Practical Numerical Simulations: Assignment 3

Kevin Grainger

November 2024

Contents

1 Problem 1

1.1 Theory

The Laplace equation is classified as an elliptic PDE (discriminant < 0). Elliptical PDE's allow the construction of a solution using iterative methods, this is because they have a stable and smooth solution (making iterative updates accurate). As well as this the boundary conditions of Elliptical PDE's dictate the entire solution of our PDE's domain, allowing for the construction of the domain ω starting with only the boundary conditions

Constructing the numerical solution to this PDE first involves the use of a point based grid system. We need to imagine the domain (ω) of our two dimensional PDE as a square of Area = 1. The borders of this unit square represent our boundary conditions. This grid will contain n^2 points, the value of which is calculated by the average of the four points surrounding (x+h,x-h,y+h,y-h). The Dirichlet boundary values can start this process off.

In this exercise we will produce a sparse linear system, meaning a matrix with values concentrated on and neighboring the diagonal. There are numerous methods for numerically solving these types of systems, in our case we will be using an altered version of the Gauss-Seibel method called, 'Successive Over-Relaxation. This method relies on an empirical weighting of elements in the Gauss-Seibel method. This weighting factor, called the 'relaxation factor', greatly speeds up convergence. Once our system is converged we will have an overall solution for our ϕ over domain ω.I will derive and outline the details of our method below:

1.1.1 Finite Difference Method

The use of the finite difference method imposes a grid on our chosen PDE domain. We can then approximate Laplace's equation at each grid-point by averaging the 4 nearest neighbors.
To derive this, we can start by finding an approximate expression for the 4 nearest neighbor points, separated bu 'h'. By Taylor-expansion:

$$\phi(x+h,y) = \phi(x,y) + h\frac{\partial\phi}{\partial x} + \frac{1}{2}h^2\frac{\partial^2\phi}{\partial x^2} + \frac{1}{6}h^3\frac{\partial^3\phi}{\partial x^3} + O(h^4) \tag{1}$$

$$\phi(x-h,y) = \phi(x,y) - h\frac{\partial\phi}{\partial x} + \frac{1}{2}h^2\frac{\partial^2\phi}{\partial x^2} - \frac{1}{6}h^3\frac{\partial^3\phi}{\partial x^3} + O(h^4) \tag{2}$$

$$\phi(x,y+h) = \phi(x,y) + h\frac{\partial\phi}{\partial y} + \frac{1}{2}h^2\frac{\partial^2\phi}{\partial y^2} + \frac{1}{6}h^3\frac{\partial^3\phi}{\partial y^3} + O(h^4) \tag{3}$$

$$\phi(x,y-h) = \phi(x,y) - h\frac{\partial\phi}{\partial y} + \frac{1}{2}h^2\frac{\partial^2\phi}{\partial y^2} - \frac{1}{6}h^3\frac{\partial^3\phi}{\partial y^3} + O(h^4) \tag{4}$$

Adding equations (1),(2) and (3)(4).

$$\phi(x+h,y) + \phi(x-h,y) = 2\phi(x,y) + h^2\frac{\partial^2\phi}{\partial x^2} + O(h^4) \tag{5}$$

$$\phi(x,y+h) + \phi(x,y-h) = 2\phi(x,y) + h^2\frac{\partial^2\phi}{\partial y^2} + O(h^4) \tag{6}$$

Adding these equations (5)(6).

$$\phi(x,y) = \frac{1}{4}[\phi(x+h,y) + \phi(x-h,y) + \phi(x,y+h) + \phi(x,y-h) + (h^2(\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2})) + O(h^4)] \quad (7)$$

We are working with the Laplace equation, so we know:

$$\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = 0 \quad (8)$$

$$\phi(x,y) = \frac{1}{4}[\phi(x+h,y) + \phi(x-h,y) + \phi(x,y+h) + \phi(x,y-h) + O(h^4)] \quad (9)$$

We know we will be calculating these ϕ values in iterations, thus we need to define an order of operation:

$$\phi^{k+1}_{x,y} = \frac{1}{4}(\phi^{k+1}_{x-1,y} + \phi^{k+1}_{x,y-1} + \phi^{k}_{x+1,y} + \phi^{k}_{x,y+1}) \quad (10)$$

Meaning $\phi_{x-1,y}$ and $\phi_{x,y-1}$ are from the current iteration, $\phi_{x+1,y}$ and $\phi_{x,y+1}$ come from the previous iteration, and $h = 1$. This order will make more sense after we outline the use of the Grid in Section 1.1.2.
We now have an equation that can construct an iterative method to solve the Laplace equation. Although we can technically use this to solve our PDE, the convergence is very slow, and the number of iterations needed is large. Our next step is outline a method which can accelerate us towards the final solution (Gauss-Seidel & Successive Over-Relaxation Method).

To briefly illustrate how these methods would work, calculate the difference in consecutive grid points (named δ_k). As we near convergence, these points (given sufficient resolution of the grid) should have very similar values.

$$\delta_k(x,y) = \frac{1}{4}[\phi(x+h,y) + \phi(x-h,y) + \phi(x,y+h) + \phi(x,y-h)] - \phi_k(x,y) \quad (11)$$

If we involve this 'delta' figure in the iteration, scaled by λ:

$$\phi_{k+1} = \phi_k(x,y) + (\lambda+1)d_k(x,y) \quad (12)$$

The result of this expression means we take larger steps (informed by the previous grid points) if we are further from the solution. As δ_k will be larger. But as we approach convergence, δ_k becomes smaller, thus this amplification has little effect. This outlines the basis of the convergence aiding methods we will discuss in Section 1.1.3 & Section 1.1.4 (Gauss-Seidel & SOR).

1.1.2 Sparse Systems of Linear Equations & Grid System

We can now create a sparse linear system using the equations derived. To start we need to create an indexed grid system, and using our order of calculation to calculate each grid point.

In the iterative process, we will evaluate points from the boundary inwards (or visa-versa). I am going to derive an expression for the inner points of our square in terms of the boundary points for an example system (using Equation (9)):

$$\phi_{1,1} - \frac{1}{4}(\phi_{2,1} + \phi_{1,2}) = \frac{1}{4}(\phi_{0,1} + \phi_{1,0}) \quad (13)$$

$$\phi_{2,1} - \frac{1}{4}(\phi_{1,1} + \phi_{2,2}) = \frac{1}{4}(\phi_{2,0} + \phi_{3,1}) \tag{14}$$

$$\phi_{1,2} - \frac{1}{4}(\phi_{2,2} + \phi_{1,2}) = \frac{1}{4}(\phi_{0,2} + \phi_{1,3}) \tag{15}$$

$$\phi_{2,2} - \frac{1}{4}(\phi_{2,1} + \phi_{1,2}) = \frac{1}{4}(\phi_{2,3} + \phi_{3,2}) \tag{16}$$

Thinking of this system of linear equations as an eigenvalue problem that creates our Dirichlet boundary conditions, we can create a matrix equation for the system. Our linear system acts on a vector composed of unknown internal points, to output our boundary data (in the form as above). A small example of such a system is pictured, it is important to note we are moving left to right, bottom to top.

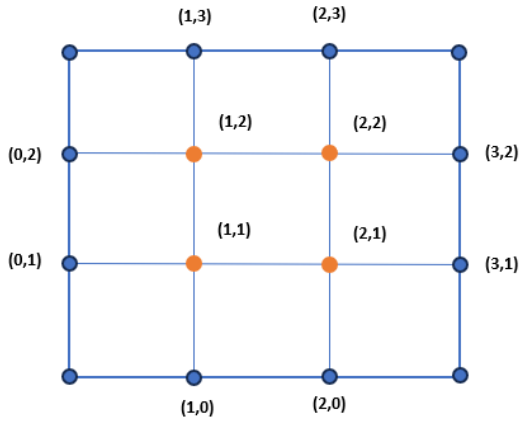


Figure 1: Example 3x3 Matrix, with Dirichlet boundary conditions

$$\begin{bmatrix} 1 & -\frac{1}{4} & -\frac{1}{4} & 0 \\ -\frac{1}{4} & 1 & 0 & -\frac{1}{4} \\ -\frac{1}{4} & 0 & 1 & -\frac{1}{4} \\ 0 & -\frac{1}{4} & -\frac{1}{4} & 1 \end{bmatrix} \begin{bmatrix} \phi_{1,1} \\ \phi_{2,1} \\ \phi_{1,2} \\ \phi_{2,2} \end{bmatrix} = \begin{bmatrix} \frac{1}{4}(\phi_{0,1} + \phi_{1,0}) \\ \frac{1}{4}(\phi_{2,0} + \phi_{3,1}) \\ \frac{1}{4}(\phi_{3,1} + \phi_{2,0}) \\ \frac{1}{4}(\phi_{2,3} + \phi_{3,2}) \end{bmatrix} \tag{17}$$

We have the beginning of a sparse linear system, that creates the boundary conditions from the inner points of the domain (via our finite difference and 4 nearest neighbor average).

$$A\phi = b \tag{18}$$

This Matrix 'A' and it's properties will be manipulated in order to speed up convergence in our system.

