Diffusion equation

So far, we have explored ordinary differential equations where physical quantities depend only on one variable (usually time). However, many physical phenomena involve quantities that change in both space and time simultaneously. These situations require partial differential equations (PDEs). The diffusion equation is one of the most important PDEs in physics, appearing in heat conduction, particle diffusion, and even quantum mechanics (as the time-dependent Schrödinger equation).

```
#| edit: false
#| echo: false
#| execute: true

import numpy as np
import matplotlib.pyplot as plt
from scipy import sparse
from scipy.sparse import linalg  # for sparse.linalg.spsolve
from scipy.integrate import odeint

# Set default plotting parameters
plt.rcParams.update({
    'font.size': 12,
    'lines.linewidth': 1,
    'lines.markersize': 5,
    'axes.labelsize': 11,
    'xtick.labelsize': 10,
    'ytick.labelsize': 10,
    'xtick.top': True,
    'xtick.direction': 'in',
    'ytick.right': True,
    'ytick.direction': 'in',
})
```

```
def get_size(w, h):
    return (w/2.54, h/2.54)
```

Physical Model

Understanding Diffusion

Imagine dropping a drop of ink into a glass of still water. Initially, all the ink particles are concentrated in one small region. Over time, you'll observe the ink spreading outward, gradually coloring the entire glass of water. This spreading process is called **diffusion**.

At the microscopic level, diffusion results from the random motion of particles (Brownian motion). Each ink particle undergoes a "random walk" - constantly changing direction due to collisions with water molecules. While individual particle motion is unpredictable, the collective behavior follows predictable statistical laws.

The animation below demonstrates this process: particles start concentrated at the center and gradually spread throughout the domain due to their random motion.

The Diffusion Equation

Rather than tracking individual particles, we work with **concentration** - the number of particles per unit volume at each point in space. The diffusion equation describes how this concentration changes over time:

$$\frac{\partial c(\mathbf{r}, t)}{\partial t} = D\nabla^2 c(\mathbf{r}, t) \tag{1}$$

Where:

- $c(\mathbf{r}, t)$ is the concentration at position $\mathbf{r}$ and time t
- D is the **diffusion coefficient** (units: length^2/time)
- ∇^2 is the Laplacian operator (measures the "curvature" of the concentration)

The diffusion coefficient D characterizes how quickly diffusion occurs - larger D means faster spreading.

```
//| echo: false
//| fig-align: center
width = 600
height = 600
margin = ({top: 20, right: 30, bottom: 20, left: 40})
plotHeight = 100

viewof simulation = {
  // Create main container
  const container = d3.create("div")
    .style("display", "flex")
    .style("flex-direction", "column");

  // Create SVG for particle simulation
  const svg = container.append("svg")
    .attr("width", width)
    .attr("height", height - plotHeight)
    .attr("viewBox", [0, 0, width, height - plotHeight]);

  // Create SVG for histogram
  const histogramSvg = container.append("svg")
    .attr("width", width)
    .attr("height", plotHeight)
    .attr("viewBox", [0, 0, width, plotHeight]);

  const numParticles = 1000;
  const D = 0.5; // Diffusion coefficient
  const numBins = 80;

  // Create particles at the center
  const particles = Array.from({length: numParticles}, () => ({
    x: width / 2,
    y: (height - plotHeight) / 2,
    vx: 0,
    vy: 0
  }));

  // Setup scales for histogram
  const xScale = d3.scaleLinear()
    .domain([0, width])
    .range([margin.left, width - margin.right]);

  const yScale = d3.scaleLinear()
    .domain([0, numParticles/5])
    .range([plotHeight - margin.bottom, margin.top]);

  // Create histogram generator
  const histogram = d3.bin()
    .domain(xScale.domain())
    .thresholds(xScale.ticks(numBins))
    .value(d => d.x);

  // Create histogram group
```

Simplifying to One Dimension

To make our numerical solution manageable, we'll consider diffusion along a single direction (say, the x-axis). This reduces our equation to:

$$\frac{\partial c(x,t)}{\partial t} = D\frac{\partial^2 c(x,t)}{\partial x^2} \tag{2}$$

This equation tells us: *The rate of change of concentration at any point equals the diffusion coefficient times the curvature of the concentration profile at that point.*

From Continuous to Discrete

To solve this equation numerically, we must discretize both space and time - converting our continuous problem into a discrete one that computers can handle.

