8 Conservation laws and heat flow

A conservation law is a balance law, and corresponds to an equation that describes how a quantity is balanced in some system throughout a given process. (Consider how this is related to conservation laws in physics.) For example, suppose we are keeping track of some measurable quantity in a physical system (e.g. heat, water, etc). The fundamental conservation law then states that the rate of change of the total quantity in the system is equal to the rate of the quantity flowing into the system plus the rate at which the quantity is produced by sources inside the system.

Derivation of the Conservation equation in multiple dimensions

Suppose Ω is a region in $\mathbb{R}^n$, and $V \subset \Omega$ is bounded with a reasonably well-behaved boundary ∂V. Let $u(\vec{x}, t)$ represent the density (concentration) of some quantity throughout Ω. Let $\vec{n}(x)$ represent the normal direction to V at $x \in \partial V$, and let $\vec{J}(\vec{x}, t)$ be the flux vector for the quantity, so that $\vec{J}(\vec{x}, t) \cdot \vec{n}(x)\, dA$ represents the rate at which the quantity leaves V by crossing a boundary element with area dA. Note that the total amount of the quantity in V is

$$\int_V u(\vec{x}, t)\, dt,$$

and the rate at which the quantity enters V is

$$-\int_{\partial V} \vec{J}(\vec{x}, t) \cdot \vec{n}(x)\, dA.$$

We let the source term be given by $g(\vec{x}, t, u)$; we may interpret this to mean that the rate at which the quantity is produced in V is

$$\int_V g(\vec{x}, t, u)\, dt.$$

Then the integral form of the conservation law for u is expressed as

$$\frac{d}{dt}\int_V u(\vec{x}, t)\, d\vec{x} = -\int_{\partial V} \vec{J} \cdot \vec{n}\, dA + \int_V g(\vec{x}, t, u)\, d\vec{x}.$$

If u and J are sufficiently smooth functions, then we have

$$\frac{d}{dt}\int_V u\, d\vec{x} = \int_V u_t\, d\vec{x},$$

and

$$\int_{\partial V} \vec{J} \cdot \vec{n}\, dA = \int_V \nabla \cdot \vec{J}\, d\vec{x}.$$

Since this holds for all nice subsets $V \subset \Omega$ with V arbitrarily small, we obtain the differential form of the conservation law for u:

$$u_t + \nabla \cdot \vec{J} = g(\vec{x}, t, u),$$

where ∇ is the gradient function and $\nabla \cdot \vec{J} = \frac{\partial J_1}{\partial x_1} + \cdots + \frac{\partial J_n}{\partial x_n}$

Constitutive Relations

Currently our conservation law appears in the form

$$u_t + \nabla \cdot \vec{J} = g(\vec{x}, t, u).$$

Thus the conservation law consists of one equation and 2 unknowns (u and J). To this equation we add other equations, called constitutive relations, which are used to fully determine the system.

For example, suppose we wish to describe the flow of heat. Since heat flows from warmer regions to colder regions, and the rate of heat flow depends on the difference in temperature between regions, we usually assume that the flux vector $\vec{J}$ is given by

$$\vec{J}(x,t) = -\nu \nabla u(x,t),$$

where ν is a diffusion constant and $\nabla u(x,t) = [\partial_{x_1} u \dots \partial_{x_n} u]^T$. This constitutive relation is called Fick's law, and is the basic model for any diffusive process. Substituting into the conservation law we obtain

$$u_t - \nu \Delta u(x,t) = g(\vec{x}, t, u)$$

where Δ is the Laplacian operator, and $\Delta u(x,t) = \frac{\partial^2 u}{\partial x_1^2} + \cdots + \frac{\partial^2 u}{\partial x_n^2}$. The function g represents heat sources/sinks within the region.

Numerically modeling heat flow

Consider the heat flow equation in one dimension together with appropriate initial conditions and homogeneous Dirichlet boundary conditions:

$$\begin{aligned} &u_t = \nu u_{xx}, \quad x \in [a,b], \quad t \in [0,T], \\ &u(a,t) = 0, \quad u(b,t) = 0, \\ &u(x,0) = f(x). \end{aligned}$$

We will look for an approximation U_i^j to $u(x_i, t_j)$ on the grid $x_i = a + hi$, $t_j = kj$, where h and k are small changes in x and t respectively and i and j are indices. Note that the index i ranges over different spacial grid points and the index j ranges over different time steps. We will denote the approximate value of u at the i'th grid point and the j'th time step as U_i^j.

