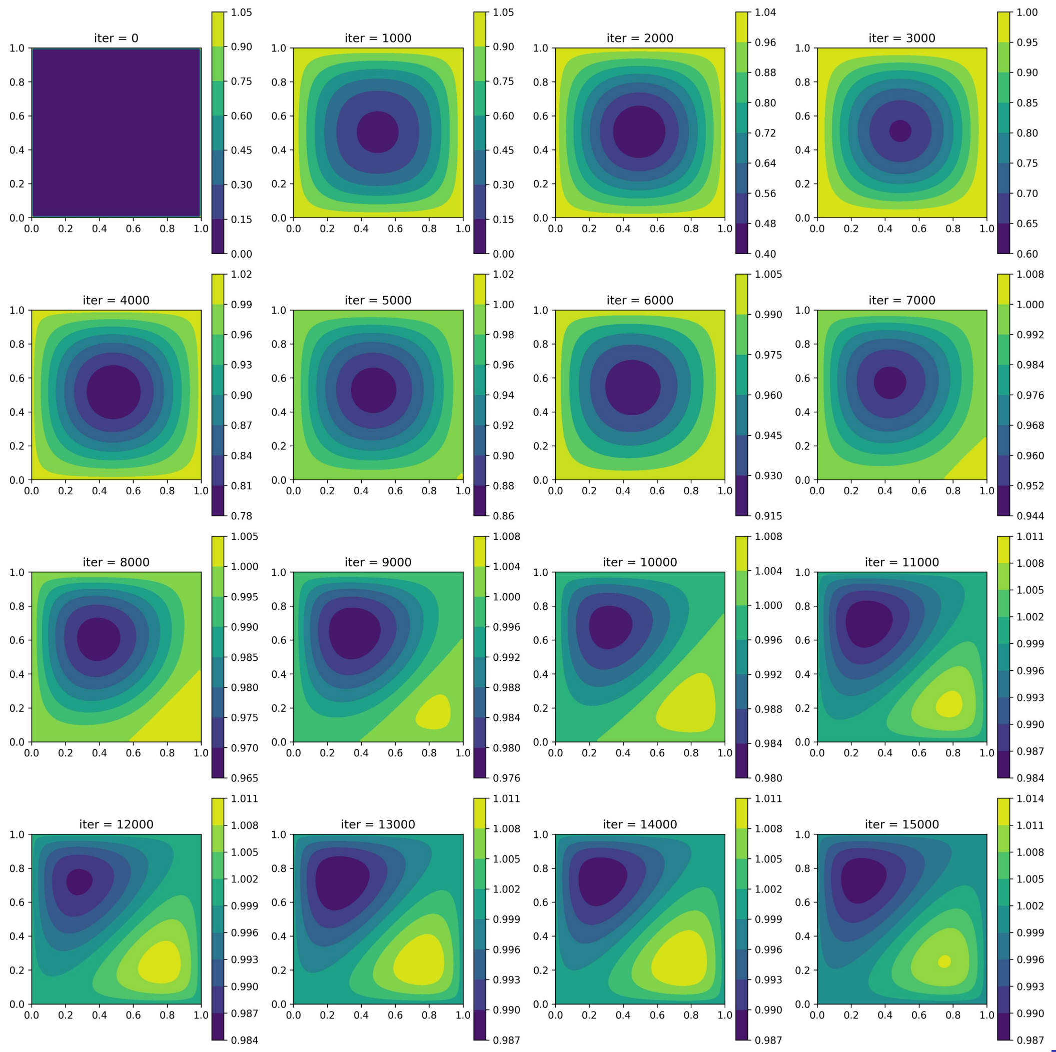
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

