CO25: Laplace's Equation Report

Alex McInerny, shil5991

March 4, 2024

1 Abstract

We used the successive over-relaxation method in `MATLAB` to numerically approximate a 2D solution of Laplace's equation (using finite differences) in a uniform, square grid with specified, distinct boundary conditions on each edge. For different over-relaxation parameters and given boundary conditions, we plotted heat maps of the approximation and analyse the nature with which the algorithm converges to the true (analytic) solution to a desired degree of accuracy. We verified that the optimal over relaxation parameter ($\alpha \approx 1.39$) provides fast convergence, and one that is not within the valid range (1 to 2) leads to an incorrect, divergent solution. Furthermore, we discuss how for small α, convergence is monotonic, however oscillatory nature increases as α increases.

2 Introduction

Solving Laplace's equation (a second order partial differential equation) has incredibly important applications through all science and mathematics. Examples where solving Laplace's equation is crucial include fluid flow, heat conduction, and electrostatics. In order to calculate a solution of Laplace's equation, boundary conditions relating to the physical situation must be specified.

Obtaining a solution analytically (i.e., calculating a elementary expression for the solution) can often be very tricky and, for some boundary conditions, may be impossible. It is therefore important that there are other ways to approximate solutions, such that physical insight may be found to problems that would otherwise be impossible to solve. Unsurprisingly there are many ways to do this, but the one we shall focus on is called the finite difference approximation. The main idea is to descretise the PDE, approximating the partial derivatives at each position, which gives a system of equations to solve. We shall then solve this system using the successive over relaxation numerical method, giving a final, approximate solution to Laplace's equation with the desired boundary conditions.

The code written uses these techniques to solve a specific boundary value problem, however it is made generally to allow for different situations, and self contained so no user-input is required to achieve any required outputs.

In the following report, we consider in a more detailed manor the theory behind the techniques used, how the code was implemented, then we discuss the results and conclude by discussing their meaning, any research which may have furthered the results and limitations of the study.

3 Theoretical Background

3.1 Finite Differences Method

Laplace's equation $\nabla^2\psi = \frac{\partial^2\psi}{\partial x^2} + \frac{\partial^2\psi}{\partial y^2} = 0$, in some domain $\mathcal{D}$, can be approximated [1] by

$$\frac{\psi_{i+1,j} - 2\psi_{i,j} + \psi_{i-1,j}}{(\delta x)^2} + \frac{\psi_{i,j+1} - 2\psi_{i,j} + \psi_{i,j-1}}{(\delta y)^2} \approx 0, \tag{1}$$

where $\mathcal{D}$ contains an imaginary grid of points as shown in Fig.(1), and $\delta x, \delta y$ are the distances between the points in the x, y directions. This *finite difference approximation* [2] comes from approximating each term with centered differences. i.e., Considering Laplace's equation at some point (i, j) then Taylor expanding each second derivative in that points vicinity and substituting in those approximations;

$$\left.\frac{\partial^2\psi}{\partial x^2}\right|_{(i,j)} \approx \frac{\psi(x+\delta x, y) - 2\psi(x,y) + \psi(x-\delta x)}{(\delta x)^2} + \mathcal{O}\big((\delta x)^2\big) := \frac{\psi_{i+1,j} - 2\psi_{i,j} + \psi_{i-1,j}}{(\delta x)^2} + \mathcal{O}\big((\delta x)^2\big). \tag{2}$$

The error of this finite difference approximation [3], defined by the difference between approximate and analytic solution, has two main sources; *round off error*, which is the loss of precision due to computers rounding of floating point numbers, and *local truncation error*, which arises due to the discretisation of the problem. It can be shown [4] that this discretisation error is of the order of the step size squared (see Eq.(2)) thus approximation accuracy (and also simulation time) increase dramatically with decreased step size.