1.1.3 Gauss-Seidel Method

To give background on the Successive Over-Relaxed Method we need to start from the Gauss-Seidel method. The Gauss-Seidel method is used to numerically solve large system of linear equations

using far less iterations, by decomposing our systems' matrix 'A'.
We can start with a basic iterative scheme, that converges to our original system (Aϕ = b)

$$P\phi^{k+1} = b + (P - A)\phi^k \tag{19}$$

We see from the example matrix in 1.1.2, that we have a symmetric matrix and sparse linear system (many zero entries). This mean we can decompose the matrix into its diagonal, upper and lower portions. Where:

$$D \rightarrow Diagonal \tag{20}$$
$$L \rightarrow Lower\ triangle \tag{21}$$
$$U \rightarrow Upper\ triangle \tag{22}$$
$$\tag{23}$$

$$A\phi = b \tag{24}$$
$$(D + L + U)\phi = b \tag{25}$$
$$(D + L)\phi = b - U\phi \tag{26}$$

Using an order of calculation we can turn this into an iteration we can use.

$$(D + L)\phi^{k+1} = (b - Ux^k) \tag{27}$$

$$\phi^{k+1} = (D + L)^{-1}(b - Ux^k) \tag{28}$$

It is clear that for this method to work we need 'D+L' to be invertible. Given D+L is simply the bottom half of a matrix including the diagonal, there is no row or column of the matrix that is entirely composed of zero entries. This mean we will always have $|det(D + L)| > 0$, thus D+L is invertible.

$$x_{k+1} = (D + L)^{-1}(b - (A - (D + L))x^k \tag{29}$$
$$x^{k+1} = x^k + (D + L)^{-1}(b + Ax^k) \tag{30}$$

The order of operation is very important, as changes to order change the iteration scheme entirely. As mentioned, our grid order will over top to bottom, left to right. Meaning $\phi_{x-1,y}$ and $\phi_{x,y-1}$ are from the current iteration, $\phi_{x+1,y}$ and $\phi_{x,y+1}$ come from the previous iteration.

Using equation (27) we can derive (again) our iteration equation:

$$(D + L)\phi^{k+1} = (b - Ux^k)$$

$$L\phi \rightarrow -\phi_{x-1,y} - \phi_{x,y-1}$$
$$U\phi \rightarrow -\phi_{x+1,y} - \phi_{x,y+1}$$
$$D\phi \rightarrow 4\phi x, y$$

$$\phi^{k+1}_{x,y} = \frac{1}{4}(\phi^{k+1}_{x-1,y} + \phi^{k+1}_{x,y-1} + \phi^{k}_{x+1,y} + \phi^{k}_{x,y+1}) \tag{31}$$

1.1.4 Successive Over-Relaxed Method

Although functional, the Gauss-Seidel Method takes many iterations to converge. To fix this we can use a weighting parameter ω, the relaxation factor, this works similar to (11), it gives a weighting to the previous worked values in each iteration.

$$x^{k+1} \propto x^k + \omega_{correction} \tag{32}$$

Notice by looking at our system $Ax = b$ and equation (25), the Lower diagonal 'L' represents the current iteration ϕ^{k+1} and 'D' the upper diagonal represents the previous iteration. So to give more weight to the previous steps of the current iteration we can have $\frac{1}{\omega}D + L$. This exact form only makes sense for the purpose of the calculation.

$$A\phi = b \tag{33}$$

$$(\frac{1}{\omega}D + L + U)\phi = b \tag{34}$$

$$(\frac{1}{\omega}D + L)\phi^{k+1} = b - U\phi^k \tag{35}$$

Noting that still $A = U + L + D$.

$$(\frac{1}{\omega}D + L)\phi^{k+1} = b + (A - \frac{1}{\omega}D + L)\phi^k \tag{36}$$

$$(\frac{1}{\omega}D + L)\phi^{k+1} = b + (U + D + L - \frac{1}{\omega}D + L)\phi^k \tag{37}$$

$$(\frac{1}{\omega}D + L)\phi^{k+1} = b + (U + D - \frac{1}{\omega}D)\phi^k \tag{38}$$

$$(D + \omega L)\phi^{k+1} = b\omega + (1 - \omega)D\phi^k + \omega U\phi^k \tag{39}$$

Finally in equation (39) we have a weighting attached to L, which as we discussed represents the previous grid points of the current iteration. If we amplify these values, we start to take larger steps forward.
Noting again that D represents our previous steps of this iteration, and U are of the previous iterations.

$$D\phi^{k+1} = \phi^{k+1}_{x,y} \tag{40}$$

$$\omega L\phi^{k+1} = \frac{\omega}{4}(\phi^{k+1}_{x-1,y} + \phi^{k+1}_{x,y-1}) \tag{41}$$

$$(1 - \omega)D\phi_k = (1 - \omega)\phi^k_{x,y} \tag{42}$$

$$\omega U\phi_k = \frac{\omega}{4}(\phi^k_{x+1,y} + \phi^k_{x,y+1}) \tag{43}$$

$$\phi^{k+1}_{x,y} = (1 - \omega)\phi^k_{x,y} + \frac{\omega}{4}(\phi^{k+1}_{x-1,y} + \phi^{k+1}_{x,y-1} + \phi^k_{x+1,y} + \phi^k_{x,y+1}) \tag{44}$$

$b\omega$ remains a constant.

1.2 Code Execution

1.2.1 Grid Construction

This code is centered around the use of 2D arrays. We store the values of ϕ over our domain Ω, using the N x N array phi[N][N]. We can pull out specific ϕ values for any x-y coordinate, phi[i][j],

but first we have to convert our Cartesian coordinates into grid indices.
Where N is the dimension of our square grid, IE. there are N points on each side of the grid, and N^2 points on the grid.
First we define our step size in both directions. It is important to note that for a side of 'N' points, we have [0 → N-1] 'steps' between points. Meaning we get step size:

$$\Delta x, \Delta y = \frac{1}{N-1} \tag{45}$$

To find an exact (x,y) point on the grid we need to convert them to an index on the grid. If our points start from [0,0] in the bottom left of the grid, and the step size is always $\frac{1}{N-1}$, our x and y indexes are as such:

$$x = i * (stepsize) \rightarrow x = \frac{i}{N-1} \rightarrow i = x(N-1) \tag{46}$$

$$y = j * (stepsize) \rightarrow x = \frac{j}{N-1} \rightarrow j = y(N-1) \tag{47}$$

$$\tag{48}$$

So to land exactly on a position $\frac{2}{5}, \frac{1}{2}$ we need the have a grid devisable into 10 by 10 sections. A side of 10 lengths has 11 points, meaning our N must be devisable by 11.

$$N = k(11), k \in \mathbf{N}$$

We also need to set the boundaries for the box and line within ω. This is done by defining there corners (x-y start and end points), so we can fill in the lines with a for-loop.

```
const int N = 253;              // Grid dimension
const double tol = 1e-7;        // Tolerance to define convergence
const double omega = 1.8;       // Relaxation factor
double phi[N][N];               // Storing PHI as a 2D array [x][y]
bool boundary[N][N];            // To keep the boundary points constant we need to omit them from the SOR process
//Used a 2D array to store define our boundary points, [x][y]==true if boundary, [x][y]==false if not boundary, we will have a N x N true/false grid.

//Defining the boundaries given for the shapes (boundaries) within the PDE domain//
//Box contour
int box_x1 = 2 * (N-1) / 10;
int box_x2 = 4 * (N-1) / 10;
int box_y1 = 7 * (N-1) / 10;
int box_y2 = 9 * (N-1) / 10;
//Line contour
int line_x = 8 * (N-1) / 10;
int line_y1 = 1 * (N-1) / 10;
int line_y2 = 6 * (N-1) / 10;
```

Figure 2: Initializing the System

From here we need define our boundaries of domain ω and the objects (box & line). To illustrate this geometry in array form:

- phi[i][N-1] Top side.
- phi[N-1][i] Right side.
- phi[i][0] Bottom side.

