3
Numerical Solutions of PDEs

There's no sense in being precise when you don't even know what you're talking about.- John von Neumann (1903-1957)

Much of the book has dealt with finding exact solutions to some generic problems. However, many problems of interest cannot be solved exactly. The heat, wave, and Laplace equations are linear partial differential equations and can be solved using separation of variables in geometries in which the Laplacian is separable. However, once we introduce nonlinearities, or complicated non-constant coefficients into the equations, some of these methods do not work. Even when separation of variables or the method of eigenfunction expansions gave us exact results, the computation of the resulting series had to be done on a computer and inevitably one could only use a finite number of terms of the expansion. So, therefore, it is sometimes useful to be able to solve differential equations numerically. In this chapter we will introduce the idea of numerical solutions of partial differential equations. We will introduce the finite difference method and the idea of stability. Other common approaches may be added later.

3.1 The Finite Difference Method

The heat equation can be solved using separation of variables. However, many partial differential equations cannot be solved exactly and one needs to turn to numerical solutions. The heat equation is a simple test case for using numerical methods. Here we will use the simplest method, finite differences.

Let us consider the heat equation in one dimension,

$$u_t = ku_{xx}.$$

Boundary conditions and an initial condition will be applied later. The starting point is figuring out how to approximate the derivatives in this equation.

Recall that the partial derivative, u_t, is defined by

$$\frac{\partial u}{\partial t} = \lim_{\Delta t \to \infty} \frac{u(x, t + \Delta t) - u(x, t)}{\Delta t}.$$

Therefore, we can use the approximation

$$\frac{\partial u}{\partial t} \approx \frac{u(x, t + \Delta t) - u(x, t)}{\Delta t}. \tag{3.1}$$

Forward difference approximation.

This is called a forward difference approximation.

In order to find an approximation to the second derivative, u_{xx}, we start with the forward difference

$$\frac{\partial u}{\partial x} \approx \frac{u(x + \Delta x, t) - u(x, t)}{\Delta x}.$$

Then,

$$\frac{\partial u_x}{\partial x} \approx \frac{u_x(x + \Delta x, t) - u_x(x, t)}{\Delta x}.$$

We need to approximate the terms in the numerator. It is customary to use a backward difference approximation. This is given by letting $\Delta x \to -\Delta x$ in the forward difference form,

Backward difference approximation.

$$\frac{\partial u}{\partial x} \approx \frac{u(x, t) - u(x - \Delta x, t)}{\Delta x}. \tag{3.2}$$

Applying this to u_x evaluated at $x = x$ and $x = x + \Delta x$, we have

$$u_x(x, t) \approx \frac{u(x, t) - u(x - \Delta x, t)}{\Delta x},$$

and

$$u_x(x + \Delta x, t) \approx \frac{u(x + \Delta x, t) - u(x, t)}{\Delta x}.$$

Inserting these expressions into the approximation for u_{xx}, we have

$$\begin{aligned} \frac{\partial^2 u}{\partial x^2} &= \frac{\partial u_x}{\partial x} \\ &\approx \frac{u_x(x + \Delta x, t) - u_x(x, t)}{\Delta x} \\ &\approx \frac{\frac{u(x+\Delta x,t) - u(x,t)}{\Delta x} - \frac{u(x,t) - u(x-\Delta x,t)}{\Delta x}}{\Delta x} \\ &= \frac{u(x + \Delta x, t) - 2u(x, t) + u(x - \Delta x, t)}{(\Delta x)^2}. \end{aligned} \tag{3.3}$$

This approximation for u_{xx} is called the central difference approximation of u_{xx}.

Central difference approximation of u_{xx}.

Combining Equation (3.1) with (3.3) in the heat equation, we have

$$\frac{u(x, t + \Delta t) - u(x, t)}{\Delta t} \approx k \frac{u(x + \Delta x, t) - 2u(x, t) + u(x - \Delta x, t)}{(\Delta x)^2}.$$

Solving for $u(x, t + \Delta t)$, we find

$$u(x, t + \Delta t) \approx u(x, t) + \alpha \left[u(x + \Delta x, t) - 2u(x, t) + u(x - \Delta x, t) \right], \tag{3.4}$$

where $\alpha = k\dfrac{\Delta t}{(\Delta x)^2}$.

In this equation we have a way to determine the solution at position x and time $t + \Delta t$ given that we know the solution at three positions, x, $x + \Delta x$, and $x + 2\Delta x$ at time t.

$$u(x, t + \Delta t) \approx u(x, t) + \alpha \left[u(x + \Delta x, t) - 2u(x, t) + u(x - \Delta x, t)\right]. \qquad (3.5)$$

A shorthand notation is usually used to write out finite difference schemes. The domain of the solution is $x \in [a, b]$ and $t \geq 0$. We seek approximate values of $u(x, t)$ at specific positions and times. We first divide the interval $[a, b]$ into N subintervals of width $\Delta x = (b - a)/N$. Then, the endpoints of the subintervals are given by

$$x_i = a + i\Delta x, \quad i = 0, 1, \ldots, N.$$

Similarly, we take time steps of Δt, at times

$$t_j = j\Delta t, \quad j = 0, 1, 2, \ldots.$$

This gives a grid of points (x_i, t_j) in the domain.

At each grid point in the domain we seek an approximate solution to the heat equation, $u_{i,j} \approx u(x_i, t_j)$. Equation (3.5) becomes

$$u_{i,j+1} \approx u_{i,j} + \alpha \left[u_{i+1,j} - 2u_{i,j} + u_{i-1,j}\right]. \qquad (3.6)$$

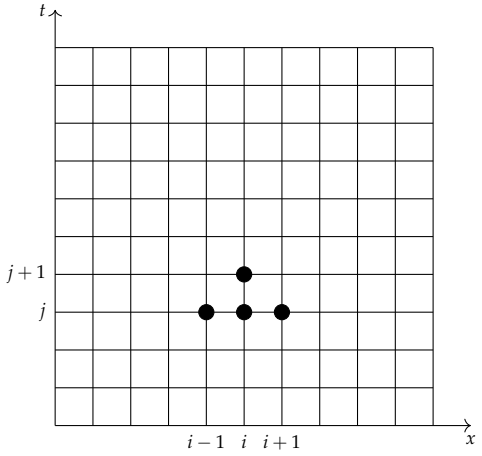

Figure 3.1: This *stencil* indicates the four types of terms in the finite difference scheme in Equation (3.6). The black circles represent the four terms in the equation, $u_{i,j}$ $u_{i-1,j}$ $u_{i+1,j}$ and $u_{i,j+1}$.

Equation (3.6) is the finite difference scheme for solving the heat equation. This equation is represented by the stencil shown in Figure 3.1. The black circles represent the four terms in the equation, $u_{i,j}$ $u_{i-1,j}$ $u_{i+1,j}$ and $u_{i,j+1}$.

Let's assume that the initial condition is given by

$$u(x, 0) = f(x).$$

Then, we have $u_{i,0} = f(x_i)$. Knowing these values, denoted by the open circles in Figure 3.2, we apply the stencil to generate the solution on the $j = 1$ row. This is shown in Figure 3.2.

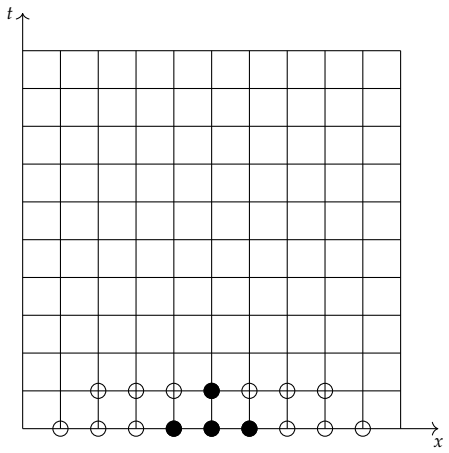

Figure 3.2: Applying the stencil to the row of initial values gives the solution at the next time step.

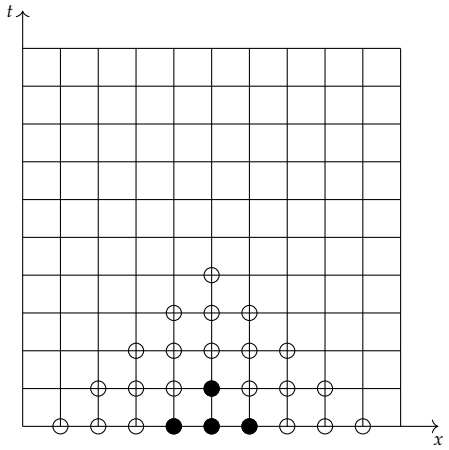

Figure 3.3: Continuation of the process provides solutions at the indicated points.

Further rows are generated by successively applying the stencil on each row, using the known approximations of $u_{i,j}$ at each level. This gives the values of the solution at the open circles shown in Figure 3.3. We notice that the solution can only be obtained at a finite number of points on the grid.

In order to obtain the missing values, we need to impose boundary conditions. For example, if we have Dirichlet conditions at $x = a$,

$$u(a,t) = 0,$$

or $u_{0,j} = 0$ for $j = 0, 1, \ldots$, then we can fill in some of the missing data points as seen in Figure 3.4.

The process continues until we again go as far as we can. This is shown in Figure 3.5.

We can fill in the rest of the grid using a boundary condition at $x = b$. For Dirichlet conditions at $x = b$,

$$u(b,t) = 0,$$

or $u_{N,j} = 0$ for $j = 0, 1, \ldots$, then we can fill in the rest of the missing data points as seen in Figure 3.6.

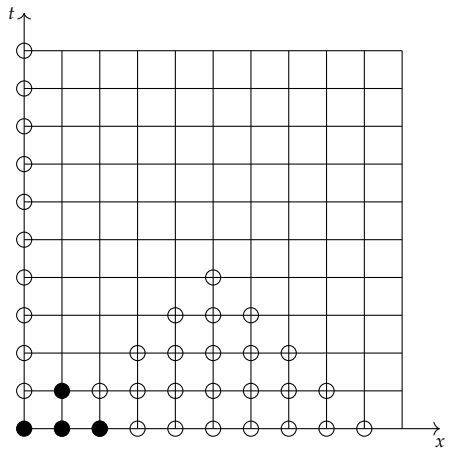

Figure 3.4: Knowing the values of the solution at $x = a$, we can fill in more of the grid.

