University of Sydney

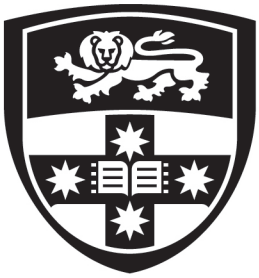
Engineering Analysis

AMME2000

ASSIGNMENT 2

Author:
Clay Johnson

Student ID:
450410211



THE UNIVERSITY OF
SYDNEY

Contents

Abstract **2**

1 Section 1: Analytical solution to the 2D Heat Equation **3**
1.1 Boundary Conditions . . . 3
1.2 First Dirichlet Boundary Condition . . . 3
1.2.1 Derivation of Fourier Series Coefficients . . . 5
1.3 Second Dirichlet Boundary Condition . . . 6
1.4 Solution Plot . . . 7

2 Section 2: Numerical Solutions to the Laplace Equation **8**
2.1 Forward in Time, Central in Space (FTCS) . . . 8
2.2 Iterative Central Difference Approximation . . . 10
2.3 Matrix Method Central Difference Approximation . . . 12

3 Section 3: Numerical solution to Poisson's Equation **14**
3.1 Applying the Matrix Method to Poisson's Equation . . . 14
3.2 Contour Plots . . . 15
3.3 Temperature Variance . . . 16

Appendix **17**

Abstract

In this report, several solutions to a heat problem (Figure 1) are developed and explored. The analytical solution is derived and plotted to provide a baseline for comparison and understanding. Several numerical solutions are implemented and their performance and accuracy is discussed.

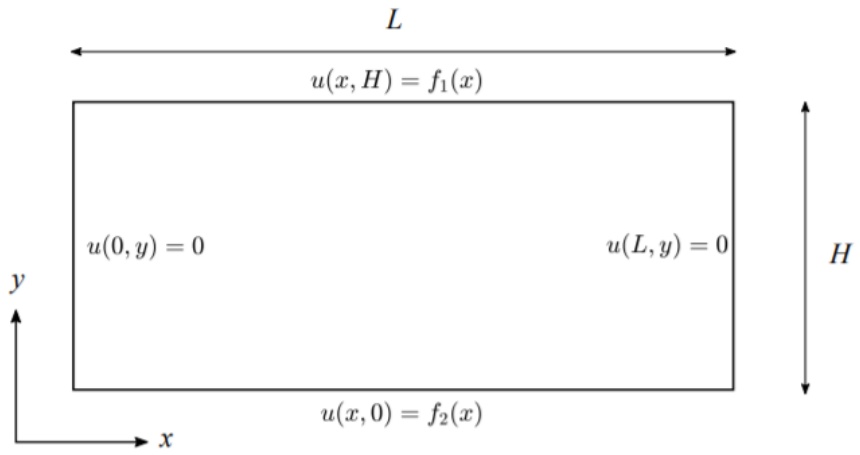


Figure 1: Diagram of temperature boundary conditions and domain size

The temperature profiles at the top and bottom of the domain are

$$f_1(x) = \frac{-4T_1}{L^2}x(x-L) \tag{1}$$

$$f_2(x) = \frac{-4T_2}{L^2}x(x-L) \tag{2}$$

1 Section 1: Analytical solution to the 2D Heat Equation

In this section the analytical solution is derived and discussed. The approach for solving problems with multiple Dirichlet boundary conditions is demonstrated in Figure 2.

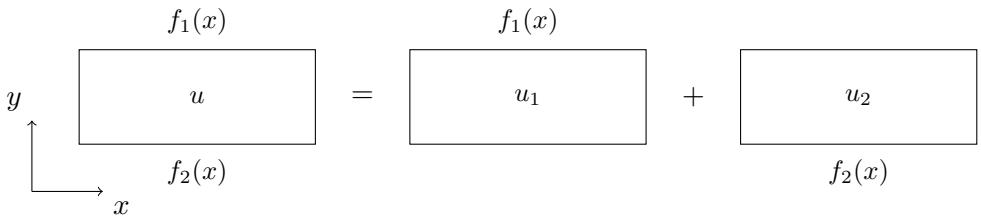


Figure 2: Solution approach for multiple Dirichlet boundary conditions

Using this approach, the full solution is obtained by summing solutions to simpler sub-problems

$$u(x,y) = u_1(x,y) + u_2(x,y)$$

1.1 Boundary Conditions

The boundary conditions are rewritten for the two sub-problems. Noting that both u_1 and u_2 are zero at $x = 0$ and $x = L$, the boundary conditions are

$$u_1(x,y) = \begin{cases} 0 & y = 0 \\ f_1(x) & y = H \end{cases} \quad \text{and} \quad u_2(x,y) = \begin{cases} f_2(x) & y = 0 \\ 0 & y = H \end{cases}$$

1.2 First Dirichlet Boundary Condition

We assume that the dependence of the solution on each dimension can be separated such that

$$\boxed{u_1(x,y) = F(x)G(y)} \tag{3}$$

Taking the second partial derivatives of u_1 with respect to x and y and introducing a separation constant results in two ordinary differential equations

$$F_{xx} + p^2 F = 0 \text{ and } G_{yy} - p^2 G = 0$$

Solving the ODE for F results in a Fourier series of the form

$$F_n(x) = A_n^{c1} cos(px) + B_n^{c1} sin(px)$$

Applying the boundary condition at $x = 0$ implies that $A_n^{c1} = 0$, meaning the Fourier series has sine terms only. In that case, the boundary condition at $x = L$ is satisfied when px is an integer multiple of pi. The Fourier series solution for a single mode n is then

$$\boxed{F_n(x) = B_n^{c1} sin(px) \text{ where } p = n\pi/L} \tag{4}$$

The second ODE can be solved for G with the value for p determined above

$$G_n(y) = A_n^{c2} e^{py} + B_n^{c2} e^{-py}$$

Applying the boundary condition at $y = 0$ implies that $B_n^{c2} = -A_n^{c2}$. Factoring out the constant and noting that $2sinh(x) = e^x - e^{-x}$ the solution for a single mode n is

$$\boxed{G_n(y) = 2A_n^{c2} sinh(py) \text{ where } p = n\pi/L} \tag{5}$$

Combining Eqn.(4) and Eqn.(5) gives the solution for a single mode n as

$$\boxed{u_n(x,y) = A_n^{c3} sinh(py) sin(px)} \tag{6}$$

Where we have combined the two coefficients so that $A_n^{c3} = 2A_n^{c2} B_n^{c1}$. The full solution described in Eqn.(1) is then a summation of Eqn.(6) for all modes beginning at $n = 1$ since $n = 0$ is zero

$$u_1(x,y) = \sum_{n=1}^{\infty} A_n^{c3} sinh(py) sin(px)$$

Apply the non-zero boundary condition at $y = H$ to solve for the remaining constant A_n^{c3}

$$u_1(x,H) = f_1(x) = \sum_{n=1}^{\infty} A_n^{c3} sinh(pH) sin(px)$$

Note that $A_n^{c3} sinh(pH)$ is replaced with B_n as it is constant for a single mode n. Making this substitution gives a Fourier sine series in x from which we can compute B_n

$$u_1(x,H) = f_1(x) = \sum_{n=1}^{\infty} B_n sin(px)$$

Recalling that $p = n\pi/L$ the full solution with the substitution from the previous step is then

$$\boxed{u_1(x,y) = \sum_{n=1}^{\infty} B_n \frac{sinh\left(\frac{n\pi y}{L}\right)}{sinh\left(\frac{n\pi H}{L}\right)} sin\left(\frac{n\pi x}{L}\right)} \tag{7}$$