- phi[0][i] Left side.

Figure 2 shows the full setup, including the contour of the box and line.
We don't want these boundary points to be changed by our SOR function, as this would make the boundary conditions meaningless. To do this, I assigned a bool true/false value to each point on the grid, this needed another array spanning the domain, called boundary[N][N]. We then give our boundary points the value boundary[x][y]==true, and we can then use an if function within our SOR method to skip over these points in the iterations.

```
//We need to cycle through our grid, assigning the boundary condition values and assigning boundary status boundary[x][y]==true
void boundary_conditions() {
    memset(boundary, 0, sizeof(boundary));  // Set all elements of the 2D array to false
    //Domain 1x1 square
    for (int i = 0; i < N; i++) {
        double x = double (i) / double (N - 1);
        double y = double (i) / double (N - 1);

        phi[i][N-1] = x;      // Top side= x
        boundary[i][N-1] = true;
        phi[N-1][i] = y;      // Right side= y
        boundary[N-1][i] = true;
        phi[i][0] = 0.0;      // Bottom side= 0
        boundary[i][0] = true;
        phi[0][i] = 0.0;      // Left side= 0
        boundary[0][i] = true;
    }
    //Box x-contour value
    for (int j = box_y1; j <= box_y2; j++) {
        phi[box_x1][j] = 1.0; // Left side of box A
        boundary[box_x1][j] = true;

        phi[box_x2][j] = 1.0;   // Right side of box A
        boundary[box_x2][j] = true;
    }
    //Box y-contour value
    for (int i = box_x1; i <= box_x2; i++) {
        phi[i][box_y1] = 1.0; // Bottom side of box A
        boundary[i][box_y1] = true;

        phi[i][box_y2] = 1.0;   // Top side of box A
        boundary[i][box_y2] = true;
    }
    // Line B contour value
    for (int j = line_y1; j <= line_y2; j++) {
        phi[line_x][j] = 0.0; // Line B with phi = 0
        boundary[line_x][j] = true;
    }
} //End of boundary conditions function
```

Figure 3: Assigning value==true to the Boundaries

1.2.2 SOR Function

The function needed to preform the Successive Over-Relaxation method is simple enough. It is only required to make the iteration computation, hold the new value of ϕ, and track the error ($delta$) between iterations.
First we need to set the phi[i][j] function to zero. At this point we have a N x N array full of zeros, except some of the boundaries. We begin by setting up a while loop, with our convergence condition [delta < tolerance], this keep the SOR process running until we reach acceptable convergence. Next

we need nested while loops, as we need to cycle through both the i & j variables of our arrays. Within this is the 'if' function, as mentioned, to exclude any boundary points from the process: if[boundary[i][j]==false]. We then iterate using the 4 point average using the nearest neighbors (as described in the 'Theory' section), and calculate the difference of ϕ between iterations.

```
//new phi value using the SOR increment, as outlined in document
double phi_1 = (1 - omega) * phi[i][j] + omega * 0.25 * (phi[i+1][j] + phi[i-1][j] + phi[i][j+1] + phi[i][j-1]);
// Measure the difference (delta) between new and old phi values
double delta = fabs(phi_1 - phi[i][j]);
```

Figure 4: Iteration and Delta calculation

We then assign this new 'phi_1' to be our old phi for the next iteration.
To track convergence we have to keep a 'max global error' value for our domain Ω. This value represents the maximum change between the current iteration and last iteration of any grid point. This means with each iteration and for every grid point, the delta value is compared to the 'max global delta'. If this delta value is greater than the 'max global delta' it becomes the new 'max global delta'. To start the while loop we set max_global_delta=1 , then within the loop max_global_delta is set back to zero for every iteration. SO as we start to have convergence the max global delta should fall.
To summarize, we compute the largest changed found in the grid for each iteration, if this change is within tolerance (1e-7) we can close our while loop, and reasonably assume convergence.

```
// SOR function to apply the Sucessive Over-Relaxtion method [while keeping boundary conditions fixed]
void SOR() {
    int iterations = 0;
    double global_max_delta = 1.0; // must start at a higher value than tolerance in order to start while loop

    //while-loop set to end when differecne between current and previous value is within tolerance
    while (global_max_delta >= tol) {
        global_max_delta = 0.0;                     //set back to zero

        for (int i = 1; i < N - 1; i++) {      //nested for-loops to cycle through the 2D array [i][j]
            for (int j = 1; j < N - 1; j++) {
                if (boundary[i][j] == false) { // for-loop only runs if the point is not a boundary point

                //new phi value using the SOR increment, as outlined in document
                double phi_1 = (1 - omega) * phi[i][j] + omega * 0.25 * (phi[i+1][j] + phi[i-1][j] + phi[i][j+1] + phi[i][j-1]);
                // Measure the difference (delta) between new and old phi values
                double delta = fabs(phi_1 - phi[i][j]);

                if (delta > global_max_delta) {      //if our difference is greater than the difference found previously, we assign it as the new delta
                    global_max_delta = delta;       //we are effectivily finding the maximum change in our grid, and when this is acceptibly small we can say we have convergence
                }
                phi[i][j] = phi_1;
            } }
        }

        iterations++; //increment iteration
    }
```

Figure 5: SOR function

Finally, we need to call these various functions in the correct order.

- Set phi[i][j] to zero.
- Call boundary_conditions() to define our borders.
- Call the SOR function.
- Call and Print Derivative_value() function

```cpp
int main() {

    memset(phi, 0, sizeof(phi));               //setting the phi[x][y] array to zero
    boundary_conditions();                     //setting the initial boundary values
    SOR();                                     //calling our Sucess Over-Relaxation function
    cout << "∂ψ/∂y(2/5,1/2) =" << derivative_value() << endl;   //Printing the derivstive value for our point

    // printing grid values in CSV format as I used excel to make my graphs
    ofstream csvFile("phi_values.csv");        //Cretaing the CSV file
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {

            csvFile <<phi[i][j]<<",";          //Storing the Phi values
        }
        csvFile<<endl;                         //Due to excel needing separate columns for 3D graphing, we create a new line each loop
    }
    csvFile.close();

    cout << "phi values in 'phi_values.csv'" << endl;

    return 0;
}
```

Figure 6: Driver Code

We see at the end, that we print the data into a CSV file. This is because I used Excel to create my surface graphs, as well as this I print only each 'i' (horizontal) line of values. This is done by calling a new line 'endl' after each iteration of the outer for loop (the 'i' for-loop). This means we print a N x N square of data, allowing for easier plotting.

1.2.3 Derivative Calculation

The derivative is calculated by the central difference method as derived in the last assignment.

$$\frac{\partial \phi}{\partial y} \approx \frac{\phi(i, j+1) - \phi(i, j-1)}{2\Delta y} \tag{49}$$

Our ϕ points are directly above each other on the grid, we can use them to get the change Δ over a given distance:

$$\Delta x, \Delta y = \frac{1}{N-1} \tag{50}$$

But to find the exact point $(\frac{2}{5}, \frac{1}{2})$ on the grid we need to convert them to an index on the grid.

$$x = \frac{2}{5} = \frac{i}{N-1} \rightarrow i = (2/5)(N-1) \tag{51}$$

$$y = \frac{1}{2} = \frac{j}{N-1} \rightarrow j = (1/2)(N-1) \tag{52}$$

$$\tag{53}$$

```cpp
double derivative_value() {            // compute the derivative ∂φ/∂y(2/5, 1/2)

    //transofrming the x-y coordintes to grid points
    int i = 2 * (N - 1) / 5;           // x = 2/5
    int j = (N - 1) / 2;               // y = 1/2
    double dy = 1.0 / (N - 1);         //Step size dy=dx=(N-1)^-1


    return (phi[i][j+1] - phi[i][j-1]) / (2 * dy);  //using finite difference/ first principals
}
```

Figure 7: Function to calculate Derivative

1.3 Extra Programs for Analysis:

1.3.1 Relaxation Factor Optimization

The choice of ω impacts the number of iterations needed to converge the system. We can roughly find the optimal choice of ω by running our SOR function for various ω values. Instead of rewriting the SOR function, I called SOR() within a for loop that inserts values of ω spaced by 0.05.

```cpp
// 'for' a range of omega vales, we call SOR()
for (double omega = 0.05; omega <= 1.95; omega += 0.025) {

    //Important to reset phi and boundary conditions for each omega value (needed as we are not within the SOR funtions)
    memset(phi, 0, sizeof(phi));
    boundary_conditions();
    csvFile << omega << "," << SOR(omega) << "\n";  // Call SOR method with given omega, (cycled through by for-loop) & printing the omega and iterations to CSV file
}
```

Figure 8: SOR function - range of *omega* values

Beyond this change the rest of the code is identical. The same method was used for Q2, so I did not include this in the Q2 section.