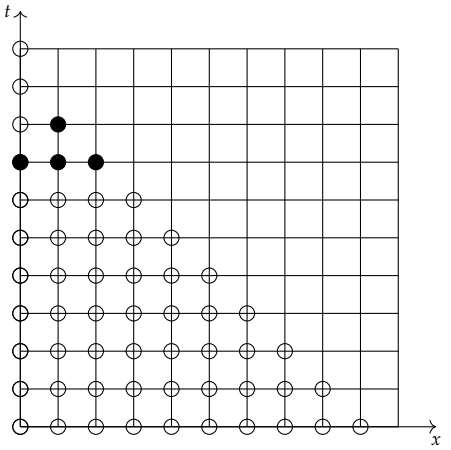

Figure 3.5: Knowing the values of the solution at other times, we continue to fill the grid as far as the stencil can go.

We could also use Neumann conditions. For example, let

$$u_x(a,t) = 0.$$

The approximation to the derivative gives

$$\frac{\partial u}{\partial x}\Big|_{x=a} \approx \frac{u(a+\Delta x,t) - u(a,t)}{\Delta x} = 0.$$

Then,

$$u(a+\Delta x,t) = u(a,t),$$

or $u_{0,j} = u_{1,j}$, for $j = 0, 1, \ldots$. Thus, we know the values at the boundary and can generate the solutions at the grid points as before.

We now have to code this using software. We can use MATLAB to do this. An example of the code is given below. In this example we specify the length of the rod, $L = 1$, and the heat constant, $k = 1$. The code is run for $t \in [0, 0.1]$.

The grid is created using $N = 10$ subintervals in space and $M = 50$ time steps. This gives dx = Δx and dt = Δt. Using these values, we find the numerical scheme constant $\alpha = k\Delta t/(\Delta x)^2$.

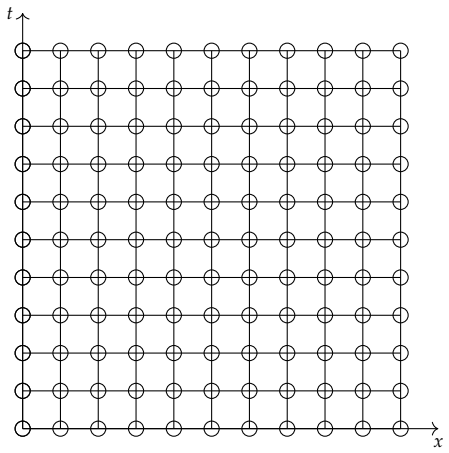

Figure 3.6: Using boundary conditions and the initial condition, the grid can be fill in through any time level.

Nest, we define $x_i = i * dx$, $i = 0, 1, \ldots, N$. However, in MATLAB, we cannot have an index of 0. We need to start with $i = 1$. Thus, $x_i = (i-1) * dx$, $i = 1, 2, \ldots, N+1$.

Next, we establish the initial condition. We take a simple condition of

$$u(x, 0) = \sin \pi x.$$

We have enough information to begin the numerical scheme as developed earlier. Namely, we cycle through the time steps using the scheme. There is one loop for each time step. We will generate the new time step from the last time step in the form

$$u_i^{new} = u_i^{old} + \alpha \left[u_{i+1}^{old} - 2u_i^{old} + u_{i-1}^{old} \right]. \tag{3.7}$$

This is done using $u0(i) = u_i^{new}$ and $u1(i) = u_i^{old}$.

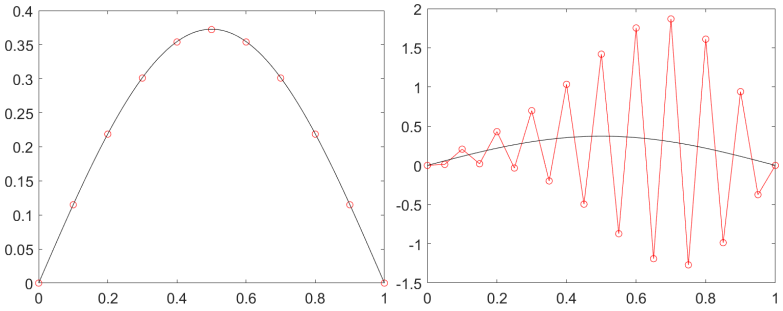

Figure 3.7: Plot of the exact solution and numerical solution of the heat equation for $\Delta t = 0.002$ for $\Delta x = 0.10$ (left) and $\Delta x = 0.20$ (right).

At the end of each time loop we update the boundary points so that the grid can be filled in as discussed. When done, we can plot the final solution. If we want to show solutions at intermediate steps, we can plot the solution earlier. In Figure 3.7 we plot the exact solution, $u(x, t) = e^{-\pi^2 t} \sin \pi x$, and numerical solution of the heat equation for $\Delta t = 0.002$ for $\Delta x = 0.10$ (left) and $\Delta x = 0.20$ (right). Even though the solution appears good for $\Delta x = 0.10$, doubling the number of points seems to have produced some strange result. This is due to the conditional stability of the scheme. In the next two sections

we discuss the truncation error and stability of this scheme. We conclude with the MATLAB implementation.

```
% Solution of the Heat Equation Using a Forward Difference Scheme
% Initialize Data
%     Length of Rod, Time Interval
%     Number of Points in Space, Number of Time Steps
L=1;
T=0.1;
k=1;
N=10;
M=50;
dx=L/N;
dt=T/M;
alpha=k*dt/dx^2;

% Position
for i=1:N+1
    x(i)=(i-1)*dx;
end

% Initial Condition
for i=1:N+1
    u0(i)=sin(pi*x(i));
end

% Partial Difference Equation (Numerical Scheme)
for j=1:M
   for i=2:N
      u1(i)=u0(i)+alpha*(u0(i+1)-2*u0(i)+u0(i-1));
   end
   u1(1)=0;
   u1(N+1)=0;
   u0=u1;
end

% Plot solution
plot(x, u1);
```

3.2 *Truncation Error*

In the previous section we found a finite difference scheme for numerically solving the one dimensional heat equation. We have from Equations (3.5) and (3.6),

$$u(x,t+\Delta t) \approx u(x,t)+\alpha\left[u(x+\Delta x,t)-2u(x,t)+u(x-\Delta x,t)\right], \quad (3.8)$$

$$u_{i,j+1} \approx u_{i,j} + \alpha \left[u_{i+1,j} - 2u_{i,j} + u_{i-1,j} \right], \tag{3.9}$$

where $\alpha = k\Delta t/(\Delta x)^2$. For points $x \in [a, b]$ and $t \geq 0$, we use the scheme to find approximate values of $u(x_i, t_j) = u_{i,j}$ at positions $x_i = a + i\Delta x, \quad i = 0, 1, \ldots, N$, and times $t_j = j\Delta t, \quad j = 0, 1, 2, \ldots$.

In implementing the scheme, we have found that there are errors introduced just like when using Euler's Method for ordinary differential equations. These truncations errors can be found by applying Taylor approximations just like we had for ordinary differential equations. In the schemes (3.8) and (3.9), we have not used equality. In order to replace the approximation by an equality, we need to estimate the order of the terms neglected in a Taylor series expansions of the time and space derivatives that we have approximated.

We begin with the time derivative approximation. We used the forward difference formula (3.1),

$$\frac{\partial u}{\partial t} \approx \frac{u(x, t + \Delta t) - u(x, t)}{\Delta t}. \tag{3.10}$$

This formula can be derived from the Taylor series expansion of $u(x, t + \Delta t)$ about $\Delta t = 0$,

$$u(x, t + \Delta t) = u(x, t) + \frac{\partial u}{\partial t}(x, t)\Delta t + \frac{1}{2!}\frac{\partial^2 u}{\partial t^2}(x, t)(\Delta t)^2 + O((\Delta t)^3).$$

Here we use "big O" notation where $O((\Delta t)^3)$ indicates that the terms not listed are of the order $(\Delta t)^3$ or smaller.

Solving for $\frac{\partial u}{\partial t}(x, t)$, we obtain

$$\frac{\partial u}{\partial t}(x, t) = \frac{u(x, t + \Delta t) - u(x, t)}{\Delta t} - \frac{1}{2!}\frac{\partial^2 u}{\partial t^2}(x, t)\Delta t + O((\Delta t)^2).$$

We see that we have obtained the forward difference approximation (3.1) with the added benefit of knowing something about the error terms introduced in the approximation. Namely, when we approximate u_t with the forward difference approximation (3.1), we are making an error of

$$E(x, t, \Delta t) = -\frac{1}{2!}\frac{\partial^2 u}{\partial t^2}(x, t)\Delta t + O((\Delta t)^2).$$

We have truncated the Taylor series to obtain this approximation and we say that

$$\frac{\partial u}{\partial t} = \frac{u(x, t + \Delta t) - u(x, t)}{\Delta t} + O(\Delta t) \tag{3.11}$$

is a first order approximation in Δt.

In a similar manor, we can obtain the truncation error for the $u_{xx}-$ term. However, instead of starting with the approximation we used in Equation (3.3), we will derive a term using the Taylor series expansion of $u(x + \Delta x, t)$ about $\Delta x = 0$. Namely, we begin with the expansion

$$\begin{aligned} u(x + \Delta x, t) &= u(x, t) + u_x(x, t)\Delta x + \frac{1}{2!}u_{xx}(x, t)(\Delta x)^2 + \frac{1}{3!}u_{xxx}(x, t)(\Delta x)^3 \\ &\quad + \frac{1}{4!}u_{xxxx}(x, t)(\Delta x)^4 + \ldots. \end{aligned} \tag{3.12}$$

We want to solve this equation for u_{xx}. However, there are some obstructions, like needing to know the u_x term. So, we seek a way to eliminate lower order terms. One way is to note that replacing Δx by $-\Delta x$ gives

$$\begin{aligned} u(x-\Delta x,t) &= u(x,t) - u_x(x,t)\Delta x + \frac{1}{2!}u_{xx}(x,t)(\Delta x)^2 - \frac{1}{3!}u_{xxx}(x,t)(\Delta x)^3 \\ &\quad + \frac{1}{4!}u_{xxxx}(x,t)(\Delta x)^4 + \ldots. \end{aligned} \tag{3.13}$$

Adding these Taylor series, we have

$$\begin{aligned} u(x+\Delta x,t) + u(x-\Delta x,t) &= 2u(x,t) + u_{xx}(x,t)(\Delta x)^2 \\ &\quad + \frac{2}{4!}u_{xxxx}(x,t)(\Delta x)^4 + O((\Delta x)^6). \end{aligned} \tag{3.14}$$

We can now solve for u_{xx} to find

$$\begin{aligned} u_{xx}(x,t) &= \frac{u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)}{(\Delta x)^2} \\ &\quad + \frac{2}{4!}u_{xxxx}(x,t)(\Delta x)^2 + O((\Delta x)^4). \end{aligned} \tag{3.15}$$

Thus, we have that

$$u_{xx}(x,t) = \frac{u(x+\Delta x,t) - 2u(x,t) + u(x+\Delta x,t)}{(\Delta x)^2} + O((\Delta x)^2)$$

is the second order in Δx approximation of u_{xx}.

