9. PDE 1

When you open the toolkit of differential equations you see the hammers and saws of engineering and physics for the past two centuries and for the foreseeable future.
–Benoit Mandelbrot

9.1. Intro to PDEs

Partial differential equations (PDEs) are differential equations involving the partial derivatives of an unknown multivariable function. In most of this chapter and the next we will examine two classical problems from physics: heat transport phenomena and wave phenomena. Do not think, however, that just because we are focusing on these two primary examples that this is the extent of the utility of PDEs. Basically, every scientific field has been impacted by (or has directly impacted) the study of PDEs. Any phenomenon that can be modelled via the change in multiple continuous variables (not restricted to space and time) is likely governed by a PDE model.

There is a wealth of wonderful theory for finding analytic solutions to many special classes of PDEs. However, most PDEs simply do not lend themselves to analytic solutions that we can write down in terms of the regular mathematical operations of sums, products, powers, roots, trigonometric functions, logarithms, etc. For these PDEs we must turn to numerical methods to approximate the solution.

Recall that numerical solutions to ODEs were approximations of the value of the unknown function at a discrete set of times. Similarly, numerical solutions to PDEs are going to be approximations of the value of the unknown function at a discrete set of points in time AND space.

What we will cover in this chapter will include one primary and powerful technique for approximating solutions to PDEs: **the finite difference method**. There are many other techniques for approximating solutions to PDEs, and the field of numerical PDEs is still an active area of mathematical and scientific research.

9.2. The Heat Equation

You have probably met the heat equation, also known as the diffusion equation, in a previous module. The heat equation is a partial differential equation that describes how heat diffuses through a material. The heat equation is a parabolic PDE and is given by

$$\frac{\partial u}{\partial t} = D\nabla^2 u$$

where $u(t, x)$ is the temperature of the material at time t and position x and D is the diffusion coefficient. The heat equation is a simple model for heat diffusion but also describes diffusion in general, like the diffusion of a solute in a solvent or of plants in a field or, ... well, you get the idea.

You of course know that the heat equation is easy to solve analytically, given that it is a linear homogeneous PDE with constant coefficients. However, the finite difference method is a powerful tool for solving similar PDEs that do not have simple analytic solutions. The advantage of using the heat equation as a test case for the finite difference method is that we can easily verify the accuracy of our numerical solutions by comparing them to the known analytic solutions.

9.2.1. In One Spatial Dimension

For the sake of simplicity we will start by considering the heat equation in 1 spatial dimension:

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2}.$$

We will also use the alternative notation

$$u_t = Du_{xx}, \tag{9.1}$$

where the subscripts denote partial derivatives.

Exercise 9.1. Just as we did in Chapter 7 to approximate solutions to ODEs, we will start by partitioning the spatial domain into finitely many pieces and we will partition time into finitely many pieces. We do this by introducing a grid of points (t_n, x_i) where $t_n = t_0 + n\,\Delta t$ and $x_i = x_0 + i\,\Delta x$. Then we want to build a numerical approximation to the function $u(t, x)$ at these grid points.

First we need to introduce some notation for the numerical solution. As you will see in a moment, there is a lot to keep track of in numerical PDEs so careful indexing and well-chosen notation is essential. Let U_i^n be the approximation of the solution to $u(t, x)$ at the point $t = t_n = t_0 + n\,\Delta t$ and $x = x_i = x_0 + i\,\Delta x$ (since we have two variables we need two indices). For example, U_4^1 is the value of the approximation at time t_1 and at the spatial point x_4.

Next we need to approximate both derivatives u_t and u_{xx} in the PDE using methods that we have used before. Now would be a good time to go back to Section 5.2 and refresh your memory for how we build approximations of derivatives.

(a) Use the forward-difference formula to approximate the time derivative u_t at the point $t = t_n$ and $x = x_i$.

$$u_t(t_n, x_i) \approx \frac{??? - ???}{???}.$$

(b) Use the centred-difference formula to approximate the second spatial derivative u_{xx} at the point $t = t_n$ and $x = x_i$.

$$u_{xx}(t_n, x_i) \approx \frac{??? - ??? + ???}{???}.$$

(c) Put your answers from parts (a) and (b) together using the 1D heat equation (Eq. 9.1)

$$\frac{??? - ???}{\Delta t} = D\left(\frac{??? - ??? + ???}{\Delta x^2}\right).$$

Be sure that your indexing is correct: the superscript n is the index for time and the subscript i is the index for space.