1.3.2 Required Parameters for Convergence:

```cpp
int main() {

    ofstream csvFile("N_tol_derivative_values.csv");
    csvFile << "N,tol,Derivative_Value" << endl;

    // Vhe only change from the other programs is that we must call our function in a loop over different N and tolerance values
    //And apply changes by assigning the N and tol variablles to a temporary varibel that is being changed in the for-loop
    for (int altered_N = 50; altered_N <= 500; altered_N += 50) {
        for (double altered_tol = 1e-1; altered_tol >= 1e-9; altered_tol /= 10) {
            N = altered_N;      // change grid dimension
            tol = altered_tol;  // change tolerance

            memset(phi, 0, sizeof(phi));  // set phi array to zero
            boundary_conditions();        // set boundary conditions as ==true
            SOR();
            cout << "N = " << N << ", tol = " << tol
                 << ", ∂φ/∂y(2/5,1/2) = " << derivative_value() << endl;
            csvFile << N << "," << altered_tol << "," << derivative_value() << endl;    //Output CSV file with N, tol and repective derivaqtive value, we analise these in the report
        }
    }
    csvFile.close();

    return 0;
}
```

Figure 9: Code needed to alter the N and tol values, and recording resulting $\frac{\partial \varphi}{\partial y}$ at $\left(\frac{2}{5}, \frac{1}{2}\right)$ value

This code works by changing the variables N and tolerance in the main code, this means our system is called for each combination of N and tol, for a given range. To do this we first need to define phi[500][500] and boundary [500][500] to ensure our array is large enough for every N. The result of this code is a saved CSV file with every combination of N and tol, along with the corresponding $\frac{\partial \varphi}{\partial y}$ at $\left(\frac{2}{5}, \frac{1}{2}\right)$. There is no other change to the program otherwise, or in any change in Q2.

1.4 Results

1.4.1 ϕ Solution

Graph of ϕ over given domain [N=253 and ω=1.8]

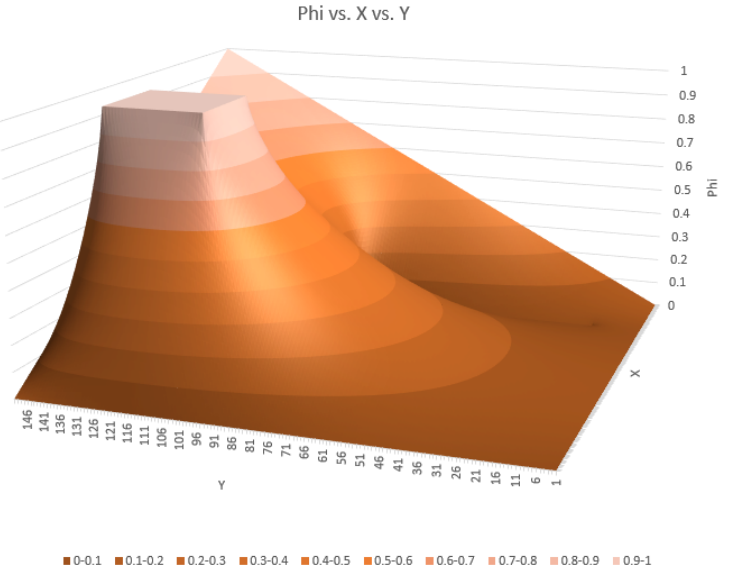
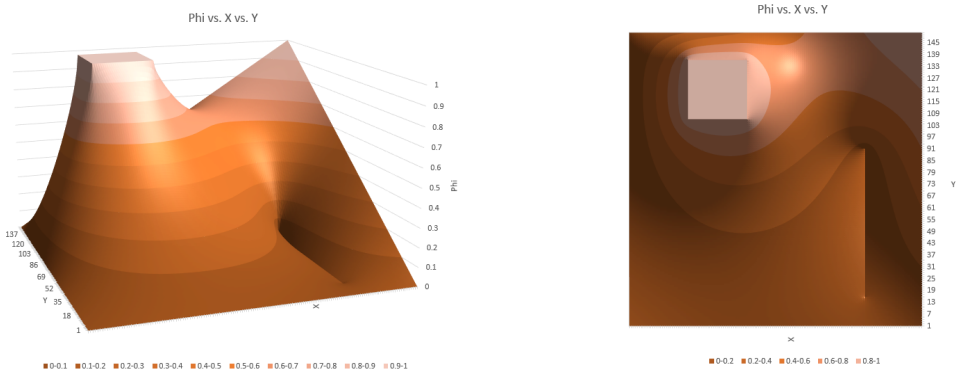


Figure 10: Phi over domain Ω



(a) Phi over domain Ω - View from lower side

(b) Contour of Phi over Ω

Figure 11: Visualization of Phi over the domain Ω: (a) 3D view from lower side, (b) contour representation.

1.4.2 Over-relaxation Parameter

The over relaxation parameter was found purely through trail and error, to be $1.8 < \omega < 1.95$, the optimal value changes as you change the dimension 'N'. I created a separate program to find the optimal ω, to an accuracy of ± 0.05. For the below graph, I used N=250. I graphed the number of iterations needed to converge for difference values of ω:

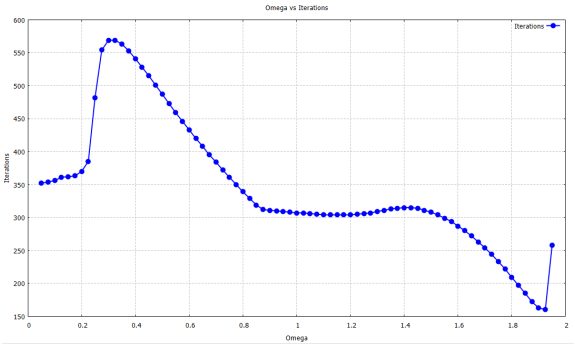


Figure 12: Iterations as a function of Omega

We see a minimum occur around $\omega \approx 1.9$.

1.4.3 Derivative Value

As discussed, finding the most accurate information on the point $(\frac{2}{5}, \frac{1}{2})$, we need an 'N' divisible by 11 (see 1.2.1). If we know the program is landing exactly on point $(\frac{2}{5}, \frac{1}{2})$, we exclude any over/under shooting error. In 'Table 1' see how the SOR accuracy and convergence scales with 'N'.

Tolerance (tol)	N	ω	$\frac{\partial \varphi}{\partial y}\left(\frac{2}{5}, \frac{1}{2}\right)$	$\pm\Delta$
1×10^{-7}	110	1.8	1.71532	$9.17e-10+O(h^2)$
1×10^{-7}	154	1.8	1.70927	$6.53e-10+O(h^2)$
1×10^{-7}	209	1.8	1.74164	$4.78e-10+O(h^2)$
1×10^{-7}	253	1.8	1.73206	$3.97e-10+O(h^2)$
1×10^{-7}	308	1.8	1.7306	$3.25e-10+O(h^2)$
1×10^{-7}	352	1.8	1.72613	$2.84e-10+O(h^2)$
1×10^{-7}	407	1.8	1.72517	$2.46e-10+O(h^2)$
1×10^{-7}	506	1.8	1.72151	$1.98e-10+O(h^2)$

Table 1: Values of $\frac{\partial \varphi}{\partial y}$ at $\left(\frac{2}{5}, \frac{1}{2}\right)$ for different values of N with $\omega = 1.8$ and tolerance 1×10^{-7}

We see 3 s.f convergence around N = 352.
Note the uncertainty caused by our convergence criteria (tolerance) and h-artifacts (from method derivation)(derived in Section 1.3.5) is of the form:

$$\pm O(\frac{tol}{h}) + O(h^2) \tag{54}$$

This creates the requirement for accuracy:

$$tol << h$$

$$tol << \frac{1}{N-1}$$

See Section 1.3.5 for value of $\frac{\partial \varphi}{\partial y}$ at $\left(\frac{2}{5}, \frac{1}{2}\right)$ for different tolerance values.

