Numerical Methods for PDEs: An Interactive Journey

Renu Dhadwal

Table of contents

Chapter 1

Numerical Methods for PDEs: An Interactive Journey

Chapter 2

Welcome

Welcome to this interactive journey through **Numerical Methods for Partial Differential Equations (PDEs)**.

You will explore the subject through **dialogues between Acharya and Pavni**, discovering history, classification, and numerical methods step by step.

2.1 Levels

- Level 1: Origins of PDEs
- Level 2: Classification of PDEs
- Level 3: Physical Interpretation and boundary conditions
- Level 4: Finite Difference Methods- Introduction with application to the Heat Eqaution
- Level 5 : Hyperbolic PDEs
- Level 5.5: Numerics for Hyperbolic PDEs
- Level 6 : Von-Neumann Stability Analysis
- Level 7 : Elliptic PDEs

Chapter 3

Stay tuned as new levels unlock — from classification of PDEs to discretization techniques and beyond!

Chapter 4

Level 1: A Historical Glimpse of PDEs

Pavni: Acharya, where did all these partial differential equations come from? They seem so abstract.

Acharya: Ah, Pavni, they did not begin in abstraction. They began with strings, heat, and the mysteries of the heavens.

Pavni: Strings? You mean music?

Acharya: Exactly! In 1746, *d'Alembert* studied how a vibrating string produces sound. From that, he wrote down the **wave equation**. Soon after, *Euler* extended it to drums and membranes. So you see, music and mathematics are deeply connected.

Pavni: That's beautiful! And the heat equation?

Acharya: That came from *Joseph Fourier* in the early 1800s. He asked: *How does heat spread through a solid body?* His answer was the **heat equation**, and in solving it he gave us the gift of **Fourier series**.

Pavni: So Fourier series were born from studying heat?

Acharya: Precisely. And then *Laplace* studied gravitational attraction and derived the **Laplace equation**, describing potentials in physics. *Poisson* later generalized it with the **Poisson equation**, where sources appear inside the field.

Pavni: So each new equation came from a real phenomenon.

Acharya: Just so. Later, in the 19th century, *Navier* and *Stokes* wrote down the equations of fluid motion — the **Navier–Stokes equations**. Even today, their mysteries are not fully solved.

Pavni: (smiling) So PDEs are not just formulas on paper. They are echoes of sound, flows of heat, gravity, and water.

Acharya: Well said, Pavni. They are the language by which nature speaks to us.

Chapter 5

Level 2: Classification of PDEs

Pavni: Acharya, last time you told me how PDEs were born from strings, heat, and flows. But there seem to be so many kinds of PDEs. How do we organize them?

Acharya: A good question, Pavni. Mathematicians classify PDEs in several ways, much like a botanist classifies plants. Let us begin with the simplest.

Pavni: I am listening.

Acharya: First, by **order**. A PDE is first-order if the highest derivative is first order, second-order if the highest is second, and so on. For example, the transport equation $u_t + cu_x = 0$ is first-order, while the heat equation $u_t = \alpha u_{xx}$ is second-order.

Pavni: That part seems simple enough. What else?

Acharya: Next comes **linearity**. A PDE is linear if the unknown function and its derivatives appear only linearly — not multiplied together, not inside a sine or square. The heat equation is linear. But if you had a term like uu_x, that would make it nonlinear.

Pavni: So, nonlinear PDEs are trickier?

Acharya: Very much so! Nonlinearity makes life both harder and richer.

Pavni: Are there other distinctions?

Acharya: Yes. Equations can be **homogeneous** or **inhomogeneous** depending on whether the right-hand side is zero. Laplace's equation is homogeneous, Poisson's equation is inhomogeneous.

Acharya: More generally, a non-homogeneous PDE can be written as:

$$F(t, x_1, x_2, \dots, x_n, u, u_t, u_{x_1}, \dots, u_{x_n}, u_{tt}, u_{x_1 x_1}, \dots) = g(x_1, x_2, \dots, x_n, t),$$

where g is a nonzero source or forcing term. If $g \equiv 0$, the PDE is homogeneous.

Pavni: So the right-hand side introduces an external influence, like heat sources or forces?

Acharya: Exactly. For example, $u_{xx} + u_{yy} = f(x, y)$ is the Poisson equation with a source term $f(x, y)$. .

Pavni: I think I follow. But I have also heard words like elliptic and hyperbolic. What do they mean?

Acharya: Ah, those arise for **second-order PDEs in two variables**. Suppose we have:

$$Au_{xx} + Bu_{xy} + Cu_{yy} + \cdots = 0.$$

We look at the discriminant: $B^2 - 4AC$.
- If it is less than 0, the PDE is **elliptic** (like Laplace's equation).
- If it equals 0, the PDE is **parabolic** (like the heat equation).
- If it is greater than 0, the PDE is **hyperbolic** (like the wave equation).

Pavni: This reminds me of conic sections! Circles, parabolas, and hyperbolas.

Acharya: Exactly. The analogy is deliberate — both come from the same quadratic form.

Pavni: And physically?

Acharya: Elliptic equations describe steady states — like equilibrium temperature distributions. Parabolic equations describe diffusion, the smoothing of irregularities over time. Hyperbolic equations describe wave-like motion, signals traveling with finite speed.

Pavni: That helps me imagine them. So classification is not just a game, but it tells us the nature of the solutions.

Acharya: Well said, Pavni. And it also guides how we design numerical methods. An elliptic problem requires different strategies than a hyperbolic one.

Pavni: Then I am eager to learn those strategies!

Acharya: Patience, Pavni. First we must prepare the ground. Classification is the map; numerical methods are the journey.

5.1 Mini Quizzes

Quiz 1: Identify the order
Which of the following is a second-order PDE?
1. $ u_t + cu_x = 0 $
2. $ u_t = u_{xx} $
3. $ u u_x = 0 $

Answer 1

Equation (2) $ u_t = u_{xx} $ is second-order because of the u_{xx} term.

Quiz 2: Linearity check
Which PDE is nonlinear?
1. $ u_t = u_{xx} $
2. $ u_t + u u_x = 0 $

Answer 2

Equation (2) is nonlinear because of the product term uu_x.

Quiz 3: Homogeneous or inhomogeneous
Classify: $ u_{xx} + u_{yy} = f(x,y) $.

Answer 3

It is **inhomogeneous**, since the right-hand side is not zero.

Quiz 4: Type of second-order PDE
For $ u_{xx} + 2u_{xy} + u_{yy} = 0 $, compute $ B^2 - 4AC $. What type is it?

Answer 4

Here, $A = 1, B = 2, C = 1$.
$ B^2 - 4AC = 2^2 - 4(1)(1) = 0 $.
So it is **parabolic**.

Quiz 5: Physical meaning
Match each equation with its physical interpretation:
- Heat equation
- Wave equation
- Laplace's equation

(a) Steady state

(b) Wave-like motion

(c) Diffusion in time

Answer 5

- Heat equation $\rightarrow$ (c) Diffusion in time
- Wave equation $\rightarrow$ (b) Wave-like motion
- Laplace's equation $\rightarrow$ (a) Steady state

Chapter 6

Level 3: PDEs, their Physical Meaning, and Boundary Conditions

Pavni: Acharya, PDEs still feel mysterious. What do they *really* mean?

Acharya: Good question. PDEs describe how a quantity changes with respect to **both time and space**. Think of:
- Heat spreading in a rod
- Waves traveling along a string
- Fluid flowing through a pipe

All of these involve rates of change in multiple directions, and that's why PDEs come into play.

Pavni: So PDEs are the language of physics in extended domains?

Acharya: Exactly. But to make their solutions unique and physically meaningful, we need **boundary conditions**. Let's explore them one by one.

6.1 Dirichlet Condition

Acharya: Dirichlet means fixing the value of the solution at the boundary.

Pavni: Like holding both ends of a rod at 100 °C?

Acharya: Precisely. It represents physical situations where the boundary is controlled by an external source—like contact with a thermostat.

6.2 Neumann Condition

Acharya: Neumann means fixing the derivative, often representing **flux**.

Pavni: So in the rod, saying no heat flows out means the temperature gradient at the end is zero?

Acharya: Exactly. That's an insulated boundary.

6.3 Robin (Mixed) Condition

Acharya: Robin mixes the two:

$$au + b\frac{\partial u}{\partial n} = c.$$

Pavni: Is that like when heat escapes to the air?

Acharya: Yes—convective cooling. The flux depends on both the temperature at the boundary and the environment.

Quick Recap

- **Dirichlet** → Value fixed (e.g., temperature = 100 °C).
- **Neumann** → Flux fixed (e.g., insulated boundary).
- **Robin** → Combination (e.g., convective heat loss).

6.3.1 Mini-Quiz

1. A vibrating string held fixed at both ends uses which boundary condition?

   Answer
   Dirichlet. The displacement of the string is zero at both ends.
2. If a wall is perfectly insulated, what type of boundary condition applies to temperature?

   Answer
   Neumann. The derivative (temperature gradient) is zero, meaning no heat flux.
3. Which boundary condition models cooling of hot coffee in a room?

   Answer

Robin. Heat loss depends on both the coffee's surface temperature and the room temperature (convection).

Pavni: Now I see it! PDEs tell the story inside the domain, and boundary conditions set the rules at the edges.

Acharya: Well said. Together, they form the complete model of a physical system.

Chapter 7

Level 4: Finite Differences and the Heat Equation

Pavni: Acharya, how do we actually *approximate* derivatives on a computer?

Acharya: We use the **finite difference method**. A computer only works with discrete points, so we replace continuous derivatives with **difference quotients** on a **grid**. These come directly from the **Taylor series**, and because we truncate the series, each formula comes with a predictable **error term**.

Pavni: Can you give me an example?

Acharya: Suppose we have points $x_0, x_1, \dots, x_N$ with spacing Δx. Around a point x_i, Taylor's theorem gives us expansions for $u(x_{i+1})$ and $u(x_{i-1})$. By combining them, we obtain difference formulas:

- **Forward difference** for the first derivative:

$$u'(x_i) = \frac{u(x_{i+1}) - u(x_i)}{\Delta x} - \tfrac{1}{2}u''(\xi)\,\Delta x,$$

  so the truncation error is $\mathcal{O}(\Delta x)$.

- **Backward difference**:

$$u'(x_i) = \frac{u(x_i) - u(x_{i-1})}{\Delta x} + \tfrac{1}{2}u''(\xi)\,\Delta x,$$

  also error $\mathcal{O}(\Delta x)$.

- **Central difference**:

$$u'(x_i) = \frac{u(x_{i+1}) - u(x_{i-1})}{2\Delta x} - \tfrac{1}{6}u^{(3)}(\xi)\,(\Delta x)^2,$$

  so the error is $\mathcal{O}((\Delta x)^2)$ — more accurate.

- **Second derivative** (central difference):

$$u''(x_i) = \frac{u(x_{i-1}) - 2u(x_i) + u(x_{i+1})}{(\Delta x)^2} - \tfrac{1}{12}u^{(4)}(\xi)\,(\Delta x)^2,$$

  with error $\mathcal{O}((\Delta x)^2)$.

Pavni: So these formulas are not exact — they always come with an error term?

Acharya: Precisely. That error is called the **local truncation error**. When we build numerical schemes for PDEs, these errors accumulate step by step, and we'll have to balance them with stability.

Pavni: I see. So finite differences give us algebraic formulas for derivatives, but with a known accuracy.