The derivation of B_n is shown below, beginning with an examination of the standard derivation of Fourier Sine series coefficients

$$B_n = \frac{1}{L} \int_{-L}^{L} f(x) sin\left(\frac{n\pi x}{L}\right) dx$$

1.2.1 Derivation of Fourier Series Coefficients

The standard derivation of coefficients requires periodicity of the integrand on the domain $[-L, L]$. Upon inspection it is clear that our integrand is not periodic since $f(x)$ is a parabola centred at $L/2$, with roots at 0 and L. However, the domain of our problem is only the region between the roots of the parabola, which can be expressed as a Fourier sine series (we can see this is true by our derivation of Eqn.(4)). If that region of $f(x)$ was the right half of a periodic function on $[-L, L]$ then the standard derivation of constants would apply. Such a function would be odd and the corresponding integrand would be both periodic and even. For an even function, the integral can be rewritten with bounds $[0, L]$. This is significant because now we are only integrating over the domain of interest, where we have shown the standard derivation of coefficients is valid for $f(x)$.

$$\boxed{B_n = \frac{2}{L}\int_0^L f(x) sin\left(\frac{n\pi x}{L}\right) dx} \tag{8}$$

Substituting Eqn.(1) into Eqn.(8) and letting $p = n\pi/L$ gives

$$B_n = \frac{2}{L}\int_0^L \frac{-4T}{L^2} x(x-L) sin(px) dx$$

The constant term is brought out the front and the integral is distributed across the expanded form of the expression

$$= -\frac{8T_1}{L^3}\left[\int_0^L x^2 sin(px)dx - L\int_0^L x sin(px)dx\right]$$

Integrate by parts ($\int uv^{'}dx = uv - \int u^{'}vdx$), letting $u = x^2$ and $v^{'} = sin(px)$ for the first integral. For the second integral let $u = x$ and $v^{'} = sin(px)$

$$= -\frac{8T_1}{L^3}\left[-\frac{x^2}{p}cos(px)\Big|_0^L + \frac{2}{p}\int_0^L x cos(px)dx - L\left[-\frac{x}{p}cos(px)\Big|_0^L + \frac{1}{p}\int_0^L cos(px)dx\right]\right]$$

The first integral cannot be evaluated as it is so we apply integration by parts a second time, letting $u = x$ and $v^{'} = cos(px)$

$$= -\frac{8T_1}{L^3}\left[-\frac{x^2}{p}cos(px)\Big|_0^L + \frac{2}{p}\left[\frac{x}{p}sin(px)\Big|_0^L - \frac{1}{p}\int_0^L sin(px)dx\right] - L\left[-\frac{x}{p}cos(px)\Big|_0^L + \frac{1}{p^2}sin(px)\Big|_0^L\right]\right]$$

Evaluate the remaining integral and apply the limits to integrated terms

$$= -\frac{8T_1}{L^3}\left[-\frac{L^2}{p}cos(pL) + \frac{L^2}{p} + \frac{2}{p}\left[\frac{L}{p}sin(pL) + \frac{1}{p^2}cos(pL) - \frac{1}{p^2}\right] - L\left[-\frac{L}{p}cos(pL) + \frac{L}{p} + \frac{1}{p^2}sin(pL)\right]\right]$$

After collecting like terms and simplifying where appropriate, the expression is

$$= -\frac{8T_1}{L^3}\left[\frac{1}{p^3}(2cos(pL) - 2 + pLsin(pL))\right]$$

Remembering $p = n\pi/L$ and that sine is zero for integer multiples of pi

$$\boxed{B_n = -8T_1\frac{(2cos(n\pi) - 2)}{n^3\pi^3}} \tag{9}$$

Substituting the result from Eqn.(9) into Eqn.(7) yields the full solution to the case with Dirichlet boundary condition at the top of the problem domain

$$\boxed{u_1(x,y) = -8T_1 \sum_{n=1}^{\infty} \frac{(2cos(n\pi) - 2)}{n^3\pi^3} \frac{sinh\left(\frac{n\pi y}{L}\right)}{sinh\left(\frac{n\pi H}{L}\right)} sin\left(\frac{n\pi x}{L}\right)} \tag{10}$$

1.3 Second Dirichlet Boundary Condition

The solution method for the case with Dirichlet boundary condition at the bottom of the problem domain is the same as the method in Section 1.2 with a shift in reference axis. We move the y axis by H such that the problem is now defined on (x, q) where $q = y - H$. The problem domain doesn't change in x but the y domain is now $-H \leq q \leq 0$. The governing equation for this case is then

$$\boxed{u_2(x,q) = F(x)G(q)} \tag{11}$$

Rewriting the boundary conditions in terms of q for u_2 (noting there is no change at $x = 0$ and $x = L$ as x has not been transformed) gives

$$u_2(x,q) = \begin{cases} 0 & q = 0 \\ f_2(x) & q = -H \end{cases}$$

The symmetry of this problem to the previous problem means the derivation of Eqn.(4) and Eqn.(5) applies here too. We can see this is true since the Dirichlet boundary condition is at the non-zero edge of the y domain in both cases. As such, Eqn.(6) can be rewritten here in terms of x and q

$$u_n(x,q) = A_n^{c3} sinh(pq) sin(px)$$

It follows then that the full solution is of the same form as Eqn.(7) but defined on (x, q) and characterised by the non-zero boundary condition at $q = -H$

$$\boxed{u_2(x,q) = \sum_{n=1}^{\infty} B_n \frac{sinh\left(\frac{n\pi q}{L}\right)}{sinh\left(\frac{-n\pi H}{L}\right)} sin\left(\frac{n\pi x}{L}\right)} \tag{12}$$

Given the form of $f_1(x)$ is the same as $f_2(x)$, the careful treatment of B_n in Section 1.2 holds here as well. As such, the full solution to the case with Dirichlet boundary condition at the bottom of the problem domain is the same except q replaces y. By recalling $q = y - H$ the solution on (x, y) is

$$\boxed{u_2(x,y) = -8T_2 \sum_{n=1}^{\infty} \frac{(2cos(n\pi) - 2)}{n^3\pi^3} \frac{sinh\left(\frac{n\pi(y-H)}{L}\right)}{sinh\left(\frac{-n\pi H}{L}\right)} sin\left(\frac{n\pi x}{L}\right)} \tag{13}$$

1.4 Solution Plot

The temperature profile for $u(x, y)$ is plotted in Figure 3. The plot was made using the output of analyticalSol, a custom function in MATLAB (see appendix). The function computes the first 100 non-zero Fourier terms, producing a consistent and reliable plot. The mesh size increases with the function input while preserving the 1:3 domain ratio and ensuring uniform grid spacing. In this plot the function input was 3, corresponding to a grid resolution of 240×80 or nearly 20 thousand unique points plotted.