1.4.4 General Convergence Criteria of the SOR Method

We can introduce a value called the residual 'r'. This represents the difference between the left and right hand side of our system.

$$r = A\phi - b \tag{55}$$

We want to have a small maximum residual.

$$||r^k|| = max_{xy}|A\phi^k - b| \tag{56}$$

The general convergence criteria for an iteration method on a sparse linear system:

$$d^k = \phi^{k+1} - \phi k \tag{57}$$

General case:

$$P\phi^{k+1} = b(P - A)\phi^k \tag{58}$$

Our case: $P = (\frac{1}{\omega}D + L)$

$$(\frac{1}{\omega}D + L)\phi^{k+2} = b + ((\frac{1}{\omega}D + L) - A)\phi^{k+1} \tag{59}$$

$$(\frac{1}{\omega}D + L)\phi^{k+1} = b + ((\frac{1}{\omega}D + L) - A)\phi^{k} \tag{60}$$

Note our iteration takes the form:

$$x^{k+1} = (D + \omega L) - 1[\omega b - [\omega U + (\omega - 1)D]x^k] \tag{61}$$

$$(\frac{1}{\omega}D + L)(\phi^{k+2} - \phi^{k+1}) = (\frac{1}{\omega}D + L - A)(\phi^{k+1} - \phi^k) \tag{62}$$

$$(\frac{1}{\omega}D + L)d^{k+1} = (\frac{1}{\omega}D + L - A)d^k \tag{63}$$

$$d^{k+1} = (I - (\frac{1}{\omega}D + L))^{-1}A)d^k \tag{64}$$

If this value is to progress to zero the eigenvalues must be:

$$|\lambda(I - (\frac{1}{\omega}D + L)^{-1}A))| = |G| < 0 \tag{65}$$

This is the equivalent of saying the spectral radius of the iteration matrix C_ω is less than 1.
Starting from equation (35):

$$(D + \omega L)\phi^{k+1} = b\omega + (1 - \omega)D\phi^k + \omega U\phi^k \tag{66}$$

$$C_\omega = (D+\omega)^{-1}[(1-\omega)D - \omega U] \tag{67}$$

$$\rho[C_\omega] < 1 \tag{68}$$

This has been proven to converge for $0 < \omega < 2$. We can go further and provide the criteria for (68) to be true.
In general, convergence depends on matrix A. A must be be symmetric on the diagonal.

$$det(\lambda D + zL + \frac{1}{z}U) = det(\lambda D + L + U), \quad for \lambda \in \mathbf{C} \tag{69}$$

Finally, Jacobi's method must be convergent for:

$$\mu = \rho(\mathbf{I} - D^{-1}A) < 1 \tag{70}$$

Another way to say this is to require diagonal dominance of matrix 'A'. That is:

$$|a_{ii}| > \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}| \tag{71}$$

Which we see from our example 3x3 system that this is the case for matrix 'A'. And any larger system will follow the same pattern as the geometry is the same, and the coefficients are always $\frac{1}{4}$ regardless of the system size.

1.4.5 Accuracy of Results

Convergence Criteria, h-Artifacts & Uncertainty:
We pick up errors from the convergence criteria (tolerance) and from the derivation of our iterations scheme. We can quantify this, in a similar as in the 'Theory' section, but specifically regarding the Laplace equation.

$$\phi(x+h, y) = \phi(x, y) + h\frac{\partial \phi}{\partial x} + \frac{h^2}{2!}\frac{\partial^2 \phi}{\partial x^2} + O(h^4)... \tag{72}$$

$$\phi(x-h, y) = \phi(x, y) - h\frac{\partial \phi}{\partial x} + \frac{h^2}{2!}\frac{\partial^2 \phi}{\partial x^2} + O(h^4)... \tag{73}$$

Adding equations (72) & (73) and solving for the second derivative and the higher order term.

$$\frac{1}{h^2}(\phi(x+h, y) + \phi(x-h, y) - 2\phi(x, y)) = \frac{\partial^2 \phi}{\partial x^2} + O(h^2) \tag{74}$$

We can repeat this along the y direction.

$$\frac{1}{h^2}(\phi(x, y+h) + \phi(x, y-h) - 2\phi(x, y)) = \frac{\partial^2 \phi}{\partial y^2} + O(h^2) \tag{75}$$

Leaving:

$$\frac{1}{h^2}(\phi(x, y+h) + \phi(x, y-h) - 2\phi(x, y)) = \frac{\partial^2 \phi}{\partial y^2} + O(h^2) \tag{76}$$

We see our SOR method will have an uncertainty for each phi value of:

$$\pm\Delta = tol + O(h^2) \tag{77}$$

This truncation leads to h artifacts.

Uncertainty in derivative calculations can be found to be $\propto O(h^2)$:

$$\frac{\phi(x+h,y) - \phi(x-h,y)}{2h} = \dot{\phi} + \frac{h^2}{6}\dddot{\phi} + \frac{tol}{h} = \dot{\phi} \pm O(h^2) + \frac{tol}{h} \tag{78}$$

We again have an error $\propto O(h^2)$.
To minimize error, we require:

$$\begin{gathered} h >> tol \\ h = \frac{1}{N-1} \\ \frac{1}{N-1} << tol \end{gathered} \tag{79}$$

This is a good rule to follow as we vary our N and tolerance values. What is meant by h artifacts is the collective error built up through calculations using h, where h is the grid spacing. A source of this error in our case is due to the omission of $O(h^2)$ terms, as seen in the derivation above. This means that we have an error proportional to h^2, this means that by increasing the resolution of our grid, IE. decreasing the value of 'h'. We can make our error due to h artifacts essentially negligible. In the results section we used the tolerance figure (1e-7) and h^2 to calculate an uncertainty in our figures.
Tolerance:
Another source of error is due to the fact the iteration scheme can only run for a finite 'k' iterations, this will always leave a small residual error ($r = A\phi - b$). But by using a small tolerance value we can minimize this truncation of the solution.

Due to the method for convergence method I used ('max global error', see Section 1.2.2), this program requires a very small tolerance to give accurate results. They begin to diverge after tolerance $> 1e-5$. I can demonstrate this by numerical experiment:

Tolerance (tol)	N	ω	$\frac{\partial \varphi}{\partial y}\left(\frac{2}{5}, \frac{1}{2}\right)$
1×10^{-2}	353	1.8	0.0839
1×10^{-3}	353	1.8	1.59509
1×10^{-4}	353	1.8	1.87275
1×10^{-5}	353	1.8	1.7478
1×10^{-6}	353	1.8	1.73055
1×10^{-7}	353	1.8	1.7288
1×10^{-8}	353	1.8	1.72863

Table 2: Values of $\frac{\partial \varphi}{\partial y}$ at $\left(\frac{2}{5}, \frac{1}{2}\right)$ for different tolerances with $N = 353$ and $\omega = 1.8$

We need a tolerance of at least 1e-7 for 3 s.f of accuracy, for N=353, this would be less for small grids.
This is because my while-loop relies on the value 'max_global_delta', this (as discussed) is the maximum change recorded on the grid between iterations, if this is left too large ($> 1e-5$), there is a chance we have not reached full convergence. The delta could potentially be in a false minimum, and jump above tolerance the next iteration (if the while-loop continued), thus going onto affect our measurements (Eg. derivative calculation). This is far less likely if the tolerance is very small.

Grid Resolution
As explored in the Results section, we get different values for different inputted N values. The higher the N value the more grid points in our fixed ω domain. This means our grid points are closer together and thus our approximation, that $\phi_{x,y}$ can be taken as the average of it's four nearest grid neighbors, becomes more accurate. As well as this, the overshoot/undershoot when N is not a multiple of 11 becomes smaller as N grows larger due to small step size 'h'.