(d) Rearrange your result from part (c) to solve for U_i^{n+1}:

$$U_i^{n+1} = ??? + \frac{D\Delta t}{\Delta x^2}\left(??? - ??? + ???\right).$$

The iterative scheme which you just derived is called the **forward difference scheme** for the heat equation. Notice that the term on the left is the only term at the next time step $n + 1$. So, for every spatial point x_i we can build U_i^{n+1} by evaluating the right-hand side of the finite difference scheme.

(e) The numerical errors made by using the forward difference scheme we just built come from two sources: from the approximation of the time derivative and from the approximation of the second spatial derivative. Fill in the question marks in the powers of the following expression:

$$\text{Numerical Error} = \mathcal{O}(\Delta t^{???}) + \mathcal{O}(\Delta x^{???}).$$

(f) Explain what the result from part (e) means in plain English.

There are many different finite difference schemes due to the fact that there are many different ways to approximate derivatives (See Section 5.2). One convenient way to keep track of which information you are using and what you are calculating in a finite difference scheme is to use a **finite difference stencil image**. Figure 9.1 shows the finite difference stencil for the approximation to the heat equation that you built in the previous exercise. In this figure we are showing that the function values U_{i-1}^n, U_i^n, and U_{i+1}^n at the points x_{i-1}, x_i, and x_{i+1} at time step t_n are used to calculate U_i^{n+1}. We will build similar stencil diagrams for other finite difference schemes throughout this chapter.

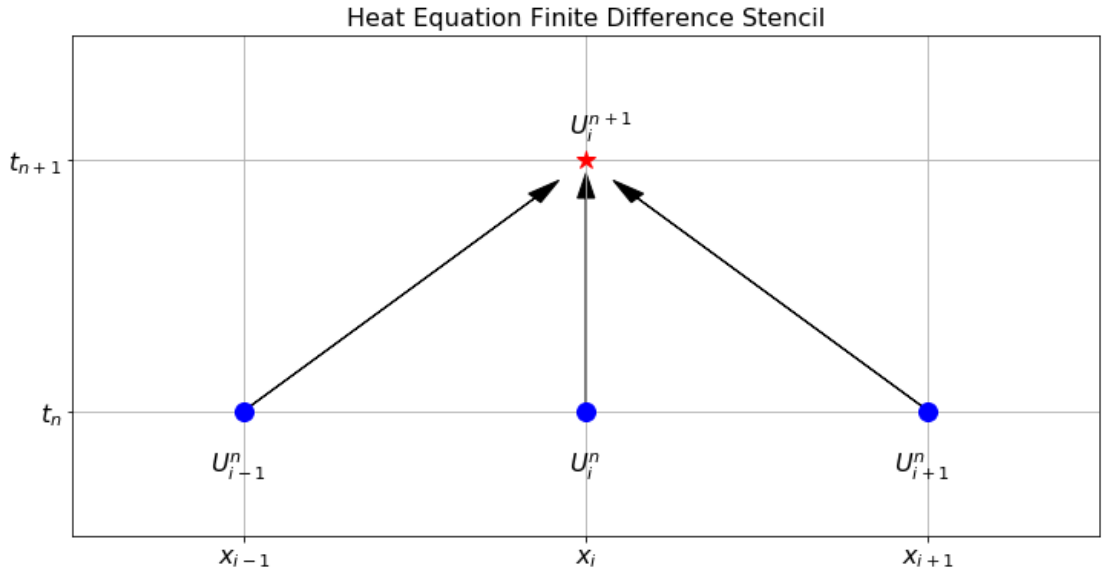

Figure 9.1.: The forward difference stencil for the 1D heat equation.

Exercise 9.2. Now we want to implement your answer to part (d) of the previous exercise to approximate the solution to the following problem: Solve

$$u_t = 0.1u_{xx}$$

on the domain $0 < x < 1$ and $0 < t < 1$ with the initial condition

$$u(0, x) = \sin(2\pi x)$$

and boundary conditions

$$u(t, 0) = 0, \text{ and } u(t, 1) = 0.$$

For this purpose divide the x domain into 20 equal pieces and the t domain into 100 equal pieces.