Spatial Discretization

We divide our spatial domain into a grid of points separated by distance Δx. Instead of knowing $c(x,t)$ at every point x, we'll track concentrations at specific grid points: $c_0, c_1, c_2, \ldots, c_N$.

The second spatial derivative (curvature) at point i can be approximated using three neighboring points:

$$\left.\frac{\partial^2 c(x,t)}{\partial x^2}\right|_{x=x_i} \approx \frac{c_{i+1}^n - 2c_i^n + c_{i-1}^n}{\Delta x^2} \tag{3}$$

This finite difference formula captures the essential physics: if the concentration at point i is higher than the average of its neighbors, the curvature is negative, and diffusion will tend to reduce the concentration at point i.

Matrix Representation

We can represent this spatial discretization as a matrix operation. For our concentration vector $\mathbf{C} = [c_0, c_1, c_2, \ldots, c_N]^T$, the spatial derivative becomes:

$$\frac{\partial^2 \mathbf{C}}{\partial x^2} = \frac{1}{\Delta x^2}\mathbf{M}\mathbf{C}$$

where $\mathbf{M}$ is the second-difference matrix:

$$\mathbf{M} = \begin{bmatrix} -2 & 1 & 0 & 0 & \cdots & 0 \\ 1 & -2 & 1 & 0 & \cdots & 0 \\ 0 & 1 & -2 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 1 & -2 & 1 \\ 0 & \cdots & 0 & 0 & 1 & -2 \end{bmatrix}$$

Each row of this matrix implements the finite difference stencil for one grid point.

i Dirichlet Boundary Conditions (Fixed Values)

The matrix above assumes so-called **Dirichlet boundary conditions**, where the concentration is fixed at specific values at the boundaries (typically zero). This represents situations where particles are absorbed when they reach the boundaries.
Physical example: A pipe with absorbing walls where particles disappear upon contact.
Matrix structure: The first and last rows have the standard `-2, 1` pattern, enforcing $c_{-1} = 0$ and $c_{N+1} = 0$.

Neumann Boundary Conditions (Zero Flux)

For **open boundary conditions** (zero flux: $\frac{\partial c}{\partial x} = 0$ at boundaries), the matrix would be modified:

$$\mathbf{M}_{\text{Neumann}} = \begin{bmatrix} -1 & 1 & 0 & 0 & \cdots & 0 \\ 1 & -2 & 1 & 0 & \cdots & 0 \\ 0 & 1 & -2 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 1 & -2 & 1 \\ 0 & \cdots & 0 & 0 & 1 & -1 \end{bmatrix}$$

These boundary conditions are now called **Neumann boundary conditions**.
Physical example: A system with reflective or impermeable boundaries where no material can cross.
Matrix changes: First and last diagonal entries change from `-2` to `-1`.

Temporal Discretization: The Crank-Nicolson Scheme

For time discretization, we use the **Crank-Nicolson scheme**, which provides excellent stability and accuracy. The key idea is to evaluate the spatial terms at the average of two consecutive time steps:

$$\frac{c_i^{n+1} - c_i^n}{\Delta t} = \frac{D}{2}\left[\left.\frac{\partial^2 c}{\partial x^2}\right|^{n+1} + \left.\frac{\partial^2 c}{\partial x^2}\right|^{n}\right] \tag{4}$$

Rearranging this equation and using our matrix notation:

$$\left(\mathbf{I} - \frac{\Delta t \cdot D}{2}\mathbf{M}\right)\mathbf{C}^{n+1} = \left(\mathbf{I} + \frac{\Delta t \cdot D}{2}\mathbf{M}\right)\mathbf{C}^{n} \tag{5}$$

This gives us our solution algorithm:

$$\mathbf{A}\mathbf{C}^{n+1} = \mathbf{B}\mathbf{C}^{n} \tag{6}$$

where $\mathbf{A} = \left(\mathbf{I} - \frac{\Delta t \cdot D}{2}\mathbf{M}\right)$ and $\mathbf{B} = \left(\mathbf{I} + \frac{\Delta t \cdot D}{2}\mathbf{M}\right)$.