A common method for modeling ordinary and partial differential equations is the finite difference method, so-named because equations containing derivatives are replaced with equations containing difference schemes. These difference schemes can often be found using Taylor's theorem. For example, the equation

$$u(x, t_j + k) = u(x, t_j) + u_t(x, t_j)k + \mathcal{O}(k^2)$$

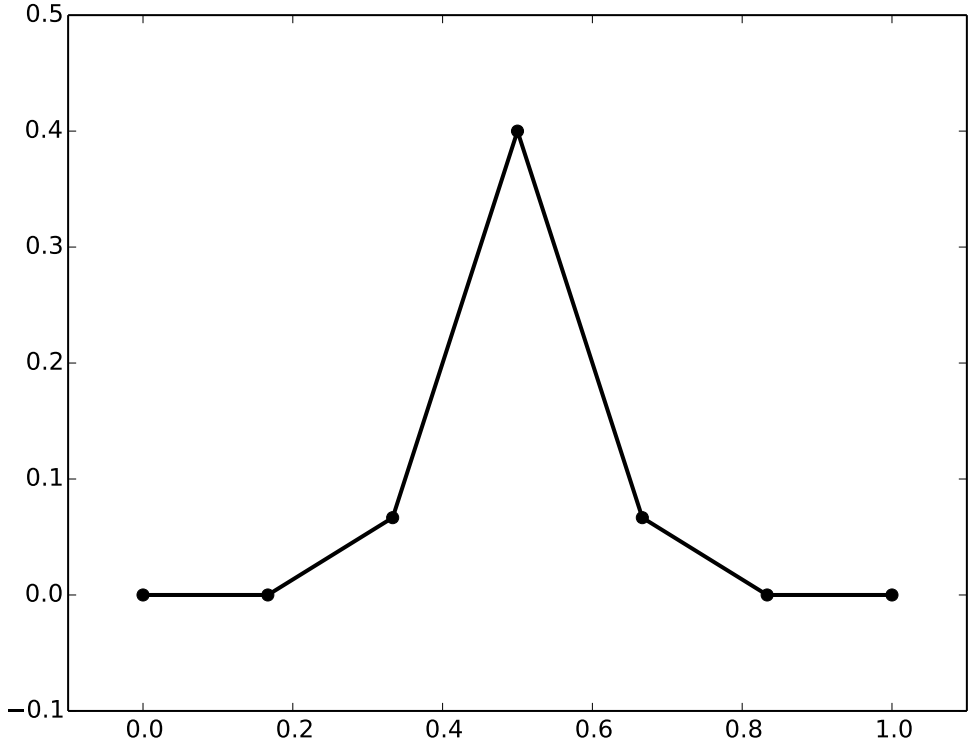

Figure 8.1: The graph of U^0, the approximation to the solution $u(x, t = 0)$ for Problem 1.

yields a first-order forward difference approximation to $u_t(x, t_j)$, namely,

$$u_t(x, t_j) = \frac{u(x, t_j + k) - u(x, t_j)}{k} + \mathcal{O}(k).$$

Similarly, by adding the equations

$$u(x_i + h, t) = u(x_i, t) + u_x(x_i, t)h + u_{xx}(x_i, t)\frac{h^2}{2} + u_{xxx}(x_i, t)h^3 + \mathcal{O}(h^4),$$
$$u(x_i - h, t) = u(x_i, t) + u_x(x_i, t)(-h) + u_{xx}(x_i, t)\frac{(-h)^2}{2} + u_{xxx}(x_i, t)(-h)^3 + \mathcal{O}(h^4),$$

we obtain a second-order centered difference approximation to $u_{xx}(x_i, t)$:

$$u_{xx}(x_i, t_j) = \frac{u(x_i + h, t_j) - 2u(x_i, t_j) - u(x_i - h, t_j)}{h^2} + \mathcal{O}(h^2).$$