Acharya: Exactly. That's the foundation of finite difference methods.

7.1 Application: The Heat Equation

Acharya: Consider the 1D heat equation:

$$u_t = \alpha^2 u_{xx}^2, \quad 0 < x < 1,\ t > 0$$

with boundary conditions $u(0,t) = u(1,t) = 0$ and initial profile $u(x,0) = f(x)$.

We set up a grid: - In space: $x_i = i\Delta x,\ i = 0, \dots, N$
- In time: $t^n = n\Delta t,\ n = 0, 1, 2, \dots$

At each point, let $u_i^n \approx u(x_i, t^n)$.

Pavni: And now we replace derivatives?

Acharya: Correct.
- Time derivative (forward difference):

$$u_t(x_i, t^n) \approx \frac{u_i^{n+1} - u_i^n}{\Delta t}$$

- Spatial second derivative (central difference):

$$u_{xx}(x_i, t^n) \approx \frac{u_{i-1}^n - 2u_i^n + u_{i+1}^n}{(\Delta x)^2}$$

Plugging these into the PDE, we get:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \alpha^2 \, \frac{u_{i-1}^n - 2u_i^n + u_{i+1}^n}{(\Delta x)^2}.$$

Rearranging:

$$u_i^{n+1} = u_i^n + \lambda\,(u_{i-1}^n - 2u_i^n + u_{i+1}^n),$$

where $\lambda = \frac{\alpha^2 \Delta t}{(\Delta x)^2}$.

7.2 Stability Condition

Pavni: So we can just keep applying this formula to march forward in time?

Acharya: Yes, but with a caveat. This scheme, called **FTCS (Forward Time, Central Space)**, is only stable if:

$$\lambda \leq \tfrac{1}{2}.$$

Pavni: So if Δt is too large, the scheme fails?

Acharya: Exactly. The numerical solution will blow up, even though the true solution is stable. Choosing Δt small enough ensures stability.

Why the FTCS stability condition is $\lambda \leq \frac{1}{2}$

If the update is

$$u^{(n+1)} = Au^{(n)},$$

and the initial error is e^0, then after n steps the error is

$$e^{(n)} = A^n e^0.$$

Using an induced matrix norm:

$$\|e^{(n)}\| = \|A^n e^0\| \;\leq\; \|A^n\|\,\|e^0\| \;\leq\; \|A\|^n \|e^0\|.$$

- If $\|A\| \leq 1$, the error never grows.
- If $\|A\| < 1$, the error decays as $n \to \infty$.
- In general, the asymptotic condition is that the **spectral radius** $\rho(A) < 1$.

7.2.0.1 Eigenvalues of the FTCS matrix

For the FTCS tridiagonal matrix A (symmetric Toeplitz), the eigenvalues are

$$\mu_k = 1 - 4\lambda \sin^2\left(\frac{k\pi}{2(N-1)}\right), \qquad k = 1, 2, \dots, N-2,$$

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

Thus the spectral radius is

$$\rho(A) = \max_{1\leq k\leq N-2} |\mu_k| = \max_{1\leq k\leq N-2} \left|1 - 4r\sin^2\left(\tfrac{k\pi}{2(N-1)}\right)\right|.$$

7.2.0.2 Stability condition

- As $N \to \infty$ (i.e. $\Delta x \to 0$), the maximum of $\sin^2(\cdot)$ tends to 1.
- Therefore the most restrictive case is

$$|1 - 4\lambda| \le 1.$$

- This simplifies to

$$0 \ \le \ \lambda \ \le \ \tfrac{1}{2}.$$

Thus, the **FTCS scheme is stable if and only if**

$$\lambda = \frac{\alpha \Delta t}{(\Delta x)^2} \ \le \ \tfrac{1}{2}.$$

7.3 Mini-Quizzes

Quiz 1:
Let $u(x) = x^2$. With $\Delta x = 0.1$, approximate $u'(1)$ using:
1. Forward difference
2. Central difference

Compare with the exact derivative $u'(1) = 2$. Which is more accurate?

Answer 1

- Forward difference:

$$\frac{u(1+\Delta x) - u(1)}{\Delta x} = \frac{(1.1)^2 - 1^2}{0.1} = \frac{1.21 - 1}{0.1} = 2.1.$$

- Central difference:

$$\frac{u(1+\Delta x) - u(1-\Delta x)}{2\Delta x} = \frac{(1.1)^2 - (0.9)^2}{0.2} = \frac{1.21 - 0.81}{0.2} = 2.0.$$

- Exact derivative: $u'(1) = 2$.

Conclusion: The central difference gives the exact value here (error 0), while the forward difference has error 0.1. Central is more accurate (as expected — it is second-order).

Quiz 2:

Suppose $\alpha = 1$, $\Delta x = 0.1$. What is the maximum Δt for stability in the explicit scheme?

(Hint: $\lambda = \dfrac{\alpha \Delta t}{(\Delta x)^2} \leq \frac{1}{2}$.)

Answer 2

We need

$$\lambda = \frac{\alpha\, \Delta t}{(\Delta x)^2} \leq \frac{1}{2}.$$

With $\alpha = 1$ and $\Delta x = 0.1$, $(\Delta x)^2 = 0.01$. So

$$\frac{\Delta t}{0.01} \leq \frac{1}{2} \quad \Rightarrow \quad \Delta t \leq 0.01 \times \frac{1}{2} = 0.005.$$

Maximum allowable $\Delta t = 0.005$.

7.4 Heat equation — Matrix Method with Non-zero Dirichlet Conditions

Pavni: We already saw how to discretize the heat equation with FTCS. But can we write the whole scheme in a more compact way?

Acharya: Yes. That's where the **matrix method** comes in. Let's recall the PDE:

$$u_t = \alpha^2 u_{xx}, \quad 0 < x < 1,\ t > 0,$$

with Dirichlet conditions

$$u(0,t) = L, \qquad u(1,t) = R,$$

and initial condition $u(x,0) = f(x)$.

Pavni: So we still set up the grid in space and time?

Acharya: Exactly. The FTCS update at interior nodes is

$$u_i^{j+1} = u_i^j + \lambda\,(u_{i-1}^j - 2u_i^j + u_{i+1}^j), \quad i = 1, \dots, N_x - 2,$$

where $\lambda = \frac{\alpha^2 \Delta t}{\Delta x^2}$.

Pavni: That's a lot of coupled equations. How do we collect them?

Acharya: We put all interior values into a vector

$$u^{(j)} = \begin{bmatrix} u_1^j \\ u_2^j \\ \vdots \\ u_{N_x-2}^j \end{bmatrix}.$$

Pavni: And the update rule becomes a matrix multiplication?

Acharya: Yes. We can write

$$u^{(j+1)} = A\,u^{(j)} + b,$$

where A is tridiagonal and b accounts for the boundary conditions:

$$A = \begin{bmatrix} 1-2\lambda & \lambda & & & \\ \lambda & 1-2\lambda & \lambda & & \\ & \ddots & \ddots & \ddots & \\ & & \lambda & 1-2\lambda & \lambda \\ & & & \lambda & 1-2\lambda \end{bmatrix}, \qquad b = \begin{bmatrix} \lambda L \\ 0 \\ \vdots \\ 0 \\ \lambda R \end{bmatrix}.$$

Pavni: So if $L = R = 0$, then $b = 0$ and we just have $u^{(j+1)} = Au^{(j)}$.

Acharya: Precisely. That's the beauty of the matrix method: it organizes the scheme into a linear algebra update.

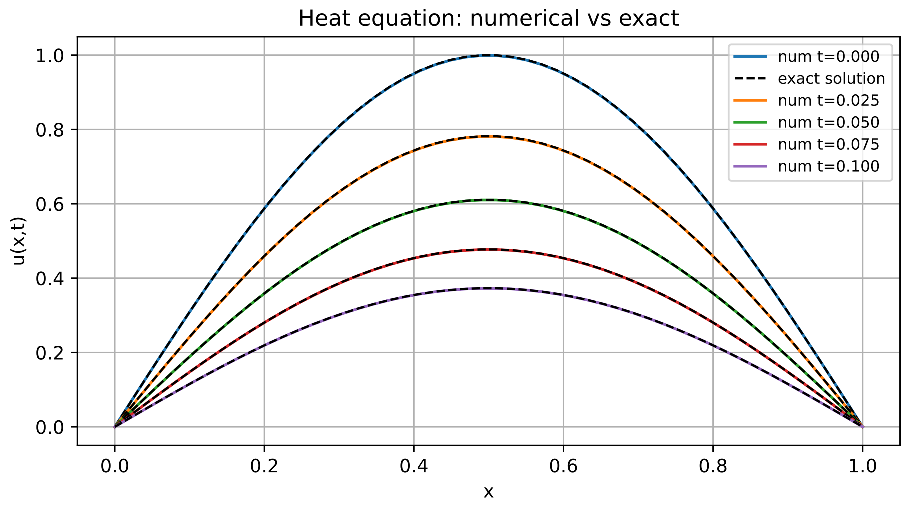
7.4.1 Implementation in Python

Let us now implement the method in Python and compare with the exact solution for $f(x) = \sin(\pi x)$:

$$u(x,t) = e^{-\pi^2 \alpha t}\,\sin(\pi x).$$

```python
import numpy as np
import matplotlib.pyplot as plt

alpha = 1.0
Nx    = 50
T     = 0.1
L, R  = 0.0, 0.0

dx = 1.0/(Nx-1)
dt_max = dx*dx/(2*alpha)
s  = 0.4
dt = s*dt_max
Nt = int(T/dt)
dt = T/Nt
r  = alpha*dt/dx**2

x  = np.linspace(0.0, 1.0, Nx)
u0 = np.sin(np.pi * x)

m = Nx - 2
main = (1 - 2*r) * np.ones(m)
```

```python
off  = r * np.ones(m-1)
A = np.diag(main) + np.diag(off,1) + np.diag(off,-1)

b = np.zeros(m)
b[0], b[-1] = r*L, r*R

u = u0.copy()
u_in = u[1:-1].copy()
snapshots = []
snap_times = np.linspace(0, Nt-1, 5, dtype=int)

for j in range(Nt):
    u_in = A @ u_in + b
    u[1:-1] = u_in
    if j in snap_times or j == Nt-1:
        snapshots.append((j*dt, u.copy()))

plt.figure(figsize=(8,4))
first_exact = True
for t_here, u_snap in snapshots:
    plt.plot(x, u_snap, label=f"num t={t_here:.3f}")
    u_exact = np.exp(-np.pi**2*alpha*t_here) * np.sin(np.pi*x)
    if first_exact:
        plt.plot(x, u_exact, 'k--', linewidth=1.2, label='exact solution')
        first_exact = False
    else:
        plt.plot(x, u_exact, 'k--', linewidth=1.2, label="_nolegend_")

plt.xlabel("x"); plt.ylabel("u(x,t)")
plt.title("Heat equation: numerical vs exact")
plt.legend(fontsize="small")
plt.grid(True)
plt.show()
```



7.4.2 Remarks

- The **linear algebra structure** $u^{(j+1)} = Au^{(j)} + b$ makes the scheme compact and systematic.
- The stability restriction $\lambda \leq \frac{1}{2}$ follows from analyzing the eigenvalues of A.
- For large N_x, A is sparse and tridiagonal — in practice, use `scipy.sparse.diags` for efficiency.