Numerical Solution

Now we implement our numerical scheme in Python. We need to specify both **boundary conditions** (what happens at the edges) and **initial conditions** (the starting concentration profile).

Problem Setup

We'll solve diffusion on a domain of length $L = 1$ with **Dirichlet boundary conditions**: the concentration is fixed at zero at both ends. This might represent a situation where particles are absorbed when they reach the boundaries.

For our initial condition, we'll use a Gaussian (bell-shaped) distribution centered at $x = L/2$:

$$c(x, 0) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-L/2)^2}{2\sigma^2}} \tag{7}$$

```
#| autorun: false

# Domain parameters
L = 1.0          # domain length
NX = 500         # number of spatial grid points
dx = L/(NX+1)    # spatial step size
x = np.linspace(dx, L-dx, NX)  # spatial grid (excluding boundaries)

# Time parameters
T = 0.5          # total simulation time
```

```
dt = dx**2/4     # time step (chosen for stability)
NT = int(T/dt)   # number of time steps

# Physical parameter
D = 0.1          # diffusion coefficient

# Stability analysis for Crank-Nicolson scheme
stability_number = D * dt / (dx**2)
max_stable_dt = dx**2 / (2*D)  # Conservative estimate
```

i Parameter Choices

The parameter selection in this diffusion equation implementation involves several carefully considered choices:
Spatial Parameters:

- **Domain length (L = 1.0)**: A unit domain simplifies the mathematics and provides a clear reference scale
- **Grid points (NX = 500)**: This provides high spatial resolution with $dx \approx 0.002$, ensuring accurate capture of concentration gradients
- **Grid spacing calculation**: $dx = L/(NX + 1)$ accounts for the boundary points being excluded from the interior grid

Temporal Parameters: The critical choice here is $dt = dx^2/4$. This is not arbitrary but based on stability theory:

- For explicit schemes, the stability condition is $D \cdot dt/dx^2 \leq 0.5$
- Here, $dt = dx^2/4$ ensures $D \cdot dt/dx^2 = D/4 \approx 0.025$ (with $D = 0.1$), well within the stable range
- This conservative choice prioritizes numerical stability over computational speed

Physical Parameter: The diffusion coefficient $D = 0.1$ provides a moderate diffusion rate that allows observable spreading over the simulation time $T = 0.5$.

i How to analyze stability

Understanding numerical stability is crucial for reliable simulations. The key insight is that when we discretize differential equations, we must ensure that small errors don't grow exponentially and destroy our solution.

The Stability Number

The **stability number** $r = D \cdot dt/dx^2$ is a dimensionless parameter that tells us whether our numerical scheme will behave well:

- **Physical meaning**: r represents the ratio of how far information can diffuse in one time step ($\sqrt{D \cdot dt}$) compared to our grid spacing (dx)
- **Mathematical meaning**: It measures how much the solution can change between adjacent grid points in one time step

Interpreting Stability Values

Values $r > 0.5$:

- Can cause catastrophic instability in explicit schemes (solution "blows up")
- Even in stable implicit schemes like Crank-Nicolson, accuracy degrades significantly
- **Tip**: If your simulation produces wildly oscillating or growing solutions, check if r is too large

Values $r < 0.01$:

- Very conservative - the solution will be accurate but computationally expensive
- You're taking much smaller time steps than necessary
- **Tip**: If your simulation runs very slowly, you might be able to increase dt safely

Values $0.01 \leq r \leq 0.5$:

- The "sweet spot" for most diffusion problems
- Good balance of computational efficiency and numerical accuracy
- **Tip**: Aim for this range in your simulations

Why Crank-Nicolson Still Needs Stability Analysis

While the Crank-Nicolson scheme is theoretically "unconditionally stable" (it won't blow up for any time step), accuracy still matters:

- Large time steps can introduce significant numerical errors
- The solution might be stable but completely wrong
- **Rule of thumb**: Stability analysis helps ensure both stability AND accuracy

Practical Stability Analysis Steps

1. **Calculate your stability number**: $r = D \cdot dt/dx^2$
2. **Check the range**: Is $0.01 \leq r \leq 0.5$?