These difference approximations give us the $\mathcal{O}(h^2 + k)$ explicit method

$$\begin{aligned} \frac{U_i^{j+1} - U_i^j}{k} &= \nu \frac{U_{i+1}^j - 2U_i^j + U_{i-1}^j}{h^2}, \\ U_i^{j+1} &= U_i^j + \frac{\nu k}{h^2}(U_{i+1}^j - 2U_i^j + U_{i-1}^j). \end{aligned} \tag{8.1}$$

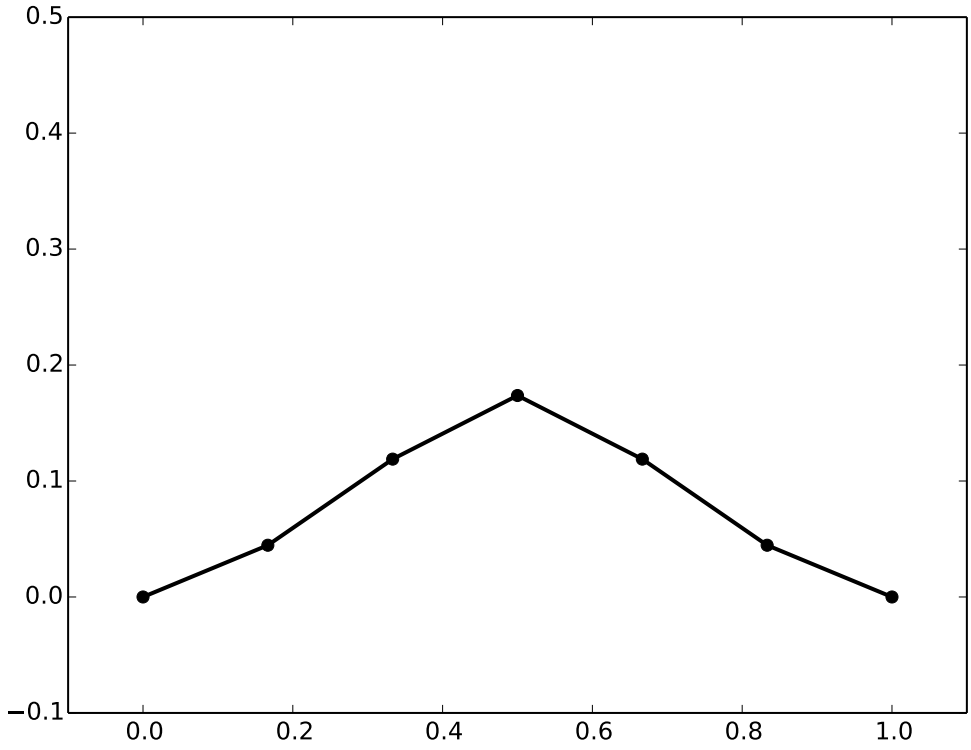

Figure 8.2: The graph of U^4, the approximation to the solution $u(x, t = .4)$ for Problem 1.

This method can be written in matrix form as

$$U^{j+1} = AU^j,$$

where A is the tridiagonal matrix given by

$$A = \begin{bmatrix} 1-2\lambda & \lambda & & & & \\ \lambda & 1-2\lambda & \lambda & & & \\ & \ddots & \ddots & \ddots & & \\ & & \lambda & 1-2\lambda & \lambda & \\ & & & \lambda & 1-2\lambda & \end{bmatrix},$$

$\lambda = \nu k/h^2$, and U^j represents the approximation at time t_j. We can get this method started by using the initial condition given in our problem, so that $U_i^0 = f(x_i)$.

Note

Finite difference schemes, though they can be *represented* using matrix multiplication, should not be *implemented* using raw matrix multiplication. Using NumPy, it is best to vectorize the

difference scheme so that you do not have to loop over the spatial indices. If you are using a language with faster loops (like C, C++, Fortran, or Cython), it could work well to loop directly through the indices in both time and space.

To account for boundary conditions using this differencing scheme, simply set the boundary points to the appropriate values in the initial conditions, then avoid modifying them as you update for each time step. This would be the equivalent of replacing the first and last rows of the matrix representation of the differencing scheme with the first and last rows of the identity matrix.