Numerical approximation given by Wolfram

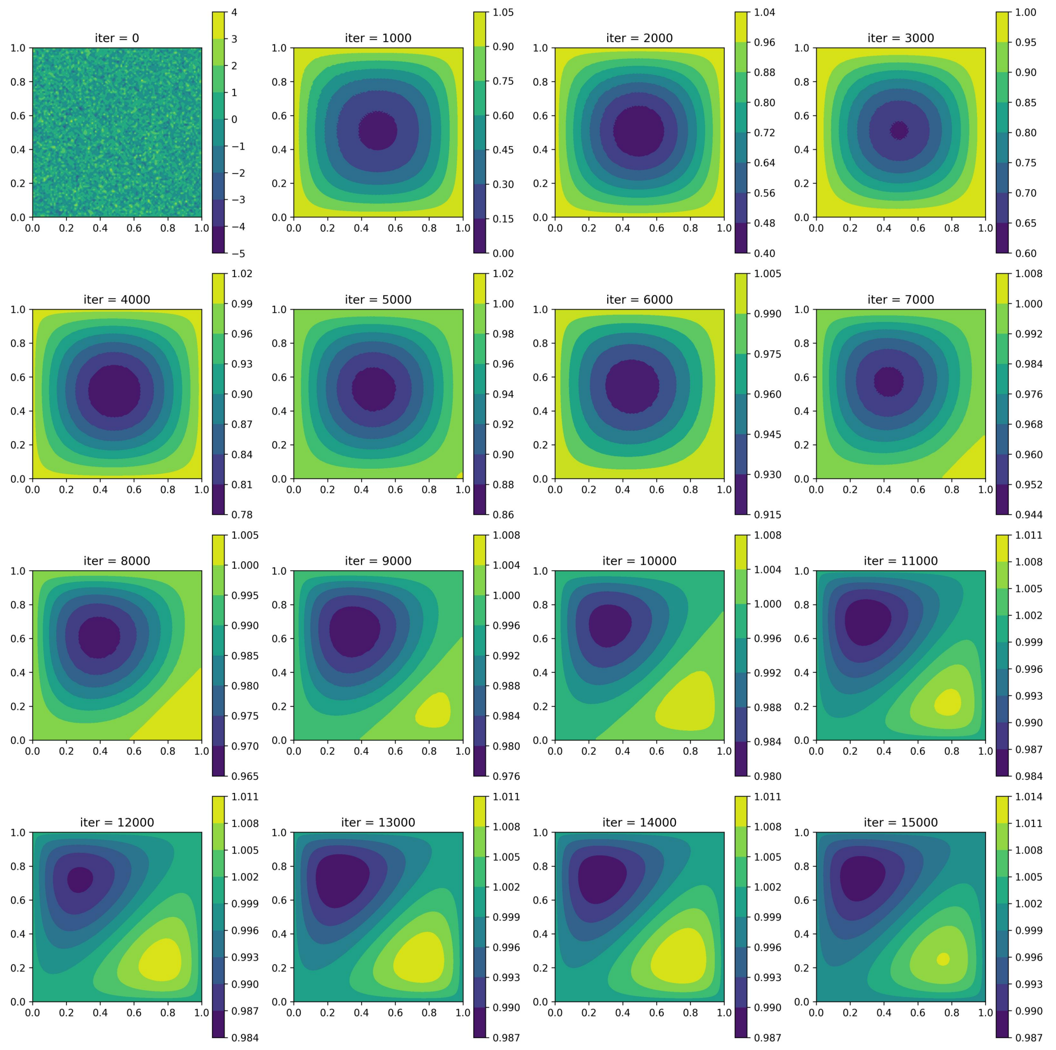
My program is outputing the correct approximation to the PDE solution.

Results: Convergence of Jacobi Method

Initialized with all 0s



Initialized with std. normal



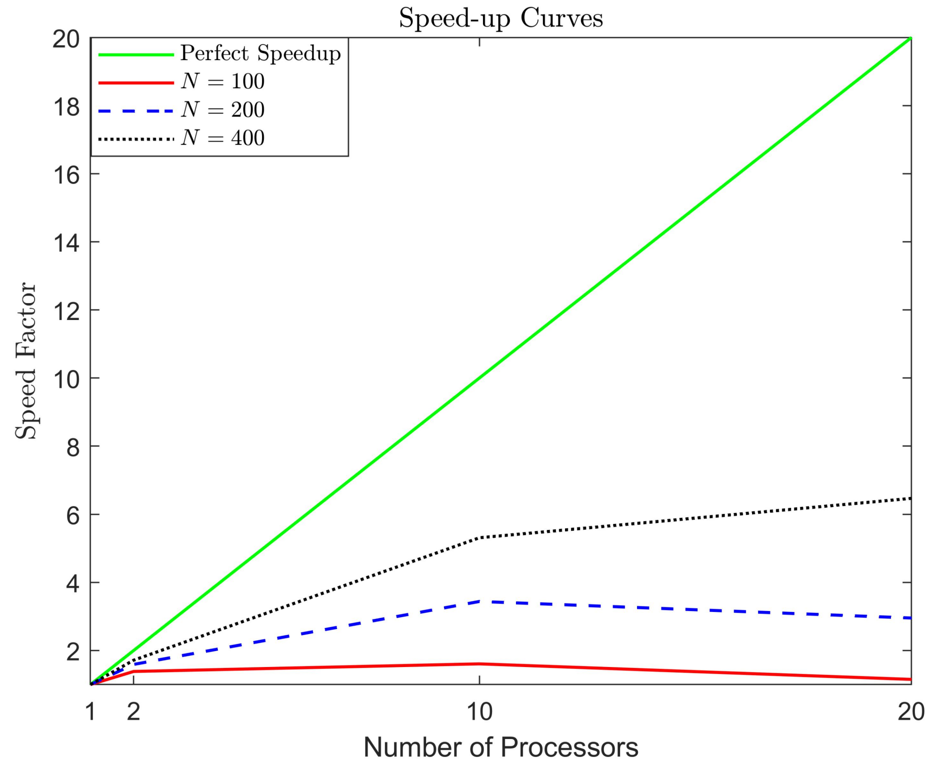
Results: Parallel Speed-up Results

	$P = 1$	$P = 2$	$P = 10$	$P = 20$
$N = 100$	2.679693	1.934774	1.666728	2.326345
$N = 200$	28.899304	18.155115	8.402078	9.783527
$N = 400$	354.117078	206.414760	66.682876	54.755312

Table 1: Running Time in Seconds

	$P = 2$	$P = 10$	$P = 20$
$N = 100$	1.385	1.607	1.152
$N = 200$	1.592	3.440	2.954
$N = 400$	1.716	5.310	6.467

Table 2: Speed-up Factors $S(P, N) = \frac{T(1,N)}{T(P,N)}$



The speed-up gets better when the scale gets larger; this happens because the communication cost becomes less significant compared to the computation cost. Therefore, my program is efficient and scalable.
We can expect almost a linear speed up asymptotically when N goes to infinity.

Thank you