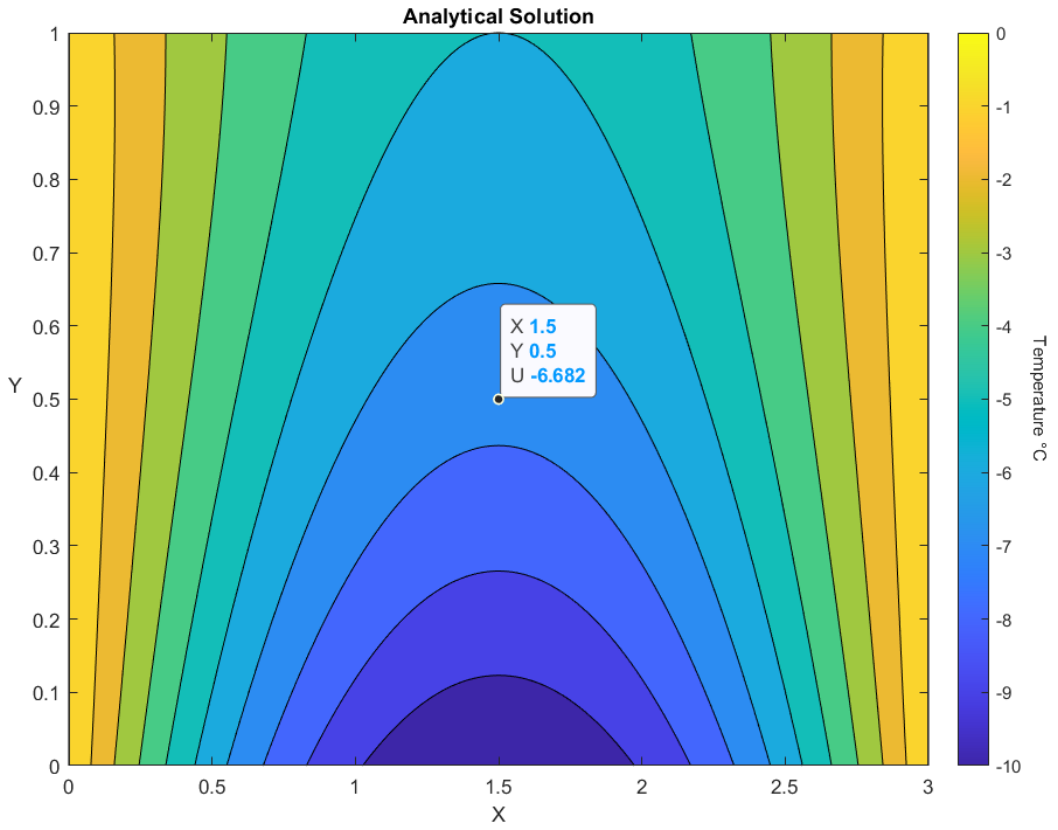


Figure 3: Solution profile with the temperature at the centre displayed

Figure 4 demonstrates the known behaviour of the problem at the boundaries; the temperature is zero on the left and right edge, halfway along the top edge is right on the $-5°C$ level line and similarly halfway along the bottom edge coincides with $-10°C$. Overall the temperature distribution throughout the domain is continuous, so the plot confirms the expected behaviour of the problem.

2 Section 2: Numerical Solutions to the Laplace Equation

This section explores three numerical algorithms for the Laplace equation and verifies them against the solution in Section 1. Each algorithm will be evaluated in terms of accuracy and computational efficiency. All computation is performed on an Intel Core i7-7700HQ CPU @ 2.80GHz; a modern processor that should provide a good indication of a given algorithm's computational efficiency.

2.1 Forward in Time, Central in Space (FTCS)

The FTCS scheme is a finite difference method for numerically approximating solutions to parabolic partial differential equations. The method uses the initial conditions of the problem to evolve the solution forward in time. Conceptually, the change in temperature of a point in space, for a small time step, depends on the temperature of the neighbouring points. That is, hot regions will cool and cold regions will heat until the distribution of heat in the system is at equilibrium

$$\boxed{u_t(x,y,t) = D\big(u_{xx}(x,y,t) + u_{yy}(x,y,t)\big)} \tag{14}$$

The spatial components of Eqn.(14) are discretised by applying the central difference method twice. The resulting discretisation is central-in-space (meaning the previous, current and next point are used to approximate the change in temperature for the current point) with second order error

$$u_{xx}(x_i,t^n) = \frac{u^n_{i+1} - 2u^n_i + u^n_{i-1}}{\Delta x^2} \quad \text{and} \quad u_{yy}(y_j,t^n) = \frac{u^n_{j+1} - 2u^n_j + u^n_{j-1}}{\Delta y^2}$$

The temporal component of Eqn.(14) is discretised with a forward-in-time difference method called Euler time integration. In order to ensure stability of the scheme, the time step is constrained such that it effectively reduces the error to second order

$$u_t(x_i,y_j,t^n) = \frac{u^{n+1}_{i,j} - u^n_{i,j}}{\Delta t}$$

Substituting the numerical approximations for u_{xx}, u_{yy} & u_t and choosing equal grid spacing such that $\Delta x = \Delta y = \Delta s$, gives the FTCS scheme for the 2D Heat Equation

$$\boxed{u^{n+1}_{i,j} = u^n_{i,j} + \frac{D\Delta t}{\Delta s^2}(u^n_{i+1} - 2u^n_i + u^n_{i-1} + u^n_{j+1} - 2u^n_j + u^n_{j-1})} \tag{15}$$

This numerical approximation is implemented as a function in MATLAB with the tic and toc commands placed either side of the main loop structure in the code (see appendix). The algorithm is run for 5 increasingly finer grid resolutions and their comparative difference to the analytical solution in Section 1 is plotted in Figure 4. The time taken for the computation of each is plotted in Figure 5.

Figure 4 initially indicates that accuracy of the FTCS increases with finer grid spacing. However, after $n = 3$ the difference between the FTCS and the analytical solution actually increases. Whilst initially troubling, a closer look at the values computed for higher resolutions suggests that MATLAB struggles with repeated near zero floating point arithmetic.

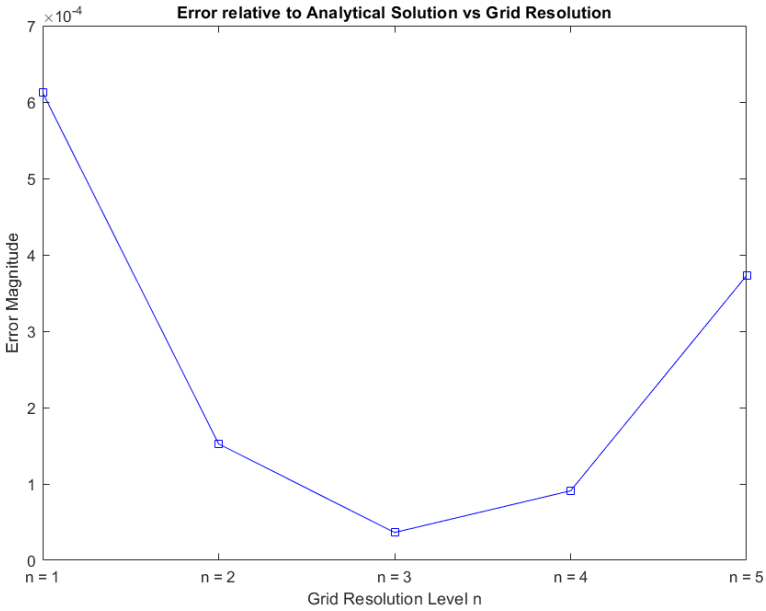


Figure 4: Error and grid resolution for the FTCS stencil

Figure 5 shows fast computation time for $n = 1, 2$ and 3 before the implementation becomes significantly more computationally expensive. At $n = 5$ this solution took 29 minutes and 41 seconds.