7.5 Backward Difference (Implicit) Scheme for the Heat Equation

Pavni: Acharya, the FTCS scheme works only if $\lambda \leq \frac{1}{2}$. Is there a method that avoids this restriction?

Acharya: Yes. We can use a **backward difference in time**, together with the same central difference in space. This gives the **Backward Euler scheme**, which is implicit but unconditionally stable.

Pavni: Implicit? What does that mean?

Acharya: It means that the new values u^{n+1} appear on both sides of the equation, so we must solve a system of equations at each time step.

7.5.1 Derivation

Start from the PDE:

$$u_t = \alpha^2 u_{xx}.$$

- Approximate the time derivative with a **backward difference**:

$$u_t(x_i, t^{n+1}) \;\approx\; \frac{u_i^{n+1} - u_i^n}{\Delta t}.$$

- Approximate the spatial second derivative at the new time level:

$$u_{xx}(x_i, t^{n+1}) \;\approx\; \frac{u_{i-1}^{n+1} - 2u_i^{n+1} + u_{i+1}^{n+1}}{(\Delta x)^2}.$$

The scheme becomes:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \alpha^2 \, \frac{u_{i-1}^{n+1} - 2u_i^{n+1} + u_{i+1}^{n+1}}{(\Delta x)^2}.$$

Rearrange:

$$-\lambda \, u_{i-1}^{n+1} + (1 + 2\lambda) \, u_i^{n+1} - \lambda \, u_{i+1}^{n+1} = u_i^n, \qquad \lambda = \frac{\alpha^2 \Delta t}{(\Delta x)^2}.$$

7.5.2 Matrix Form

Let $u^{(n)}$ be the vector of interior values at time step n. Then

$$Bu^{(n+1)} = u^{(n)},$$

where

$$B = \begin{bmatrix} 1+2\lambda & -\lambda & & & \\ -\lambda & 1+2\lambda & -\lambda & & \\ & \ddots & \ddots & \ddots & \\ & & -\lambda & 1+2\lambda & -\lambda \\ & & & -\lambda & 1+2\lambda \end{bmatrix}_{(N_x-2)\times(N_x-2)}.$$

So each step requires solving the linear system

$$u^{(n+1)} = B^{-1} u^{(n)}.$$

7.5.3 Stability

Pavni: Doesn't that make it more expensive than FTCS?

Acharya: It does, because we must solve a tridiagonal system at every time step.
But the reward is **unconditional stability**: for any $\Delta t > 0$ and $\Delta x > 0$, the scheme does not blow up.

Pavni: So no restriction like $\lambda \leq \frac{1}{2}$?

Acharya: Exactly. Backward Euler is stable for all λ.
It is only first-order accurate in time (like FTCS), but still second-order in space.

7.5.4 Remarks

- Backward Euler is more robust but requires solving a linear system at each step.

- For large systems, efficient algorithms like the **Thomas algorithm** (specialized Gaussian elimination for tridiagonal matrices) are used.

- In practice, one balances cost (explicit FTCS, cheap but conditionally stable) against robustness (implicit Backward Euler, unconditionally stable).

Why the Backward Euler scheme is unconditionally stable

If the update is

$$u^{(n+1)} = Au^{(n)},$$

and the initial error is e^0, then after n steps the error is

$$e^{(n)} = A^n e^0.$$

As before, the asymptotic condition is that the **spectral radius** $\rho(A) < 1$.

7.5.4.1 Eigenvalues of the Backward Euler iteration matrix

The Backward Euler scheme for the heat equation is

$$\frac{U^{n+1} - U^n}{\Delta t} = \alpha^2 AU^{n+1},$$

which rearranges to

$$U^{n+1} = (I - \lambda A)^{-1} U^n, \qquad \lambda = \frac{\alpha^2 \Delta t}{(\Delta x)^2}.$$

- If $\lambda_i(A)$ are the eigenvalues of the discrete Laplacian A, then the eigenvalues of the iteration matrix are

$$\mu_i = \frac{1}{1 - \lambda_i(A)}.$$

- For the 1D Laplacian with Dirichlet BCs,

$$\lambda_i(A) = -4\sin^2\left(\frac{i\pi}{2m}\right), \qquad i = 1, 2, \dots, m-1.$$

Thus

$$\mu_i = \frac{1}{1 + 4\lambda \sin^2\left(\frac{i\pi}{2m}\right)}.$$

7.5.4.2 Stability condition

- Since $\lambda > 0$ and $\sin^2(\cdot) \geq 0$, the denominator is always greater than 1.

- Therefore

$$0 < \mu_i < 1, \qquad \forall i.$$

This means all eigenvalues of the iteration matrix lie strictly inside the unit circle.

Thus, the **Backward Euler scheme is unconditionally stable**: - No restriction on Δt.
- Every mode decays monotonically.
- In contrast, FTCS required $\lambda \leq \frac{1}{2}$ for stability.

7.5.5 Interactive plots to see the eigenvalues of FTCS and Backward Differencce methods

```
Unable to display output for mime type(s): text/html

Unable to display output for mime type(s): text/html

Unable to display output for mime type(s): text/html
```

How to interact with the plots - Move the **slider** to change $\lambda = \frac{\alpha^2 \Delta t}{(\Delta x)^2}$.

- **FTCS plot**: notice that for $\lambda > 0.5$ some eigenvalues leave $[-1, 1]$, indicating instability.
- **Backward Euler plot**: eigenvalues always remain in $(0, 1)$, showing unconditional stability.
- Hover over points to see their values, zoom by dragging, double-click to reset.

Chapter 8

Crank–Nicolson scheme

Pavni: We've studied FTCS and Backward Euler for the heat equation. FTCS was simple, but had that annoying stability restriction $\lambda \leq \frac{1}{2}$.

Acharya: Yes. And Backward Euler solved that problem — it was unconditionally stable.

Pavni: But both of them are only *first-order accurate in time*, right?

Acharya: Exactly. And that's a problem if we want more accuracy without shrinking Δt too much.

Pavni: So FTCS is unstable unless $\lambda \leq \frac{1}{2}$, and Backward Euler is stable but too diffusive. We're still missing something better.

Acharya: That's where the **Crank–Nicolson scheme** comes in. The idea is to apply the trapezoidal rule in time — using the *average* of the spatial derivative at times n and $n + 1$.

Pavni: So instead of evaluating only at the old time (FTCS) or the new time (Backward Euler), we take a balance of both?

Acharya: Exactly. That trick makes the scheme **second-order accurate in time** and **unconditionally stable**. It's like combining the strengths of both FTCS and Backward Euler.

Pavni: Ah, so Crank–Nicolson is really about accuracy as well as stability. That sounds worth deriving!
Pavni: Let's derive Crank–Nicolson starting from the heat equation $u_t = \alpha^2 u_{xx}$. How do we get the time-centered scheme?

Acharya: Start by integrating the PDE in time over one step, from t^n to t^{n+1} at a fixed x:

$$u(x, t^{n+1}) - u(x, t^n) = \alpha^2 \int_{t^n}^{t^{n+1}} u_{xx}(x, t)\, dt.$$

Pavni: Now approximate that time integral — I remember the trapezoidal (averaging) rule.

Acharya: Exactly. Apply the trapezoidal rule to the integral on the right:

$$\int_{t^n}^{t^{n+1}} u_{xx}(x,t)\,dt \approx \frac{\Delta t}{2}\big(u_{xx}(x,t^n) + u_{xx}(x,t^{n+1})\big)$$

with pointwise quadrature error $O(\Delta t^3)$ (so after division by Δt the time-discretization error is $O(\Delta t^2)$).

Pavni: Then substitute that into the integrated PDE.

Acharya: We get

$$u(x,t^{n+1}) - u(x,t^n) = \alpha^2 \frac{\Delta t}{2}\big(u_{xx}(x,t^n) + u_{xx}(x,t^{n+1})\big) + O(\Delta t^3).$$

Pavni: Next we discretize u_{xx} in space with the centered second difference.

Acharya: Right. At grid point x_j,

$$u_{xx}(x_j,t^\ell) \approx \frac{U^\ell_{j-1} - 2U^\ell_j + U^\ell_{j+1}}{(\Delta x)^2},$$

which in vector form for interior nodes is $\dfrac{1}{(\Delta x)^2}AU^\ell$, where A is the usual tridiagonal Laplacian (stencil $[1,-2,1]$).

Substituting gives (collecting interior points into U^n)

$$U^{n+1} - U^n = \frac{\alpha^2 \Delta t}{2(\Delta x)^2}(AU^n + AU^{n+1}) + (\text{truncation terms}).$$

Pavni: Introduce the nondimensional parameter λ?

Acharya: Yes, set $\lambda = \dfrac{\alpha^2\Delta t}{(\Delta x)^2}$. Then

$$U^{n+1} - U^n = \frac{\lambda}{2}(AU^n + AU^{n+1}) + (\text{truncation terms}).$$

Bring U^{n+1} terms to the left to obtain

$$\Big(I - \tfrac{\lambda}{2}A\Big)U^{n+1} = \Big(I + \tfrac{\lambda}{2}A\Big)U^n.$$

Now, let's write these matrices explicitly. The left-hand side matrix is

$$L = I - \tfrac{\lambda}{2}A = \begin{bmatrix} 1+\lambda & -\frac{\lambda}{2} & 0 & \cdots & 0 \\ -\frac{\lambda}{2} & 1+\lambda & -\frac{\lambda}{2} & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & -\frac{\lambda}{2} & 1+\lambda & -\frac{\lambda}{2} \\ 0 & \cdots & 0 & -\frac{\lambda}{2} & 1+\lambda \end{bmatrix}.$$

The right-hand side matrix is

$$R = I + \tfrac{\lambda}{2}A = \begin{bmatrix} 1-\lambda & \frac{\lambda}{2} & 0 & \cdots & 0 \\ \frac{\lambda}{2} & 1-\lambda & \frac{\lambda}{2} & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \frac{\lambda}{2} & 1-\lambda & \frac{\lambda}{2} \\ 0 & \cdots & 0 & \frac{\lambda}{2} & 1-\lambda \end{bmatrix}.$$

So the Crank–Nicolson scheme can be written as

$$\boxed{L\,U^{n+1} = R\,U^n}.$$

Pavni: Great — and the eigenvalue form and stability follow from the spectral map of A.

Acharya: Right — each eigenvalue α_k of A is mapped to

$$\mu_k = \frac{1 + \frac{\lambda}{2}\alpha_k}{1 - \frac{\lambda}{2}\alpha_k},$$

and substituting $\alpha_k = -4\sin^2\!\left(\frac{k\pi}{2(N+1)}\right)$ yields

$$\mu_k = \frac{1 - 2\lambda\sin^2\!\left(\frac{k\pi}{2(N+1)}\right)}{1 + 2\lambda\sin^2\!\left(\frac{k\pi}{2(N+1)}\right)}.$$

This shows $|\mu_k| \le 1$, so CN is unconditionally stable.