Combining these results, we find that the heat equation is approximated by

$$\frac{u(x,t+\Delta t) - u(x,t)}{\Delta t} = \frac{u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)}{(\Delta x)^2} + O\left((\Delta x)^2, \Delta t\right).$$

This has local truncation error that is first order in time and second order in space.

3.3 *Stability*

Another consideration for numerical schemes for the heat equation is the stability of the scheme. In implementing the finite difference scheme,

$$u_{m,j+1} = u_{m,j} + \alpha\left[u_{m+1,j} - 2u_{m,j} + u_{m-1,j}\right], \tag{3.16}$$

$\alpha = k\Delta t/(\Delta x)^2$, one finds that the solution goes crazy when α is too big. In other words, if you try to push the individual time steps too far into the future, then something goes haywire. We saw this in Figure 3.7. Even though Δx was halved, α went from 0.20 to 0.80. We can determine the onset of instability by looking at the solution of this equation for $u_{m,j}$. [Note: We changed index i to m to avoid confusion later in this section.] We will see that there needs to be a restriction placed on α.

The scheme is actually what is called a partial difference equation for $u_{m,j}$. We could write it in terms of differences, such as $u_{m+1,j} - u_{m,j}$ and $u_{m,j+1} - u_{m,j}$. The time steps are one unit and the spatial points are at most two units apart. We can see this in the stencils in Figure 3.1. So, this is a second order partial difference equation similar to the idea that the heat equation is a second order partial differential equation. The heat equation can be solved using the method of separation of variables. The difference scheme can also be solved in a similar fashion. We will show how this leads to product solutions.

We begin by assuming that $u_{mj} = X_m T_j$, a product of functions of the indices m and j. [Recall that sequences are functions whose domain consist of a subset of integers. For example, $X_m = X(m)$.] Inserting this guess into the finite difference equation, we have

$$\begin{aligned} u_{m,j+1} &= u_{m,j} + \alpha \left[u_{m+1,j} - 2u_{m,j} + u_{m-1,j} \right], \\ X_m T_{j+1} &= X_m T_j + \alpha \left[X_{m+1} - 2X_m + X_{m-1} \right] T_j, \\ \frac{T_{j+1}}{T_j} &= \frac{\alpha X_{m+1} + (1-2\alpha) X_m + \alpha X_{m-1}}{X_m}. \end{aligned} \tag{3.17}$$

Noting that we have a function of j equal to a function of m, then we can set each of these to a constant, λ. Then, we obtain two ordinary difference equations:

$$T_{j+1} = \lambda T_j, \tag{3.18}$$

$$\alpha X_{m+1} + (1-2\alpha) X_m + \alpha X_{m-1} = \lambda X_m. \tag{3.19}$$

The first equation is a simple first order difference equation and can be solved by iteration:

$$\begin{aligned} T_{j+1} &= \lambda T_j, \\ &= \lambda(\lambda T_{j-1}) = \lambda^2 T_{j-1}, \\ &= \lambda^3 T_{j-2}, \\ &= \lambda^{j+1} T_0, \end{aligned} \tag{3.20}$$

The second difference equation can be solved by making a guess in the same spirit as solving a second order constant coefficient differential equation. Namely, let $X_m = \xi^m$ for some number ξ. This gives

$$\begin{aligned} \alpha X_{m+1} + (1-2\alpha) X_m + \alpha X_{m-1} &= \lambda X_m, \\ \xi^{m-1} \left[\alpha \xi^2 + (1-2\alpha)\xi + \alpha \right] &= \lambda \xi^m \\ \alpha \xi^2 + (1 - 2\alpha - \lambda)\xi + \alpha &= 0. \end{aligned} \tag{3.21}$$

This is an equation for ξ in terms of α and λ. Due to the boundary conditions, we expect to have oscillatory solutions. So, we guess that $\xi = |\xi| e^{i\theta}$, where i is the imaginary unit.[1] We assume that $|\xi| = 1$, and thus $\xi = e^{i\theta}$ and $X_m = \xi^m = e^{im\theta}$. Since $x_m = m\Delta x$, we have $X_m = e^{ix_m\theta/\Delta x}$. We define $\beta = \theta/\Delta x$, to give $X_m = e^{i\beta x_m}$ and $\xi = e^{i\beta\Delta x}$.

[1] Recall Euler's Formula,

$$e^{i\theta} = \cos\theta + i \sin\theta.$$

So, for real $\theta, e^{i\theta}$ represents oscillations. Also, note that $|e^{i\theta}| = 1$.

Inserting this value for ξ into the quadratic equation for ξ, we have

$$\begin{aligned} 0 &= \alpha\xi^2 + (1 - 2\alpha - \lambda)\xi + \alpha \\ &= \alpha e^{2i\beta\Delta x} + (1 - 2\alpha - \lambda)e^{i\beta\Delta x} + \alpha \\ &= e^{i\beta\Delta x}\left[\alpha(e^{i\beta\Delta x} + e^{-i\beta\Delta x}) + (1 - 2\alpha - \lambda)\right] \\ &= e^{i\beta\Delta x}\left[2\alpha\cos(\beta\Delta x) + (1 - 2\alpha - \lambda)\right]. \end{aligned}$$

Solving for λ, we have

$$\lambda = 2\alpha\cos(\beta\Delta x) + 1 - 2\alpha. \tag{3.22}$$

So, we have found that

$$u_{mj} = X_m T_j = \lambda^m(a\cos\alpha x_m + b\sin\alpha x_m), \quad a^2 + b^2 = h_0^2,$$

with λ given by Equation (3.22). For the solution to remain bounded, or stable, we need $|\lambda| \leq 1$.

Therefore, we have the inequality

$$-1 \leq 2\alpha\cos(\beta\Delta x) + 1 - 2\alpha \leq 1.$$

Since $\cos(\beta\Delta x) \leq 1$, the upper bound is obviously satisfied. Since $-1 \leq \cos(\beta\Delta x)$, the lower bound is satisfied for $-1 \leq -2\alpha + 1 - 2\alpha$, or $\alpha \leq \frac{1}{2}$. Therefore, the stability criterion is satisfied when

Stability criterion.

$$\alpha = k\frac{\Delta t}{\Delta x^2} \leq \frac{1}{2}. \tag{3.23}$$

3.4 *Discretization of Laplace Equation*

Let's consider Laplace's equation in Cartesian coordinates,

$$u_{xx} + u_{yy} = 0, \quad 0 < x < L, \quad 0 < y < H$$

with the boundary conditions

$$u(0, y) = g_1(y), \quad u(L, y) = g_2(y), \quad u(x, 0) = f_1(x), \quad u(x, H) = f_2(x).$$

The boundary conditions are shown in Figure 6.12. In Chapter 1 we learned how to seek a solution of this problem using the Method of Separation of Variables. This generally leads to needing to compute a Fourier series and represent the solution in an infinite series whose coefficients we hopefully can compute. However, it might not always be possible. So, seeking a numerical solution may be an option. In this section we explore one method for numerically solving Laplace's equation on a rectangular domain.

We use finite difference approximations to approximate the second order derivatives at grid points. For a function $f(x)$, we can use the central difference approximation,

$$f''(x) = \frac{y(x + \Delta x) - 2y(x) + y(x - \Delta x)}{\Delta x^2} + O(\Delta x^2).$$

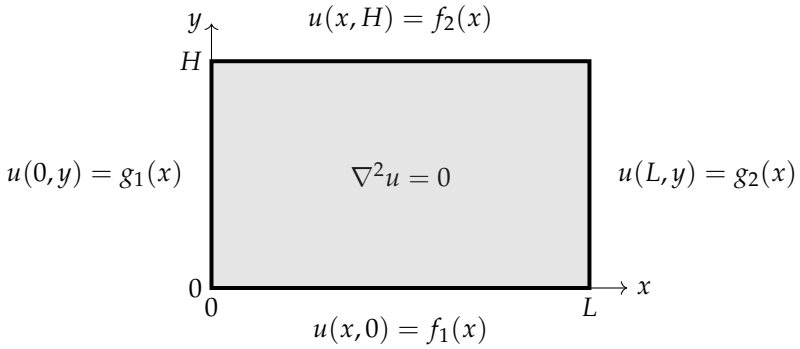

Figure 3.8: In this figure we show the domain and boundary conditions for the example of determining the solution of Laplace's equation in a rectangular region.

The domain of the solution is $x \in [0, L]$ and $y \in [0, H]$. We seek approximate values of $u(x, y)$ at specific positions in the domain. We first divide the intervals into N_x and N_y subintervals of width $\Delta x = L/N_x$. and $\Delta y = H/N_y$, respectively. This gives a grid of points (x_i, y_j) in the domain, where

$$x_i = i\Delta x, \quad i = 0, 1, \ldots, N_x, \quad y_j = j\Delta y, \quad i = 0, 1, \ldots, N_y.$$

At each grid point in the domain we seek an approximate solution to the heat equation, $u_{i,j} \approx u(x_i, y_j)$. Then, the finite difference form of Laplace's equation for the interior points ($i = 1, 2, \ldots, N_x - 1$, $j = 1, 2, \ldots, N_y - 1$) is given by

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta y^2} = 0.$$

Rearranging,

$$\begin{aligned} 0 &= \frac{\Delta y^2(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}) + \Delta x^2(u_{i,j+1} - 2u_{i,j} + u_{i,j-1})}{\Delta x^2 \Delta y^2} \\ &= \Delta y^2(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}) + \Delta x^2(u_{i,j+1} - 2u_{i,j} + u_{i,j-1}) \\ &= \Delta y^2(u_{i+1,j} + u_{i-1,j}) + \Delta x^2(u_{i,j+1} + u_{i,j-1}) - 2(\Delta x^2 + \Delta y^2)u_{i,j} \end{aligned} \tag{3.24}$$

In Figure 3.9 we show the relationships between the terms of this discrete version of Laplace's equation.