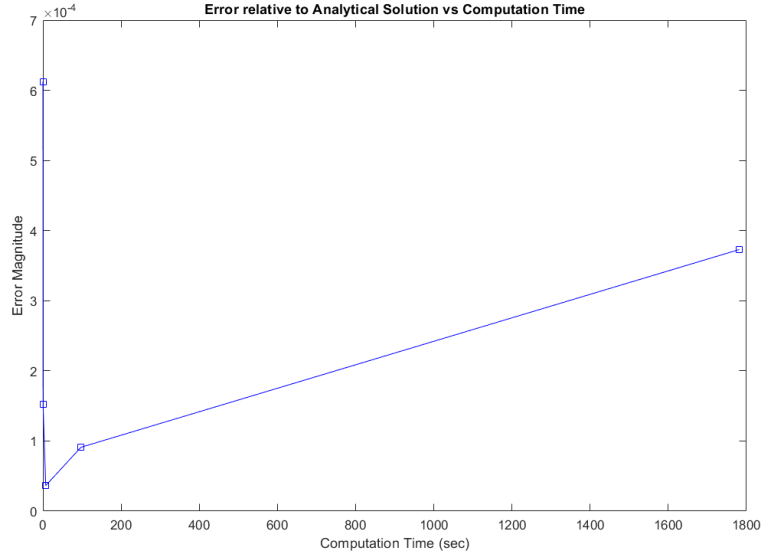


Figure 5: Solution accuracy and computation time for the FTCS stencil

2.2 Iterative Central Difference Approximation

The Central Difference approximation is another finite difference method. In this case it is used to numerically approximate a solution to the Laplace equation. Similar to the FTCS method, the Central Difference approximation models the change in temperature of a point as a function of the neighbouring points

$$u_{i,j} = \frac{1}{4}(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}) \tag{16}$$

The stencil in Eqn.(16) is implemented in MATLAB. Similar to the FTCS implementation, the algorithm has tic and toc commands either side of the main loop structure (see appendix). The algorithm progressively computes values for all x on a given y. The difference between solutions is calculated with each full pass of the algorithm until it is within the error tolerance.

Figure 6 shows a similar trend in error for the varying grid spacings. Whilst not ideal, it does confirm that both approximations behave similarly. This is somewhat intuitive given that both stencils are based on a Central Difference approximation and as such, have the same order of error.

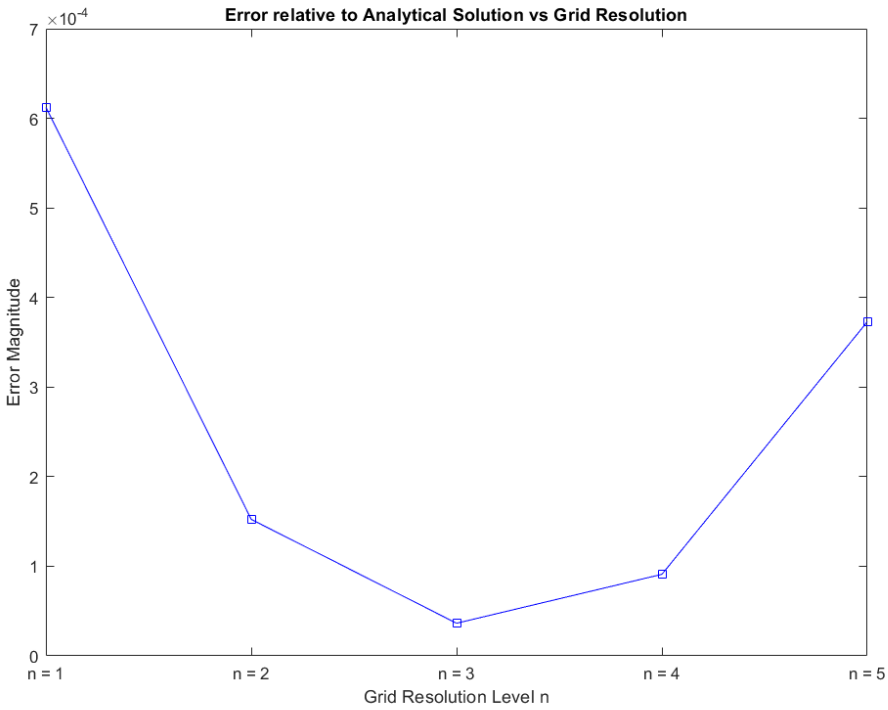


Figure 6: Error and grid resolution for the iterative implementation of the Central Difference stencil

The same trend seen in Figure 5 for the FTCS implementation is repeated here. From the similarity in stencils and results it is reasonable to infer that both methods are behaving as they should, albeit limited for larger n values. For $n = 5$ this method was slightly faster than the FTCS method, finishing at 27 minutes and 17 seconds. The major difference between the two methods is the time step in the FTCS method. Since it is proportional to grid spacing, FTCS stencils can take exponentially longer to compute for increasingly finer grid resolutions.

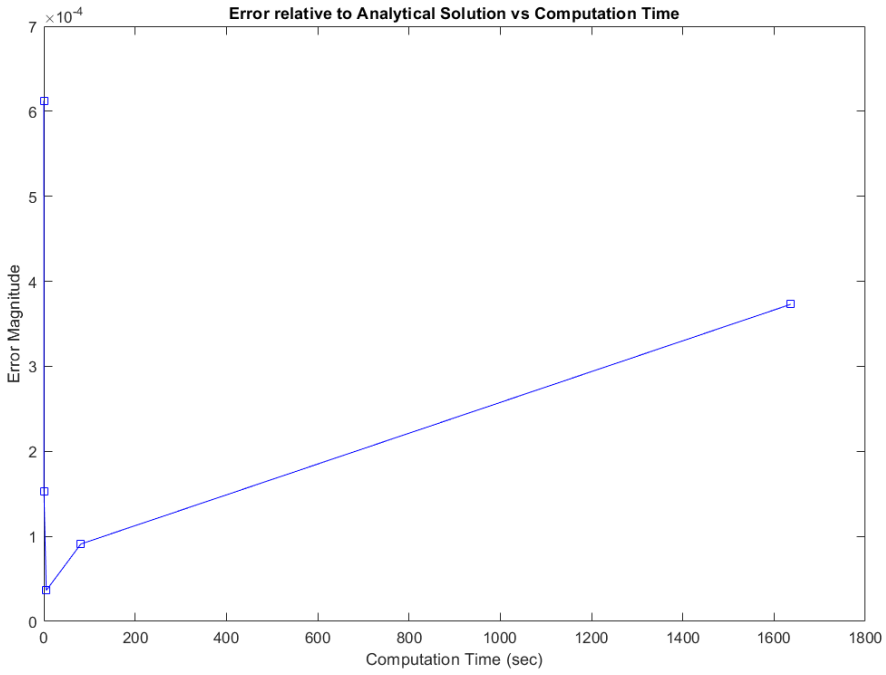


Figure 7: Solution accuracy and computation time for the iterative Central Difference stencil

2.3 Matrix Method Central Difference Approximation

The same stencil in Eqn.(16) is implemented in this section as a means of numerically approximating a solution to the Laplace equation. However, by noticing that the stencil is applied to a finite number of points without changing, the solution can be expressed as a system of simultaneous equations. Doing so means the problem can solved using matrix algebra, the operations for which are highly optimised in MATLAB. In matrix form, the equation can be written as

$$\boxed{U = A^{-1}B} \tag{17}$$

Where U is an ordered array of all unknown points, A is a matrix containing the coefficients of the stencil for each point and B is an ordered array containing the boundary conditions for the problem.