Some partial code is given below to get you started.

- First we import the proper libraries, set up the time domain, and set up the spatial domain.

```
import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interactive

# Write code to give a vector of times starting at t=0 and ending
# at t=1 that divides the interval into 100 equal pieces.

# Calculate the step size `dt`.

# Write code to give a vector of x values starting at x=0 and
# ending exactly at x=1 that divides the interval into 20 equal pieces.

# Calculate the step size `dx`.

# Specify the diffusion coefficient
D = 0.1
# The coefficient "a" appears in the forward difference scheme.
a = D*dt / dx**2

print("dt=", dt, ", dx=", dx, " and D dt/dx^2=", a)
```

- Next we build the array U so we can store all of the approximations at all times and at all spatial points. The array will have the dimensions `len(t)` by `len(x)`. We then need to enforce the boundary conditions so for all times we fill the proper portions of the array with the proper boundary conditions. Lastly, we will build the initial condition for all spatial steps in the first time step.

```
U = np.zeros((len(t),len(x)))
U[:,0] = # left boundary condition
U[:,-1] = # right boundary condition
U[0,:] = # the function for the init. condition (should depend on x)
```

- Now we step through a loop that fills the U array one row at a time. Keep in mind that we want to leave the boundary conditions fixed so we will only fill indices `1` through `-2` (stop and explain this). Be careful to get the indexing correct. For example, if we want U_i^n we use `U[n,1:-1]`, if we want U_{i+1}^n we use `U[n,2:]`, if we want U_i^{n+1} we use `U[n+1,1:-1]`, etc.

```
for n in range(len(t)-1):
    U[n+1,1:-1] = U[n,?:?] + a*( U[n,?:] - 2*U[n,?:?] + U[n,:?])
```

- We can now make a plot of the solution at the final time step to see what the solution looks like at $t = 1$.

```
plt.plot(x, U[?,?])
plt.xlabel("x")
plt.ylabel("u")
plt.grid()
```

- Let us also calculate an approximate value for $u(0.2, 0.25)$ by finding the indices in the `t` and `x` vectors that correspond to $t = 0.2$ and $x = 0.25$ and then looking up the value in the U array at those indices.

```
U[?,?]
```

Exercise 9.3. Now visualise the whole solutions over the entire time range. You can either make a plot or an animation to illustrate the time evolution of u. For each of these there a various Python packages you could use. Below is a function `plot_solution_1d()` using plotly to make a plot and a function `animate_solution_1d()` using matplotlib to make an animated 2D plot. You can use either of these or you can use your own plotting code.

```
import plotly.graph_objects as go

def plot_solution_1d(t, x, U):
    """Plots the numerical approximation to a function u(t,x).

    Args:
        t: A vector of time values.
        x: A vector of spatial values.
        U: A 2D array approximating the solution u(t,x) at each grid point.
    """
    fig = go.Figure(data=[go.Surface(z=U, x=x, y=t)])
    fig.update_layout(
        width=800, height=600,
        scene=dict(
            xaxis_title='x',
            yaxis_title='t',
            zaxis_title='u'
        )
    )
    return fig
```

```
import matplotlib.pyplot as plt
from matplotlib import animation, rc
from IPython.display import HTML

def animate_solution_1d(t, x, U):
    """Animates the numerical approximation to a function u(t,x).

    Args:
        t: A vector of time values.
        x: A vector of spatial values.
        U: A 2D array approximating the solution u(t,x) at each grid point.
    """
    fig, ax = plt.subplots()
    plt.close()
    ax.grid()
    ax.set_xlabel("x")
    ax.set_ylabel("u")
    ax.set_xlim((np.min(x), np.max(x)))
    ax.set_ylim((np.min(U), np.max(U)))
    frame, = ax.plot([], [], linewidth=2)

    # Don't display every time
    step = int(len(t)/30)+1
    frames = range(0, int(len(t)/step), 1)

    def animator(i):
        n = i*step
        ax.set_title(f"t = {t[n]:.2f}")
        frame.set_data(x, U[n,:])
        return (frame, )

    ani = animation.FuncAnimation(fig, animator, frames=frames, interval=100)
    rc('animation', html='jshtml') # embed in the HTML for Google Colab
    return ani
```