3. **If r is too large**: Decrease dt or increase dx (coarsen the grid)
4. **If r is too small**: You can safely increase dt to speed up computation
5. **Always verify**: Run a test with known analytical solution to validate your parameters

In our example, choosing $dt = dx^2/4$ gives $r \approx 0.025$, which ensures both stability and accuracy while keeping computational cost reasonable.

Initial Conditions

We create our initial Gaussian concentration profile:

```
#| autorun: false

sigma = 0.04                                    # width of initial distribution
c = np.exp(-(x-L/2)**2/(2*sigma**2))           # Gaussian profile
c = c / (sigma * np.sqrt(2*np.pi))             # normalize (optional)
c = c.reshape(-1)                               # ensure 1D array
```

Let's visualize our initial condition:

```
#| autorun: false

plt.figure(figsize=get_size(12, 8))
plt.plot(x, c, 'b-', linewidth=2, label='Initial condition')
plt.xlabel('position x')
plt.ylabel('concentration c')

plt.tight_layout()
plt.show()
```

Matrix Construction

We build the second-difference matrix using SciPy's sparse matrix tools. Let's start with the main matrix for our simulation, which implements **Dirichlet boundary conditions** (fixed concentration values of zero at both boundaries).

```
#| autorun: false

# Create second-difference matrix using sparse format for efficiency
# Method 1: Using sparse.diags (more intuitive)
diagonals = [
    np.ones(NX-1),      # upper diagonal: +1 coefficients
    -2*np.ones(NX),     # main diagonal: -2 coefficients
    np.ones(NX-1)       # lower diagonal: +1 coefficients
]
M = sparse.diags(diagonals, offsets=[1, 0, -1], shape=(NX, NX)) / (dx**2)

# Alternative Method 2: Using spdiags (more compact but less clear)
# data = np.ones((3, NX))
# data[1] = -2*data[1]
# M = sparse.spdiags(data, [-1, 0, 1], NX, NX) / (dx**2)

# Identity matrix
I = sparse.identity(NX)

# Verify matrix structure (optional check)
print(f"First few rows of M*dx²:")
print((M * dx**2).toarray()[:5, :8])
```

For comparison, here's how to construct the matrix with **Neumann boundary conditions** (zero-flux at boundaries, meaning $\frac{\partial c}{\partial x} = 0$ at both ends):

```
#| autorun: false

# Create the standard matrix first (same as above)
diagonals_neumann = [
    np.ones(NX-1),      # upper diagonal: +1 coefficients
    -2*np.ones(NX),     # main diagonal: -2 coefficients
    np.ones(NX-1)       # lower diagonal: +1 coefficients
]
M = sparse.diags(diagonals_neumann, offsets=[1, 0, -1], shape=(NX, NX)) / (dx**2)

# Convert to CSR format to allow item assignment
M = M.tocsr()

# Modify boundary rows for zero flux conditions
# First boundary:  c/ x = 0 at x = 0
M[0, 0] = -1/dx**2    # Change from -2 to -1
```

```
# Last boundary:  c/ x = 0 at x = L
M[-1, -1] = -1/dx**2  # Change from -2 to -1

print(f"First few rows of M*dx²:")
print((M * dx**2).toarray()[:5, :8])
print(f"Last few rows of M*dx²:")
print((M * dx**2).toarray()[-5:, -8:])
```

Time Evolution

Now we solve the system iteratively. Let's break this down into manageable steps:

Step 1: Pre-compute the Matrices

Remember our equation $\mathbf{AC}^{n+1} = \mathbf{BC}^n$? Since the matrices A and B contain only our grid spacing, time step, and diffusion coefficient - all of which stay constant - we can calculate them once at the beginning rather than recalculating them thousands of times in our loop. This saves enormous amounts of computation time.

```
#| autorun: false

# Pre-compute matrices (they don't change during time evolution)
A = I - (dt*D/2) * M    # Left-hand side matrix
B = I + (dt*D/2) * M    # Right-hand side matrix
```