The timing of this algorithm accounts for the fact that constructing the matrices described can be computationally expensive for a large numbers of points. As such, the tic and toc commands are placed outside the initialisation of the matrices in the code (see appendix). It should be noted that the solution can be easily generalised for most problems by fixing the grid resolution as constant. Doing so would significantly reduce the amount of computation involved in setting up the matrices for subsequent problems. In fact, only the array of boundary conditions would need to be recomputed for each problem.

Figure 8 demonstrates a significant and continuous decrease in error for increasing grid resolutions. This is extremely promising as it overcomes the major shortfall of the iterative solutions by avoiding near zero floating point arithmetic. Following the trend at $n = 4$ and 5 it doesn't seem that finer grid resolutions would yield any noticeable improvements in accuracy. It's likely that the limit of accuracy for the Central Difference model is met when $n \geq 4$.

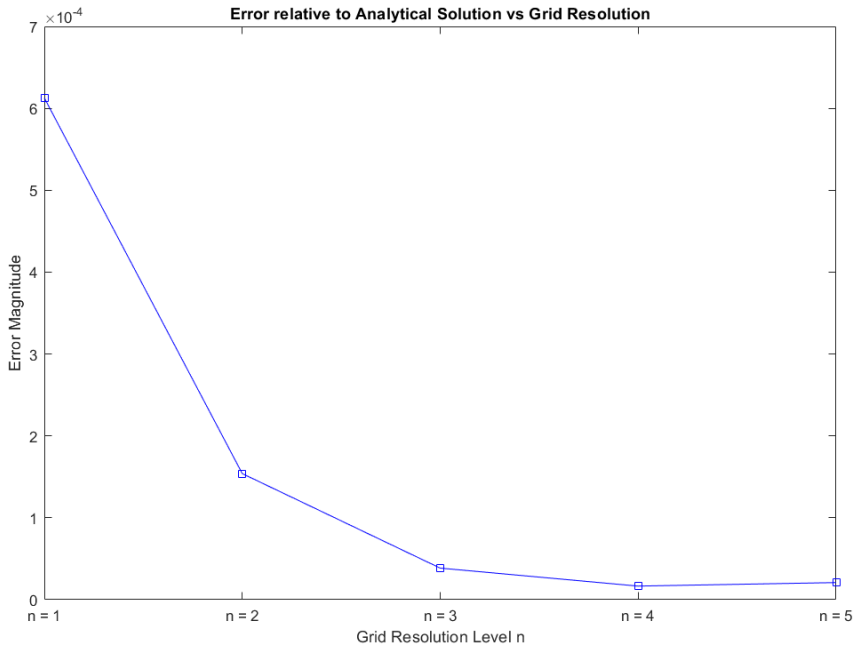


Figure 8: Error and grid resolution for the matrix method the Central Difference stencil

Further improvements over the FTCS and iterative Central Difference methods are indicated in Figure 9. Whilst all 3 methods compute the first 4 n values in roughly the same time, the matrix method is by far the more efficient method for significantly fine grid resolutions. This is most likely due to optimisation within MATLAB for matrix algebra. For instance, the 'sparse' matrix in MATLAB is used for efficiently storing and performing operations on large matrices. Taking advantage of these optimisations resulted in a roughly 30% faster solution when $n = 5$, with the matrix method running for 19 minutes and 26 seconds.

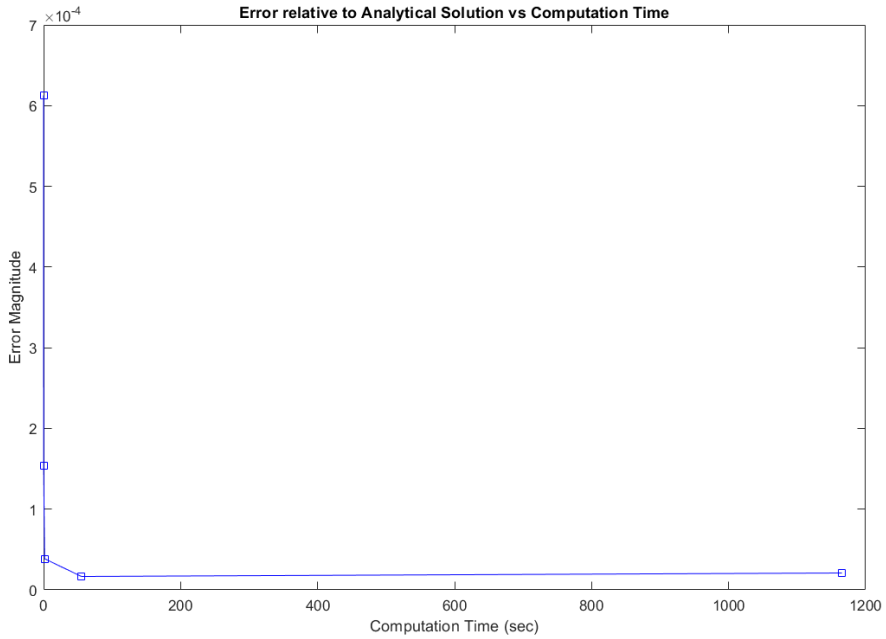


Figure 9: Solution accuracy and computation time for the matrix method Central Difference stencil

Overall, the best solution approach for this problem is the matrix method. Modelling the Central Difference stencil for each point as a system of equations and solving with MATLAB is faster and more accurate than the other methods explored in this section. The major drawback of this method is that it requires significantly more computer storage. This is due to the potentially massive coefficients matrix needed to solve this problem, which is the amount of points being solved for *squared*. For $n = 5$ in this implementation, the coefficients matrix was 307200$\times$ 307200. The default precision for matrices in MATLAB is 32 bits, meaning our coefficients matrix alone would have been almost 11 Gigabytes in size. The sparse function in MATLAB allows arrays of mostly zeros to be stored efficiently, in some cases the resulting sparse matrix is 30% the size of the original.

3 Section 3: Numerical solution to Poisson's Equation

In this section the matrix method from Section 2 is implemented for Poisson's Equation

$$u_{xx}(x,y) + u_{yy}(x,y) = p(x,y) \tag{18}$$

For this problem there are two Gaussian source/sink terms in the domain and the temperature at the boundaries is held at zero. The function $p(x,y)$ is

$$p(x,y) = P_1 e^{\left[-\left(\frac{(x-1)^2}{0.1}\right)-\left(\frac{(y-0.5)^2}{0.05}\right)\right]} + P_2 e^{\left[-\left(\frac{(x-2)^2}{0.1}\right)-\left(\frac{(y-0.5)^2}{0.05}\right)\right]} \tag{19}$$

Where $P_1 = 2°C/m^2$ and $P_2 = -4°C/m^2$

3.1 Applying the Matrix Method to Poisson's Equation

The matrix method will be used to numerically approximate Eqn.(18). This method was chosen as it is easily repurposed to solve this problem. Additionally, the results from Section 2 indicate that the matrix method is faster and more accurate than the iterative approaches. The major differences between this problem and the one solved in Section 2 is the introduction of a non-zero p function and all the boundary conditions held at zero. As such, the solution approach differs such that the Central Difference stencil in Eqn.(16) is rewritten to include the p function

$$u_{i,j} = \frac{1}{4}(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} + p(x_i, y_j)h^2) \tag{20}$$

Thus the solution matrix in Eqn.(17) becomes

$$U = A^{-1}P + A^{-1}B \tag{21}$$

Where P is an array containing the values of $p(x_i, y_j)h^2$. This is implemented in MATLAB (see appendix) and solved for $ds = 1/2^n$ where $n = 1, 2$ and 3. The two source/sink terms in Eqn.(19) will be iteratively calculated for each grid sizing and the P array will be populated with the values.