Exercise 9.4. Now wrap up your code for solving the one-dimensional heat equation as a function so that you can easily call it with different parameters.

```
def heat1d(u_0, D=0.1, t_0=0, t_max=1, N_t=100, x_left=0, x_right=1, N_x=20):
    """Solves the 1D heat equation using the forward difference method.
```

```
    This function solves the 1D heat equation with given initial and
    boundary conditions. It also prints a diagnostic message stating
    the step sizes `dt` and `dx` used and the value of `a = D*dt/dx**2`.

    Args:
        u_0: A function giving the initial condition u(0,x).
        D: The diffusion coefficient. Defaults to 0.1.
        t_0: The initial time. Defaults to 0.
        t_max: The maximum time. Defaults to 1.
        N_t: The number of time steps. Defaults to 100.
        x_left: The left boundary of the spatial domain. Defaults to 0.
        x_right: The right boundary of the spatial domain. Defaults to 1.
        N_x: The number of spatial steps. Defaults to 20.

    Returns:
        A tuple containing the following:
            t: A vector of time values.
            x: A vector of spatial values.
            U: A 2D array approximating the solution u(t,x) at each grid point.
    """
    # Your code goes here
```

Check that your function works by using it to calculate the same solution as in Exercise 9.2 and plotting the solution with `plot_solution_1d()` or `animate_solution_1d()`.

Exercise 9.5. Now run your `heat1d` function from Exercise 9.4 with the same diffusion coefficient $D = 0.1$, the same number `N_t = 100` of time steps and the same initial and boundary conditions but double the number of spatial steps to `N_x=40`. Plot the solution on the domain $t \in [0, 1]$ and $x \in [0, 1]$. Do you believe what you see? What is happening to the solution?

Exercise 9.6. You will have found that you did not get a sensible solution from your method for the previous problem. The point of this exercise is to show that value of $a = D\frac{\Delta t}{\Delta x^2}$ controls the stability of the forward difference solution to the heat equation, and furthermore that there is a threshold for a above which the forward difference scheme will be unstable. Experiment with values of Δt and Δx and conjecture the values of $a = D\frac{\Delta t}{\Delta x^2}$ that give a stable result. Your conjecture should take the form:

If $a = D\frac{\Delta t}{\Delta x^2} <$ _____ then the forward difference solution for the 1D heat equation is stable. Otherwise it is unstable.

Hint: the threshold is a simple fraction. If you think you have found a value for a at which the method is stable, run the simulation for longer (by increasing both `t_max` and `N_t` to keep the same Δt) to check that it is really stable. Close to the threshold the errors grow more slowly (albeit still exponentially).

Exercise 9.7. Consider the one dimensional heat equation with diffusion coefficient $D = 1$:

$$u_t = u_{xx}.$$

We want to solve this equation on the domain $x \in [0, 1]$ and $t \in [0, 0.1]$ subject to the initial condition $u(0, x) = \sin(\pi x)$ and the boundary conditions $u(t, 0) = u(t, 1) = 0$.

(a) Show that the function $u(t, x) = e^{-\pi^2 t}\sin(\pi x)$ is a solution to this heat equation, satisfies the initial condition, and satisfies the boundary conditions.

(b) Pick values of Δt and Δx so that you can get a stable forward difference solution to this heat equation. Then make a plot of your numerical solution.

(c) Compare your plot to the plot of the exact solution that you can get with

```
X, T = np.meshgrid(x, t)
u_exact = np.exp(-np.pi**2*T)*np.sin(np.pi*X)
plot_solution_1d(t, x, u_exact)
```

Exercise 9.8. Now let us change the initial condition to $u(0, x) = \sin(\pi x) + \sin(3\pi x)$. We will keep the same boundary conditions as before: $u(t, 0) = u(t, 1) = 0$.

(a) Show that the function $u(t, x) = e^{-\pi^2 t}\sin(\pi x) + e^{-9\pi^2 t}\sin(3\pi x)$ is a solution to this heat equation, matches this new initial condition, and matches the boundary conditions.

(b) Pick values of Δt and Δx so that you can get a stable forward difference solution to this heat equation. Make a 3d plot of your numerical solution.

(c) Compare your plot to the plot of the exact solution.