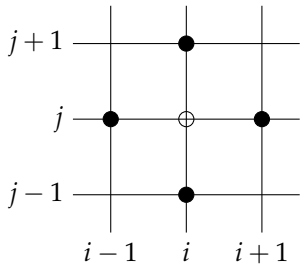

Figure 3.9: This stencil indicates the five types of terms in the finite difference scheme in Equation (3.24). The black circles represent the four terms in the equation, $u_{i-1,j}$ $u_{i+1,j}$ $u_{i,j-1}$ and $u_{i,j+1}$ and the empty circle represents $u_{i,j}$.

The method of solution differs from that of the heat equation. The only information we have is on the boundary of the domain. In Figure 3.10 we

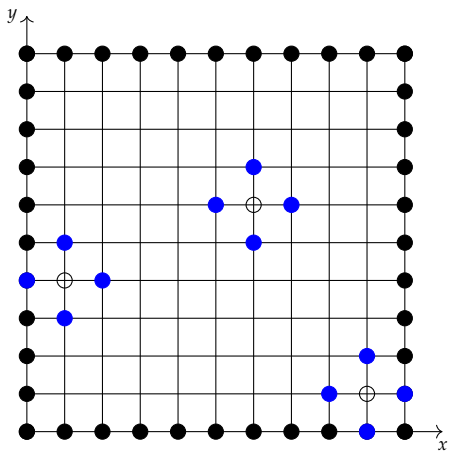

Figure 3.10: In this figure we indicate with black circles that the boundary values are known. Applying the stencil at various locations in the grid, we see that we cannot use the boundary values to approximate a value at other points.

see that we cannot use the boundary values to approximate a value at other points.

However, we can rearrange Equation (3.24) to get a more suggestive method. Namely, if we guess the solution across the grid, we can hopefully get better approximations to the solution. We assume we know approximate values at the blue circles in Figure 3.10 to compute a new values at the open circle in the center of the stencil. Thus, defining $n_x = N_x - 1$ and $n_y = N_y - 1$, we have for the interior points

$$u_{i,j}^{new} = \frac{\Delta y^2(u_{i+1,j} + u_{i-1,j}) + \Delta x^2(u_{i,j+1} + u_{i,j-1})}{2(\Delta x^2 + \Delta y^2)}, \quad i = 1, \ldots, n_x, j = 1, \ldots, n_y. \tag{3.25}$$

For now, we can simplify things a lot by letting $\Delta x = \Delta y$. Then, Equation (3.24) becomes

$$-4u_{i,j} + u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}, \quad i = 1, \ldots, n_x, j = 1, \ldots, n_y. \tag{3.26}$$

Let's consider what this means for $n_x = n_y = n$ for different n and develop a matrix form of the scheme.

3.5 *Matrix Formulation of the Scheme*

For $n = 2$, we see from Figure 3.11 that there are four interior points. Making these points the first term in the scheme (3.26), we arrive at four equations for the four unknowns, $u_{1,1}$, $u_{1,2}$, $u_{2,1}$, and $u_{2,2}$,

$$\begin{aligned} 0 &= -4u_{1,1} + u_{i2,1} + u_{0,1} + u_{1,2} + u_{1,0} \\ 0 &= -4u_{1,2} + u_{i2,2} + u_{0,2} + u_{1,3} + u_{1,1} \\ 0 &= -4u_{2,1} + u_{3,1} + u_{1,1} + u_{2,2} + u_{2,0} \\ 0 &= -4u_{2,2} + u_{3,2} + u_{1,2} + u_{2,3} + u_{2,1}. \end{aligned} \tag{3.27}$$

The highlighted u's lie on the boundary and their values are known.

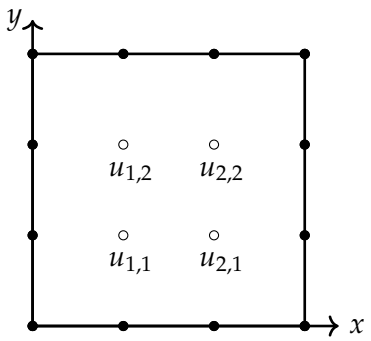

Figure 3.11: Example of a 4×4 grid with a 2×2 inner region. The goal is that given boundary values on can find approximate solutions to Laplace's Equation using Equation (3.26).

We can rearrange these linear equations into a matrix equation, $S\Phi = b$

$$\left[\begin{array}{cc:cc} -4 & 1 & 1 & 0 \\ 1 & -4 & 0 & 1 \\ \hdashline 1 & 0 & -4 & 1 \\ 0 & 1 & 1 & -4 \end{array}\right] \left[\begin{array}{c} u_{1,1} \\ u_{1,2} \\ u_{2,1} \\ u_{2,2} \end{array}\right] = - \left[\begin{array}{c} u_{0,1} \\ u_{0,2} \\ u_{3,1} \\ u_{3,2} \end{array}\right] - \left[\begin{array}{c} u_{1,0} \\ u_{2,0} \\ u_{1,3} \\ u_{2,3} \end{array}\right]$$

We have moved the boundary values to the right hand side and arranged them into two vectors. The first vector contains boundary conditions on the sides of the region and the second vector contains values of the solution along the bottom and top of the region.

The 4×4 matrix shows some symmetries. We can write the S matrix using a tensor product, also known as a direct produce or Kronecker product. The tensor product between an $m \times n$ matrix A and a $p \times q$ matrix B gives a $np \times mq$ matrix $C = A \otimes B$ with elements

$$c_{rs} = a_{ij} b_{k\ell}, \quad r = p(i-1) + k, \quad s = q(j-1) + \ell. \tag{3.28}$$

For example, if A is a 2×2 matrix, then

$$A \otimes B = \left[\begin{array}{cc} a_{11}B & a_{12}B \\ a_{21}B & a_{22}B \end{array}\right]$$

In particular, consider the matrices

$$M = \left[\begin{array}{cc} -2 & 1 \\ 1 & -2 \end{array}\right] \quad \text{and} \quad I = \left[\begin{array}{cc} 1 & 0 \\ 0 & 1 \end{array}\right].$$

We can compute two tensor products,

$$\begin{aligned} I \otimes M &= \left[\begin{array}{cc} 1 & 0 \\ 0 & 1 \end{array}\right] \otimes \left[\begin{array}{cc} -2 & 1 \\ 1 & -2 \end{array}\right] \\ &= \left[\begin{array}{cccc} -2 & 1 & 0 & 0 \\ 1 & -2 & 0 & 0 \\ 0 & 0 & -2 & 1 \\ 0 & 0 & 1 & -2 \end{array}\right] \\ M \otimes I &= \left[\begin{array}{cc} -2 & 1 \\ 1 & -2 \end{array}\right] \otimes \left[\begin{array}{cc} 1 & 0 \\ 0 & 1 \end{array}\right] \end{aligned}$$

$$= \begin{bmatrix} -2 & 0 & 1 & 0 \\ 0 & -2 & 0 & 1 \\ 1 & 0 & -2 & 0 \\ 0 & 1 & 0 & -2 \end{bmatrix}. \tag{3.29}$$

Adding these, we have the S matrix for the $n = 2$ case:

$$\begin{bmatrix} -2 & 1 & 0 & 0 \\ 1 & -2 & 0 & 0 \\ 0 & 0 & -2 & 1 \\ 0 & 0 & 1 & -2 \end{bmatrix} + \begin{bmatrix} -2 & 0 & 1 & 0 \\ 0 & -2 & 0 & 1 \\ 1 & 0 & -2 & 0 \\ 0 & 1 & 0 & -2 \end{bmatrix} = \begin{bmatrix} -4 & 1 & 1 & 0 \\ 1 & -4 & 0 & 1 \\ 1 & 0 & -4 & 1 \\ 0 & 1 & 1 & -4 \end{bmatrix}.$$

More compactly, we have shown that

$$S = M \otimes I + I \otimes M.$$

Similarly, for $n = 3$ we obtain

$$\left[\begin{array}{ccc:ccc:ccc} -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 0 & 0 & 1 & 0 & 0 & 0 \\ \hdashline 1 & 0 & 0 & -4 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & -4 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & -4 & 0 & 0 & 1 \\ \hdashline 0 & 0 & 0 & 1 & 0 & 0 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 \end{array}\right] \begin{bmatrix} u_{1,1} \\ u_{1,2} \\ u_{1,3} \\ u_{2,1} \\ u_{2,2} \\ u_{2,3} \\ u_{3,1} \\ u_{3,2} \\ u_{3,3} \end{bmatrix} = - \begin{bmatrix} u_{0,1} \\ u_{0,2} \\ u_{0,3} \\ 0 \\ 0 \\ 0 \\ u_{4,1} \\ u_{4,2} \\ u_{4,3} \end{bmatrix} - \begin{bmatrix} u_{1,0} \\ 0 \\ u_{1,3} \\ u_{2,0} \\ 0 \\ u_{2,3} \\ u_{3,0} \\ 0 \\ u_{3,3} \end{bmatrix} \tag{3.30}$$

We see that these matrix equations take the form

$$S\Phi = b, \tag{3.31}$$

where S is a block tridiagonal matrix and b involves the values of the solution on the boundary. Again we have separated out the vertical and horizontal values as two column vectors for clarity. We have also defined the vector of unknowns

$$\Phi = \begin{bmatrix} u_{1,1} \\ u_{2,1} \\ u_{3,1} \\ u_{1,2} \\ u_{2,2} \\ u_{3,2} \\ u_{1,3} \\ u_{2,3} \\ u_{3,3} \end{bmatrix}.$$

It is a simple matter to generalize the process when $\Delta x = \Delta y$ and $n_x = n_y = n$. In the case for $n = 4$, we have the 4×4 blocks

$$M = \begin{bmatrix} -4 & 1 & 0 & 0 \\ 1 & -4 & 1 & 0 \\ 0 & 1 & -4 & 1 \\ 0 & 0 & 1 & -4 \end{bmatrix} \quad \text{and} \quad I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

