Computational Fluid Dynamics

Aero 523 Course Notes: Fall 2019

Krzysztof Fidkowski
University of Michigan, Ann Arbor, MI, 48109, USA

September 1, 2019

Contents

Chapter 1

Mathematics and Programming Review

1.1 Notation

In this text we will use the following notation:

c : (regular font) a **scalar**

$\vec{v}$: (arrow) a **spatial vector**, e.g.

$$\vec{v} = v_1\hat{x}_1 + v_2\hat{x}_2 + v_3\hat{x}_3,$$

where $\hat{x}_i$ is the unit vector in the i^{th} Cartesian direction.

$\mathbf{u}$: (bold) a **state vector**, e.g.

$$\mathbf{u} = [\rho, \rho u, \rho v, \rho E]^T.$$

$\vec{\mathbf{F}}$: (bold and arrow) a **combined spatial and state vector**, e.g.

$$\vec{\mathbf{F}} = \mathbf{F}_1\hat{x}_1 + \mathbf{F}_2\hat{x}_2 + \mathbf{F}_3\hat{x}_3.$$

$\mathbf{A}$: (bold, typically uppercase) a **matrix**, e.g.

$$\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}.$$

We will use index notation, in which v_i denotes the i^{th} component of a vector $\mathbf{v}$ (spatial or state). A repeated index in an expression implies summation, e.g.

$$v_i w_i = \sum_i v_i w_i = \mathbf{v} \cdot \mathbf{w}.$$

∂_i is used to denote differentiation with respect to x_i. So the **gradient** of a scalar field p is the vector

$$\nabla p \;=\; \frac{\partial p}{\partial x_1}\hat{x}_1 + \frac{\partial p}{\partial x_2}\hat{x}_2 + \frac{\partial p}{\partial x_3}\hat{x}_3 = \partial_i p\, \hat{x}_i$$

The **divergence** of a vector field $\vec{v}$ is the scalar $\nabla \cdot \vec{v} = \partial_i v_i$.

When a rank n vector $\mathbf{f}$ is a function of a rank m vector $\mathbf{u}$, the **Jacobian** of $\mathbf{f}$ with respect to $\mathbf{u}$ is an $n \times m$ matrix of partial derivatives,

$$\frac{\partial \mathbf{f}}{\partial \mathbf{u}} = \begin{bmatrix} \frac{\partial f_1}{\partial u_1} & \frac{\partial f_1}{\partial u_2} & \dots & \frac{\partial f_1}{\partial u_m} \\ \frac{\partial f_2}{\partial u_1} & \frac{\partial f_2}{\partial u_2} & \dots & \frac{\partial f_2}{\partial u_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial u_1} & \frac{\partial f_n}{\partial u_2} & \dots & \frac{\partial f_n}{\partial u_m} \end{bmatrix}. \tag{1.1.1}$$

1.2 Linear Algebra

The superscript T following a vector or matrix denotes the **transpose** operation, which means flipping rows into columns and vice-versa. Hence, the dot/inner product between two column vectors $\mathbf{u}, \mathbf{v}$ of rank n, is

$$\mathbf{u} \cdot \mathbf{v} = \mathbf{u}^T \mathbf{v} = \mathbf{v}^T \mathbf{u} = \sum_{i=1}^{n} u_i v_i = u_i v_i = \text{ a scalar.} \tag{1.2.1}$$

Two vectors are orthogonal if their dot product is zero. We can multiply a column vector $\mathbf{u} \in \mathbb{R}^m$ from the left by a matrix $\mathbf{A} \in \mathbb{R}^{n \times m}$. This operation produces a vector $\mathbf{v} = \mathbf{A}\mathbf{u} \in \mathbb{R}^n$, the entries of which are dot products between $\mathbf{u}$ and the rows of $\mathbf{A}$,

$$\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1m} \\ A_{21} & A_{22} & \dots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \dots & A_{nm} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_m \end{bmatrix}.$$

Note that such a matrix-vector product requires m multiplications and additions per entry of $\mathbf{v}$, for a total of $\mathcal{O}(mn)$ operations[1]. In a similar way we can multiply matrices, say $\mathbf{A} \in \mathbb{R}^{n \times m}$ and $\mathbf{B} \in \mathbb{R}^{m \times p}$ (note the inner dimension m has to match), to obtain a matrix $\mathbf{C} = \mathbf{A}\mathbf{B} \in \mathbb{R}^{n \times p}$ with entries $C_{ik} = A_{ij} B_{jk}$,

$$\begin{bmatrix} C_{11} & C_{12} & \dots & C_{1p} \\ C_{21} & C_{22} & \dots & C_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ C_{n1} & C_{n2} & \dots & C_{np} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1m} \\ A_{21} & A_{22} & \dots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \dots & A_{nm} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} & \dots & B_{1p} \\ B_{21} & B_{22} & \dots & B_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ B_{m1} & B_{m2} & \dots & B_{mp} \end{bmatrix}.$$

[1] In an order of magnitude estimate of operations we treat multiplications and additions similarly and ignore leading multiplicative factors.