Problem 1. Consider the specific initial boundary value problem

$$\begin{aligned} &u_t = .05u_{xx}, \quad x \in [0,1], \quad t \in [0,1] \\ &u(0,t) = 0, \quad u(1,t) = 0, \\ &u(x,0) = 2\max\{.2 - |x - .5|, 0\}. \end{aligned} \tag{8.2}$$

Approximate the solution $u(x,t)$ at time $t = .4$ by taking 6 subintervals in the x dimension and 10 subintervals in time. The graphs for U^0 and U^4 are given in Figures 8.1 and 8.2.

For the next problem, we need to show how Matplotlib can be used to create a 2D animation. The following is a simple working example that animates a sine wave.

```
import numpy as np
from matplotlib import animation, pyplot as plt

def sine_animation(res=100):
    # Make the x and y data.
    x = np.linspace(-1, 1, res+1)[:-1]
    y = np.sin(np.pi * x)
    # Initialize a matplotlib figure.
    f = plt.figure()
    # Set the x and y axes by constructing an axes object.
    plt.axes(xlim=(-1,1), ylim=(-1,1))
    # Plot an empty line to use in the animation.
    # Notice that we are unpacking a tuple of length 1.
    line, = plt.plot([], [])
    # Define an animation function that will update the line to
    # reflect the desired data for the i'th frame.
    def animate(i):
        # Set the data for updated version of the line.
        line.set_data(x, np.roll(y, i))
        # Notice that this returns a tuple of length 1.
        return line,
    # Create the animation object.
    # 'frames' is the number of frames before the animation should repeat.
    # 'interval' is the amount of time to wait before updating the plot.
    # Be sure to assign the animation a name so that Python does not
    # immediately garbage collect (delete) the object.
```

```
    a = animation.FuncAnimation(f, animate, frames=y.size, interval=20)
    # Show the animation.
    plt.show()

# Run the animation function we just defined.
sine_animation()
```

Problem 2. Solve the specific initial boundary value problem

$$\begin{aligned} &u_t = u_{xx}, \quad x \in [-12, 12], \quad t \in [0, 1], \\ &u(-12, t) = 0, \quad u(12, t) = 0, \\ &u(x, 0) = \max\{1 - x^2, 0\} \end{aligned} \tag{8.3}$$

using the first order explicit method 8.1. Use 140 subintervals in the x dimension and 70 subintervals in time. The initial and final states are shown in Figure 8.3. Animate your results.

Explicit methods usually have a stability condition, called a CFL condition (for Courant-Friedrichs-Lewy). For method 8.1 the CFL condition that must be satisfied is that

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

Repeat your computations using 140 subintervals in the x dimension and 66 subintervals in time. Animate the results. For these values the CFL condition is broken; you should easily see the result of this instability in the approximation U^{66}.

Implicit methods often have better stability properties than explicit methods. The Crank-Nicolson method, for example, is unconditionally stable and has order $\mathcal{O}(h^2 + k^2)$. To derive the Crank-Nicolson method, we use the following approximations:

$$\begin{aligned} u_t(x_i, t_{j+1/2}) &= \frac{u(x_i, t_{j+1}) - u(x_i, t_j)}{k} + \mathcal{O}(k^2), \\ u_{xx}(x_i, t_{j+1/2}) &= \frac{u_{xx}(x_i, t_{j+1}) + u_{xx}(x_i, t_j)}{2} + \mathcal{O}(k^2). \end{aligned}$$

The first equation is a Finite Difference approximation, and the second is a midpoint approximation. These approximations give the method

$$\begin{aligned} \frac{U_i^{j+1} - U_i^j}{k} &= \frac{1}{2}\left(\frac{U_{i+1}^j - 2U_i^j + U_{i-1}^j}{h^2} + \frac{U_{i+1}^{j+1} - 2U_i^{j+1} + U_{i-1}^{j+1}}{h^2}\right), \\ U_i^{j+1} &= U_i^j + \frac{k}{2h^2}\left(U_{i+1}^j - 2U_i^j + U_{i-1}^j + U_{i+1}^{j+1} - 2U_i^{j+1} + U_{i-1}^{j+1}\right). \end{aligned} \tag{8.4}$$