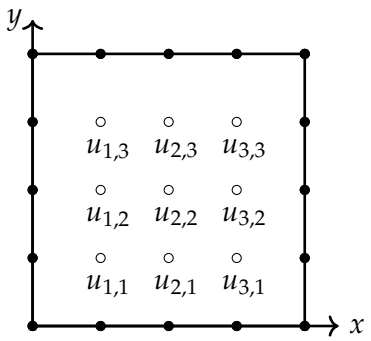

Figure 3.12: Example of a 5×5 grid with a 3×3 inner region. The numerical scheme for solving Laplace's equation is in Equation (3.30)

The S matrix is then,

$$S = \begin{bmatrix} M & I & 0 & 0 \\ I & M & I & 0 \\ 0 & I & M & I \\ 0 & 0 & I & M \end{bmatrix},$$

or written out A is given as

$$\left[\begin{array}{cccc|cccc|cccc|cccc}
-4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & & & & & & & & \\
1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & & & & & & & & \\
0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & & & & & & & & \\
0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 & & & & & & & & \\
\hline
1 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & & & & \\
0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & & & & \\
0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & & & & \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 & & & & \\
\hline
 & & & & 1 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
 & & & & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 \\
 & & & & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 \\
 & & & & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 1 \\
\hline
 & & & & & & & & 1 & 0 & 0 & 0 & -4 & 1 & 0 & 0 \\
 & & & & & & & & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 \\
 & & & & & & & & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 \\
 & & & & & & & & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4
\end{array}\right]$$

Again, we can use the tensor product of the 4×4 identity matrix with

$$M = \begin{bmatrix} -2 & 1 & 0 & 0 \\ 1 & -2 & 1 & 0 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & 1 & -2 \end{bmatrix}$$

to find that $S = M \otimes I + I \otimes M$.

3.6 *Numerical Solution of the 2D Laplace Equation*

The goal is to solve Equation (3.31). Since it is a linear system, there are several ways to approach the solution. We can solve it directly,

$$\Phi = S^{-1} b,$$

using Gaussian elimination. We could also use iterative methods as we alluded to earlier before Equation (3.25). It is best to do a simple example.

Example:

We consider the solution of Laplace's equation on the unit square,

$$u_{xx} + u_{yy} = 0, \quad 0 < x < 1, \quad 0 < y < 1,$$

with the boundary conditions

$$u(0,y) = 0, \quad u(1,y) = 1, \quad u(x,0) = \sin \pi x, \quad u(x,1) = 0.$$

We learned how to solve Laplace's equation in the first chapter using separation of variables. In this problem we have to solve two separate problems to accommodate the boundary conditions.

$$\begin{aligned} w_{xx} + w_{yy} &= 0, \quad w(0,y) = w(1,y) = w(x,1) = 0, w(x,0) = \sin \pi x, \\ v_{xx} + v_{yy} &= 0, \quad v(0,y) = v(1,y) = v(x,1) = 0, v(1,y) = 1. \end{aligned} \tag{3.32}$$

Then, $u(x,y) = w(x,y) + v(x,y)$ will solve the original problem.

The product solutions for these two problems are

$$\begin{aligned} w_n(x,y) &= \sin n\pi x \sinh n\pi(1-y), \\ v_n(x,y) &= \sin n\pi y \sinh n\pi x. \end{aligned} \tag{3.33}$$

This gives the general solution to the full problem as[2]

$$u(x,y) = \sum_{n=1}^{\infty} a_n \sin n\pi x \sinh n\pi(1-y) + \sum_{n=1}^{\infty} b_n \sin n\pi y \sinh n\pi x.$$

You can verify that $u(0,y) = 0$ and $u(x,1) = 0$. We still need to satisfy $u(1,y) = 1$ and $u(x,0) = \sin \pi x$. For the latter condition, we see that $a_n = 0$ for $n > 1$. This leaves for $n = 1$

$$u(x,0) = a_1 \sin \pi x \sinh \pi = \sin \pi x,$$

or $a_1 = (\sinh \pi)^{-1}$.

For the other condition, we have

$$u(1,y) = \sum_{n=1}^{\infty} (b_n \sinh n\pi) \sin n\pi y = 1.$$

This is a Fourier sine series for $y \in [0,1]$. The expansion coefficient for this sine series is $b_n \sinh n\pi$. We learned from the previous chapter that the coefficients are found by integration giving

$$\begin{aligned} b_n \sinh n\pi &= 2 \int_0^1 \sin n\pi y \, dy \\ &= \frac{2}{n\pi}(1 - \cos n\pi). \end{aligned} \tag{3.34}$$

So, we have the exact solution

$$u(x,y) = \sin \pi x \frac{\sinh \pi(1-y)}{\sinh \pi} + \sum_{n=1}^{\infty} \frac{2}{n\pi}(1 - \cos n\pi) \sin n\pi y \frac{\sinh n\pi x}{\sinh n\pi}. \tag{3.35}$$

[2] This problem could have been applied more generally to a rectangle $[0,L] \times [0,H]$ and not a unit square. In that case, the solution and boundary conditions would need to be modified to give a rescaled model. The solution subject to the boundary conditions Here $u(x,0) = \sin \frac{\pi x}{L}$ and $u(L,y) = 1$, takes the form

$$u(x,y) = \sin \frac{\pi x}{L} \frac{\sinh \frac{\pi}{L}(H-y)}{\sinh \frac{\pi H}{L}} + \sum_{n=1}^{\infty} \frac{2}{n\pi}(1 - \cos n\pi) \sin \frac{n\pi y}{H} \frac{\sinh \frac{n\pi x}{H}}{\sinh \frac{n\pi L}{H}}.$$

We can now numerically integrate this problem. We do this in MATLAB. We begin by setting up the problem as seen in the code below. Should be able to provide Python code.

```
clear
global L H dx dy Nx Ny  x y

% Inside dimensions - so far nx=ny, dx=dy
nx=3;
ny=3;
% Outside dimensions
Nx=nx+1;
Ny=ny+1;
% Domain size
L=1;
H=0.75;
% Increments
dx=L/Nx;
dy=H/Ny;
% Independent variables
x=0:dx:L;
y=0:dy:H;
% Initialize u(x,y) plus Boundary Conditions
U=zeros(Nx+1,Ny+1);
U(1,:)=0;                % left
U(Nx+1,:)=1;             % right
U(:,1)=sin(pi*x/L);      % bottom
U(:,Ny+1)=0;             % top
\end{verbatim}
%\end{quote}

Next, we obtain matrices $A$ and $b.$

%\begin{quote}
\begin{verbatim}
% A matrix
k=nx;               % Need to adjust for nonsquare grid
A=getA(k);

% Inside boundary values
bvert=zeros(1,k^2);
bhor=zeros(1,k^2);

for i=1:k
    bvert(i) = U(1,i+1);
    bvert(k^2-i+1) = U(k+2, k-i+2);
    bhor((i-1)*k+1) = U(i+1, 1);
```

```
    bhor((i-1)*k+k) = U(i+1, k+2);
end

b=-bhor-bvert;
```

Here we used a function to generate matrix A, called `getA`. It is given by

```
function A=getA(k)
e = ones(k,1);
a = spdiags([e -2*e e] , [-1 0 1], k, k);
I = speye (k, k);
A = kron(a,I) + kron(I,a);
end
```

This code relies on the Kronecker product (tensor product) in Equation (3.28). In MATLAB is is implemented using the `kron` function.

Now, we are in a position to solve Equation (3.31). One could compute A^{-1}, if it exists. But, in MATLAB one can use the backslash to solve a linear system. This is equivalent to using Gaussian elimination when A is a square matrix. Also, since b in the above code is a row vector, we first turn it into a column vector, `b'`. Then, `Phi=A\b'` produces a column vector. In order to related the solution to the inner grid points, we reshape `Phi` into a $k \times k$ matrix. %beginquote

```
Phi=A\b';
Phi=reshape(Phi,k,k);
U(2:k+1,2:k+1)=Phi';
Ugaussian=U';
myplot(Ugaussian,'Gaussian Solution')
\end\begin{lstlisting}
%\end{quote}

Finally, we plot the solution using {\tt myplot} which provides a
     plot of the solution normalized so that the maximum value is
     set to one. The resulting plot is in Figure \ref{fig:
    gaussian}.
%\begin{quote}

 \begin{lstlisting}[style=Matlab-editor]
function myplot(U,mytitle)
global L H dx dy Nx Ny x y
maxU=max(U,[],'all');
nU=U/maxU;
contourf(x,y,nU,200,'linecolor','none')
colormap(jet)
colorbar
caxis([-1,1])
xlabel('$x$','Interpreter','latex','Fontsize',14)
ylabel('$y$','Interpreter','latex','Fontsize',14)
```

```
title(mytitle,'Interpreter','latex','Fontsize',14)
axis equal
end
```

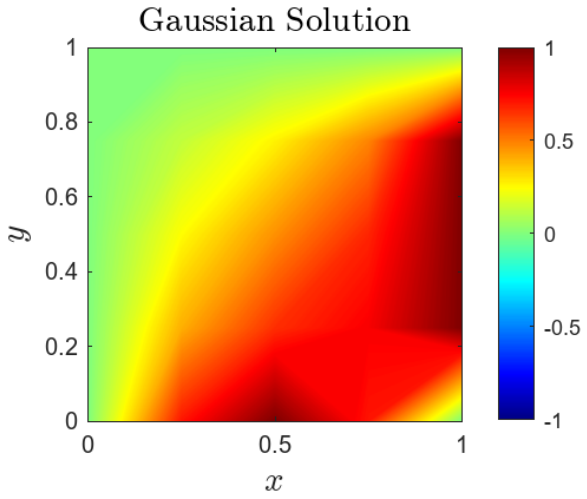

Figure 3.13: Solution of the Laplace equation example for $n = 3$ using Gaussian elimination through MATLAB's backslash operation.