Crank–Nicolson: derivation summary & stability

Starting point (integrated PDE):

$$u(x,t^{n+1}) - u(x,t^n) = \alpha^2 \int_{t^n}^{t^{n+1}} u_{xx}(x,t)\,dt.$$

Trapezoidal rule (time) + centered difference (space):

$$\int_{t^n}^{t^{n+1}} u_{xx}\,dt \approx \frac{\Delta t}{2}\big(u_{xx}(t^n) + u_{xx}(t^{n+1})\big), \qquad u_{xx}(x_j,t^\ell) \approx \frac{(AU^\ell)_j}{(\Delta x)^2}.$$

Matrix form (with $\lambda = \dfrac{\alpha^2\Delta t}{(\Delta x)^2}$):

$$\boxed{\left(I - \tfrac{\lambda}{2}A\right)U^{n+1} = \left(I + \tfrac{\lambda}{2}A\right)U^n\,.}$$

Eigenvalues (mode-wise): if $s_k = \sin(\frac{k\pi}{2(N+1)})$ then

$$\boxed{\mu_k = \frac{1 - 2\lambda s_k^2}{1 + 2\lambda s_k^2}\,, \quad k = 1, \ldots, N.}$$

Stability check (short): - For $\lambda \geq 0$ and $0 \leq s_k^2 \leq 1$ the denominator $1 + 2\lambda s_k^2 > 0$.
- Hence

$$|\mu_k| = \frac{|1 - 2\lambda s_k^2|}{1 + 2\lambda s_k^2} \leq 1,$$

and for nontrivial modes with $\lambda > 0$ strict inequality holds.
Conclusion: Crank–Nicolson is **unconditionally stable** and second-order accurate in time and space (global error $O(\Delta t^2 + \Delta x^2)$).

```
Unable to display output for mime type(s): text/html
```

How to interact:
- Move the slider to vary $\lambda = \dfrac{\alpha^2 \Delta t}{(\Delta x)^2}$.
- Hover on a marker to see the exact (k, μ_k).
- Zoom by dragging, double-click to reset.
Observe: all μ_k satisfy $|\mu_k| \leq 1$ for every $\lambda \geq 0$, confirming unconditional stability; for small λ the modes are near 1, for large λ high-frequency modes move nearer to -1 but remain bounded in magnitude by 1.

Chapter 9

Level 5 — Hyperbolic PDEs

Pavni: Acharya, we've studied the heat equation. From theory we know that it seems to *spread* information everywhere. Does every PDE behave like that?

Acharya: That's a good observation, Pavni. Not every PDE diffuses information infinitely fast like the parabolic heat equation.
Some equations describe situations where information or disturbances travel at a *finite speed* — like waves or moving traffic.

Pavni: So these are called *hyperbolic equations*?

Acharya: Exactly. Hyperbolic equations are the mathematical way of describing signals or quantities that *move* through a medium rather than *diffuse*.
For example — a vibrating string, sound waves in air, or even cars moving on a road.

Pavni: Interesting! But why are they so different from diffusion equations like the heat equation?

Acharya: Because diffusion equations allow disturbances to spread instantaneously everywhere — a small change at one point affects all others immediately. That's called **infinite propagation speed**.
But hyperbolic equations restrict influence to certain *paths* in the (x, t)-plane, called **characteristics**.
Information travels only along these paths at a finite speed.

Pavni: That sounds more physical. After all, signals, sound, or cars don't move infinitely fast!

Acharya: Exactly. Let's begin with a simple and familiar example — **traffic flow**.
It beautifully captures the idea of *conservation* and *finite-speed motion.*

9.1 The Traffic Flow Model

Pavni: So, where do we begin?

Acharya: Let's define some quantities.

- $\rho(x,t)$: density of cars at position x and time t (cars/km)
- $v(x,t)$: average speed of cars (km/h)
- $q(x,t)$: flow rate — number of cars passing a point per unit time (cars/hour)

Pavni: So q must depend on both how many cars there are and how fast they move?

Acharya: Precisely. That leads us to the simple physical relation:

$$q = \rho v.$$

Why $q = \rho v$?

If a small segment of road of length Δx contains $\rho\,\Delta x$ cars,
and each car moves with velocity v, then in time Δt each car covers $v\,\Delta t$ km.
So, the number of cars passing a fixed point in time Δt is

$$(\rho\,\Delta x) \cdot \frac{v\,\Delta t}{\Delta x} = \rho v\,\Delta t.$$

Dividing by Δt gives $q = \rho v$.

9.2 Conservation of Cars

Acharya: Now, think of a small stretch of road between x_1 and x_2.
Cars can only leave or enter through the ends.

$$\frac{d}{dt}\int_{x_1}^{x_2} \rho(x,t)\,dx = q(x_1,t) - q(x_2,t).$$

Pavni: That looks like conservation of mass!

Acharya: Exactly — just conservation of the *number of cars.*
The total number of cars in the interval changes only because of the *net flow* at the boundaries.

Now, let's use the **Fundamental Theorem of Calculus** to convert this integral form into a *local differential equation.*

By the Fundamental Theorem of Calculus,

$$q(x_2,t) - q(x_1,t) = \int_{x_1}^{x_2} q_x(x,t)\,dx.$$

Substitute this into our conservation statement:

$$\frac{d}{dt}\int_{x_1}^{x_2} \rho(x,t)\,dx = -\int_{x_1}^{x_2} q_x(x,t)\,dx.$$

Assuming ρ is smooth enough to interchange differentiation and integration:

$$\int_{x_1}^{x_2} \rho_t(x,t)\,dx = -\int_{x_1}^{x_2} q_x(x,t)\,dx.$$

Combine both integrals:

$$\int_{x_1}^{x_2} [\rho_t(x,t) + q_x(x,t)]\,dx = 0.$$

Acharya: Now, Pavni, what does this equation tell us?

Pavni: It says that the integral of $\rho_t + q_x$ over *any* interval is zero.

Acharya: Exactly! And that can only happen if the integrand itself is zero *everywhere*.
Hence, we get the **local conservation law**:

$$\boxed{\rho_t + q_x = 0.}$$

Pavni: So the integral form expresses the total number of cars in a segment being conserved,
and this differential form expresses that conservation *at every point* on the road!

Acharya: Perfectly said. This is the basic **conservation law** for one-dimensional flow.
Next, if we assume each car moves at a constant speed a, we'll get the **linear advection equation.**

Pavni: So this describes a traffic pattern moving forward without changing its shape?

Acharya: Exactly. The entire profile of car density just shifts rightward with speed a.

9.3 Try to Predict

Acharya: Before solving it, can you guess how the density will evolve if initially the density is a bump — say,

$$\rho(x,0) = e^{-x^2}?$$

Pavni: I think it will move to the right, keeping its shape.

Acharya: Perfect intuition. We'll now confirm this mathematically using **characteristics** — special curves along which information travels.

9.4 Characteristics and the Solution of the Linear Advection Equation

Pavni: Acharya, now that we have derived the equation

$$\rho_t + a\rho_x = 0, \quad a > 0,$$

for cars moving at constant speed, how do we actually find $\rho(x,t)$ from this?

Acharya: Excellent question. This equation may look simple, but it contains a beautiful idea — that **information travels along certain paths** in the (x,t)-plane. These paths are called *characteristics*.

9.4.1 Deriving the Characteristic Curves

Acharya: Let's think of ρ as a function of both x and t.
Suppose we move along a curve $x = x(t)$ in the (x,t)-plane.
By the chain rule, the total derivative of ρ along that curve is

$$\frac{d\rho}{dt} = \rho_t + \frac{dx}{dt}\rho_x.$$

Now, if we choose $\dfrac{dx}{dt} = a$, then

$$\frac{d\rho}{dt} = \rho_t + a\rho_x = 0.$$

That means ρ is **constant** along any curve that satisfies $\dfrac{dx}{dt} = a$.

Pavni: So those are the characteristic curves?

Acharya: Exactly! Integrating $\frac{dx}{dt} = a$ gives

$$x = at + x_0, \quad \text{or} \quad x - at = x_0 = \text{constant}.$$

Each line $x - at = \text{constant}$ is a **characteristic line**.

9.4.2 The Meaning of Characteristics

Pavni: What does it mean that ρ is constant along these lines?

Acharya: It means that the value of ρ at time t and position x
is exactly the same as its value at the point where that characteristic line started on the x-axis (that is, at time $t = 0$).

So,

$$\rho(x, t) = \rho_0(x - at),$$

where $\rho_0(x)$ is the initial density at $t = 0$.

Pavni: Oh! So the initial profile just shifts by at?

Acharya: Exactly — it moves to the right if $a > 0$ and to the left if $a < 0$.
No change in shape — pure translation.

9.4.3 Verifying the Solution

Pavni: Let's check if this really satisfies the PDE.

Acharya: Sure! Let $\rho(x, t) = \rho_0(x - at)$.
Then

$$\rho_t = -a\rho_0'(x - at), \quad \rho_x = \rho_0'(x - at).$$

Substitute into $\rho_t + a\rho_x = 0$:

$$(-a\rho_0') + a\rho_0' = 0.$$

It satisfies the equation perfectly.

9.4.4 Understanding Finite Propagation Speed

Acharya: The key property of this equation is that information moves at a *finite speed.*

If a disturbance is initially present only between $x = 0$ and $x = 1$,
then at time t it will be found only between $x = at$ and $x = 1 + at$.
Every point outside this region remains unaffected.

Pavni: So information doesn't spread instantly like in the heat equation?

Acharya: Exactly. In the **heat equation**, even a small bump affects the whole line immediately — infinite propagation speed.
But here, the influence travels only along straight lines $x - at = \text{constant}$ — that's **finite-speed propagation**.

9.4.5 Example

Let's take an initial density

$$\rho_0(x) = \begin{cases} 1, & 0 < x < 1, \\ 0, & \text{otherwise.} \end{cases}$$

Then

$$\rho(x,t) = \begin{cases} 1, & at < x < 1 + at, \\ 0, & \text{otherwise.} \end{cases}$$

Pavni: So the block of cars just moves forward as a group?

Acharya: Exactly — like a moving traffic wave with constant shape.

9.4.6 Visualizing Characteristics

Each characteristic line shows where information travels:

```
t ↑
  |
  |       /    /    /    /    /    /    /
  |      /    /    /    /    /    /    /
  |     /    /    /    /    /    /    /
  |____/____/____/____/____/____/____/→ x
```

9.5 Domain of Dependence and Domain of Influence

Pavni: Acharya, you mentioned earlier that information in hyperbolic equations travels along characteristics. But how exactly do we describe *which* parts of the initial data influence the solution at a given point?

Acharya: Excellent question, Pavni. To answer that, we introduce two key ideas — the **domain of dependence** and the **domain of influence**. These describe how information moves through space and time.

9.5.1 What do these mean?

Acharya: Let's begin intuitively.

- The **domain of dependence (DoD)** of a point (x, t) is the set of points in the initial data that can *affect* the solution at (x, t).
  In other words: *Which initial points influence the value here?*
- The **domain of influence (DoI)** of a point $(x_0, 0)$ is the set of space–time points (x, t) that can be affected by that initial point.
  In other words: *Where does the information starting at x_0 go?*

Pavni: So the DoD looks backward in time, and the DoI looks forward?

Acharya: Exactly. They're like mirror images of each other.
For hyperbolic PDEs, these regions are bounded by **characteristics** — the paths along which information travels at finite speed.

9.5.2 Example 1: Linear Advection

Pavni: Let's try this with the advection equation again:

$$u_t + a\,u_x = 0, \quad u(x, 0) = u_0(x).$$

Acharya: Good. The characteristics are the straight lines

$$x - at = \text{constant}.$$

So each point (x, t) connects to exactly one point on the x–axis: $x_0 = x - at$.

Pavni: So the value $u(x, t)$ depends only on $u_0(x - at)$?