This method can be written in matrix form as

$$BU^{j+1} = AU^j,$$

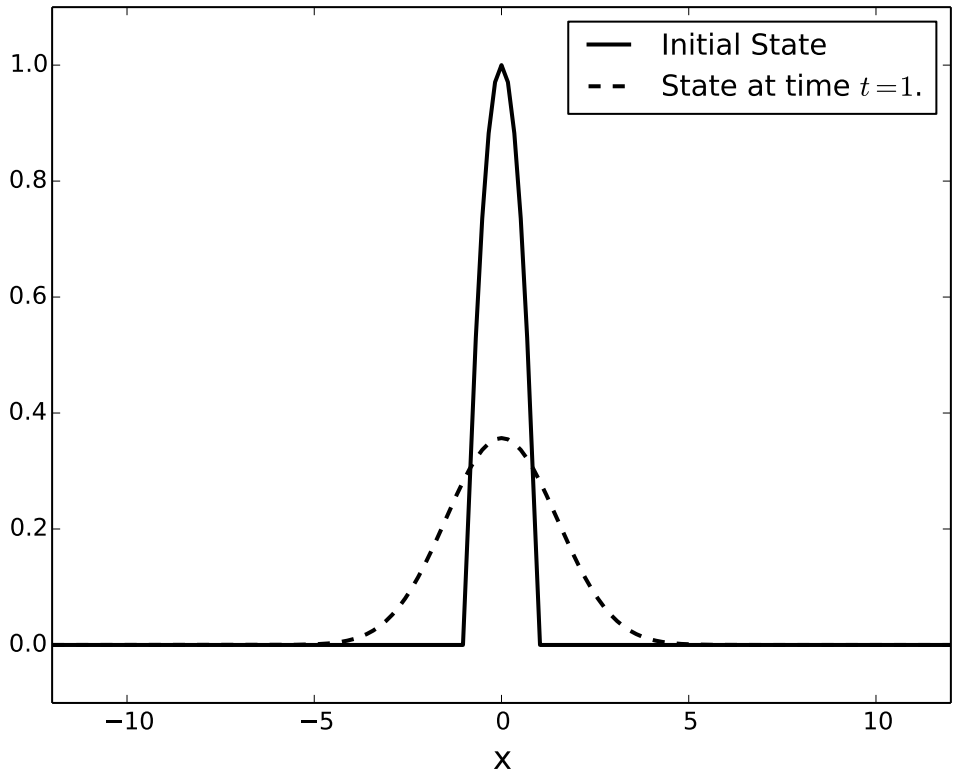

Figure 8.3: The initial and final states for equation Problem 2.

where A and B are tridiagonal matrices given by

$$B = \begin{bmatrix} 1+2\lambda & -\lambda & & & \\ -\lambda & 1+2\lambda & -\lambda & & \\ & \ddots & \ddots & \ddots & \\ & & -\lambda & 1+2\lambda & -\lambda \\ & & & -\lambda & 1+2\lambda \end{bmatrix},$$

$$A = \begin{bmatrix} 1-2\lambda & \lambda & & & \\ \lambda & 1-2\lambda & \lambda & & \\ & \ddots & \ddots & \ddots & \\ & & \lambda & 1-2\lambda & \lambda \\ & & & \lambda & 1-2\lambda \end{bmatrix},$$

where $\lambda = \nu k/(2h^2)$, and U^j represents the approximation at time t_j. Note that here we have defined λ differently than we did before!

How do we know if a numerical approximation is reasonable? One way to determine this is to compute solutions for various step sizes h and see if the solutions are converging to something. To be more specific, suppose our finite difference method is $\mathcal{O}(h^p)$ accurate. This means that the error $E(h) \approx Ch^p$ for some constant C as $h \to 0$ (i.e., for $h > 0$ small enough).