We can compare the solution in Figure 3.13 with the exact solution in Equation (3.34). That solution involves an infinite sum. So, we plot a partial sum keeping enough terms to indicate an approximation to the full sum. One such function to produce the exact solution is below.

```
function v=exact(N)
global L H dx dy Nx Ny x y
Mx=length(x);
My=length(y);
v=zeros(Mx,My);
for i=1:Mx
    for j=1:My
        v(i,j)=sin(pi*x(i)/L).*sinh(pi*(H-y(j))/L)/sinh(pi*H/L);
        for n=1:N
            v(i,j)=v(i,j)...
            +2*(1-cos(n*pi))/(n*pi)*sin(n*pi*y(j)/H)...
            *sinh(n*pi*x(i)/H)/sinh(n*pi*L/H);
        end
    end
end
end
```

In Figure 3.14 we see the exact solution using $N = 100$ terms of the Fourier sine series. We see similarities with the plot in Figure 3.13. In order to compare solutions, we should compute some type of quantitative difference between the two results.

Another method for solving the finite difference scheme (3.26) is to use an iterative method. We rewrite the system as[3]

[3] From Equation (3.24), we can also write

$$u_{i,j} = \frac{\Delta y^2(u_{i+1,j} + u_{i-1,j}) + \Delta x^2(u_{i,j+1} + u_{i,j-1})}{2(\Delta x^2 + \Delta y^2)}$$

This would work with a more general grid.

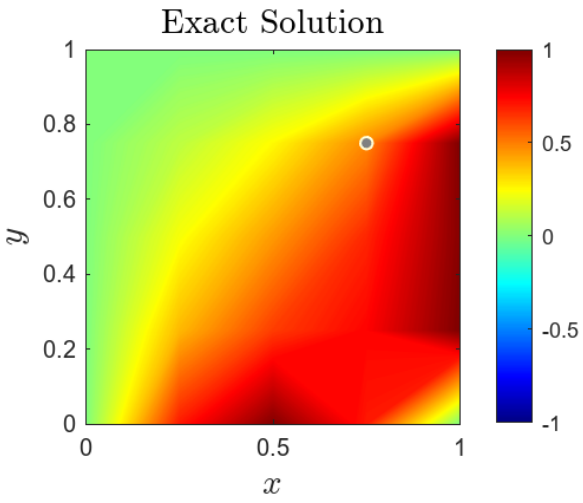

Figure 3.14: Solution of the Laplace equation example for $n = 3$ using the exact solution in Equation (3.34).

$$u_{i,j} = \frac{1}{4}\left(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}\right), \quad i = 1, \ldots, n_x, j = 1, \ldots, n_y. \tag{3.36}$$

The idea is to guess $u_{i,j}$, insert the guess on the right hand side of the equation and output a better guess. Then, take the new approximation and keep going until the procedure seems to converge to a solution. If we let the kth iteration produce $u_{i,j}^{(k)}$, then the scheme could be written as

$$u_{i,j}^{(k+1)} = \frac{1}{4}\left(u_{i+1,j}^{(k)} + u_{i-1,j}^{(k)} + u_{i,j+1}^{(k)} + u_{i,j-1}^{(k)}\right), \quad i = 1, \ldots, n_x, j = 1, \ldots, n_y, \tag{3.37}$$

where we begin with a guess $u_{i,j}^{(0)}$. The only thing we currently know are the boundary values. So, we can use those to generate a better approximation. For the inside points we can set the values to some approximation based on the characteristics of the problem or simply set all values to zero.

We will call this process a Jacobi iterative method. The MATLAB code that captures this with an input of the boundary values and number of iterations is shown in the function `Jacobi` and implemented by calling

```
Ujacobi=Jacobi(U(1,:), U(Nx+1,:),U(:,1),U(:,Ny+1),10);
```

```
function W=Jacobi(bcl, bcr,bcb,bct,N)
global L H dx dy Nx Ny x y

W=zeros(Nx+1,Ny+1);
den=2*(dx^2+dy^2);

W(1,:)=bcl;          % left
W(Ny+1,:)=bcr;       % right
W(:,1)=bcb;          % bottom
W(:,Nx+1)=bct;       % top

for k=1:N;
for i=2:Nx
```

```
    for j=2:Ny
        W(i,j)=(dy^2*(W(i+1,j)+W(i-1,j)) ...
            +dx^2*(W(i,j+1)+W(i,j-1)))/den;
    end
end
end
end
```

For the $n = 3$ grid and boundary conditions, we iterate the scheme ten times. In Figure 3.15 we show the results. Again, they look similar to those in Figures 3.14 and 3.13.

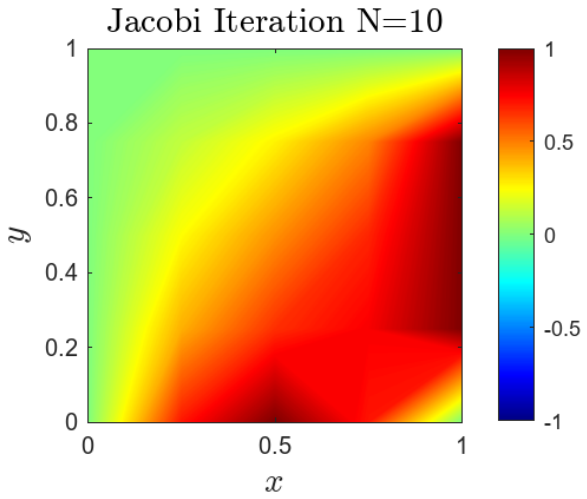

Figure 3.15: Solution of the Laplace equation example for $n = 3$ using the Jacobi iteration scheme for $N = 10$ iterations.

If we wish to provide more accurate solutions, then we need a finer grid. So, we let $n = 50$ and do not change anything else. We show in Figure 3.16 these using the exact solution in Equation (3.34), the Gaussian solution, and the Jacobi iteration for $N = 10, 50$. The Gaussian solution has the same features as the exact solution. However, we see that the Jacobi scheme has not yet converged.

So, we continue iterating the Jacobi scheme. In Figure 3.17 several plots as the iteration proceeds. We see that for $N = 1000$ iterations, the solution is looking more like the exact solution.

3.7 *Jacobi, Gauss-Seidel, and SOR Iterative Schemes*

The Jacobi iterative scheme is just one possible iterative scheme. It seems to converge slowly. Will it always converge? Can we speed up the convergence? Are there better schemes? These are just some of the questions a numerical analyst might ask. But first, we need to recast the Jacobi scheme in matrix form.

We begin with the numerical scheme in the form $A\Phi = b$, where b contains the boundary values and A is the full $n_x \times n_y$ matrix. The equivalent of solving the system for u_{ij} would be to move the diagonal elements to

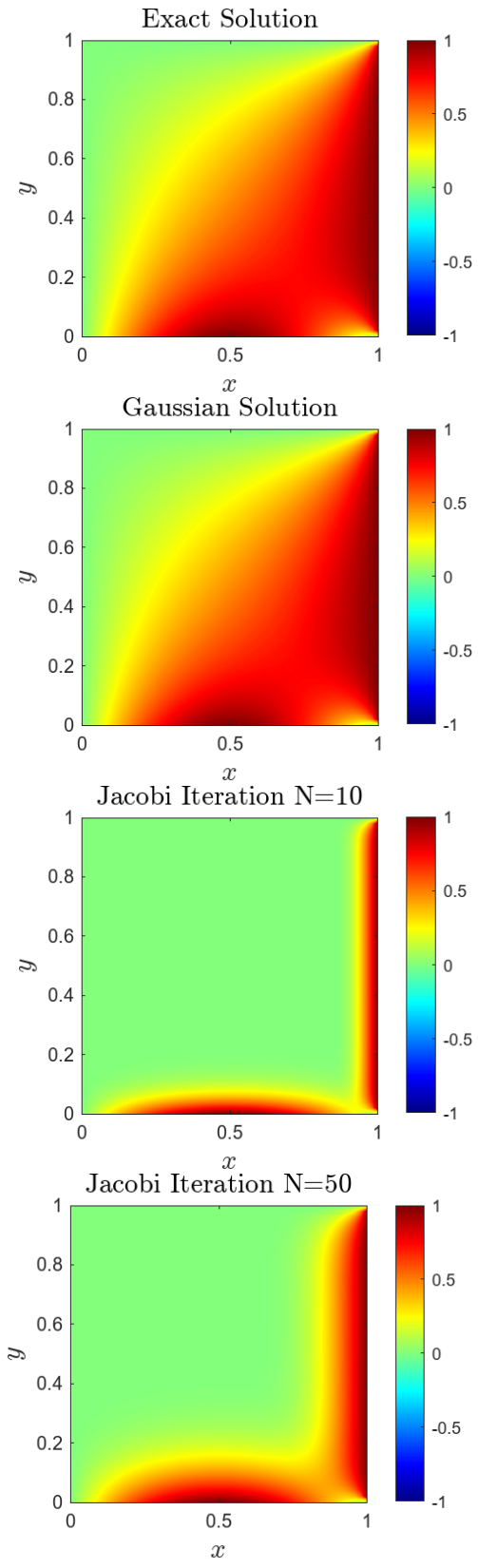

Figure 3.16: Solution of the Laplace equation example for $n = 50$ using the exact solution in Equation (3.34), the Gaussian solution, and the Jacobi iteration for $N = 10, 50$. The Gaussian solution has the same features as the exact siolution. However, we see that 10 or 50 iterations of the Jacobi scheme do not appear correct.