Acharya: Exactly. That means:

- The **domain of dependence** of (x, t) is the **single point** $x_0 = x - at$.
- The **domain of influence** of an initial point $(x_0, 0)$ is the **straight line** $x = x_0 + at$.

Pavni: So for advection, both DoD and DoI are just lines — not regions?

Acharya: Right. A *single characteristic line* carries all the information.
This is why the advection equation has such a clean propagation behavior — each point of initial data simply moves at speed a without interacting with others.

9.5.3 Example 2: The 1-D Wave Equation

Pavni: What happens for the wave equation?

$$u_{tt} = c^2 u_{xx}.$$

Acharya: Ah, this one is a bit richer. The general solution is given by d'Alembert's formula:

$$u(x,t) = F(x-ct) + G(x+ct).$$

Pavni: So now we have two families of characteristics — one moving right, one left?

Acharya: Precisely! For this equation: - Right-moving characteristics: $x - ct =$ constant
- Left-moving characteristics: $x + ct =$ constant

Now, to find the value at (x,t), you need both F and G — meaning information from **two points** on the initial line:

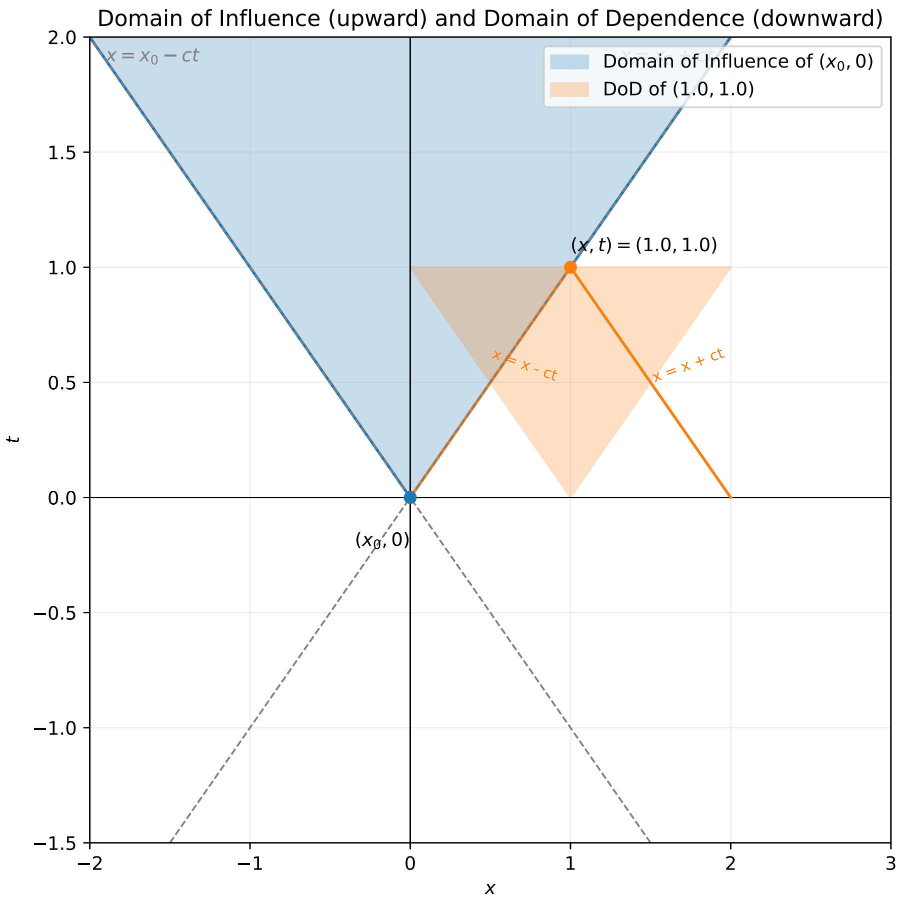
$$x - ct \quad \text{and} \quad x + ct.$$

Pavni: So the DoD of (x,t) is the interval between those two points?

Acharya: Yes. The **domain of dependence** is

$$[x - ct,\, x + ct].$$

And the **domain of influence** of an initial point $(x_0, 0)$ is the cone-shaped region.



9.6 Nonlinear Advection and the Formation of Shocks

Pavni: Acharya, the linear advection equation made perfect sense — the whole profile just moves without changing shape.
But traffic jams don't behave that nicely! Sometimes cars slow down abruptly and the jam gets worse over time.
Does our model capture that?

Acharya: Excellent question, Pavni. The linear advection equation assumes that every car moves with the *same constant speed a*.
But in reality, cars move **slower when the road is crowded** — so the speed depends on the density of cars, ρ.

Pavni: So v depends on ρ? Like $v = v(\rho)$?

Acharya: Exactly. That makes the flux $q = \rho v(\rho)$ a nonlinear function of ρ. Substituting this into the conservation law

$$\rho_t + q_x = 0,$$

we get the **nonlinear advection equation**:

$$\rho_t + \big(f(\rho)\big)_x = 0,$$

where $f(\rho) = \rho v(\rho)$.

9.6.1 Example: Traffic Flow Model

Pavni: What kind of $v(\rho)$ should we take?

Acharya: A simple and realistic choice is

$$v(\rho) = 1 - \rho,$$

meaning cars move at unit speed when the road is empty ($\rho = 0$),
and stop completely when the road is jammed ($\rho = 1$).

Then

$$f(\rho) = \rho(1 - \rho) = \rho - \rho^2,$$

and our equation becomes

$$\rho_t + (\rho - \rho^2)_x = 0.$$

Pavni: Expanding that gives

$$\rho_t + (1 - 2\rho)\rho_x = 0.$$

So it looks just like the advection equation, but the wave speed now depends on ρ itself!

Acharya: Exactly! This is called the **nonlinear advection equation** or **inviscid Burgers' equation**.

9.7 Characteristics in the Nonlinear Case

Pavni: Earlier, for $\rho_t + a\rho_x = 0$, we had straight characteristic lines $x - at =$ constant.
How does it work here when the speed depends on ρ?

Acharya: Let's find out. As before, along a curve $x = x(t)$, the total derivative is

$$\frac{d\rho}{dt} = \rho_t + \frac{dx}{dt}\rho_x.$$

From the PDE,

$$\rho_t = -(1-2\rho)\rho_x.$$

Substitute this:

$$\frac{d\rho}{dt} = \rho_t + \frac{dx}{dt}\rho_x = \big[-(1-2\rho) + \frac{dx}{dt}\big]\rho_x.$$

For $\frac{d\rho}{dt} = 0$ along a characteristic, we must have

$$\frac{dx}{dt} = 1 - 2\rho.$$

Pavni: So, ρ is constant along a curve whose slope depends on ρ itself?

Acharya: Precisely!
Each "traffic packet" (region of constant density) moves with its own speed $1 - 2\rho$.
That's the big difference from the linear case — now the **characteristics themselves depend on the solution**.

9.8 When Characteristics Meet — Shock Formation

Pavni: What happens if cars ahead are slower than those behind?
Would the faster ones eventually catch up?

Acharya: That's the perfect intuition. Let's see what that means mathematically.

Suppose initially $\rho(x, 0)$ decreases with x — that is, higher density (slower cars) ahead, and lower density (faster cars) behind.
Then the characteristic speed $1 - 2\rho$ is **larger behind** and **smaller ahead**.

That means characteristics **converge** — they start to intersect.
When that happens, two different characteristic lines try to assign *different values* of ρ at the same (x, t).

Pavni: So the solution becomes *multivalued*? That's not physical.

Acharya: Exactly — in reality, when cars catch up, a **shock** forms — a sharp front separating high and low densities.
The PDE solution must then be replaced by a *weak solution* that allows for discontinuities satisfying the **Rankine–Hugoniot condition**.

9.9 The Shock Speed (Rankine–Hugoniot Condition)

Let the shock position be $x = s(t)$.
On the left of the shock, density is ρ_L; on the right, it's ρ_R.

Conservation of cars gives the jump condition

$$\frac{ds}{dt} = \frac{f(\rho_L) - f(\rho_R)}{\rho_L - \rho_R}.$$

Pavni: So the shock moves at a speed equal to the slope of the line joining the two points on the flux curve $f(\rho)$?

Acharya: Perfect!
That's a nice geometric interpretation — the **secant slope** between $(\rho_L, f(\rho_L))$ and $(\rho_R, f(\rho_R))$.

For our example $f(\rho) = \rho(1 - \rho)$, we get

$$\frac{ds}{dt} = \frac{\rho_L(1 - \rho_L) - \rho_R(1 - \rho_R)}{\rho_L - \rho_R} = 1 - (\rho_L + \rho_R).$$

9.10 Example: A Forming Traffic Jam

Pavni: Let's take an example!

Acharya: Suppose the initial density is

$$\rho(x, 0) = \begin{cases} 0.8, & x < 0, \\ 0.2, & x > 0. \end{cases}$$

Then $\rho_L = 0.8$ (slow cars ahead) and $\rho_R = 0.2$ (fast cars behind).
The shock speed is

$$s' = 1 - (0.8 + 0.2) = 0.$$

Pavni: So the shock doesn't move — it stays fixed in place?

Acharya: Exactly! That represents a *stationary traffic jam* — cars pile up until the density behind adjusts.
If we reverse the situation — slower cars behind faster ones — the characteristics diverge instead, creating a **rarefaction wave** rather than a shock.

9.11 Key Takeaways

- Linear advection: constant-speed characteristics → waves move unchanged.
- Nonlinear advection: speed depends on ρ → characteristics can intersect.
- When they intersect, shocks form — discontinuous but physically meaningful solutions.
- The shock speed is determined by the Rankine–Hugoniot condition.
- This mechanism underlies shock waves, traffic jams, and compressible fluid flows.

Pavni: Wow, so from traffic flow we've reached the idea of *shock waves*!
That's amazing — how a simple conservation law captures such complex behavior.

Acharya: That's the beauty of hyperbolic equations — they describe how *information* and *discontinuities* propagate in nature.

9.12 Shock Waves

Pavni: Acharya, we saw how nonlinear advection can either *spread out* into a rarefaction or *pile up* into a shock. Can we work through a concrete example — step by step — and actually *see* the shock form?

Acharya: Absolutely. Let's use the simple traffic-inspired Burgers'-type model we had earlier:

$$\rho_t + (1 - 2\rho)\,\rho_x = 0,$$

so the characteristic speed is

$$c(\rho) = 1 - 2\rho.$$

We will pick an initial condition that produces a shock:

$$\rho(x, 0) = \begin{cases} 0.2, & x < 0, \\ 0.8, & x > 0. \end{cases}$$

Pavni: That's the case where the left side is sparse (fast) and the right side is dense (slow), so I expect traffic from the left to run into slow traffic on the right. That should make a shock, right?

Acharya: Exactly. Let's make the statements precise.

- Left state: $\rho_L = 0.2 \Rightarrow c_L = 1 - 2(0.2) = 0.6$ (moves right).
- Right state: $\rho_R = 0.8 \Rightarrow c_R = 1 - 2(0.8) = -0.6$ (moves left).

The two families of characteristics coming from left and right **collide** — they cross in the (x, t)-plane — so the continuous solution breaks down and a shock forms.

Pavni: And the shock speed is given by the Rankine–Hugoniot condition?