Step 2: Set Up Storage

As our simulation runs through time, we want to save snapshots of how the concentration profile looks at different moments. Think of this like taking photos of the spreading ink drop at regular intervals. We'll store these "snapshots" in a list so we can analyze and visualize the evolution later.

```
#| autorun: false

# Create storage for our results
data = []                   # List to store concentration profiles
time_indices = []           # List to track which time steps we stored
data.append(c.copy())      # Store initial condition
time_indices.append(0)     # Store initial time index
```

Step 3: Main Time Stepping Loop

This is where the real physics happens! At each time step, we take our current concentration profile and use our diffusion equation to predict what it will look like one small time step later. We repeat this process thousands of times to watch the diffusion unfold. It's like predicting the future in tiny increments - each prediction becomes the starting point for the next prediction.

The key insight is that we're solving $\mathbf{A}\mathbf{C}^{n+1} = \mathbf{B}\mathbf{C}^n$ at each step: we know $\mathbf{C}^n$ (current concentration), so we calculate $\mathbf{B}\mathbf{C}^n$ and then solve for the unknown $\mathbf{C}^{n+1}$ (future concentration).

```
#| autorun: false

# Main time stepping loop
print("Starting time evolution...")

for n in range(NT):
    # Calculate the right-hand side: B * c^n
    rhs = B @ c

    # Solve the linear system: A * c^(n+1) = rhs
    c = sparse.linalg.spsolve(A, rhs)
    c = np.array(c)  # Convert to regular numpy array

    # Store result every 10 time steps to save memory
    if n % 10 == 0 or n == NT-1:
        data.append(c.copy())
        time_indices.append(n + 1)  # +1 because we're storing the result after the step

    # Print progress occasionally
    if n % (NT//10) == 0:
        print(f"  Progress: {100*n/NT:.1f}% complete")
```

Step 4: Quick Validation Check

Even the best numerical methods can sometimes produce unphysical results due to rounding errors or inappropriate parameters. As good computational physicists, we should always check whether our results make sense from a physics perspective. Are there negative concentrations (impossible!)? Do particles disappear from the boundaries as expected? Is mass conserved (particles don't just vanish)?

```
#| autorun: false

# Check if our solution is physically reasonable
```

```
print("\n  Quick validation checks:")

# 1. No negative concentrations (unphysical)
if np.min(c) < -1e-10:
    print(f"   Warning: Found negative concentrations!")
else:
    print("  All concentrations are positive")

# 2. Boundary conditions
if abs(c[0]) < 1e-10 and abs(c[-1]) < 1e-10:
    print("  Boundary conditions satisfied (zero at edges)")
else:
    print(f"   Boundary values: left = {c[0]:.2e}, right = {c[-1]:.2e}")

# 3. Mass conservation
initial_mass = np.trapz(data[0], x)
final_mass = np.trapz(data[-1], x)
mass_change = abs(final_mass - initial_mass) / initial_mass
print(f"  Mass conservation error: {mass_change:.2e}")
```

Visualization and Analysis

Evolution of Concentration Profiles

Let's visualize how the concentration profile evolves over time:

```
#| autorun: false
#| fig-align: center

plt.figure(figsize=get_size(12, 8))

# Plot profiles at available time intervals
plot_indices = np.linspace(0, len(data)-1, min(10, len(data)), dtype=int)
colors = plt.cm.viridis(np.linspace(0, 1, len(plot_indices)))

for i, data_idx in enumerate(plot_indices):
    time_step = time_indices[data_idx]
    time_value = time_step * dt
    plt.plot(x, data[data_idx], color=colors[i],
             label=f't = {time_value:.3f}', linewidth=2)
```

```
plt.xlabel('position x')
plt.ylabel('concentration c')

plt.tight_layout()
plt.show()
```

Notice how the sharp initial peak gradually spreads out and flattens over time, while the total area under the curve remains constant (conservation of mass).