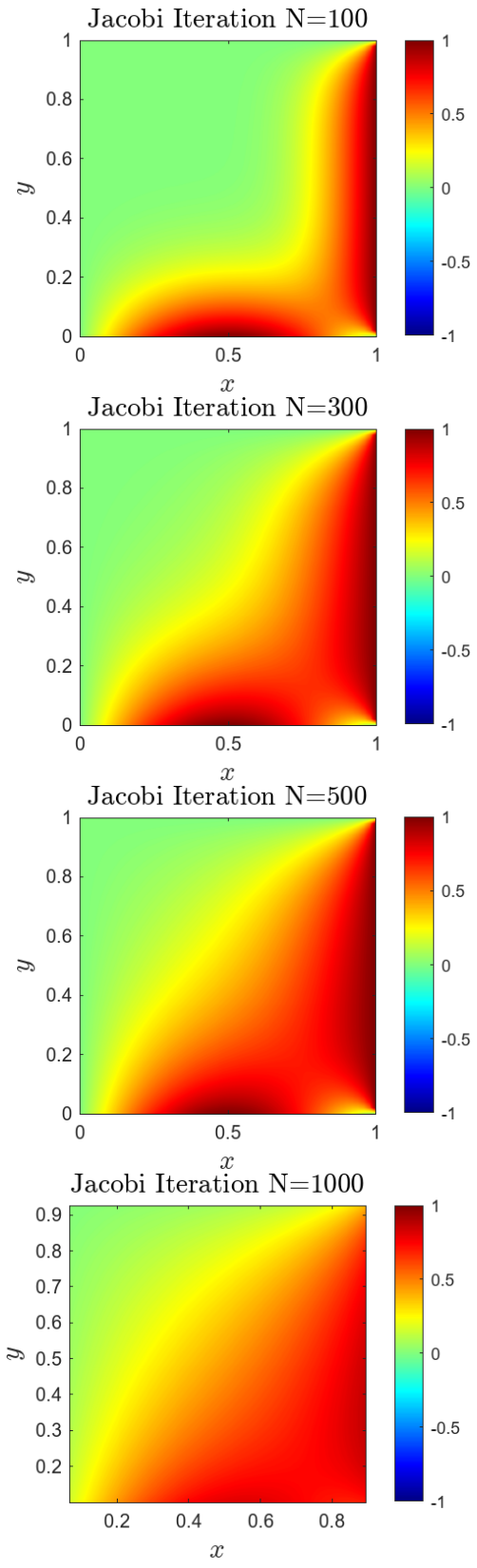

Figure 3.17: Several iterations of the Jacobi scheme for the solution of the 2D Laplace equation.

diagonal matrix D. Removing the diagonal will split the elements of A into those above the diagonal, part of an upper triangular matrix U, and those below the diagonal, part of an upper triangular matrix L. Thus, we can write $A = L + D + U$. Now, we can rewrite $A\Phi = b$ as

$$\begin{aligned}(L+D+U)\Phi &= b \\ D\Phi &= b-(L+U)\Phi \\ \Phi &= D^{-1}\left[b-(L+U)\Phi\right]. \end{aligned} \tag{3.38}$$

The iterative process then follows as

$$\Phi^{(k+1)} = D^{-1}\left[b-(L+U)\Phi^{(k)}\right], \quad k = 0,1,2,\ldots.$$

This is the Jacobi matrix iteration scheme. Convergence is based on fixed point theorems and we will not go into this here.

To implement this in MATLAB we first obtain the decomposition of matrix A. This is done with the following code:

```
% Obtain upper/lower/diagonal parts of A
[c,r] = meshgrid(1:size(A,1),1:size(A,2));
diagA =A;
diagA(c ~= r) = 0;
trilA = A-diagA;
trilA(c>r) = 0;
triuA = A-diagA;
triuA(c<r) = 0;
triuA;
```

Now, we can set up the scheme assuming we have already defined the boundary values as seen previously.

```
W=zeros(Nx+1,Ny+1);
W(1,:)=U(1,:);          % left
W(Ny+1,:)=U(Ny+1,:);    % right
W(:,1)=U(:,1);          % bottom
W(:,Nx+1)=U(:,Nx+1);    % top

% Jacobi
T = -diagA^(-1)*(trilA+triuA);
P=zeros(k*k,1);
M=1000;
for i=1:M
    P=T*P+diagA^(-1)*b';
end
W(2:k+1,2:k+1)=reshape(P,k,k)';
myplot(W',['Jacobi Matrix Iteration  N=',num2str(M)])
```

There are other ways to rearrange the matrices before performing the iteration. Consider the following:

$$(L+D+U)\Phi = b$$

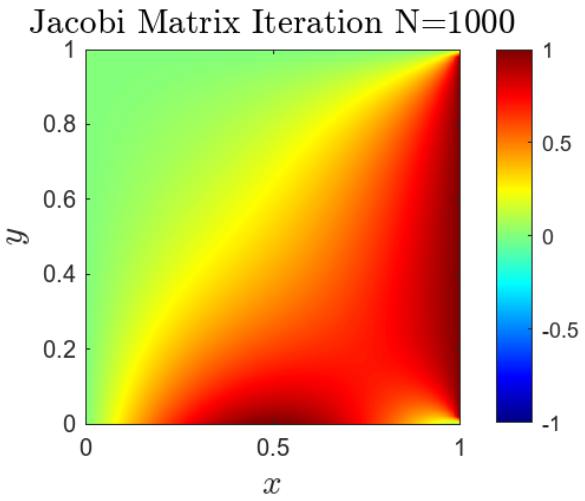

Figure 3.18: Solution of the Laplace equation example for $n = 50$ using the Jacobi Matrix iteration scheme for $N = 1000$ iterations.

$$\begin{aligned}(L+D)\Phi &= b - U\Phi \\ \Phi &= (L+D)^{-1}(b - U\Phi). \end{aligned} \tag{3.39}$$

The iterative process that results is called the Gauss-Seidel iterative scheme and is given by

$$\Phi^{(k+1)} = (L+D)^{-1}(b - U\Phi^{(k)}), \quad k = 0, 1, 2, \ldots.$$

We implement this in MATLAB:

```
% Gauss-Seidel
P=zeros(k*k,1);
M=500;
for i=1:M
    P=(trilA+diagA)^(-1)*(b'-triuA*P);
end
W(2:k+1,2:k+1)=reshape(P,k,k)';
myplot(W',['Gauss-Seidel Iteration  N=',num2str(M)])
```

We get the plot in Figure 3.19.

Finally, we move around the Gauss-Seidel terms.

$$\begin{aligned}\Phi^{(k+1)} &= (L+D)^{-1}(b - U\Phi^{(k)}) \\ (L+D)\Phi^{(k+1)} &= b - U\Phi^{(k)} \\ D\Phi^{(k+1)} &= -L\Phi^{(k+1)} + b - U\Phi^{(k)} \\ \Phi^{(k+1)} &= D^{-1}\left[b - L\Phi^{(k+1)} + b - U\Phi^{(k)}\right]. \end{aligned} \tag{3.40}$$

Now, subtract $\Phi^{(k)}$ to obtain the change in the approximation at each step of the iteration process.

$$\begin{aligned}\Phi^{(k+1)} - \Phi^{(k)} &= D^{-1}\left[b - L\Phi^{(k+1)} + b - U\Phi^{(k)}\right] - \Phi^{(k)} \\ \Phi^{(k+1)} - \Phi^{(k)} &= D^{-1}\left[b - L\Phi^{(k+1)} - U\Phi^{(k)} - D\Phi^{(k)}\right]. \end{aligned} \tag{3.41}$$

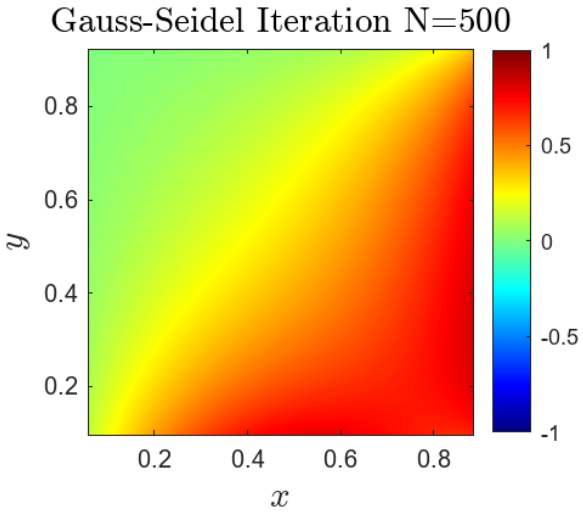

Figure 3.19: Solution of the Laplace equation example for $n = 50$ using the Gauss-Seidel iteration scheme for $N = 10$ iterations.

Instead of adding this correction to the previous step to get $\Phi^{(k+1)}$,

$$\Phi^{(k+1)} = \Phi^{(k)} + \left(\Phi^{(k+1)} - \Phi^{(k)}\right),$$

we can add a multiple of it,

$$\Phi^{(k+1)} = \Phi^{(k)} + \omega\left(\Phi^{(k+1)} - \Phi^{(k)}\right),$$

where if $0 < \omega < 1$ it is called under-relaxation and if $1 < \omega < 2$, it is called over-relaxation. In the later case, we develop a scheme called successive over-relaxation, or SOR.It is given in terms of matrices as

$$\Phi^{(k+1)} = \Phi^{(k)} + \omega D^{-1}\left[b - L\Phi^{(k+1)} - U\Phi^{(k)} - D\Phi^{(k)}\right]. \tag{3.42}$$

An equivalent approach is through a manipulation of the matrix equations. From $A\Phi = b$, we have

$$D\Phi = b - (L + U)\Phi$$

and for some constant ω we have $\omega A\Phi = \omega b$. Then, we write

$$\omega L\Phi = \omega b - \omega(D + U)\Phi.$$

Combining these two expressions, we obtain

$$\begin{aligned}(D + \omega L)\Phi &= \omega b + D\Phi - \omega(D + U)\Phi \\ &= \omega b - [\omega U + (\omega - 1)D]\,\Phi\end{aligned}$$

$$\Phi = (D + \omega L)^{-1}\omega b - (D + \omega L)^{-1}\left[\omega U + (\omega - 1)D\right]\Phi. \tag{3.43}$$

Writing $T = -(D + \omega L)^{-1}\left[\omega U + (\omega - 1)D\right]$ and $c = (D + \omega L)^{-1}\omega b$, we can write the scheme as

$$\Phi^{(k+1)} = T\Phi^{(k)} + c.$$

The implementation is shown in the code and the solution plot is seen in Figure 3.20.

```
% SOR
omega=1.25;
c = (diagA+omega*trilA)^(-1)*b';
T = -(diagA+omega*trilA)^(-1)*(omega*triuA+(omega-1)*diagA);
P=zeros(k*k,1);
M=1000;
for i=1:M
    P=T*P+c;
end
W(2:k+1,2:k+1)=reshape(P,k,k)';
myplot(W',['SOR  ','$\omega =$',num2str(omega),' N=',num2str(M)])
```

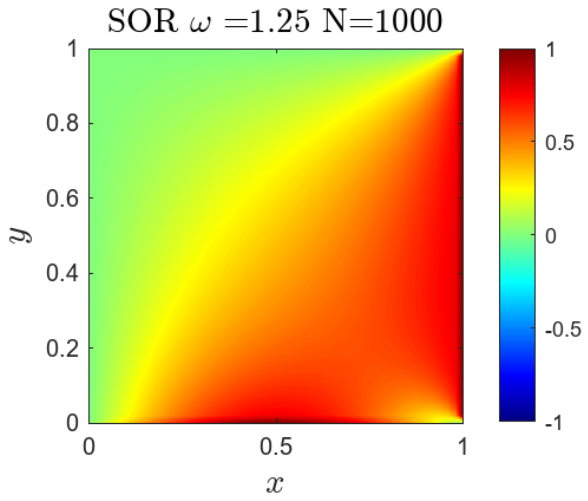

Figure 3.20: Solution of the Laplace equation example for $n = 3$ using the SOR iteration scheme for $N = 10$ iterations.