Acharya: Correct. For a conservation law $\rho_t + f(\rho)_x = 0$ the shock speed s satisfies

$$s = \frac{f(\rho_L) - f(\rho_R)}{\rho_L - \rho_R}.$$

Here $f(\rho) = \rho(1 - \rho)$, so for our numbers

$$f(\rho_L) = 0.2(0.8) = 0.16, \qquad f(\rho_R) = 0.8(0.2) = 0.16,$$

hence $s = \dfrac{0.16 - 0.16}{0.2 - 0.8} = 0.$
So this particular shock is *stationary* — the discontinuity sits at $x = 0$ while characteristics run into it.

9.13 Interactive Exploration — Shock vs Rarefaction

Acharya: Let's play! Try changing the left and right densities using the sliders below.
You'll see how the characteristics behave — whether they *collide* (shock) or *spread out* (rarefaction).

```
Unable to display output for mime type(s): text/html
```

```
Unable to display output for mime type(s): text/html
```

9.14 A Moving Shock: When the Traffic Jam Travels

Pavni: Acharya, earlier we saw a stationary shock where the boundary between two traffic regions just sat still.
Can the shock *move*?

Acharya: Certainly, Pavni.
Let's take another case:

$$\rho(x,0) = \begin{cases} 0.6, & x < 0, \\ 0.2, & x > 0. \end{cases}$$

Here the left side is denser — more cars — while the right side is freer.
Let's see what happens.

9.14.1 Compute the fluxes

Acharya: Our flux function is still $f(\rho) = \rho(1-\rho)$.
So,

$$f(0.6) = 0.24, \quad f(0.2) = 0.16.$$

The shock speed is found from the **Rankine–Hugoniot condition**:

$$s = \frac{f(\rho_L) - f(\rho_R)}{\rho_L - \rho_R} = \frac{0.24 - 0.16}{0.6 - 0.2} = 0.2.$$

Pavni: So the shock moves *to the right* with speed 0.2?

Acharya: Exactly.
Because the flux on the left is higher, the congested region pushes into the free region —
the "traffic jam" moves forward!

9.14.2 What happens to the characteristics?

Acharya: Let's look at the characteristic speeds:

$$c_L = 1 - 2\rho_L = 1 - 1.2 = -0.2, \quad c_R = 1 - 2\rho_R = 1 - 0.4 = 0.6.$$

So on the left, lines tilt gently left; on the right, they tilt right.
They intersect, and the envelope of all those intersections forms the **shock curve**:

$$x = st = 0.2t.$$

9.14.3 Visualizing the moving shock

```python
import numpy as np
import plotly.graph_objects as go

presets = [
    ("Shock: rightward, s=0.2 (0.1,0.7)", 0.1, 0.7),
    ("Rarefaction (0.6,0.2)", 0.6, 0.2),
    ("Shock: stationary, s=0 (0.4,0.6)", 0.4, 0.6),
    ("Shock: leftward, s=-0.1 (0.3,0.8)", 0.3, 0.8)
]

def make_preset_frames(rho_L, rho_R, preset_idx, t_max=3.0, n_frames=28, n_char=40):
    """Build frames for a Riemann problem for Burgers equation u_t + (u(1-u))_x = 0."""
    f = lambda r: r*(1-r)
    a = lambda r: 1 - 2*r
    fL, fR = f(rho_L), f(rho_R)

    # Shock speed (Rankine-Hugoniot)
    if not np.isclose(rho_L, rho_R):
        s = (fL - fR) / (rho_L - rho_R)
    else:
        s = a(rho_L)

    c_L, c_R = a(rho_L), a(rho_R)

    # Entropy condition: shock is physical if characteristics converge: c_L > c_R
    # For the equation rho_t + (1-2*rho)*rho_x = 0, this is the correct condition
    is_shock = (c_L > c_R)

    t_frames = np.linspace(0, t_max, n_frames)
    frames = []

    for k, tk in enumerate(t_frames):
        traces = []

        if is_shock:
            # Shock case: rho_L > rho_R, characteristics collide
            # Left region (x < 0 at t=0): density = rho_L, speed = c_L
            x0_left = np.linspace(-2.0, -0.01, n_char)
            for x0 in x0_left:
                ts = np.linspace(0, tk, 18)
                # Characteristic starts at (x0, 0) and moves with speed c_L
                xs = x0 + c_L * ts
                traces.append(go.Scatter(x=xs, y=ts, mode="lines",
                                         line=dict(color="royalblue", width=0.9),
                                         showlegend=False))

            # Right region (x > 0 at t=0): density = rho_R, speed = c_R
            x0_right = np.linspace(0.01, 2.0, n_char)
            for x0 in x0_right:
                ts = np.linspace(0, tk, 18)
                # Characteristic starts at (x0, 0) and moves with speed c_R
                xs = x0 + c_R * ts
                traces.append(go.Scatter(x=xs, y=ts, mode="lines",
                                         line=dict(color="royalblue", width=0.9),
                                         showlegend=False))

            # Shock trajectory: starts at (0,0), moves with speed s
            ts_shock = np.linspace(0, tk, 50)
            xs_shock = s * ts_shock
            traces.append(go.Scatter(x=xs_shock, y=ts_shock, mode="lines",
                                     line=dict(color="red", width=3),
                                     name=f"shock s={s:.3f}"))
        else:
            # Rarefaction case: rho_L < rho_R, characteristics spread out
            # Left region
            x0_left = np.linspace(-2.0, -0.01, n_char//2)
            for x0 in x0_left:
                ts = np.linspace(0, tk, 18)
                xs = x0 + c_L * ts
                traces.append(go.Scatter(x=xs, y=ts, mode="lines",
                                         line=dict(color="royalblue", width=0.9),
                                         showlegend=False))

            # Right region
            x0_right = np.linspace(0.01, 2.0, n_char//2)
            for x0 in x0_right:
                ts = np.linspace(0, tk, 18)
                xs = x0 + c_R * ts
                traces.append(go.Scatter(x=xs, y=ts, mode="lines",
                                         line=dict(color="royalblue", width=0.9),
                                         showlegend=False))

            # Rarefaction fan: characteristics from origin with speeds between c_L and c_R
            speeds = np.linspace(c_L, c_R, 41)
            for xi in speeds:
                ts = np.linspace(0, tk, 18)
                xs = xi * ts
                traces.append(go.Scatter(x=xs, y=ts, mode="lines",
                                           line=dict(color="green", width=1.2),
                                           showlegend=False, opacity=0.7))

            # Boundary characteristics of rarefaction fan
            traces.append(go.Scatter(x=c_L * np.linspace(0, tk, 2), y=np.linspace(0, tk
                                         mode="lines", line=dict(color="darkgreen", width=
                                         name=f"c_L={c_L:.2f}"))
            traces.append(go.Scatter(x=c_R * np.linspace(0, tk, 2), y=np.linspace(0, tk
                                         mode="lines", line=dict(color="darkgreen", width=
                                         name=f"c_R={c_R:.2f}"))

        frame_name = f"p{preset_idx}_f{k}"
        frames.append(go.Frame(data=traces, name=frame_name))

    return frames, s, is_shock

# Build all presets
all_frames = []
preset_meta = []
for i, (label, rL, rR) in enumerate(presets):
    frames_i, s_i, is_shock_i = make_preset_frames(rL, rR, i)
    all_frames.extend(frames_i)
    frame_names_i = [fr.name for fr in frames_i]
    preset_meta.append({
        "label": label, "rho_L": rL, "rho_R": rR,
        "s": s_i, "is_shock": is_shock_i,
        "frame_names": frame_names_i,
        "first_frame_name": frames_i[0].name
    })

# Pad frames so all have same number of traces
max_traces = max(len(fr.data) for fr in all_frames)
for fr in all_frames:
    n_missing = max_traces - len(fr.data)
    if n_missing > 0:
        fr.data += tuple(go.Scatter(x=[None], y=[None], visible=False) for _ in range(

# Build figure
default_idx = 0
first_frame = next(fr for fr in all_frames if fr.name == preset_meta[default_idx]["fir
fig = go.Figure(frames=all_frames)
fig.add_traces(first_frame.data)

# Slider and menus
default_frames = preset_meta[default_idx]["frame_names"]
```

```
steps = [dict(method="animate",
              args=[[name], dict(mode="immediate", frame=dict(duration=60, redraw=True))],
              label=f"{k}") for k, name in enumerate(default_frames)]
slider = [dict(active=0, pad={"t": 40}, steps=steps, currentvalue={"prefix": "frame: "})]

updatemenus = [
    dict(type="buttons", direction="left", x=0.02, y=0.95,
         buttons=[
             dict(label="Play", method="animate",
                  args=[default_frames, {"frame": {"duration": 80, "redraw": True}, "fromcurrent"
             dict(label="Pause", method="animate", args=[[None], {"frame": {"duration": 0, "redra
         ]),
    dict(type="buttons", direction="down", x=1.02, y=0.8,
         buttons=[
             dict(label=meta["label"], method="animate",
                  args=[meta["frame_names"],
                        {"frame": {"duration": 80, "redraw": True}, "mode": "immediate"}])
             for meta in preset_meta
         ])
]

fig.update_layout(
    title=preset_meta[default_idx]["label"] + " - Riemann presets",
    xaxis_title="x",
    yaxis_title="t (time ↑)",
    xaxis=dict(range=[-1.5, 1.5]),
    yaxis=dict(range=[0, 3]),
    sliders=slider,
    updatemenus=updatemenus,
    width=900, height=520,
    margin=dict(t=80, l=60, r=140, b=60)
)

fig.show()
```

```
Unable to display output for mime type(s): text/html
```

Chapter 10

From Traffic Flow to Numerical Schemes

We just modeled traffic flow and arrived at the **linear transport (advection) equation**

$$u_t + a\, u_x = 0, \qquad a > 0,$$

where $u(x,t)$ is the density and a is the propagation speed.

Let a uniform grid be

$$x_j = j\,\Delta x, \qquad t^n = n\,\Delta t, \qquad u_j^n \approx u(x_j, t^n), \qquad \lambda = \frac{a\,\Delta t}{\Delta x}.$$

Pavni: Last time we derived $u_t + a\,u_x = 0$ from traffic flow. How do we solve it numerically?

Acharya: First, try the tempting **FTCS** (Forward in time, Central in space):

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} + a\,\frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x} = 0 \implies u_j^{n+1} = u_j^n - \frac{\lambda}{2}\,(u_{j+1}^n - u_{j-1}^n).$$

Pavni: It looks symmetric and simple. Why not use it?

Acharya: Two reasons:

1. **Direction of information (traffic logic).** For $a > 0$, information travels **from left to right**. FTCS is symmetric— it lets the **downstream** value u_{j+1}^n influence the update equally with the **upstream** value u_{j-1}^n. That clashes with the one-way nature of characteristics.

2. **Instability (preview).** FTCS for pure advection amplifies some Fourier modes (no built-in damping) and blows up. We'll prove this with von Neumann analysis in the next level.

Pavni: So our spatial difference should look **upstream**.

Acharya: Exactly. That idea is **upwinding**.