9.2.2. Different Boundary Conditions

In any initial and boundary value problem such as the heat equation, the boundary are often of Dirichlet or Neumann type. In Dirichlet boundary conditions the values of the solution at the boundary are specified. In contrast, Neumann boundary conditions specify the flux at the boundary instead of the value of the solution.

Exercise 9.9 (Time-dependent Dirichlet Boundary Condition)**.** Modify your 1D heat equation code to plot and to animate an approximate solution of the diffusion equation $u_t = 0.5u_{xx}$ with $x \in (0, 1)$, $u(0, x) = \sin(2\pi x)$, $u(t, 0) = 0$ and $u(t, 1) = \sin(5\pi t)$. See if everyone in your group gets the same plot and animation.

Exercise 9.10 (Neumann Boundary Condition)**.** Consider the 1D heat equation $u_t = u_{xx}$ with boundary conditions $u_x(t, 0) = 0$ and $u(t, 1) = 0$ with initial condition $u(0, x) = \cos(\pi x/2)$. So at $x = 0$ we are imposing a Neumann boundary condition and at $x = 1$ we are imposing a Dirichlet boundary condition. Check that the initial condition satisfies these boundary conditions. As the heat profile evolves in time the Neumann boundary condition $u_x(t, 0) = 0$ says that the slope of the solution needs to be fixed at 0 at the left-hand boundary.

(a) The Neumann boundary condition $u_x(t, 0) = 0$ can be approximated with the first order approximation

$$u_x(t_n, 0) \approx \frac{U_1^n - U_0^n}{\Delta x} \text{ for all } n.$$

If we set this approximation to 0 (since $u_x(t, 0) = 0$) and solve for U_0^n we get an additional constraint at every time step of the numerical solution to the heat equation:

$$U_0^n = ??? \text{ for all } n.$$

(b) Modify your 1D heat equation code to implement this Neumann boundary condition, plot the numerical solution and verify visually that the Neumann boundary is satisfied.

9.3. Other Parabolic PDEs

The heat equation is a parabolic PDE and the forward-difference method that we have developed can be adapted to work for other parabolic PDEs.

9.3.1. Reaction-Diffusion Equations

For example, the heat equation can be modified to include a reaction term. The reaction-diffusion equation is a PDE that models the diffusion of a substance in space and time with a reaction term that describes the rate of change of the substance due to some reaction. While it has its origin in chemistry, it shows up in many other fields as well, for example in ecology and epidemiology.

Exercise 9.11 (Fisher-KPP Equation)**.** Modify your 1D heat equation code to calculate an approximate solution of the Fisher-KPP equation

$$u_t = u_{xx} + u(1-u)$$

with $t \in [0, 10]$, $x \in (0, 50)$, boundary conditions

$$u(t, 0) = 0, \qquad u(t, 50) = 1$$

and initial condition

$$u(0, x) = \frac{1 + \tanh\left(\dfrac{x - 40}{2}\right)}{2}.$$

Use `animate_solution_1d()` to visualize the solution. How does the solution change as time evolves?

9.3.2. Advective-Diffusion Equations

The diffusion term usually arises from random spatial motion of particles. However, in some cases the particles are advected by a flow field. In this case we need to add an advection term to the diffusion equation. The advection-diffusion equation is a PDE that models the diffusion of a substance in space and time with an advection term that describes the rate of change of the substance due to some flow field.

Exercise 9.12. Modify your 1D heat equation code to plot an approximate solution of the following simple advection-diffusion equation:

$$u_t = 0.1u_{xx} - u_x$$

Use the forward difference formula for the u_x term and the centred difference formula for the u_{xx} term. Use the initial condition $u(0, x) = \sin(\pi x)$, Dirichlet boundary conditions $u(t, 0) = 0$ and $u(t, 1) = 0$, and $t \in [0, 1]$. Use 20 spatial steps and 100 time steps. Make a plot and an animation of the solution.

9.4. Stability

While exploring the explicit finite-difference method for solving the 1d heat equation $u_t = Du_{xx}$ we encountered the stability condition

$$a = \frac{D\Delta t}{(\Delta x)^2} \leq \frac{1}{2}.$$

We now want to understand where this condition comes from.