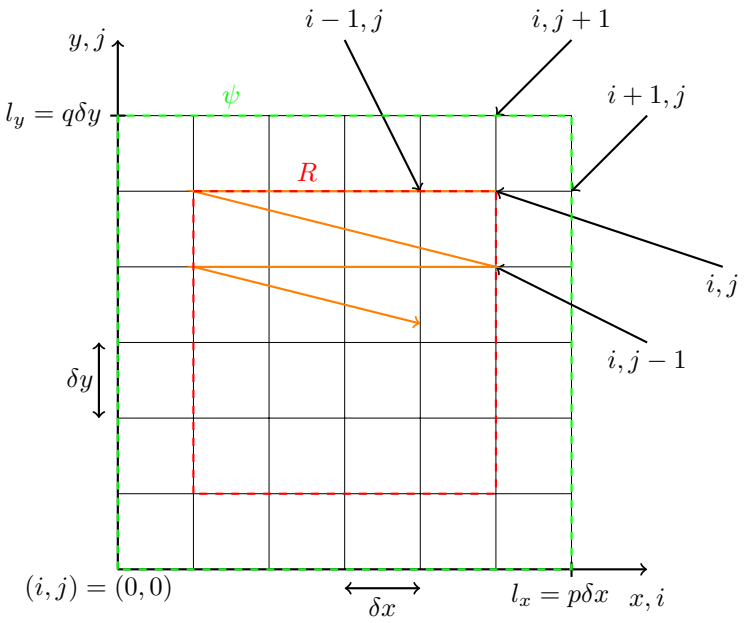


Figure 1: Depicts a grid layout for $\delta x = \delta y$ with $p = q = 7$. ψ is defined for all points within the green dotted line, and specified on the boundary, and R is defined within the red dotted line. Arrows in the top right indicate the indexes used in numerical calculations. The orange line indicates the systematic manner in which R and ψ are iterated over.

We can apply this approximation of ψ to each point inside of a domain (the green dotted line in Fig.(1)), giving a system of linear equations for $\psi_{i,j}$. This is quite tricky to solve analytically, so an iterative procedure may be used as follows. Eq.(1) can be resolved for $\psi_{i,j}$ as

$$\psi_{i,j} = \frac{\frac{\psi_{i-1,j}+\psi_{i-1,j}}{(\delta x)^2} + \frac{\psi_{i,j+1}+\psi_{i,j-1}}{(\delta y)^2}}{2\left(\frac{1}{(\delta x)^2} + \frac{1}{(\delta y)^2}\right)}. \tag{3}$$

Using an the *successive over-relaxation* method allows us to solve this with an iterative approach. We consider

$$\psi^{m+1}_{i,j} = \alpha\overline{\psi}_{i,j} + (1-\alpha)\psi^m_{i,j}, \tag{4}$$

where m is an *iteration number*, $\overline{\psi}_{i,j}$ is calculated via Eq.(3), and α is the *over relaxation parameter*.

For the domain being a uniform grid (constraining $\delta x = \delta y$), Eq.(4) can be written as

$$\psi^{m+1}_{i,j} = \psi^m_{i,j} + \frac{\alpha R^m_{i,j}}{4}, \tag{5}$$

where the *iterated residuals* matrix elements $R^m_{i,j}$ are given by

$$R^m_{ij} = \psi^m_{i-1,j} + \psi^m_{i-1,j} + \psi^m_{i,j+1} + \psi^m_{i,j-1} - 4\psi^m_{i,j}. \tag{6}$$

We can iterate through the grid as shown in Fig.(1), sweeping through systematically as is shown by the orange line, applying Eq.(5) and Eq.(6) in order to solve for ψ in the region.

The residuals are reduced each iteration (assuming convergence), and when the residuals R^m_{ij} are zero (or within some arbitrarily close value), $\psi^{m+1}_{i,j} = \psi^m_{i,j}$ and since the solution is found, iteration may be stopped. If the residuals don't decrease to within some bound of zero within a specified number of iterations, the process should be terminated to prevent wasting computation power.

The over-relaxation parameter $\alpha \in (1,2)$ has optimum value given by [5]

$$\alpha_{\text{optimal}} = \frac{2}{1+\sqrt{1-\left[\frac{\cos\pi/p+\cos\pi/q}{2}\right]^2}} \overset{p=q}{=} \frac{2}{1+\sin\pi/p}. \tag{7}$$

3.2 Ohter Numerical Methods for Solving PDEs

Though the Finite Differences method (FDM) is a very commonly used method for solving PDEs due to it's usability and versatility, there are many other methods which you could use instead. Some of these include;

- Spectral methods [6]: You write a general solution as a sub of basis functions (like a Fourier series) and use mathematical methods (life FFTs) to choose coefficients of these functions which closely satisfy the differential equation and replicate the boundary conditions.
- Domain decomposition methods [7]: Split a boundary value problem into smaller subdomains, then iteratively solve those smaller domains in conjunction with each other using the boundary conditions of adjacent ones.
- Finite element methods [8]: Like the FDM it splits the problem into small discrete elements, but it uses calculus of variations in order to minimise an error function, which, when minimised, implies the solution is appropriately approximated.