3.2 Contour Plots

Figure 10 shows significant difference for varying spatial steps in the solution to Eqn.(18). Noticeably, the maximum contour level line rapidly increases between the first and second plot. This behaviour is due to the sink/source terms included in the left and right halves of the domain.

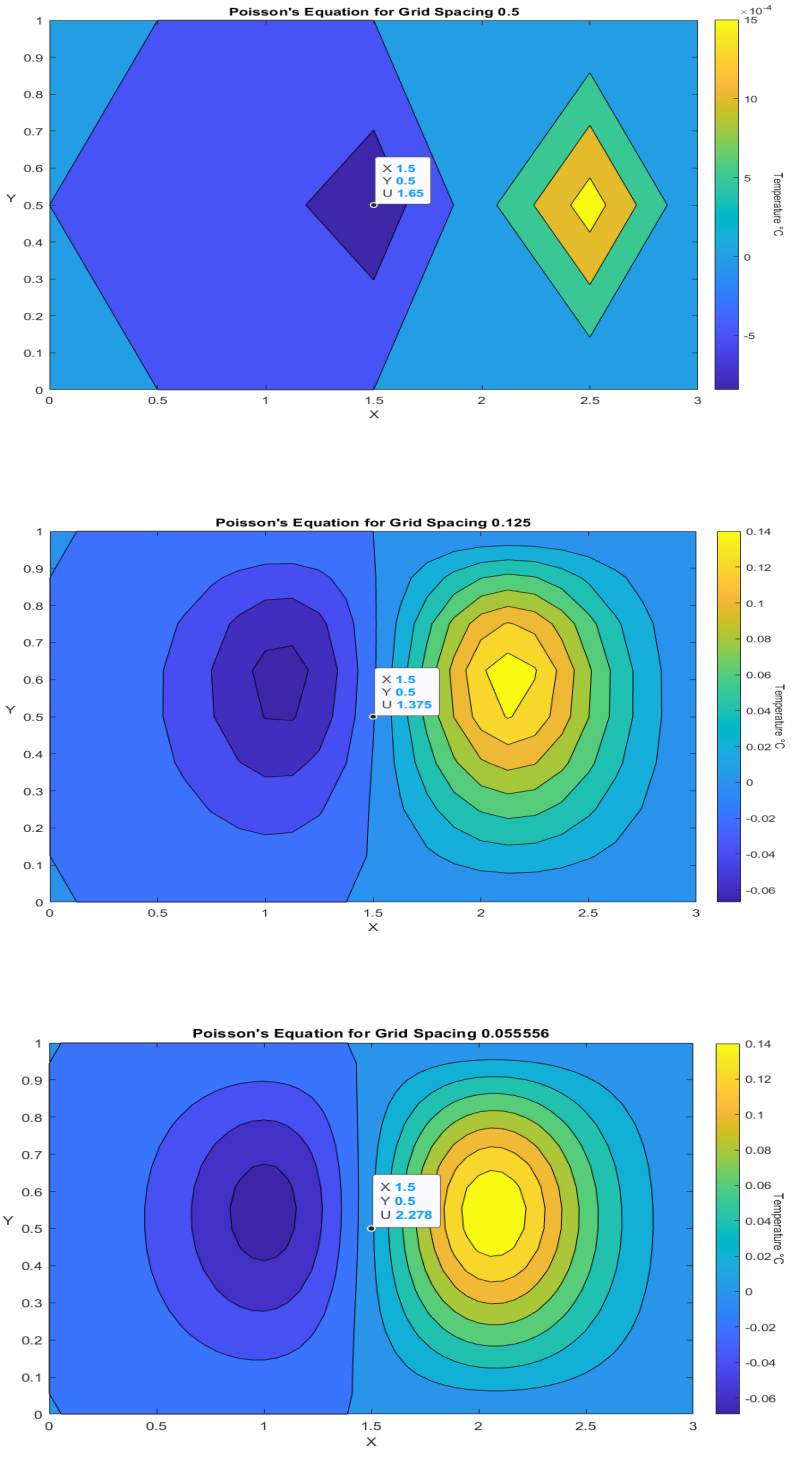


Figure 10: Solution profile for Poisson's Equation with varying spatial step

3.3 Temperature Variance

The calculated temperature at the domain centre is also different for all three plots, as shown in Figure 11. Note that the temperature at the centre doesn't change by more than a degree, this is likely caused by large changes in the gradient of the solution around the sink/source components. Such changes would be represented very differently over intervals between points of varying distance

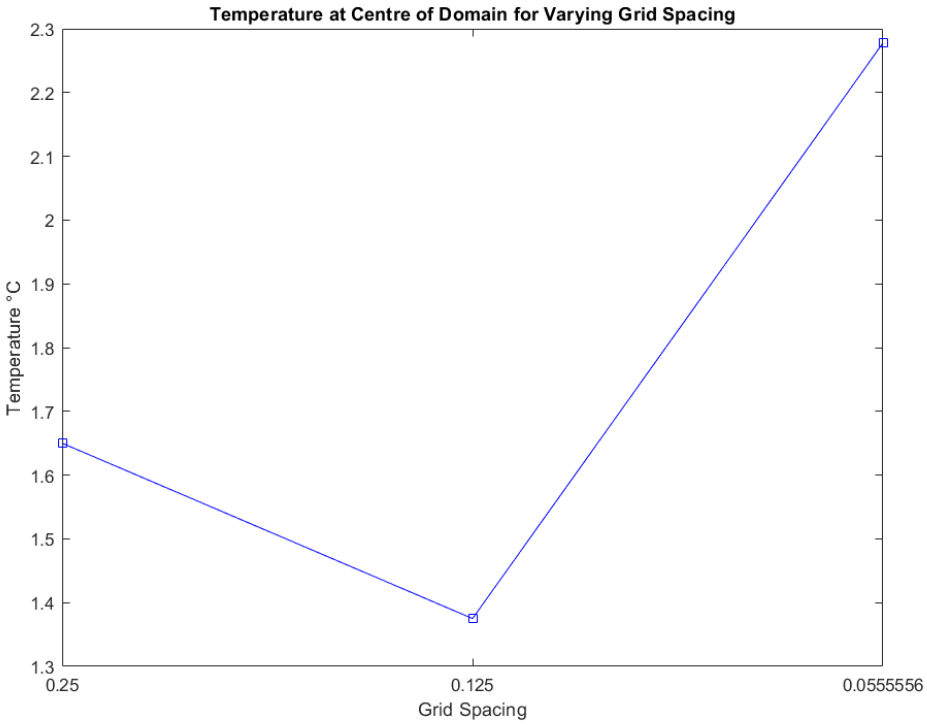


Figure 11: Change in temperature at the centre of the problem domain

The code used to generate a numerical solution to Poisson's Equation is included in the appendix. For the most part it is the same as the implementation from Section 2 where a matrix method was explored. The major difference is that an array of solutions to Eqn.(19) had to be computed iteratively, which would be incredibly inefficient for finer grid resolutions. It is likely that this would affect the overall computational time of this implementation and as such any future implementations of this method should begin by optimising that process where possible.