We start by setting up the notation for solving the heat equation using the explicit finite-difference method. We discretise the spatial variable x into N intervals with grid points $x_0, \dots, x_N$ with stepsize $\Delta x = (x_N - x_0)/N$. We discretise the time variable t into M intervals with grid points $t_0, \dots, t_M$ with stepsize $\Delta t = (t_M - t_0)/M$. We denote the approximation of $u(t_n, x_i)$ by U_i^n. The initial condition sets $U_i^0 = u(t_0, x_i)$. We work with homogeneous Dirichlet boundary conditions, so $U_0^n = U_N^n = 0$ for all n.

The approximations at the remaining points is then calculated by the formula

$$U_i^{n+1} = U_i^n + a(U_{i+1}^n - 2U_i^n + U_{i-1}^n)$$

for $i = 1, \dots, N-1$. You derived this in Section 9.2.1 using the finite-difference formulae for the derivatives from Section 5.2. We rewrite this in matrix notation:

$$\begin{pmatrix} U_1^{n+1} \\ \vdots \\ U_{N-1}^{n+1} \end{pmatrix} = \begin{pmatrix} 1-2a & a & & \\ a & 1-2a & a & \\ & \ddots & \ddots & \ddots \\ & & a & 1-2a \end{pmatrix} \begin{pmatrix} U_1^n \\ \vdots \\ U_{N-1}^n \end{pmatrix}.$$

The matrix is tridiagonal, with $1-2a$ on the diagonal and a on the two off-diagonals. We denote the matrix by A and the two vectors by $\mathbf{U}_{n+1}$ and $\mathbf{U}_n$ respectively. So we have the formula

$$\mathbf{U}_{n+1} = A\mathbf{U}_n.$$

The solution at time t_n is then given by

$$\mathbf{U}_n = A^n \mathbf{U}_0.$$

We want to understand how errors evolve over time. If errors grow exponentially over time then we call the method unstable and the method is not useful.

Let us assume that at some step, for convenience let us choose step 0, an error is introduced:

$$\tilde{\mathbf{U}}_0 = \mathbf{U}_0 + .$$

Then after n steps we have

$$\tilde{\mathbf{U}}_n = A^n \tilde{\mathbf{U}}_0 = A^n(\mathbf{U}_0 +) = A^n \mathbf{U}_0 + A^n = \mathbf{U}_n + A^n.$$

So the error at time t_n is

$$\epsilon_n = A^n \epsilon.$$

To see if the error grows exponentially over time, we expand the initial error in terms of the eigenvectors of the matrix A:

$$\epsilon = \sum_i \epsilon_i \mathbf{v}_i,$$

where

$$A\mathbf{v}_i = \lambda_i \mathbf{v}_i$$

and the sum is over all eigenvectors $\mathbf{v}_i$ of A. Then

$$\epsilon_n = \sum_i \epsilon_i \lambda_i^n \mathbf{v}_i.$$

This shows that if all eigenvalues have absolute value less than 1, then the method is stable. If at least one eigenvalue has absolute value greater than 1, then the corresponding component of the error will grow exponentially with time and the method is unstable.

There is a nice method to determine the eigenvalues of the matrix A, using Fourier analysis. We will not discuss this in this module and instead just give the result and then look at a simple example. The result (which you do not need to remember) is that the eigenvalues of the matrix A are given by

$$\lambda_k = 1 - 4a\left(\sin\left(\frac{k\pi}{2N}\right)\right)^2$$

for $k = 1, \ldots, N-1$. For stability we need $|\lambda_k| < 1$ for all k, i.e.,

$$0 \leq a\left(\sin\left(\frac{k\pi}{2N}\right)\right)^2 < \frac{1}{2}$$

for all k. The most stringent condition is that coming from $k = N-1$, so the stability condition is

$$a \leq \frac{1}{2}\left(\sin\left(\frac{(N-1)\pi}{2N}\right)\right)^{-2}.$$

In the limit $N \to \infty$ this gives the condition $a \leq 1/2$.