4 Implementation

A few functions were created to help solving the problem. These were then implemented in the main script, and the scalar field ψ was calculated for specific parameters. A description of each the functions/script used is:

4.1 Function: `solve_laplace`

`solve_laplace` computes the successive over relaxation iteration. It has inputs: `psiInitial` ($p \times q$ array–initial guess for ψ, including boundary conditions), `alpha` (float–the over-relaxation parameter), `maxIterations` (integer–the maximum number of iterations before termination), `p` and `q` (integers–the number of nodes in the grid in the x, y directions), and `accuracyCriterion` (float–the minimum value of each $R^m_{i,j}$ for early termination). It outputs; `psi` (p by q array–the final scalar field of ψ after termination), `historicalValues` ($? \times 3$ array–the value of ψ for each iteration in 3 places (bottom left, middle, top right, excluding boundary points)), and `converge` (Boolean–encoding whether the sequence converged (terminated early) or reached `maxIterations` iterations).

Firstly, it initialises any arrays needed through out the function. The program then has three nested for loops. The first iterates `i1` from 1 to `maxIterations`, the next iterates `j` from 1 to $q-2$ (sweeping down the y axis of the residuals matrix), and the last iterates `i` from 1 to $p-2$ (moving along the x axis from right to left). In the two nested for loops (the ones iterating over `i` and `j`) `R` (an array containing the elements of the residuals matrix) and `psi` are updated (note, `psi` is updated when each new residual is calculated to improve efficiency). When `j` has incremented by one, the values of `psi` at the desired points are appended to the `historicalValues` matrix. The function checks whether each element in `R` is smaller than `accuracyCriterion` using an if statement. If they are, a message stating how many iterations it took to reach convergence is printed, `convergence` is set to `True` and the code breaks, i.e., exits the first for loop (iterating over `i1`) and terminates the program. If the residues aren't smaller than the desired accuracy but `i1` iterated `maxIterations` times, a message is printed stating desired convergence levels weren't reached, and `convergence` is set to `False`. If neither of those if statements are satisfied, the program increments `i1` and repeats until either is satisfied. In this section, it was crucial to think carefully about indexes since R and ψ are offset by one),

4.2 Function: `heat_map_2d`

`heat_map_2d` takes `psi` $p \times q$ array–the potential to be plotted on a heatmap), `historicalValues` ($? \times 3$ array–the historical values of `psi` at three points), `number` (integer–the number of the figure being produced for labelling purposes), `alpha` (the value of alpha at which this `psi` was calculated), and `converge` (whether `psi` was deemed to have converged).

`heat_map_2d` is simply a function to plot both the heat map of `psi` (to visualise ψ), and a bar graph of the historical values of `psi` (in order to see how quickly it converged (or didn't converge)). The function then automatically saves the figures with distinct names so they can easily be identified. It was much easier to do this in a function as it had to be called multiple times and de-cluttered code. It does this all using generic graphing tools.

4.3 Main: `CO25_laplaces_equation`

The main script is simply used to run the desired functions under the conditions which you want. You declare the variables `p`, `q`, `length_x`, `length_y`, and specify the boundary conditions on the edge of the domain. Using a for loop, it calculates the values of `psi` at the boundary with the desired resolution, and places these in a $p \times q$ matrix of zeros,

`psiInitial`. You then choose `maxIterations`, `accuracyCriterion`, and `alphaList` (which is a list of the values of alpha you'd like to call `solve_laplace` for).

In another for loop, the code iterates through the elements in `alphaList`, calls `solve_laplace` on it and stores the results in arrays or cells as required. It also calls `heat_map_2d` to output and save the figure containing the heat map of `psi` and bar chart of `historicalValues` for each alpha. Finally, it displays the numerical values of `psi` and `historicalValues`.

4.4 Features of the code

The code made was designed to be adaptable. Features highlighting this include:

- Grid size, maximum number of iterations, and accuracy criteria can be chosen arbitrarily, though this may increase computation time.
- The boundary conditions may be changed easily, although an analytic solution will still need to be manually computed for full utility.
- Given a list of over-relaxation parameters, the code successively computes the solution for each and plots them, along with their convergence properties. You don't need to run code multiple times for different α.
- Figures are automatically labeled with relevant data, and saved with a specific name as a .png file.

5 Analysis and Results

5.1 Analytic solution

Considering the case of the unit square domain with $l_x = l_y = 1$ divided into a grid by $p = q = 7$

$$\psi_{\text{boundary}} = \begin{cases} 0 & x = 0 \quad \text{and} \quad y \in [0,1] \\ 0 & x \in [0,1] \quad \text{and} \quad y = 0 \\ \sin 1 \sinh y & x = 1 \quad \text{and} \quad y \in [0,1] \\ \sin x \sinh 1 & x \in [0,1] \quad \text{and} \quad y = 1 \end{cases}. \tag{8}$$

By observation, a solution to this problem (which, by the uniqueness theorem is the only solution) is

$$\psi(x, y) = \sin x \sinh y. \tag{9}$$

This was plotted on a heatmap in Fig.(2) using the code in Appx.(C). This provided visualisation of the boundary conditions, and the scalar field that the algorithm should converge to, allowing any problems within the code to be resolved quickly.