Appendix

The MATLAB code used in this report to compute solutions and plot figures is included below

Analytical Solution

```
function [U] = analyticalSol(n)
%% Analytical solution to a 2D heat problem
% n             - spatial step factor
% U             - analytical solution for 100 non-zero fourier terms

% solution parameters
ds = 0.1/2^n;   % spatial grid size
L = 3;          % x domain is (0,L)
H = 1;          % y domain is (0,H)
T1 = -5;        % temperature for u1
T2 = -10;       % temperature for u2

% initialise domain
x = 0:ds:L;
y = 0:ds:H;
nx = length(x);
ny = length(y);

for pos = 1:nx
    f1(pos) = (-4*T1/L^2)*x(pos)*(x(pos)-L);
    f2(pos) = (-4*T2/L^2)*x(pos)*(x(pos)-L);
end

% initialise solution output
U = zeros(ny,nx);
U(end,:) = f1;
U(1,:) = f2;

%time the computation
tic;
% run the calculation for 100 non-zero fourier terms
for nfs = 1:2:200
    bN = (2*cos(nfs*pi) - 2)/((nfs^3)*(pi^3)); % calculate fourier coefficient

    % loop over each x point
    for i = 2:nx-1
        xTerm = sin(nfs*pi*x(i)/L); % calculate current x component

        % loop over each y point
        for j = 2:ny-1
            u1 = (-8)*T1*bN*xTerm*sinh(nfs*pi*y(j)/L)/sinh(nfs*pi*H/L); % calculate one
                unique point with u1
            u2 = (-8)*T2*bN*xTerm*sinh(nfs*pi*(y(j)-H)/L)/sinh(((-nfs)*pi*H)/L); % calculate
                 one unique point with u2
            U(j,i) = U(j,i) + u1 + u2; % add solutions from u1 and u2
        end
    end
end
% finish timing
t = toc;
end
```

Forward-Time, Central-Space Approximation

```
function [U,t,Err] = ftcs(n)
%% Numerical approximation of 2D heat equation
% n             - spatial step factor
% U             - solution values
% t, Err        - time and error for simulation
A = analyticalSol(n); % analyticalSol function for error calculations

% initialise stencil variables
L = 3;          % domain length
H = 1;          % domain height
D = 0.5;        % Diffusivity less than one
ds = 0.1/2^n;   % spatial step size depends on input n
dt = ds^2/2;    % time step depends on ds
T1 = -5;        % boundary temp 1
T2 = -10;       % boundary temp 2

% define solution domain
x = 0:ds:L;
y = 0:ds:H;
f1 = x;
f2 = x;
nx = length(x);
ny = length(y);

% calculate boundary conditions
for pos = 1:nx
    f1(pos) = (-4*T1/L^2)*x(pos)*(x(pos)-L);
    f2(pos) = (-4*T2/L^2)*x(pos)*(x(pos)-L);
end

% initialise solution matrix
Unp1 = zeros(ny,nx);
Unp1(end,:) = f1;   % top boundary y = H
Unp1(1,:) = f2;     % bottom boundary y = 0
Unp1(:,1) = 0;      % left boundary = 0
Unp1(:,end) = 0;    % right boundary = 0

% loop until error converges
err = 1;        % initial error value
tol = 1e-8;     % tolerance for error convergence
tic;            % begin timing
while  err > tol
    Un = Unp1; % update solution grid

    % loop over solution grid except for boundaries
    for j = 2:ny-1
        for i = 2:nx-1
            Unp1(j,i) = Un(j,i) + (D*dt/ds^2)*(Un(j,i+1) - 2*Un(j,i) + ...
                        Un(j,i-1) + Un(j+1,i) - 2*Un(j,i) + Un(j-1,i));
        end
    end
    % calculte difference between iterations
    err = max(abs(Unp1(:)-Un(:)));
end
t = toc; % stop timing
U = Unp1; % Set output variable

%% error calculation - largest difference between two corresponding points in U and A
Err = max(abs(U(:)-A(:)));
end
```

Iterative Central Difference Approximation

```
function [U,t,Err] = lpl_cd(n)
%% Numerical approximation of Laplace equation
% n             - spatial step factor
% U             - solution values
% t, Err        - time and error for simulation
A = analyticalSol(n); % analyticalSol function for error calculations

% initialise stencil variables
L = 3;          % domain length
H = 1;          % domain height
ds = 0.1/2^n;   % spatial step size depends on input n
T1 = -5;        % boundary temp 1
T2 = -10;       % boundary temp 2

% define solution domain
x = 0:ds:L;
y = 0:ds:H;
f1 = x;
f2 = x;
nx = length(x);
ny = length(y);

% calculate boundary conditions
for pos = 1:nx
    f1(pos) = (-4*T1/L^2)*x(pos)*(x(pos)-L);
    f2(pos) = (-4*T2/L^2)*x(pos)*(x(pos)-L);
end

% initialise solution matrix
Unp1 = zeros(ny,nx);
Unp1(end,:) = f1;   % top boundary y = H
Unp1(1,:) = f2;     % bottom boundary y = 0
Unp1(:,1) = 0;      % left boundary = 0
Unp1(:,end) = 0;    % right boundary = 0

% loop until error converges
err = 1;        % initial error value
tol = 1e-8;     % tolerance for error convergence
tic;            % begin timing
while  err > tol
    Un = Unp1; % update solution grid

    % loop over solution grid except for boundaries
    for j = 2:ny-1
        for i = 2:nx-1
            Unp1(j,i) = (1/4)*(Un(j+1,i) + Un(j-1,i) + Un(j,i+1) + Un(j,i-1));
        end
    end
    % calculte difference between iterations
    err = max(abs(Unp1(:)-Un(:)));
end
t = toc; % stop timing
U = Unp1; % Set output variable

%% error calculation - largest difference between two corresponding points in U and A
Err = max(abs(U(:)-A(:)));
end
```