Example 9.1. Consider the heat equation on the spatial domain $x \in [0, 1]$ and divide this into three subintervals, so that our spatial grid consists of $x_0 = 0, x_1 = 1/3, x_2 = 2/3$ and $x_3 = 1$. The matrix A is then

$$A = \begin{pmatrix} 1-2a & a \\ a & 1-2a \end{pmatrix}.$$

The characteristic polynomial is

$$\begin{aligned}\det(A-\lambda I) &= \begin{vmatrix} 1-2a-\lambda & a \\ a & 1-2a-\lambda \end{vmatrix} \\ &= (1-2a-\lambda)^2 - a^2 \\ &= \lambda^2 - 2(1-2a)\lambda + (1-2a)^2 - a^2.\end{aligned}$$

The roots of this polynomial are

$$\lambda_{\pm} = 1 - 2a \pm a.$$

We need both of these to have a magnitude less than 1 for the method to be stable. This gives us an upper bound on the allowed a. The eigenvalue whose magnitude will increase above 1 first as a increases is $\lambda_- = 1 - 3a$, which equals -1 when $a = 2/3$ (meaning the error in that mode alternates sign every time step). So the stability condition is $a < 2/3$.

In this chapter we have developed the **explicit forward difference scheme** for the 1D heat equation, seen how to handle Dirichlet and Neumann boundary conditions, and explored variations such as reaction-diffusion and advection-diffusion equations. We have also seen that the scheme is only stable when $a = D\frac{\Delta t}{\Delta x^2} \leq \frac{1}{2}$, which places a severe constraint on the time step when the spatial resolution is fine. In the next chapter we will overcome this limitation by introducing **implicit schemes** such as the backward difference scheme and the Crank-Nicolson scheme, which are unconditionally stable and allow much larger time steps.

9.5. Exam-Style Question

(a) Consider the 1D heat equation $u_t = Du_{xx}$. Using a forward difference approximation for the time derivative and a centred difference approximation for the second spatial derivative, derive the explicit finite difference scheme for U_i^{n+1}. Use step sizes Δt and Δx, and let $U_i^n \approx u(t_n, x_i)$. [3 marks]

(b) What are the truncation errors associated with the forward difference approximation for u_t and the centred difference approximation for u_{xx}? State the overall order of the numerical scheme derived in part (a). [3 marks]

(c) What condition on $a = D\frac{\Delta t}{\Delta x^2}$ must be satisfied for this explicit forward difference scheme to be numerically stable? [1 mark]

(d) Suppose you have a Neumann boundary condition at the left boundary, $u_x(t, 0) = 0$. Using a first-order forward difference approximation for the spatial derivative, express U_0^n in terms of other grid values. [2 marks]

(e) The following incomplete Python code solves the 1D heat equation $u_t = Du_{xx}$ using the explicit scheme derived in part (a), subject to Dirichlet boundary conditions $u(t, 0) = 0$ and $u(t, 1) = 0$. Provide the missing code indicated by `...`. [4 marks]

```
import numpy as np

def solve_heat1d(u_0, D, tmax, dt, xmax, dx):
    """
    Solves the 1D heat equation.
    u_0 is a function giving the initial condition.
    """
    t = np.arange(0, tmax + dt, dt)
    x = np.arange(0, xmax + dx, dx)

    a = ...

    U = np.zeros((len(t), len(x)))

    # Apply initial condition
    U[0, :] = u_0(x)

    # Boundary conditions are naturally 0 from np.zeros

    for n in range(len(t) - 1):
        # Update interior spatial points
        U[n+1, 1:-1] = ...

    return t, x, U
```

(f) The explicit finite difference scheme from part (a) can be written in matrix form as $\mathbf{U}_{n+1} = A\mathbf{U}_n$, where $\mathbf{U}_n = (U_1^n, \ldots, U_{N-1}^n)^T$ is the vector of interior grid values at time step n and A is a $(N-1) \times (N-1)$ matrix.

(i) Write down the matrix A explicitly for the case $N = 4$ (i.e., three interior grid points), expressing all entries in terms of $a = D\Delta t/(\Delta x)^2$. [2 marks]

(ii) Using the matrix form, write down an expression for $\mathbf{U}_n$ in terms of A, n, and the initial vector $\mathbf{U}_0$. [1 mark]

(iii) Suppose that at time step 0 an error ϵ is introduced, so that the computed vector is $\tilde{\mathbf{U}}_0 = \mathbf{U}_0 + \epsilon$. Show that the error at time step n is $\epsilon_n = A^n\epsilon$. [2 marks]

(iv) Explain why and how the stability condition can be expressed in terms of the eigenvalues of A. [2 marks]