10.1 Deriving the Upwind Scheme

Acharya: For $a > 0$, approximate u_x by a **backward (upwind) difference**:

$$u_x(x_j, t^n) \approx \frac{u_j^n - u_{j-1}^n}{\Delta x}.$$

Use forward Euler in time for u_t:

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} + a\,\frac{u_j^n - u_{j-1}^n}{\Delta x} = 0.$$

Pavni: Rearranging gives the update

$$\boxed{u_j^{n+1} = u_j^n - \lambda\,(u_j^n - u_{j-1}^n), \qquad \lambda = \frac{a\Delta t}{\Delta x},\ a > 0.}$$

Acharya: If $a < 0$ (information enters from the **right**), use a **forward** difference:

$$\boxed{u_j^{n+1} = u_j^n - \lambda\,(u_{j+1}^n - u_j^n), \qquad \lambda = \frac{a\Delta t}{\Delta x} < 0.}$$

Pavni: So "upwind" means: *differentiate in the direction information arrives from.*

10.2 What Upwind Does (Qualitative)

Acharya: Upwind is **stable** under a Courant (CFL) constraint and **diffusive**: sharp fronts smear slightly because the update is a convex combination of u_j^n and its upstream neighbor.

Pavni: And FTCS?

Acharya: FTCS ignores directionality and is **unstable** for advection—save it for diffusion-type problems; not for pure transport.

Note: CFL condition (preview)

The Courant number

$$\lambda = a\Delta t/\Delta x$$

should satisfy

$$\boxed{|\lambda| \le 1}$$

so that in one time step information does not "jump over" more than one cell. We'll derive this precisely via von Neumann analysis next time.

Note: Boundary conditions

On a finite interval, specify **inflow** data only.
For $a > 0$, prescribe $u(0, t)$; the outflow boundary at $x = L$ needs no condition.
For **periodic** problems, wrap indices: $u^n_{-1} \equiv u^n_{N-1}$.

10.3 Blackboard Box (Ready to Project)

- Grid and notation: $x_j = j\Delta x,\ t^n = n\Delta t,\ u^n_j \approx u(x_j, t^n),\ \lambda = a\Delta t/\Delta x$.
- **FTCS (don't use for advection):**

$$u^{n+1}_j = u^n_j - \frac{\lambda}{2}(u^n_{j+1} - u^n_{j-1}) \quad \text{(unstable for advection).}$$

- **Upwind, $a > 0$:**

$$\boxed{u^{n+1}_j = u^n_j - \lambda\,(u^n_j - u^n_{j-1}), \quad |\lambda| \le 1.}$$

- **Upwind, $a < 0$:**

$$\boxed{u^{n+1}_j = u^n_j - \lambda\,(u^n_{j+1} - u^n_j), \quad |\lambda| \le 1.}$$

Pavni: FTCS is elegant but wrong tool for traffic—
Acharya: —whereas **upwind respects the flow of information** and gives us a reliable first solver.
Next level: stability proofs and higher-order (Lax–Friedrichs, Lax–Wendroff).

Chapter 11

Von Neumann Stability Analysis

Pavni: We already checked stability for the FTCS heat solver using the matrix (eigenvalue) approach. I want to see the Fourier-mode (von Neumann) method – the frequency-by-frequency picture.

Acharya: Sure. The von Neumann method tests how individual sinusoidal (Fourier) modes are amplified (or damped) by a linear, constant-coefficient finite-difference scheme. Because such schemes act independently on each Fourier mode, we can study one mode at a time and require every mode to be non-amplifying.

11.1 Von Neumann analysis for the FTCS scheme

Acharya: Let's start from the 1-D heat equation

$$u_t = \alpha\, u_{xx}, \qquad \alpha > 0,$$

discretized on a uniform grid $x_j = j\,\Delta x$, $t_n = n\,\Delta t$ with the explicit Forward-Time Centered-Space (FTCS) scheme

$$u_j^{n+1} = u_j^n + r\big(u_{j+1}^n - 2u_j^n + u_{j-1}^n\big), \qquad r := \alpha\,\frac{\Delta t}{\Delta x^2}.$$

(When studying errors we set $e_j^n = u_j^n - u(x_j, t_n)$.)

Pavni: Why can we assume a Fourier (sinusoidal) form for the error?

Acharya: Because the scheme is linear with constant coefficients. That means if you feed a single Fourier mode into the linear update, it stays a Fourier mode – only its amplitude changes. Fourier modes form a basis for discrete periodic signals, so studying one mode suffices.

Assume an error (or mode) of the form

$$e_j^n = \xi^n e^{ikx_j},$$

where k is the spatial wavenumber and ξ (the amplification factor) is the complex number that tells how the mode's amplitude changes per time step.

Pavni: Plug that into the error update.

Acharya: The error update mirrors the scheme:

$$e_j^{n+1} = e_j^n + r(e_{j+1}^n - 2e_j^n + e_{j-1}^n).$$

Substitute $e_j^n = \xi^n e^{ikx_j}$ and $x_{j\pm1} = x_j \pm \Delta x$:

$$\xi^{n+1} e^{ikx_j} = \xi^n e^{ikx_j} + r\,\xi^n \big(e^{ik(x_j+\Delta x)} - 2e^{ikx_j} + e^{ik(x_j-\Delta x)}\big).$$

Cancel the common nonzero factor $\xi^n e^{ikx_j}$ to get

$$\xi = 1 + r\big(e^{ik\Delta x} - 2 + e^{-ik\Delta x}\big).$$

Use $e^{i\theta} + e^{-i\theta} = 2\cos\theta$ and simplify:

$$\xi = 1 + 2r(\cos(k\Delta x) - 1) = 1 - 4r\sin^2\left(\frac{k\Delta x}{2}\right).$$

This is the amplification factor for a Fourier mode of wavenumber k.

11.2 Stability condition

Pavni: So what condition on r ensures stability?

Acharya: For stability we require that every Fourier mode is not amplified, i.e.

$$|\xi| \le 1 \quad \text{for all allowed } k.$$

For this FTCS case ξ is real, so the requirement reduces to $-1 \le \xi \le 1$.
The maximum value of ξ is 1 (when $\sin^2 = 0$), so we only need to check the minimum.

The minimum occurs when $\sin^2(\cdot) = 1$, giving $\xi_{\min} = 1 - 4r$.
Requiring $\xi_{\min} \geq -1$ gives

$$1 - 4r \geq -1 \quad \Longrightarrow \quad r \leq \tfrac{1}{2}.$$

Hence the von Neumann stability condition is

$$\boxed{\alpha \frac{\Delta t}{\Delta x^2} \leq \tfrac{1}{2} \quad \Longleftrightarrow \quad \Delta t \leq \frac{\Delta x^2}{2\alpha}.}$$

11.3 Comments and interpretation

- Long (low-frequency) modes have $\sin^2 \approx (k\Delta x/2)^2$ and so ξ is near 1; they decay slowly.
- Short (high-frequency) modes have $\sin^2$ near 1 and are damped more strongly (smaller ξ).
- If $r > 1/2$, the shortest modes have $\xi < -1$ leading to oscillatory growth and instability.
- The von Neumann method is equivalent to the matrix/eigenvalue analysis for linear constant-coefficient schemes on periodic grids. Fourier modes are eigenvectors of circulant matrices; $\xi(k)$ are the eigenvalues.

11.4 Quick exercise

Exercise.
For the heat-equation FTCS scheme above, compute $\xi(k)$ for the two extreme wavenumbers on a grid with N points and periodic boundary conditions:
$k = 0$ and the Nyquist mode $k = \pi/\Delta x$.
Interpret the values.

Click to reveal the answer

Answer (sketch).
For $k = 0$, $\sin^2(0) = 0$ so $\xi = 1$ (constant mode preserved).
For $k = \pi/\Delta x$, $\sin^2(\pi/2) = 1$ so $\xi = 1 - 4r$ (maximal damping; needs $r \leq 1/2$ for stability).

11.5 Von Neumann analysis for the Upwind and Lax–Friedrichs schemes

Pavni: Now that we finished the heat equation, can we do von Neumann analysis for a hyperbolic equation – say, the linear advection equation?

Acharya: Perfect next step. Let's take the simplest hyperbolic PDE:

$$u_t + a\,u_x = 0, \qquad a > 0.$$

The exact solution is a wave that travels to the right at speed a without changing shape.

11.5.1 1. Upwind scheme

Pavni: Remind me of the upwind finite-difference scheme?

Acharya: For $a > 0$, we use the backward spatial difference:

$$u_j^{n+1} = u_j^n - C\,(u_j^n - u_{j-1}^n),$$

where $C = a\,\frac{\Delta t}{\Delta x}$ is the Courant number.

Assume $e_j^n = \xi^n e^{ikx_j}$ and substitute to obtain

$$\xi = 1 - C(1 - e^{-ik\Delta x}) = 1 - C\big(1 - \cos(k\Delta x) + i\sin(k\Delta x)\big).$$

Then

$$|\xi|^2 = [1 - C(1 - \cos(k\Delta x))]^2 + [C\sin(k\Delta x)]^2 = 1 - 2C(1 - C)(1 - \cos(k\Delta x)).$$

Because $1 - \cos(\cdot) \geq 0$, the scheme is stable if $0 \leq C \leq 1$:

$$\boxed{0 \leq a\,\frac{\Delta t}{\Delta x} \leq 1.}$$

11.5.2 2. Lax–Friedrichs scheme

The Lax–Friedrichs scheme is

$$u_j^{n+1} = \tfrac{1}{2}(u_{j+1}^n + u_{j-1}^n) - \tfrac{C}{2}(u_{j+1}^n - u_{j-1}^n),$$

which gives

$$\xi = \cos(k\Delta x) - i\,C\sin(k\Delta x), \qquad |\xi|^2 = 1 - (1 - C^2)\sin^2(k\Delta x).$$

Stability requires $|C| \le 1$:

$$\boxed{|a|\,\frac{\Delta t}{\Delta x} \le 1.}$$

Optional check: small-wavenumber expansions

For small $k\Delta x$, expand sin and cos to second order:

- **Upwind:** leading amplitude decay $\sim e^{-(1-C)(k\Delta x)^2/2}$.
- **Lax–Friedrichs:** leading amplitude decay $\sim e^{-(1-C^2)(k\Delta x)^2/2}$.

Hence Lax–Friedrichs damps short waves more strongly.

Pavni: So both methods are stable under $|C| \le 1$, but upwind is less diffusive.

Acharya: Exactly – von Neumann analysis clearly shows the trade-offs among stability, dissipation, and dispersion.

Chapter 12

Elliptic Partial Differential Equations

Pavni: We've seen waves (hyperbolic) and diffusion (parabolic). What are elliptic PDEs about?

Acharya: Elliptic PDEs describe *steady-state* or equilibrium phenomena where time is no longer present. Typical examples are steady heat conduction, electrostatics (potential problems), and incompressible potential flow.

Pavni: So instead of marching forward in time we solve for the entire spatial field at once?

Acharya: Exactly. That changes both our modelling viewpoint and the numerical tools: we discretize the spatial domain and solve a (usually large) algebraic system.

12.1 Prototypical equations

The two classical elliptic equations in two dimensions are

- **Laplace equation** (no sources):

$$u_{xx} + u_{yy} = 0,$$

- **Poisson equation** (with sources):

$$u_{xx} + u_{yy} = f(x, y).$$

Boundary conditions determine a well-posed problem (Dirichlet, Neumann, Robin).

A second-order linear PDE in 2D written as

$$Au_{xx} + 2Bu_{xy} + Cu_{yy} + \cdots = 0$$

is **elliptic** at a point if the discriminant satisfies

$$B^2 - AC < 0.$$

This local condition corresponds to the absence of real characteristic curves — there is no natural direction for propagation.