3.8 Heat Equation Project

Each group will be assigned a specific set of initial and boundary conditions and solve the two-dimensional heat equation both analytically and numerically. Let $u = u(x, y, t)$ satisfy the following:

$$\frac{\partial u}{\partial t} = k\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \qquad 0 < x < L, \quad 0 < y < W, t > 0 \tag{3.44}$$

$$u(x, y, 0) = f(x, y) \qquad 0 < x < L, \quad 0 < y < W. \tag{3.45}$$

3.8.1 1D Heat Equation

For the first part of the project you will numerically solve the one-dimensional heat equation. Below is a copy of the MATLAB code you will be given to carry out this part of the project.

```
% Solution of the Heat Equation Using a Forward Difference Scheme

% Initialize Data
```

Group	L	W	k	$x=0$	$x=L$	$y=0$	$y=W$	$f(x,y)$
A	1	1	$\frac{1}{10}$	$u=0$	$u=0$	$u=0$	$u=0$	$xy(1-x)(1-y)$
B	2	1	$\frac{1}{5}$	$u=0$	$u=0$	$u=0$	$u_y=0$	$xy(2-x)\left(1-\frac{y}{2}\right)$
C	1	2	$\frac{1}{10}$	$u_x=0$	$u=0$	$u=0$	$u=0$	$y(1-x^2)(2-y)$
D	1	2	$\frac{1}{5}$	$u_x=0$	$u_x=0$	$u=0$	$u=0$	$x^2y^2\left(1-\frac{2}{3}x\right)(2-y)$
E	1	1	$\frac{1}{10}$	$u_x=0$	$u=0$	$u_y=0$	$u=0$	$(1-x^2)(1-y^2)$
F	1	1	$\frac{1}{5}$	$u_x=0$	$u=0$	$u=0$	$u_y=0$	$y(1-x^2)\left(1-\frac{y}{2}\right)$
G	1	1	$\frac{1}{5}$	$u_x=0$	$u_x=0$	$u=0$	$u_y=0$	$x^2y\left(1-\frac{2}{3}x\right)\left(1-\frac{y}{2}\right)$
H	2	2	$\frac{1}{15}$	$u=0$	$u_x=0$	$u=0$	$u=0$	$xy(4-x)(2-y)$
I	2	1	$\frac{1}{15}$	$u=0$	$u=0$	$u_y=0$	$u=0$	$x(2-x)(1-y^2)$

Table 3.1: These are the parameters and conditions needed for your assigned group: length, L, width, W, heat constant, k, boundary conditions, and initial condition.

```
%     Length of Rod, Time Interval
%     Number of Points in Space, Number of Time Steps
clear
L=1;
T=0.1;
k=1;
N=50;
M=500;
dx=L/N;
dt=T/M;
alpha=k*dt/dx^2;

t0 = cputime;  % Combine with t1 to time the routine

% Position
for i=1:N+1
    x(i)=(i-1)*dx;
end

% Initial Condition
for i=1:N+1
    u0(i)=x(i)*(1-x(i));
end

% Partial Difference Equation (Numerical Scheme)
for j=1:M
   for i=2:N
      u1(i)=u0(i)+alpha*(u0(i+1)-2*u0(i)+u0(i-1));
   end
```

```
    u1(1)=0;
    u1(N+1)=0;
    u0=u1;
    % Plot solution
    hold on
    if mod(j,10)==0
        plot(x, u1);
    end
    hold off
end

t1=cputime;
telapsed = t1-t0
```

3.8.2 2D Heat Equation

For the second part of the project you will solve the two-dimensional heat equation by constructing the solution to the initial-boundary value problem you are assigned from Table 1. You will find the product solutions $\phi_{n,m}(x,y,t)$ and the Fourier coefficients, $c_{n,m}$. Be careful as problems D and G also have coefficients $c_{0,m}$.

Below is a copy of MATLAB code for plotting the exact solution. Besides generating a 3D plot of the solution evolving in time, frames are captured and placed in a movie file. The movie can be played using one of the commands at the end of the file.

```
% Solution of the 2D Heat Equation Using the series solution.
% u_t = k (u_xx + u_yy)

% Initialize Data
%      L, W   = Length and Width of Playe,
%      T      = for Time Interval [0, T]
%      Nx, Ny = Number of Points in Space Grid,
%      M      = Number of Time Steps
%      dx, dy = Delta x and Delta y.
% Set your own values of L, W, T, k
clear
L=1;
W=2;
T=1;
k=1/10;
Nx=20;
Ny=20;
dx=L/Nx;
dy=W/Ny;
M=20;
dt = T/M;
```

```
% Spatial grid
[x,y]=meshgrid(0:dx:L,0:dy:W);

% Initialize u % Change to your initial condition
u=zeros(Nx+1,Ny+1);
for n=1:10
    for m=1:10
        u=u+sin(n*pi*x/L).*sin(m*pi*y/W)/n^2/m^2;
    end
end
H=max(max(u));

% Plot initial condition
surf(x,y,u,'FaceColor','red','EdgeColor','none')
camlight left;
lighting phong
xlabel('x')
ylabel('y')
title('Solution at t = 0')
axis([0,L,0,W,0,H])
frame=1;
Mov(frame)=getframe(gcf);
pause(0.5)

% Time evolution
for j=1:M
    u=zeros(Nx+1,Ny+1);
    t=j*dt;
    for n=1:10
        for m=1:10
            lambda=(n*pi/L)^2+(m*pi/W)^2;   % Change lambda
            u=u+sin(n*pi*x/L).*sin(m*pi*y/W)/n^2/m^2*exp(-k*
                lambda*t);
        end
    end

   % Plot 3D solution every 10 time steps
   %if mod(m,10)==0
        frame=frame+1;
        surf(x,y,u,'FaceColor','red','EdgeColor','none')
        camlight left;
        lighting phong
        xlabel('x')
        ylabel('y')
        title(['Solution at t = ' num2str(t)])
```

```
        axis([0,L,0,W,0,H])
        Mov(frame)=getframe(gcf);
        pause (0.5)
    %end
end

% Extra - show or create a movie
%    movie(gcf, Mov)         % plays movie
%    movie(gcf, Mov,1,2)     % plays at 2 fps
%    movie(gcf, Mov,10,5)    % repeats 10 times at 5 fps
% Create movie
%    movie2avi(Mov, 'heat2d.avi', 'compression', 'None');
```

One can also make use of the Symbolic Toolbox in MATLAB. The following defines the product solutions and computes the Fourier coefficients.

```
clear
% Problem Test
syms n x y
n = sym('n','integer');
m = sym('m','integer');

L = 1;
W = 1;
Kx = n*pi/L;
Ky = m*pi/W;

f = sin(pi*x/L)*sin(3*pi*y/W);
phi = sin(Kx*x)*sin(Ky*y);

% Fourier Coefficients
c=simplify(4/L/W*int(int(f*phi, x, [0 L]), y, [0 W]) );
\end{minted}

Now one can plot the initial condition
\begin{minted}{matlab}
% Initial Condition
N=5;
M=5;
H=.07;
ff=symsum(symsum(c*phi,n,1,N),m,1,M);
h=fsurf(ff,[0,L,0,W],'FaceColor','red','EdgeColor','none');
    camlight left;
    lighting phong
    xlabel('x')
    ylabel('y')
    title(['Solution at t = ' num2str(0)])
```

```
    axis([0,L,0,W,-1,1])
%frame=1;
%Mov(frame)=getframe(gcf);
\end{minted}

The solution can then be evolved in time.
\begin{minted}{matlab}
k=1/200;    % Heat constant

% Time Evolution
T =   1;   % Final time
Nt = 20;   % Number of steps

dt=T/Nt;
lambda = Kx^2+Ky^2;

for j=1:20
    t=j*dt;
%   frame=frame+1;
    ff=symsum(symsum(c*phi*exp(-k*lambda*t),n,1,N),m,1,M);
    %fsurf(ff,[0,L,0,W],'FaceColor','red','EdgeColor','none')
    h.Function=ff; % Alternative to using fsurf by updating the
        function plotted

    title(['Solution at t = ' num2str(t)])

%    Mov(frame)=getframe(gcf);
%    pause(.05)
end
```

Problems

1. Use the forward and backward difference formulae to find approximate values of $f'(x)$ given the following data.

a.

x	$f(x)$
1.1	0.4990
1.2	0.4348
1.3	0.3477
1.4	0.2380
1.5	0.1061

b.

x	$f(x)$
2.1	-1.7098
2.2	-1.3738
2.3	-1.1192
2.4	-0.9160
2.5	-0.747

c. The functions used to generate the tables we $f(x) = x\cos x$ and $\tan x$, respectively. What were the errors made in parts a and b?

2. Use the central difference approximation to find $f''(x)$ given the data in Problem 1. What error is made using this approximation?

3. Use $f(x_0)$, $f(x_0 \pm h)$, and $f(x_0 \pm 2h)$, for $h = \Delta x$, to find the most accurate approximation for $f'(x_0)$. What is the truncation error?

4. What does the finite difference expression $\dfrac{2u_{n,j} - 5u_{n-1,j} + 4u_{n-2,j} - u_{n-3,j}}{h^2}$ approximate?

5. Consider using a finite difference approximation for the wave equation, $u_{tt} = c^2 u_{xx}$ on $x \in [0, L]$.

a. Use centered differences in space and time to derive a finite difference model of the wave equation.

b. Draw the appropriate stencil for this difference equation and describe what conditions would be needed to proceed with numerically solving the wave equation.

c. Use the initial conditions $u(x,0) = f(x)$ and $u_t(x,0) = g(x)$ and boundary conditions $u(0,t) = u(L,t) = 0$ to fill in the missing details of the solution in part b.

d. What is the order of the truncation error for this scheme?