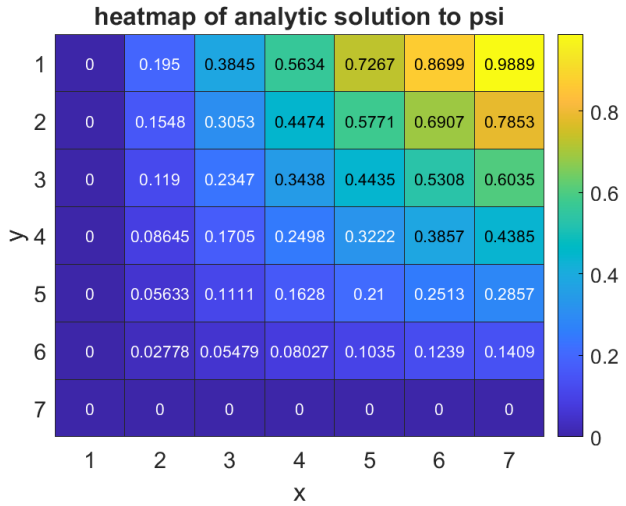
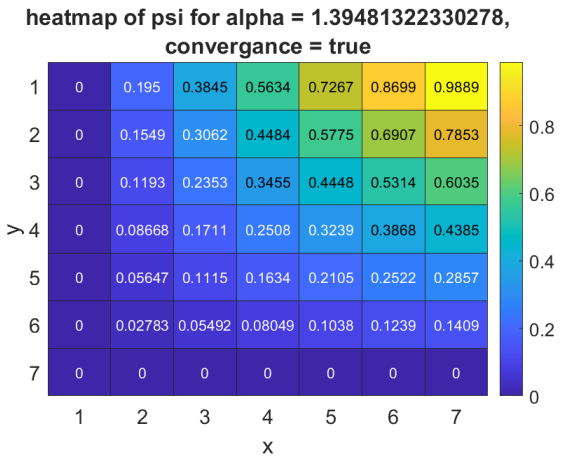


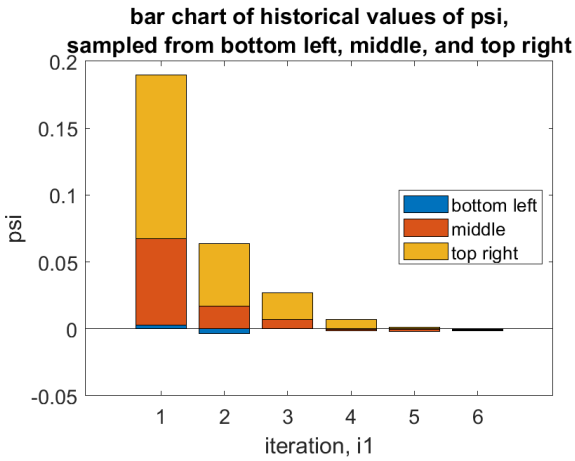
Figure 2: heatmap of analytic solution to ψ, given in Eq.(C)

5.2 Computed solutions

With constraints `maxIterations` = 30 and `accuracyCriterion` = 0.01 implemented in the code, we calculate the numerically approximated solution to Laplace's equation (with the above boundary conditions) for different values of `alpha`. These approximations were individually plotted. A heatmap of the final scalar field ψ, along with a bar graph showing historical values of ψ taken from sample points in the `psi` array during the iteration process were produced, and accuracy, convergence, and oscillatory nature were analysed.



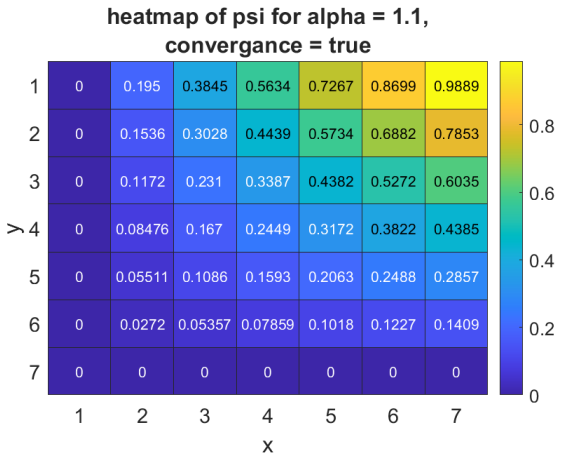
(a) heatmap of ψ for $\alpha \approx 1.39$