12.2 Boundary conditions

- **Dirichlet:** prescribe u on the boundary (value-driven).
- **Neumann:** prescribe normal derivative $\partial_n u$ on the boundary (flux-driven).
- **Robin:** linear combination of Dirichlet and Neumann.

Pavni: Which ones are easiest numerically?

Acharya: Dirichlet is simplest to impose in most finite-difference or finite-element contexts. Neumann requires consistent discretizations near the boundary (ghost points or one-sided differences).

12.3 Finite difference discretization — Laplace on a square

We illustrate with the unit square $\Omega = (0, 1)^2$ and Dirichlet boundary conditions.

Use a uniform grid with spacing $h = 1/(N + 1)$ and interior nodes (x_i, y_j) where $i, j = 1, \dots, N$.

Central differences give the five-point stencil at interior node (i, j):

$$\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h^2} = f_{i,j}.$$

Rearrange to the canonical form:

$$-u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1} + 4u_{i,j} = h^2 f_{i,j}.$$

This couples each grid point to its four neighbours.

Pavni: Wait, what happens along the edges of the square? We're only writing difference equations for the interior nodes.

Acharya: Good observation. On the boundaries, we must specify **boundary conditions** to make the problem well posed.
In this example, we assume **homogeneous Dirichlet conditions**, meaning

$$u(x, y) = 0 \quad \text{for } (x, y) \text{ on the boundary of } \Omega = (0, 1)^2.$$

That's like a square metal plate whose edges are held at zero temperature while heat is generated inside.

Pavni: So the temperature is fixed to zero all around, and the source term (f(x,y)=1) keeps the interior warm?

Acharya: Exactly. Numerically, we handle this by solving only for the **interior grid points**.
The grid spacing is (h = 1/(N+1)), so the first and last rows/columns correspond to the boundary, where (u=0).
We don't include those nodes in our system — the matrix (A) acts only on interior unknowns, which implicitly enforces (u=0) on the edges.

Pavni: That makes sense — so every (u_{i,j}) we compute corresponds to an interior temperature, with the boundaries fixed to zero.

12.4 Matrix form and properties

Ordering unknowns lexicographically (e.g. row-wise), the discrete Laplacian leads to a linear system

$$A\mathbf{u} = \mathbf{b}.$$

Key properties of the matrix A for the standard five-point Laplacian with Dirichlet BCs:

- symmetric
- sparse (banded block structure)
- positive-definite (for Poisson with Dirichlet BCs)

These properties guide solver choices — conjugate gradient (CG) is an excellent option when A is SPD.

12.5 Simple solvers (algorithmic sketch)

Pavni: Which solver should I try first?

Acharya: Start with iterative smoothers and then try Krylov methods:

- **Jacobi** (easy, textbook): parallel-friendly but slow.
- **Gauss–Seidel** (faster; sequential).
- **Successive Over-Relaxation (SOR)** (tune relaxation parameter ω).
- **Conjugate Gradient** with an incomplete Cholesky or algebraic multigrid preconditioner for large problems.

12.6 Example: Python — assemble and solve small Poisson

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.sparse import diags
from scipy.sparse.linalg import spsolve

# small example N x N interior points
N = 20
h = 1.0/(N+1)

# 1D Laplacian block
T = diags([1,-2,1], [-1,0,1], shape=(N,N)).toarray()
I = np.eye(N)
A = (np.kron(I, T) + np.kron(T, I)) / h**2

# RHS f(x,y) = 1.0
b = np.ones(N*N)

# Solve (for small N we use direct)
u = np.linalg.solve(A, b)

# reshape to 2D for plotting or inspection
U = u.reshape((N,N))

# Build coordinates for plotting
x = np.linspace(h, 1 - h, N)
y = np.linspace(h, 1 - h, N)
```

```
# --- 2D heatmap
plt.figure(figsize=(6,5))
plt.imshow(U, origin="lower", extent=[0, 1, 0, 1], aspect="auto")
plt.colorbar(label="u(x,y)")
plt.title("Poisson solution on unit square (Dirichlet 0, f 1)")
plt.xlabel("x"); plt.ylabel("y"); plt.tight_layout()
plt.show()

# --- 3D surface
from mpl_toolkits.mplot3d import Axes3D
X, Y = np.meshgrid(x, y)
fig = plt.figure(figsize=(6,5))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, U, cmap='viridis')
ax.set_title('3D surface of Poisson solution')
ax.set_xlabel('x'); ax.set_ylabel('y'); ax.set_zlabel('u(x,y)')
plt.tight_layout()
plt.show()
```

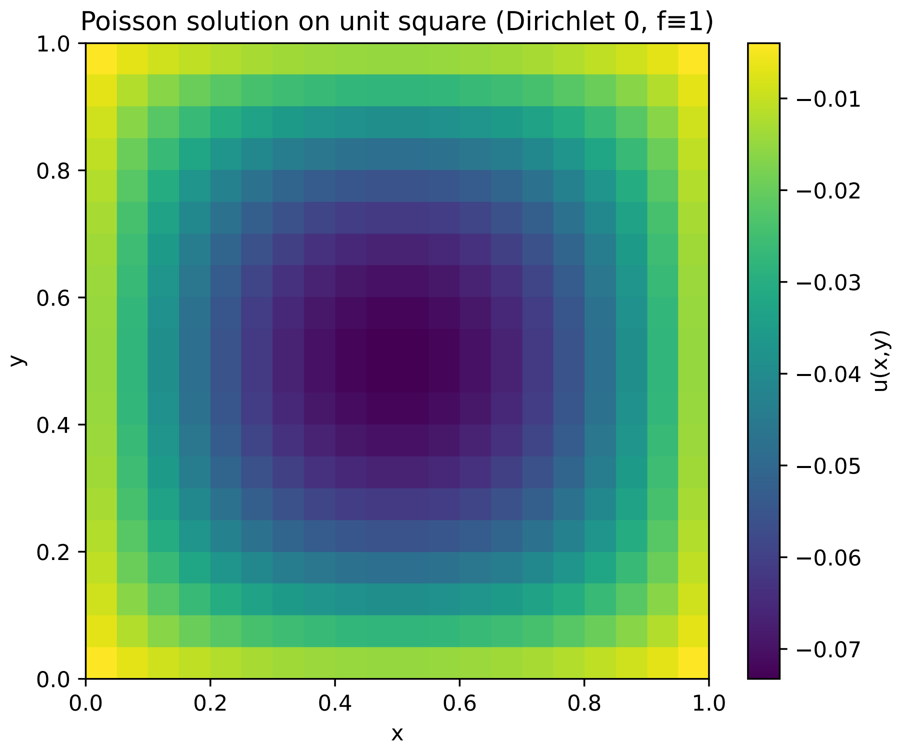
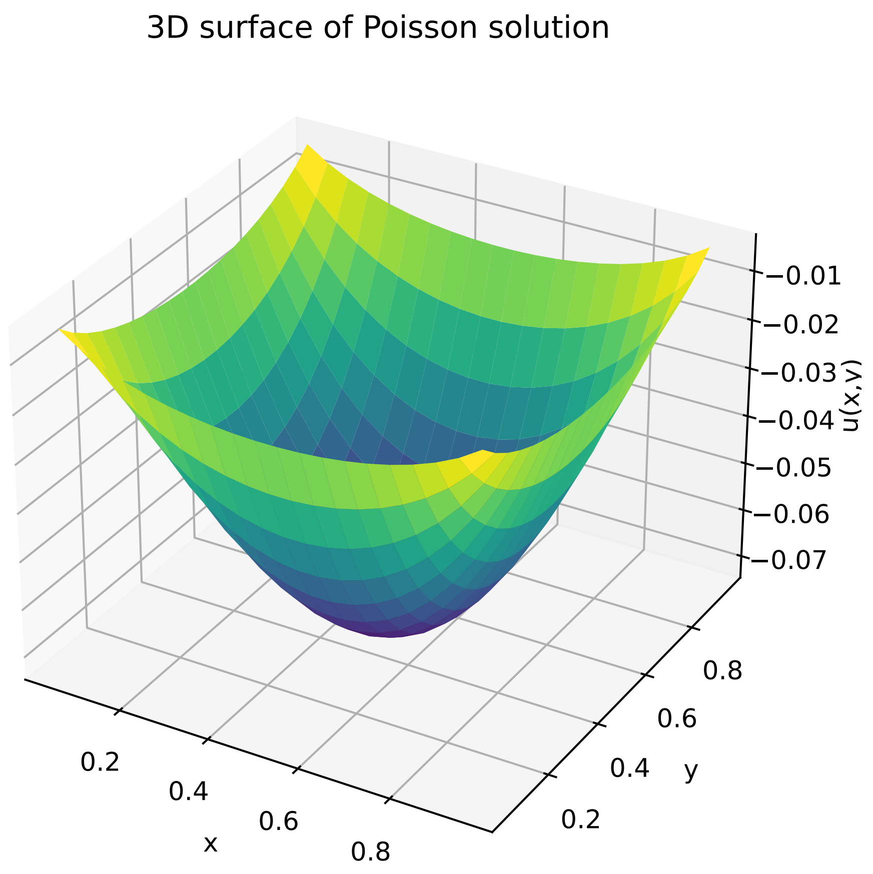


Figure 12.1: Poisson solution on unit square (Dirichlet 0, f 1) and its 3D surface.



12.7 Interpreting the results

12.8 Interpreting the results

Pavni: The plots look interesting — the surface dips down in the middle instead of forming a hill.
Why is that happening?

Acharya: Good observation! For our problem,

$$u_{xx} + u_{yy} = 1, \quad u = 0 \text{ on the boundary},$$

the Laplacian of

$$u$$

is **positive**, which means

$$u$$

must be **concave upward**.
So the solution forms a *bowl-shaped surface* — lowest at the center and rising smoothly to zero along the edges.

Pavni: So the interior is cooler, and the edges are hotter?

Acharya: Not quite. Remember, the boundary is fixed at zero temperature. The "depth" of the bowl means that

$$u$$

itself takes *negative values* inside the domain —
the diffusion operator is balancing a positive source term by lowering

$$u$$

below zero so that its curvature matches the source.
If you flipped the sign of

$$f(x, y)$$

to

$$-1$$

, you'd get the opposite: a *hill-shaped* surface with a positive peak at the center.

Pavni: Oh, so the sign of the source term determines whether the solution curves up or down?

Acharya: Exactly.
A positive source term with zero Dirichlet boundaries gives a concave-up "bowl," while a negative source gives a concave-down "hill."
The Laplacian enforces this curvature so that the spatial average of

$$u$$

compensates for the forcing.

Pavni: That's quite elegant — the geometry of the plot directly shows the sign and balance of the PDE.

Acharya: Precisely. Always interpret the numerical solution by checking:
1. The **sign** of the source term

$$f$$

,
2. The **type** of boundary condition, and
3. The **shape** (concavity) that satisfies the diffusion balance. ## Exercises — Exploring the effect of sources and boundary conditions

Pavni: These plots are quite revealing. What if we change the boundary conditions or the source term?

Acharya: Excellent question! Try modifying the code and observe how the steady-state solution responds.
Here are some guided experiments:

1. **Non-homogeneous Dirichlet boundary:**
   Keep
   $$f(x, y) = 1$$
   but set
   $$u = 100$$
   on the left edge ($x = 0$) and
   $$u = 0$$
   elsewhere.
   → *Observe how the temperature gradient forms from left to right.*

2. **Varying the internal heat source:**
   Replace

   ```
   f = np.ones((N, N))
   ```