Matrix Central Difference Approximation

```
function [U,t,Err] = lpl_matrix(n)
%% Numerical approximation of Laplace equation
% n             - spatial step factor
% U             - solution values
% t, Err        - time and error for simulation
A = analyticalSol(n); % analyticalSol function for error calculations

% initialise stencil variables
L = 3;          % domain length
H = 1;          % domain height
ds = 0.1/2^n;   % spatial step size, depends on input n
T1 = -5;        % boundary temp 1
T2 = -10;       % boundary temp 2

% initialise solution variables
x = 0:ds:L;
y = 0:ds:H;
nx = length(x);
ny = length(y);
f1 = x;
f2 = x;
for pos = 2:nx-1
    f1(pos) = (-4*T1/L^2)*x(pos)*(x(pos)-L);
    f2(pos) = (-4*T2/L^2)*x(pos)*(x(pos)-L);
end

tic; % begin timing
% initialise output matrix
U = sparse(ny,nx);  % sparse matrix for larger n values
U(end,:) = f1;      % top boundary = f1
U(1,:) = f2;        % bottom boundary = f2
U(:,1) = 0;         % left boundary = 0
U(:,end) = 0;       % right boundary = 0

% initialise coefficients matrix M
NX = nx-2;              % solution points in x minus boundary conditions
NY = ny-2;              % solution points in y minus boundary conditions
size = NX*NY;           % unknown points
M = sparse(size,size);  % sparse matrix for larger n values
M(1:size+1:size^2) = -4;            % (i,j) diagonal
M(size+1:size+1:size^2) = 1;        % (i-1,j) diagonal
M(2:size+1:size^2-size) = 1;        % (i+1,j) diagonal
M(NX+1:size+1:size^2-NX*size) = 1;  % (i,j+1) diagonal
M(NX*size+1:size+1:size^2) = 1;     % (i,j-1) diagonal
M(size*(NX-1)+(NX+1):NX*size+NX:size^2) = 0; % i+1 at x = L boundary condition
M(size*NX+NX:NX*size+NX:size^2) = 0;         % i-1 at x = 0 boundary condition

% initialise solution vector Un
Unp1 = sparse(size,1); % solution vector

% initialise coefficients vector B
B = sparse(size,1); % boundary conditions are all zero except top and bottom
B(1:NX) = f2(2:nx-1); % bottom boundary
B(size-(NX-1):size) = f1(2:nx-1); % top boundary

% matrix multiply for solution vector
Unp1 = M\-B;

% supplant converged solution vector in output matrix
% note: reshape to get vector in matrix form, transpose to orientate correctly
U(2:end-1,2:end-1) = transpose(reshape(Unp1,[NX,NY]));
t = toc; % stop timing

%% error calculation - largest difference between two corresponding points in U and A
Err = max(abs(U(:)-A(:)));
end
```

Simulation and Plotting

```
%% this script automates the grid convergence study for each method
nTicks = [1,2,3,4,5]; % for plotting

%% FTCS method
% initialise variables
ftcsT = zeros(1,5);
ftcsErr = zeros(1,5);

% calculate time taken and error for each grid resolution n=1,2,3,4,5
for n = 1:5
   [U,t,Err] = ftcs(n);
   ftcsT(n) = t;
   ftcsErr(n) = Err;
   disp("FTCS simulation " + n + " has finished successfully");
end
disp("All FTCS simulations have now finished successfully");

% error vs grid resolution
figure
plot(nTicks,ftcsErr,'b-s');
xticks([1 2 3 4 5]);
xticklabels({'n = 1','n = 2','n = 3','n = 4','n = 5'});
ylabel("Error Magnitude");
xlabel("Grid Resolution Level n");
title("Error relative to Analytical Solution vs Grid Resolution");

% error vs computation time
figure
plot(ftcsT,ftcsErr,'b-s');
ylabel("Error Magnitude");
xlabel("Computation Time (sec)");
title("Error relative to Analytical Solution vs Computation Time");

%% Central Difference method
% initialise variables
cdT = zeros(1,5);
cdErr = zeros(1,5);

% calculate time taken and error for each grid resolution n=1,2,3,4,5
for n = 1:5
   [U,t,Err] = lpl_cd(n);
   cdT(n) = t;
   cdErr(n) = Err;
   disp("Iterative central difference simulation " + n + " has finished successfully");
end
disp("All Iterative central difference simulations have now finished successfully");

% error vs grid resolution
figure
plot(nTicks,cdErr,'b-s');
xticks([1 2 3 4 5]);
xticklabels({'n = 1','n = 2','n = 3','n = 4','n = 5'});
ylabel("Error Magnitude");
xlabel("Grid Resolution Level n");
title("Error relative to Analytical Solution vs Grid Resolution");

% error vs computation time
figure
plot(cdT,cdErr,'b-s');
ylabel("Error Magnitude");
xlabel("Computation Time (sec)");
title("Error relative to Analytical Solution vs Computation Time");
```

```
%% Matrix method
% initialise variables
mtxT = zeros(1,5);
mtxErr = zeros(1,5);

% calculate time taken and error for each grid resolution n=1,2,3,4,5
for n = 1:5
   [U,t,Err] = lpl_matrix(n);
   mtxT(n) = t;
   mtxErr(n) = Err;
   disp("Matrix central difference simulation " + n + " has finished successfully");
end
disp("All Matrix central difference simulations have now finished successfully");

% error vs grid resolution
figure
plot(nTicks,mtxErr,'b-s');
xticks([1 2 3 4 5]);
xticklabels({'n = 1','n = 2','n = 3','n = 4','n = 5'});
ylabel("Error Magnitude");
xlabel("Grid Resolution Level n");
title("Error relative to Analytical Solution vs Grid Resolution");

% error vs computation time
figure
plot(mtxT,mtxErr,'b-s');
ylabel("Error Magnitude");
xlabel("Computation Time (sec)");
title("Error relative to Analytical Solution vs Computation Time");


%% play chime when simulations finish
sound(sin(1:3000));
disp("All simulations have now finished successfully");
```

Matrix Central Difference Approximation to Poisson's Equation

```
function [X,Y,U,t] = p_matrix(n)
%% Numerical approximation of Poisson's equation
% n             - spatial step factor
% X,Y,U         - meshgrid and solution values for plotting
% t             - time for error to converge

% initialise stencil variables
L = 3;
H = 1;
ds = 1/(2*n^2);   % spatial step size depends on input n
P1 = 2;
P2 = -4;

% define solution domain
x = 0:ds:L;
y = 0:ds:H;
nx = length(x);
ny = length(y);
[X,Y] = meshgrid(x,y);

% initialise output matrix
U = sparse(ny,nx);
U(end,:) = 0;    % top boundary = 0
U(1,:) = 0;      % bottom boundary = 0
U(:,1) = 0;      % left boundary = 0
U(:,end) = 0;    % right boundary = 0

% initialise coefficients matrix M
NX = nx-2;              % solution points in x minus boundary conditions
NY = ny-2;              % solution points in y minus boundary conditions
size = NX*NY;           % unknown points
M = sparse(size,size);  % sparse matrix for larger n values
M(1:size+1:size^2) = -4;            % (i,j) diagonal
M(size+1:size+1:size^2) = 1;        % (i-1,j) diagonal
M(2:size+1:size^2-size) = 1;        % (i+1,j) diagonal
M(NX+1:size+1:size^2-NX*size) = 1;  % (i,j+1) diagonal
M(NX*size+1:size+1:size^2) = 1;     % (i,j-1) diagonal
M(size*(NX-1)+(NX+1):NX*size+NX:size^2) = 0; % i+1 at x = L boundary condition
M(size*NX+NX:NX*size+NX:size^2) = 0;         % i-1 at x = 0 boundary condition

% initialise solution vector Un
Unp1 = sparse(size,1); % solution vector

% initialise P vector
P = zeros(size,1);
p = 1;
for j = 1:NY
    for i = 1:NX
        P(p) = (P1*exp(-(((x(i)-1)^2)/0.1)-(((y(j)-0.5)^2)/0.05))+P2*exp(-(((x(i)-2)^2)/0.1)
            -(((y(j)-0.5)^2)/0.05)))*ds^2;
        p = p + 1;
    end
end
P = sparse(P); % convert to sparse

% initialise coefficients vector B
B = sparse(size,1); % boundary conditions are all zero
B = -B;

% matrix multiply for new solution vector
Unp1 = M\P + M\B;

% supplant converged solution vector in output matrix
% note: reshape to get vector in matrix form, transpose to orientate correctly
U(2:end-1,2:end-1) = transpose(reshape(Unp1,[NX,NY]));
end
```