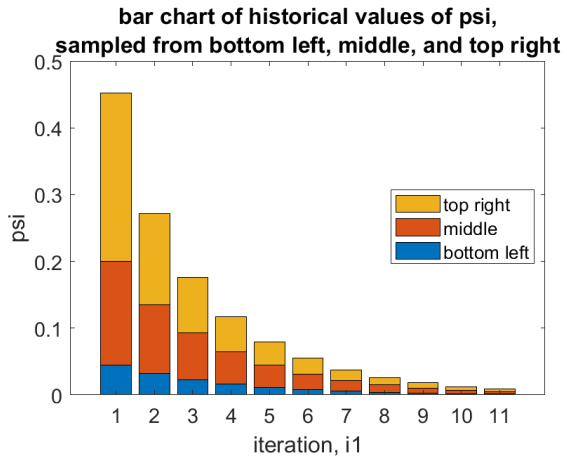
(b) bar graph of historical values of ψ sampled from bottom left, middle, and top right for $\alpha \approx 1.39$

Figure 3

For $\alpha = \alpha_{\text{optimal}} \approx 1.39$ (depicted in Fig.(3)), we had all $R_{i,j} < 0.01$ within 6 iterations, i.e., convergence was deemed to be reached within 6 iterations. Note how convergence is not fully monotonic–at iteration 2 (amongst others), the blue bar has a value less than zero, so oscillation is very minorly oscillatory.



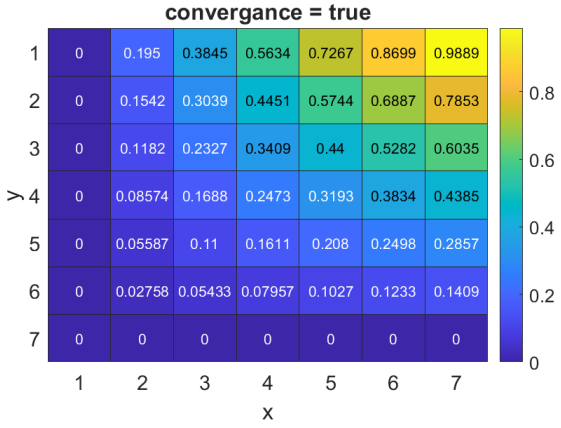
(a) heatmap of ψ for $\alpha = 1.1$



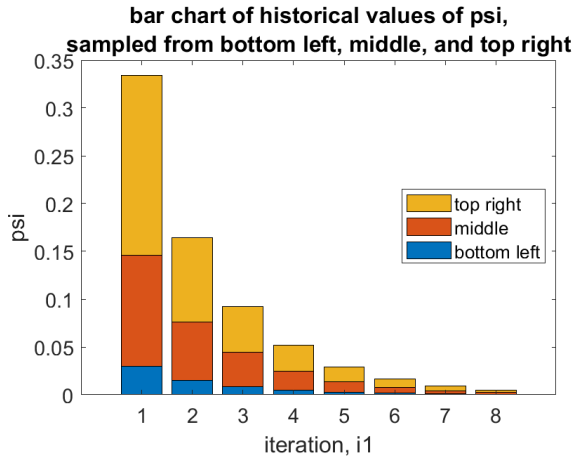
(b) bar graph of historical values of ψ sampled from bottom left, middle, and top right for $\alpha = 1.1$

Figure 4

For $\alpha = 1.1$ (in Fig.(4)), we had required convergence levels within 11 iterations. Unsurprisingly, due to an over-relaxation parameter quite far from the optimal value, the number of iterations for convergence has increased. Due to positively of each bar in the left graph, the convergence is fully monotonic.