Space-Time Evolution

A particularly insightful visualization is the **space-time plot**, which shows the entire evolution as a 2D image:

```
#| autorun: false
#| fig-align: center

plt.figure(figsize=get_size(12, 8))

# Create 2D array from our stored data
concentration_matrix = np.array(data)
stored_times = np.array([t * dt for t in time_indices])

# Create coordinate grids for contour plot
X, Y = np.meshgrid(x, stored_times)

# Create the space-time plot using filled contours
contour_plot = plt.contourf(X, Y, concentration_matrix,
                            levels=20,
                            cmap='hot')

plt.colorbar(contour_plot, label='Concentration c')
plt.xlabel('position x')
plt.ylabel('time t')

plt.tight_layout()
plt.show()
```

Quantitative Analysis

Let's analyze some key properties of our solution:

```
#| autorun: false

# Calculate the width of the distribution over time
def calculate_width(concentration_profile, x_grid):
    """Calculate the standard deviation (width) of the concentration profile"""
    total_mass = np.trapz(concentration_profile, x_grid)
    if total_mass > 1e-10:  # avoid division by zero
        mean_x = np.trapz(x_grid * concentration_profile, x_grid) / total_mass
        variance = np.trapz((x_grid - mean_x)**2 * concentration_profile, x_grid) / total_mas
        return np.sqrt(variance)
    return 0

# Calculate width at each stored time step
widths = []
times = []
analysis_step = max(1, len(data)//20)  # Analyze ~20 points for performance
for i in range(0, len(data), analysis_step):
    width = calculate_width(data[i], x)
    widths.append(width)
    times.append(time_indices[i] * dt)

plt.figure(figsize=get_size(12, 8))
plt.plot(times, widths, 'bo-', markersize=4)

# Theoretical prediction: width ~ sqrt(2*D*t)
times_theory = np.array(times)
widths_theory = np.sqrt(2 * D * times_theory + sigma**2)  # include initial width
plt.plot(times_theory, widths_theory, 'r--',
         label=r'theory: $\sqrt{2Dt + \sigma_0^2}$')

plt.xlabel('time t')
plt.ylabel('distribution width  ')

plt.tight_layout()
plt.show()
```

This analysis confirms that our numerical solution matches the theoretical prediction: the width of a diffusing distribution grows as $\sqrt{2Dt}$.

Extensions and Explorations

Different Boundary Conditions

Try modifying the boundary conditions to see their effects:

Neumann boundaries (zero flux at boundaries):

```
# Modify the matrix M to implement zero-flux conditions
M[0, 0] = -1/dx**2    # First row
M[-1, -1] = -1/dx**2  # Last row
```

Periodic boundaries (what goes out one side comes in the other):

```
# Add wraparound connections
M[0, -1] = 1/dx**2
M[-1, 0] = 1/dx**2
```

Parameter Exploration

Investigate how different parameters affect the solution:

1. **Diffusion coefficient D**: Try values from 0.01 to 1.0
2. **Initial width** : Compare narrow vs. wide initial distributions
3. **Grid resolution**: See how dx affects accuracy
4. **Time step**: Explore the stability limit

Physical Applications

This same mathematical framework applies to:

- **Heat conduction**: Replace concentration with temperature, D becomes thermal diffusivity
- **Quantum mechanics**: The time-dependent Schrödinger equation has the same mathematical structure
- **Finance**: Black-Scholes equation for option pricing
- **Biology**: Population dynamics and chemotaxis

Where to Go From Here

The diffusion equation is your gateway to the vast world of partial differential equations. Building on what you've learned here, you could explore:

1. **Two-dimensional diffusion**: Extend to diffusion in a plane
2. **Nonlinear diffusion**: When D depends on concentration
3. **Reaction-diffusion systems**: Combine diffusion with chemical reactions
4. **Wave equations**: Replace first time derivative with second
5. **Advanced numerical methods**: Finite element methods, spectral methods

The techniques you've mastered - discretization, matrix methods, and iterative solution - form the foundation for computational physics and engineering. These tools will serve you well as you tackle increasingly complex physical phenomena.