To measure the convergence of our general phi grid values we can track the 'max global delta', which is the maximum change of a grid points between iterations.
-Convergence of Phi values for different grid sizes (tol = 1e-7):

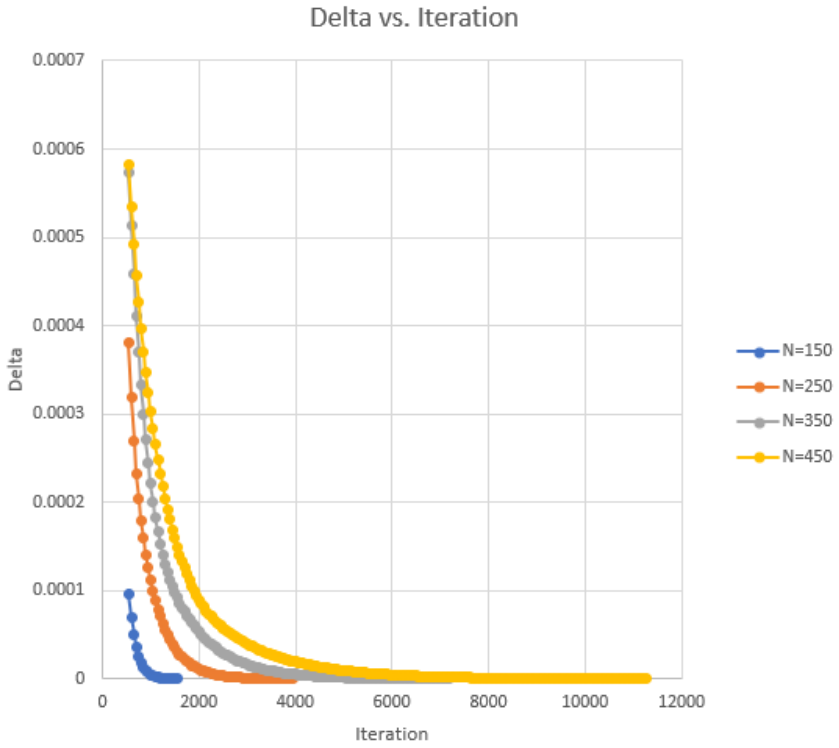


Figure 13: Max Global Delta vs. Iteration - for different grid sizes

(Note: Graph starts from iteration=500 for readability)

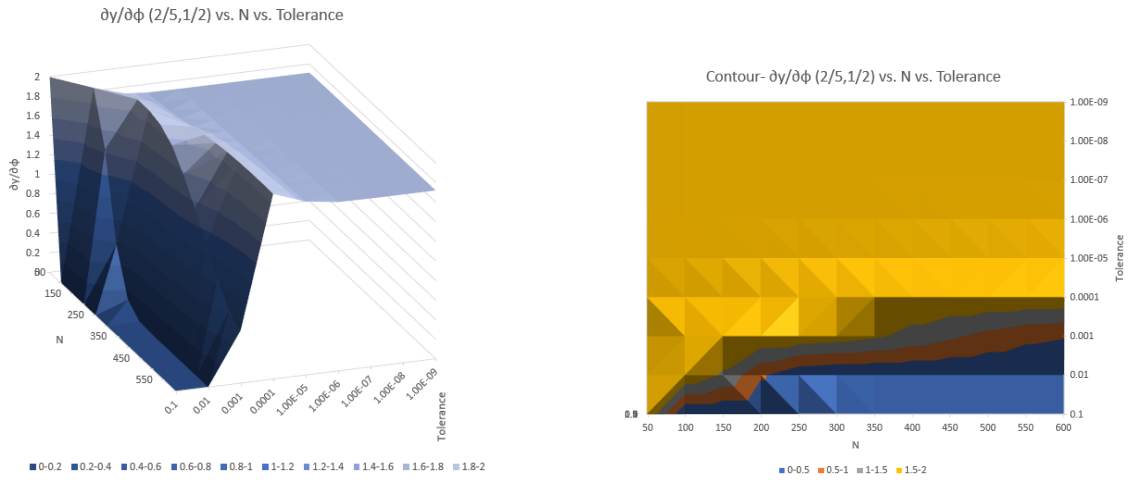
-Convergence of $\frac{\partial \phi}{\partial y}(\frac{2}{5}, \frac{1}{2})$ for different grid size:

Tolerance (tol)	N	ω	$\frac{\partial \varphi}{\partial y}\left(\frac{2}{5}, \frac{1}{2}\right)$
1×10^{-7}	100	1.8	1.67026
1×10^{-7}	150	1.8	1.68743
1×10^{-7}	250	1.8	1.70077
1×10^{-7}	350	1.8	1.70653
1×10^{-7}	400	1.8	1.70836

Table 3: Values of $\frac{\partial \varphi}{\partial y}$ at $\left(\frac{2}{5}, \frac{1}{2}\right)$ for different values of N with $\omega = 1.8$ and tolerance 1×10^{-7}

Required Parameters for Convergence:

From the graphs below we can see the relationship between the choices of parameters N and tol, and the convergence of $\frac{\partial \varphi}{\partial y}$ at $\left(\frac{2}{5}, \frac{1}{2}\right)$. The program steeply climbs towards the convergence value (≈ 1.7) once the tolerance drops below 0.001, overshooting this value slightly as it comes to a premature converge, then drops quickly back to the correct value given the right parameters. We also see once the grid is of size $N \approx 200$, the value of N makes little difference for convergence. From graph (b) we can estimate that convergence follows a roughly logarithmic pattern in terms of tolerance vs. N. The relationship loosely follows: $tol = c\, log(N-1)$.



(a) $\frac{\partial \varphi}{\partial y}\left(\frac{2}{5}, \frac{1}{2}\right)$ vs. N vs. Tolerance

(b) Contour-$\frac{\partial \varphi}{\partial y}\left(\frac{2}{5}, \frac{1}{2}\right)$ vs. N vs. Tolerance

Figure 14: 3D Representation(s) of $\frac{\partial \varphi}{\partial y}\left(\frac{2}{5}, \frac{1}{2}\right)$ with different choices of N and tol.

2 Problem 2

2.1 Theory

Neumann boundary conditions mean that instead of 'position' x-y values as our boundary, we have the derivative value normal to our boundary. On the surface we have:

$$\partial\phi \cdot \vec{n} \tag{80}$$

As shown in the first problem, we can find the derivative using the finite difference method. But the only issue is we don't have the full data needed, as we have no position data on or outside the boundary. This means that our forward difference is only O(h) accurate.
To combat this, we 'reach' inwards into the domain away from the boundary. This means we can approximate the derivative value at the boundary to the order of $O(h^2)$. This is particulay accurate for small 'h' values (large N value). This is done similar to problem 1:

$$\phi(h,y) = \phi(0,y) + h\frac{\partial\phi(0,y)}{\partial x} + O(h^2) \tag{81}$$

$$\phi(2h,y) = \phi(0,y) + 2h\frac{\partial\phi(0,y)}{\partial x} + O(h^2) \tag{82}$$

Subtracting 2 times the first equation cancels out the derivative term as such:

$$\frac{\partial\phi(0,y)}{\partial x} = \frac{4\phi(h,y) - \phi(2h,y) - 3\phi(0,y)}{2h} \tag{83}$$

So if in our case we have $\frac{\partial\phi}{\partial x} = 0$ & $\frac{\partial\phi}{\partial y}$ for x & y = 0 , we can easily solve for $\phi(0,y)\&\phi(0,x)$.

$$\phi(0,y) = \frac{4\phi(h,y) - \phi(2h,y)}{3} \tag{84}$$

Equally we can do the same for the x-boundary.

$$\phi(0,x) = \frac{4\phi(h,x) - \phi(2h,x)}{3} \tag{85}$$

The other boundaries in the unit square Ω:

$$\phi = x \rightarrow \frac{\partial\phi}{\partial x} \tag{86}$$

$$\phi = y \rightarrow \frac{\partial\phi}{\partial y} \tag{87}$$

$$\tag{88}$$

The box and line values remain unchanged as per the question.

2.2 Code Execution

For this problem I tried to keep the code as similar as possible to the code of Question 1, and only add a function to change the boundary conditions for the left and lower boundaries. I took the template of the Dirichlet code and added the function 'Neumann_bc'.