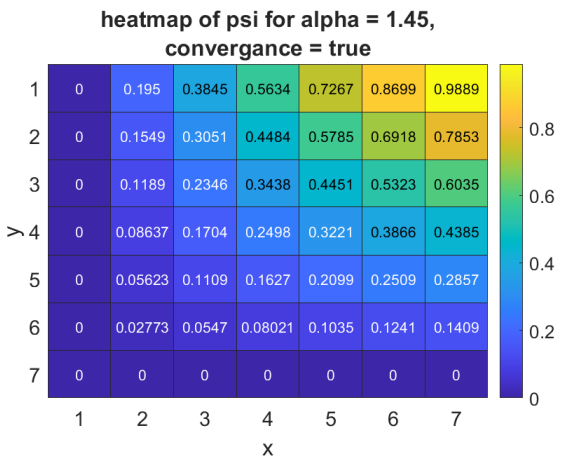
(a) heatmap of ψ for $\alpha = 1.25$



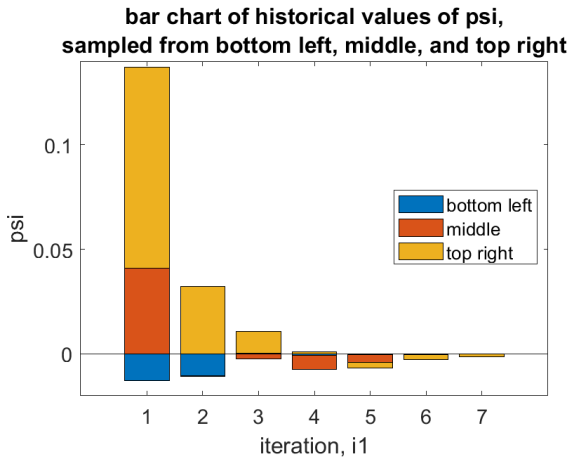
(b) bar graph of historical values of ψ sampled from bottom left, middle, and top right for $\alpha = 1.25$

Figure 5

For $\alpha = 1.25$ (Fig.(5)), convergence was reached in 8 iterations. Here, α is closer to α_{opt} so convergence is faster than the previous case, but slower than the first optimal one. Again, convergence is fully monotonic.



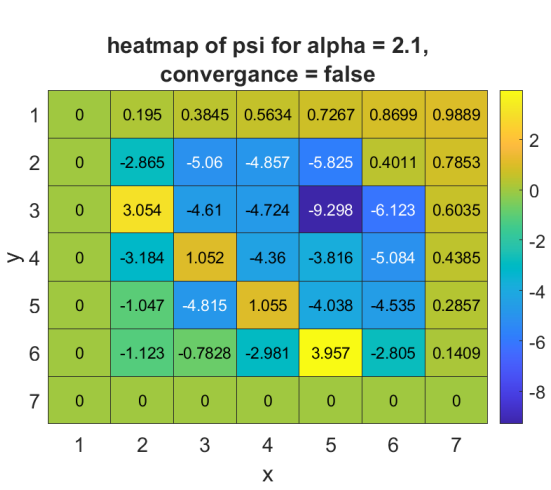
(a) heatmap of ψ for $\alpha = 1.45$



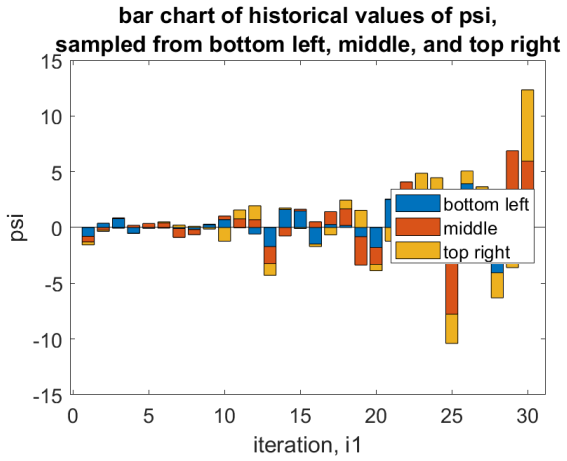
(b) bar graph of historical values of ψ sampled from bottom left, middle, and top right for $\alpha = 1.45$

Figure 6

For $\alpha = 1.45$ (depicted in Fig.(6)), we had required convergence levels within 7 iterations. It's interesting to note that, at higher α, there seems to be more oscillation in the convergence–as you can clearly tell that the left figure is oscillatory than the α_{opt} one. It seems that at higher α, the last term in Eq.(5) tends to 'overshoot', and needs to be negative in order to revert back to the actual solution–in some cases, it'll be too negative leading to the oscillation. As the residuals decrease, the oscillation dies down and the solution converges. For higher α, by Eq.(5), this overshooting is larger thus more oscillation.



(a) heatmap of ψ for $\alpha = 2.1$



(b) bar graph of historical values of ψ sampled from bottom left, middle, and top right for $\alpha = 2.1$

Figure 7

For $\alpha = 2.1$ (Fig.(7)), required convergence was not reached within 30 iterations. This is not surprising, given the condition on the over relaxation parameter was $\alpha \in [1, 2]$, so 2.1 is out of range. You can see from the left figure that 'convergence' is highly oscillatory, reinforcing the assumption about overshooting–in this case, it overshoots by more each iteration, thus the attempted solution diverges away from the analytic one. There seems to be very little pattern in the heatmap, and no way to discern the analytic solution from it–it seems like, with an out of bounds relaxation parameter, the solution will always fully diverge.

6 Conclusion

In this experiment, we discussed how to find a linear set of equations via the finite differences method, which were then solved using successive over relaxation. The main focus of the experiment was on how different values for the over-relaxation parameter affected convergence. We confirmed that the optimal over relaxation parameter for a 7×7 grid was $\alpha \approx 1.39$. We saw as the over relaxation parameter deviated from this value, the number of iterations required for convergence increased. Furthermore, we saw at higher values of α, convergence was oscillatory, whereas at lower values it was monotonic, and we discussed why. In this report, we discussed why the code (found in the appendices) was written in a sensible , such as to allow for changes in boundary conditions, grid sizes, and other parameters in order for the code to work for a general case.