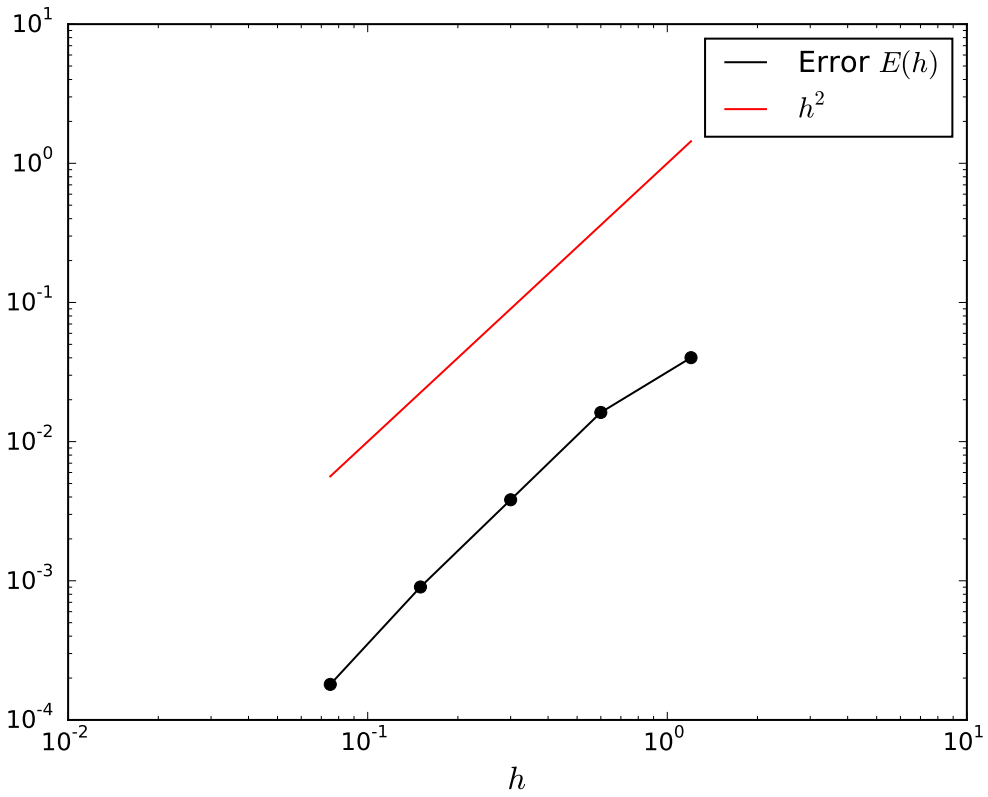

Figure 8.4: $E(h)$ represents the (approximate) maximum error in the numerical solution U to Problem 3 at time $t = 1$, using a stepsize of h.

So compute the approximation y_k for each stepsize h_k, $h_1 > h_2 > \ldots > h_m$. We will think of y_m as the true solution. Then the error of the approximation for stepsize $h_k, k < m$, is

$$E(h_k) = \max(|y_k - y_m|) \approx Ch_k^p,$$
$$\log(E(h_k)) = \log(C) + p\log(h_k).$$

Thus on a log-log plot of $E(h)$ vs. h, these values should be on a straight line with slope p when h is small enough to start getting convergence.

Problem 3. Using the Crank Nicolson method, numerically approximate the solution $u(x,t)$ of the problem

$$\begin{aligned} &u_t = u_{xx}, \quad x \in [-12, 12], \quad t \in [0, 1], \\ &u(-12, t) = 0, \quad u(12, t) = 0, \\ &u(x, 0) = \max\{1 - x^2, 0\}. \end{aligned} \tag{8.5}$$

Demonstrate that the numerical approximation at $t = 1$ converges to $u(x, t = 1)$. Do this by computing U at $t = 1$ using $20, 40, 80, 160, 320$, and 640 steps. Use the same number of steps

in both time and space. Reproduce the loglog plot shown in Figure 8.4. The slope of the line there shows the proper rate of convergence.

To measure the error, use the solution with the smallest h (largest number of intervals) as if it were the exact solution, then sample each solution only at the x-values that are represented in the solution with the largest h (smallest number of intervals). Use the ∞-norm on the arrays of values at those points to measure the error.

Notice that, since the Crank-Nicolson method is unconditionally stable, there is no CFL condition and we can use the same number of intervals in time and space.