2.2.1 Boundary Calculation using Neumann B.C's

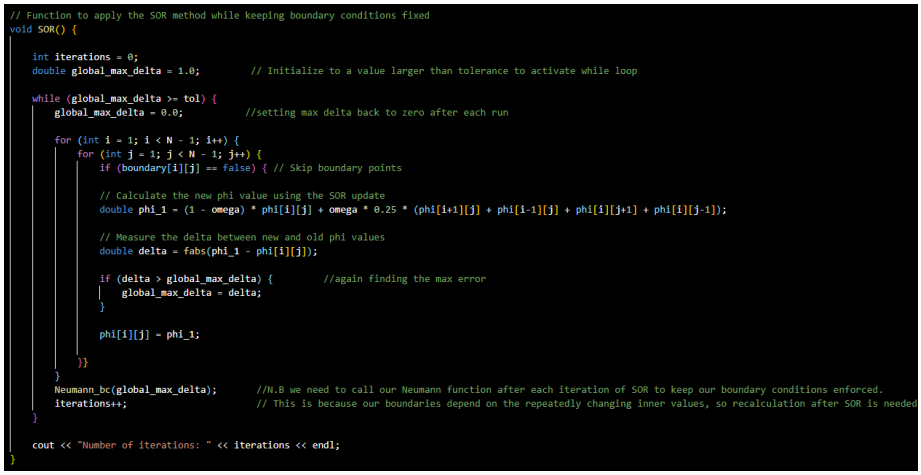
We need to use equations (84) and (85) in order to turn our Neumann boundaries into an estimate for definite values (Dirichlet). Similar to the SOR function, we loop over the lower (phi [0][j]) and left (phi[i][0]) boundaries, but we are using equation (84) and (85).

```
// applying Neumann boundary condition on the left and bottom sides
//I have added a check on the global_max_delta into this funcition
double Neumann_bc(double global_max_delta) {

    double phi_1;     //This is our new phi from the current iteration, as used in the

    // Left boundary (∂φ/∂x = 0 for x = 0)
    for (int j = 0; j < N-1; j++) {
        phi_1 = phi[0][j];
        phi[0][j] = (4.0*phi[1][j] - phi[2][j])/3.0; //As derived, we caqn calculate the boundary point using the points 1&2 'steps' into the square

        phi_1 = fabs(phi[0][j] - phi_1);            //Calculating the change after each iteration
        if(phi_1>global_max_delta){                 //As before, we use the max delta in the domain to measure if the function has converged
            global_max_delta =phi_1;                //Just as we check that the inner points are converged in the SOR, we can do the same for our new boundaries
        }                                           //Checking convergence on the boundaries can be redundant as if the cetner points have converged we can assume the same for the boundaries.
    }

    // Bottom boundary (∂φ/∂y = 0 for y = 0)
    for (int i = 0; i < N-1; i++) {
        phi_1 = phi[0][1];

        phi[i][0] = (4.0*phi[i][1] - phi[i][2])/3.0;

        phi_1 = fabs(phi[0][1] - phi_1);
        if(phi_1>global_max_delta){
            global_max_delta =phi_1;
        }
    }
    return global_max_delta;          //Returning delta
}//End of Neumann b.c function//
```

Figure 15: Function to calculate Neumann Boundaries

Notice we find the 'global max delta' to ensure convergence on the boundaries, this is likely redundant as convergence for the center points probably imply convergence on the boundaries.

We can't simply call this 'Neumann_bc' function in the main code after our 'SOR' function. This is because our boundary points are now dependent on the points within the square (phi[2h][j], phi[h][j], phi[i][2h] & phi[i][2h], where h is step size). So as the inner points are iterated, as will our boundary points (thus affecting the slope of our boundaries). The slope of our boundaries (Neumann conditions) needs to have a continuous impact on the evolution of the inner points, thus we need to enforce the condition inside of the SOR function. By calling 'Neumann_bc' after each iteration of SOR, we calculate the new boundaries based off the continuously changing 'near-boundary' grid points. These new boundaries go on to impact the evolution of the inner grid points.
If we simply called the 'Neumann_bc' function in the driver code, we would calculate the Neumann boundaries using a system that has evolved under Dirichlet boundary conditions (yielding the same result as Q1).



```
// Function to apply the SOR method while keeping boundary conditions fixed
void SOR() {

    int iterations = 0;
    double global_max_delta = 1.0;        // Initialize to a value larger than tolerance to activate while loop

    while (global_max_delta >= tol) {
        global_max_delta = 0.0;           //setting max delta back to zero after each run

        for (int i = 1; i < N - 1; i++) {
            for (int j = 1; j < N - 1; j++) {
                if (boundary[i][j] == false) { // Skip boundary points

                // Calculate the new phi value using the SOR update
                double phi_1 = (1 - omega) * phi[i][j] + omega * 0.25 * (phi[i+1][j] + phi[i-1][j] + phi[i][j+1] + phi[i][j-1]);

                // Measure the delta between new and old phi values
                double delta = fabs(phi_1 - phi[i][j]);

                if (delta > global_max_delta) {         //again finding the max error
                    global_max_delta = delta;
                }

                phi[i][j] = phi_1;

            }}
        }
        Neumann_bc(global_max_delta);       //N.B we need to call our Neumann function after each iteration of SOR to keep our boundary conditions enforced.
        iterations++;                       // This is because our boundaries depend on the repeatedly changing inner values, so recalculation after SOR is needed
    }

    cout << "Number of iterations: " << iterations << endl;
}
```

Figure 16: Revised SOR function to yield Neuamnn B.C's

2.2.2 Programs for Analysis:

This Code Remained unchanged from Q1. (Omega optimizations and Conditions for Convergence)

2.3 Results

2.3.1 ϕ Solution

We produce a 3D graph of Phi values over our X-Y domain. Notice the boundaries that were previously zero with Dirichlet conditions are no longer zero, but instead the slope is zero and the actual value of these boundaries takes the form you see in Graphs below.
Parameters (informed by Sec 2.3.3 & 2.3.4): [N=250] & [tolerance = 1e-7] & [ω ==1.8]
View from lower left side of domain

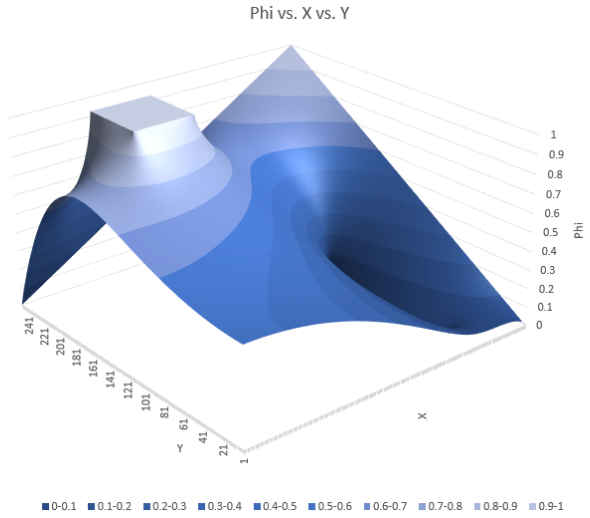


Figure 17: Neumann B.C's- Phi over Ω- Lower left view

View from lower side of square domain:

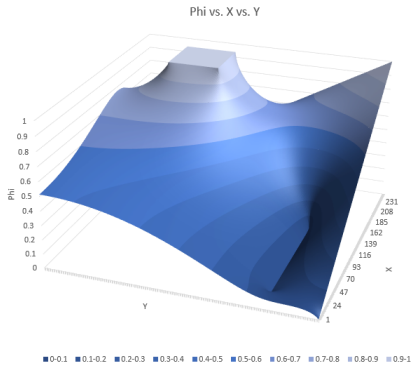


Figure 18: Neumann B.C's- Phi over Ω- Lower right view

Contour of Neumann Phi values over square domain:

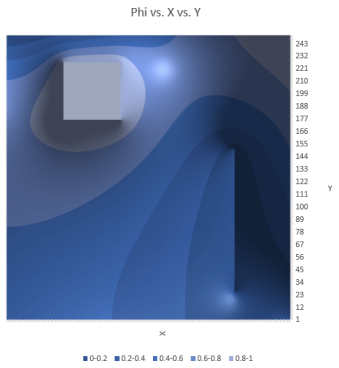


Figure 19: Contour Plot - Neumann B.C's - Domain Ω

2.3.2 Relaxation Factor

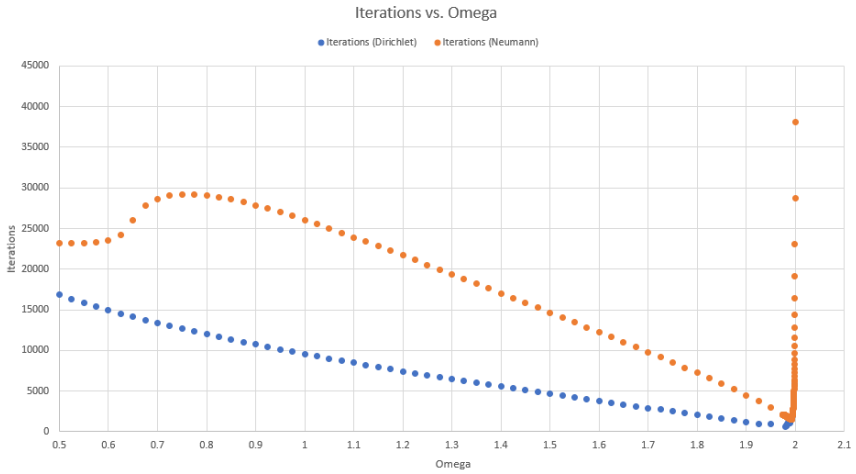


Figure 20: Iterations vs. Omega

Optimal choice for omega appears to be around $\omega = 1.98$. Notice the iterations needed for the Neumann boundary conditions are are consistently larger than that of Dirichlet. And the optimal choice of ω is slightly higher.