This research could be furthered by solving Laplace's equation under different boundary conditions, and seeing how that affected convergence. If you used a large grid size (and a higher maximum number of iterations and accuracy criteria), you could solve for a lot of values of alpha (say, 100 evenly spaced between 1 and 2), and plot a graph of the conversion times–you should expect the peak to be at α_{opt}. It would be interesting to confirm that the runtime increased on the order of the square of the grid size, which could have been done using the '`tic, toc`' feature inbuilt into `MATLAB`. Doing that would also allow us to confirm that the optimal over relaxation parameter changed based upon the size of the grid. Finally, it would be interesting to use some of the other PDE solving methods mentioned in the Theory section, however the FDM model was used due to it's utility, and other methods are often far more convoluted and may be suited to solving different problems.

References

[1] URL `https://www.andrew.cmu.edu/course/24-681/handouts/lectures/fdm_for_laplace_equation.pdf`.

[2] Wen Shen. Ch11 1. finite difference method for laplace equation in 2d. wen shen, May 2015. URL `https://www.youtube.com/watch?app=desktop&v=wBlrncpJixk&ab_channel=wenshenpsu`.

[3] Feb 2024. URL `https://en.wikipedia.org/wiki/Finite_difference_method#Accuracy_and_order`.

[4] URL `https://www3.nd.edu/~coast/jjwteach/www/www/60130/CourseLectureNotes/EE-5.pdf`.

[5] URL `https://www-teaching.physics.ox.ac.uk/practical_course/scripts/`.

[6] Feb 2024. URL https://en.wikipedia.org/wiki/Spectral_method.

[7] Apr 2023. URL https://en.wikipedia.org/wiki/Domain_decomposition_methods.

[8] Feb 2024. URL https://en.wikipedia.org/wiki/Finite_element_method.

A Code for CO25_laplaces_equation main

```
clc
clear variables
format long

% declare dimensions of domain and relevant variables.
p = 7;
q = 7;
length_x = 1;
length_y = 1;
delta_x = length_x / (p - 1);
delta_y = length_y / (q - 1);

%initialise psi, specify boundary conditions.

psiInitial = zeros(p, q);

for i1 = 1:p
    psiInitial(1, i1) = sin((i1 - 1) * delta_x) * sinh(1); %top
    psiInitial(q, i1) = 0; %bottom
end
for i2 = 1:q
    psiInitial(q-i2+1, 1) = 0; %left
    psiInitial(q-i2+1, p) = sin(1) * sinh((i2 - 1) * delta_y); %right
end

psiTrue = analytic_solution(p, q);

% maximum number of iterations and accuracy criterion
maxIterations = 30;
accuracyCriterion = 0.01;

% specify alpha (or calculate it) (from hereout, working under constraint p = q)
alpha_chosen = 1.35;
alpha_optimal = 2 / (1 + sin(pi / p));
alpha = alpha_optimal;

% initialiselists and cell arrays
psiList = {};
historicalValuesCellArray = {};
convergeList = {};
barCharts = {};

% iterate thorugh desired alpha values, call solve_laplace, plot and display data.
alphaList = [alpha, 1.1, 1.25, 1.45, 2.1];

% apply function for each alpha, and call plotting function.
for i3 = 1:length(alphaList)
    [psiList{i3}, historicalValuesCellArray{i3}, convergeList{i3}] = ...
        solve_laplace(psiInitial, alphaList(i3), maxIterations, p, q, accuracyCriterion, psiTrue);
    heat_map_2d(psiList{1, i3}, historicalValuesCellArray{1, i3}, i3, alphaList(i3), convergeList{1, i3});
    disp('The final calculated scalar field \psi was: ');
    disp(psiList{1, i3});
    disp('And the historical values for \psi in the; bottom left, middle, top left, are: ');
    disp(historicalValuesCellArray{1, i3});
end
```