2.3.3 Derivative Value

For N=352 and $\omega = 1.8$. The value of the of $\frac{\partial \phi}{\partial y}(\frac{2}{5}\frac{1}{2}) = 1.11163$

Tolerance (tol)	N	ω	$\frac{\partial \varphi}{\partial y}\left(\frac{2}{5}, \frac{1}{2}\right)$	$\pm\Delta$
1×10^{-7}	55	1.8	1.14778	$1.85e-9+O(h^2)$
1×10^{-7}	110	1.8	1.09618	$9.17e-10+O(h^2)$
1×10^{-7}	154	1.8	1.10033	$6.54e-10+O(h^2)$
1×10^{-7}	253	1.8	1.11356	$3.95e-10+O(h^2)$
1×10^{-7}	352	1.8	1.11163	$2.85e-10+O(h^2)$
1×10^{-7}	407	1.8	1.11272	$2.46e-10+O(h^2)$

Table 4: Values of $\frac{\partial \varphi}{\partial y}$ at $\left(\frac{2}{5}, \frac{1}{2}\right)$ for different values of N with $\omega = 1.8$ and tolerance 1×10^{-7}

We see 3 s.f convergence for ± 253. Note that the accuracy for Neumann boundary conditions depends even more so on the size of step 'h', due to the process of approximating the boundary with near boundary points (Equations (84) & (85)). (As before in Q1) the uncertainty caused by our convergence criteria (tolerance) and h-artifacts (from method derivation)(derived in Section 1.3.5) is of the form:

$$\pm O(\frac{tol}{h}) + O(h^2) \tag{89}$$

2.3.4 Accuracy of Results

Tolerance, Convergence and Instability:

For different tolerances we see that we land one very different values for $\frac{\partial \varphi}{\partial y}$ at $\left(\frac{2}{5}, \frac{1}{2}\right)$. This points to a much less stable system, given Dirichlet boundary conditions converge after a tolerance of 1e-4.
-Convergence of general Phi values for different N values (tol 1e-7):

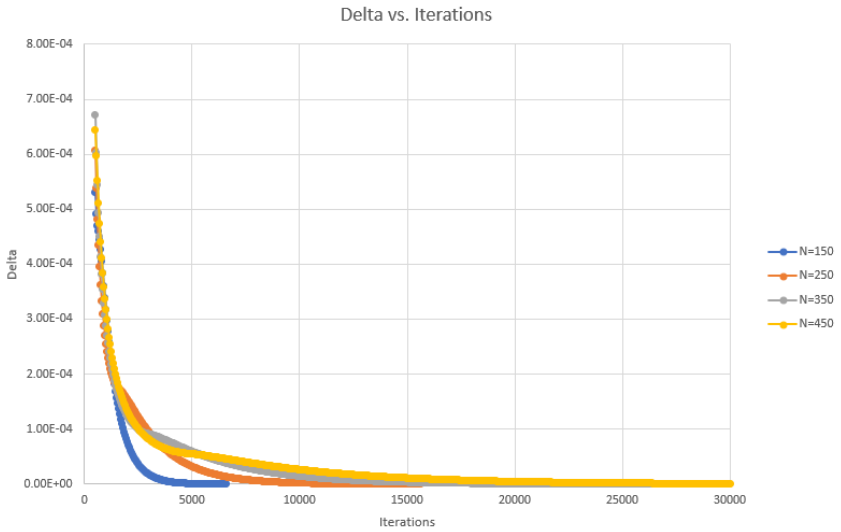


Figure 21: Max Delta vs. Iteration - Neumann B.C- For different Grid sizes.

(Note the x-axis only contains values $500 < iterations < 35000$)
We also see that thew Neumann boundary conditions program converges much slower (more iterations), this is because the more points we have converged the faster convergence occurs. With Dirichlet b.c's we start the program with already converged points along the boundaries, this speed up convergence, and the Neumann bc's program thus takes longer to reach the same level of convergence.
-Convergence of $\frac{\partial \phi}{\partial y}$ at $(\frac{2}{5}, \frac{1}{2})$ for different tolerance values:

Tolerance (tol)	N	ω	$\frac{\partial \varphi}{\partial y}\left(\frac{2}{5}, \frac{1}{2}\right)$
1×10^{-2}	352	1.8	0.0838667
1×10^{-3}	352	1.8	1.5907
1×10^{-4}	352	1.8	1.67838
1×10^{-5}	352	1.8	1.15657
1×10^{-6}	352	1.8	1.11572
1×10^{-7}	352	1.8	1.09324

Table 5: Values of $\frac{\partial \varphi}{\partial y}$ at $\left(\frac{2}{5}, \frac{1}{2}\right)$ for different values of N with $\omega = 1.8$ and tolerance 1×10^{-7}

Note the uncertainty, unchanged from Q1:

$$\pm O(\frac{tol}{h}) + O(h^2) \tag{90}$$

Required Parameters for Convergence of $\frac{\partial \varphi}{\partial y}$ at $\left(\frac{2}{5}, \frac{1}{2}\right)$: Compared to the Dirichlet boundary conditions in Q1, the program for Neumann conditions is more unstable. We see a larger peak where the program prematurely 'converges' for lower tolerances (≈ 0.001) or small grid sizes (< 250). The relationship between N and tolerance for convergence is larger the same as Q1 otherwise.

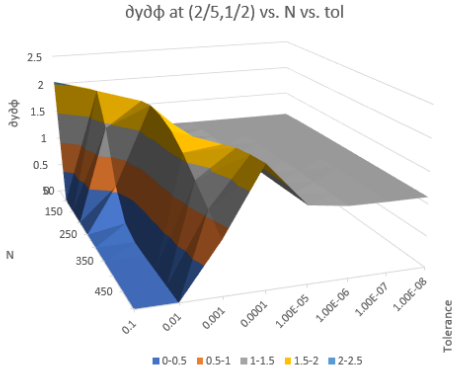


Figure 22: Values of $\frac{\partial \varphi}{\partial y}$ at $\left(\frac{2}{5}, \frac{1}{2}\right)$ for different values of N and tol

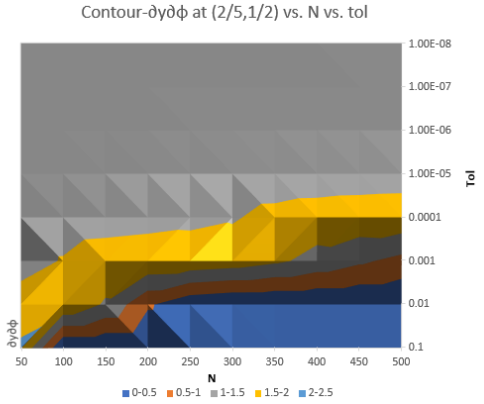


Figure 23: Contour-Values of $\frac{\partial \varphi}{\partial y}$ at $\left(\frac{2}{5}, \frac{1}{2}\right)$ for different values of N and tol

3 References:

[1] Practical Numerical Simulations Lecture Notes - TCD School of Mathematics - P. Fritzsch

[2] The Jacobi and Gauss-Seidel Iterative Methods - `https://www3.nd.edu/~zxu2/acms40390F12/Lec-7.3.pdf`

[3] A UNIFIED PROOF FOR THE CONVERGENCE OF JACOBI AND GAUSS-SEIDEL METHODS - ROBERTO BAGNARA - `https://www.cs.unipr.it/~bagnara/Papers/PDF/SIREV95.pdf`

[4] Successive Over-Relaxation - Wikipedia - `https://en.wikipedia.org/wiki/Successive_over-relaxation#cite_note-3`

[5] Wolfgang Hackbusch - Iterative Solution of Large Sparse Systems of Equations.