B Code for solve_laplace function

```
function [psi, historicalValues, converge] = solve_laplace(psiInitial, ...
    alpha, maxIterations, p, q, accuracyCriterion, psiTrue)

%     Author: Alex McInerny, Date: 07/12/2023
%     This function solves Laplace's equation (\del^2 \psi = 0) using the
%       over-relaxation method.
%
%     Inputs:
%     * psi_initial: a 2D array containing the intial \psi, including boundary values.
%     * alpha: the coefficient of over-relaxation.
%     * N_iterations: maximum number of iternations performed.
%     * p, q: number of grids in the box, in the x and y direction.
%     * accuracyCriterion: required level of accuracy
%     * psiTrue: analytic solution for psi.
%
%     Outputs:
%     * psi: a 2D array of the value of \psi up to N_iter iterations.
%     * historical_values: a 2D array (N_iter x 3) of historical values of the iteration -
%       one in the upper half, one in the middle, and one in the lower half.
%     * converge: boolean indecating whether the function converged.
%
%     Constraints:
%     * The boundaries of \psi are kept constant during the iterations.

    % initiates variables used throught
    R = ones(p-2,q-2);
    psi = psiInitial;
    historicalValues = zeros(1,3);

    % applies successive over-relaxation
    for i1 = 1:maxIterations
        for j = 1:q-2
            for i = 1:p-2
                R(j,p-i-1) = psi(j+2,p-i) + psi(j,p-i) + psi(j+1,p+1-i) + ...
                    psi(j+1,p-1-i) - 4 * psi(j+1,p-i);
                psi(j+1,p-i) = psi(j+1,p-i) + alpha / 4 * R(j, p-i-1);
            end
        end

        historicalValues(i1, 1) = psiTrue(q-2,2) - psi(q-2,2); % saves sample values of psi each iteration
        historicalValues(i1, 2) = psiTrue(floor(q/2),floor(p/2)) - psi(floor(q/2),floor(p/2));
        historicalValues(i1, 3) = psiTrue(2,p-2) - psi(2,p-2);

        % checks if solution has conveged to required level.
        if all(abs(R) < accuracyCriterion) == true(p-2,q-2)
            fprintf(['This sequence (alpha = %.15g) converged (all R_ij < %.15g)' ...
                ' within %.15g iterations.\n'], alpha, accuracyCriterion, i1);
            converge = true;
            break
        elseif i1 == maxIterations
            fprintf(['This sequence (alpha = %.15g) did not converge (some R_ij > %.15g)' ...
                ' within %.15g iterations.\n'], alpha, accuracyCriterion, i1);
            converge = false;
        else
            continue
        end
    end
```

C Code for `analytic_solution` function

```
function [psiTrue] = analytic_solution(p, q)

%     Author: Alex McInerny, Date: 07/12/2023
%     This function plots and retuerns the analytic solution to laplace's equation.
%
%     Inputs:
%     * p, q: related to the number of descrete points inside the domain.

    x = linspace(0,1,p); y = linspace(0,1,q); psiTrue = zeros(p,q);

    for j = 1:length(y)
```

```
        for i = 1:length(x)
            psiTrue(q+1-j,i) = sin(x(i)) * sinh(y(j));
        end
    end

    heatmap(psiTrue); colormap default; ax = gca; ax.FontSize = 14;
    title('heatmap of analytic solution to psi'); xlabel('x'); ylabel('y');
    saveas(gcf, 'psi_analytic_solution_heatmap', 'png')
```

D Code for `heat_map_2d` function

```
function heat_map_2d(psi, historicalValues, number, alpha, converge)

%     Author: Alex McInerny, Date: 07/12/2023
%     This function plots graphs of heatmaps to the approximate solutions to psi,
%       as well as their convergance properties, and saves them as pngs.
%
%     Inputs:
%     * psi: a 2D array of the value of \psi up to N_iter iterations.
%     * historical_values: a 2D array (N_iter x 3) of historical values of the iteration -
%       one in the upper half, one in the middle, and one in the lower half.
%     * number: figure number
%     * alpha: value of alpha for required plot
%     * converge: did the required plot converge?

    logicalString = {'false', 'true'};
    fprintf(['Figures %d for alpha = %.15g is saved as psi_heatmap_fig%d and ' ...
        'psi_historical_values_fig%d \n'], number, alpha, number, number);

    heatmap(psi); colormap default
    ax = gca; ax.FontSize = 14;
    title({sprintf('heatmap of psi for alpha = %.15g,', alpha), ...
        sprintf('convergance = %s', logicalString{converge + 1})})
    xlabel('x'); ylabel('y')
    savefig(sprintf('psi_heatmap_fig%d', number))
    saveas(gcf, sprintf('psi_heatmap_png_fig%d', number), 'png')

    bar(historicalValues, 'stacked'); ax = gca; ax.FontSize = 14;
    title({'bar chart of historical values of psi,',['sampled from ' ...
        'bottom left, middle, and top right']})
    legend({"bottom left", "middle", "top right"}, location = 'east')
    xlabel('iteration, i1'); ylabel('psi');
    savefig(sprintf('psi_historical_values_fig%d', number))
    saveas(gcf, sprintf('psi_historical_values_png_fig%d', number), 'png')
```