Note that a matrix-matrix product requires p matrix-vector multiplications, so the number of operations is $\mathcal{O}(mnp)$. Multiplying a column vector $\mathbf{v} \in \mathbb{R}^n$ on the right by a row vector $\mathbf{u} \in \mathbb{R}^m$ produces (via an outer product) a matrix $\mathbf{vu} \in \mathbb{R}^{n \times m}$.

The **determinant** of a square matrix, $\det(\mathbf{A})$, is a scalar computed from a systematic combination of products of the matrix entries. For example, for a 2×2 matrix,

$$\det\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}\right) = ad - bc.$$

For larger matrices, the determinant can be computed by combining determinants of 2×2 submatrices (minors). Properties of the determinant include:

1. $\det(\mathbf{I}) = 1$, where $\mathbf{I}$ is the identity matrix
2. $\det(\mathbf{AB}) = \det(\mathbf{A})\det(\mathbf{B})$
3. $\det(\mathbf{A}^T) = \det(\mathbf{A})$
4. $\det(\mathbf{A}^{-1}) = 1/\det(\mathbf{A})$

The **inverse** of a square matrix, $\mathbf{A}^{-1}$, if it exists, is a matrix that multiplies $\mathbf{A}$ to give the identity matrix, $\mathbf{AA}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$. A **singular** matrix is one for which the inverse does not exist, and the determinant in this case is zero. We will sometimes use the colon (:) notation to denote parts of matrices, e.g. $\mathbf{A}(i,:)$ denotes the i^{th} row of matrix $\mathbf{A}$.

The **rank** of a matrix, $\text{rank}(\mathbf{A})$ is the number of linearly-independent columns, which is the same as the number of linearly-independent rows. An $n \times n$ matrix must have *full rank* of n to be invertible. The **null space** of $\mathbf{A}$ is the set of all vectors $\mathbf{u}$ for which $\mathbf{Au} = \mathbf{0}$. The dimension of the null space is $(n - \text{rank}(\mathbf{A}))$, so a full-rank matrix has an empty null space.

An **eigenvector** $\mathbf{v} \in \mathbb{R}^n$ of an $n \times n$ matrix $\mathbf{A}$ is a nonzero vector that satisfies

$$\mathbf{Av} = \lambda\mathbf{v}, \tag{1.2.2}$$

for a scalar λ, which is the associated **eigenvalue**. Rearranging this equation shows that $\mathbf{v}$ must live in the null space of $(\mathbf{A} - \lambda\mathbf{I})$, so that λ must satisfy

$$\det(\mathbf{A} - \lambda\mathbf{I}) = 0.$$

λ could be zero, in which case the corresponding eigenvector lies in the null space of $\mathbf{A}$, and the matrix is singular. $\mathbf{v}$ scaled by a nonzero constant is still an eigenvector, the same one for our purposes. Multiple eigenvectors can have the same eigenvalue, in which case any linear combination of those eigenvectors is also an eigenvector.

The eigenvalues of a **positive definite** matrix are all real and positive. If the matrix is also symmetric, it is called **symmetric positive definite (SPD)** and has a complete set of orthogonal eigenvectors.

$\mathbf{v}$ defined in Equation 1.2.2 is a right eigenvector, whereas left eigenvectors satisfy $\mathbf{v}^T\mathbf{A} = \lambda\mathbf{v}^T$, so they are right eigenvectors of $\mathbf{A}^T$. A matrix that has a complete set of eigenvectors can be decomposed as

$$\mathbf{A} = \mathbf{R\Lambda L} = \mathbf{R\Lambda R}^{-1} = \mathbf{L}^{-1}\mathbf{\Lambda L}, \tag{1.2.3}$$

where the columns of $\mathbf{R}$ are the right eigenvectors, the rows of $\mathbf{L}$ are the left eigenvectors, and $\mathbf{\Lambda} = \text{diag}(\lambda_1, \ldots, \lambda_N)$ is a diagonal matrix of eigenvalues on the main diagonal.

The **singular value decomposition (SVD)** of a square matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ is

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T, \tag{1.2.4}$$

where $\mathbf{U}$ and $\mathbf{V}$ are **unitary** matrices (transpose is inverse) and $\mathbf{\Sigma} = \text{diag}[\sigma_1, \sigma_2, \ldots, \sigma_n]$ is a diagonal matrix of **singular values** σ_j. These are the square roots of the eigenvalues of $\mathbf{A}^T\mathbf{A}$. A singular matrix has at least one zero singular value. Note, the SVD extends to rectangular matrices, in which case $\mathbf{\Sigma}$ is padded with zero rows/columns and $\mathbf{U}$ or $\mathbf{V}$ are modified appropriately.

1.3 Norms and Function Spaces

Norms are scalar numbers that quantify sizes of vectors and functions. Norms can be induced by inner products, resulting in **Hilbert spaces**. If $(\mathbf{u}, \mathbf{v})$ denotes the inner product between $\mathbf{u}$ and $\mathbf{v}$, then the induced norm is $\|\mathbf{u}\| = \sqrt{(\mathbf{u}, \mathbf{u})}$. Norms can also be defined without inner products, and any (complete) linear space with a norm is called a **Banach space**.

Some useful norms for discrete vectors $\mathbf{u} \in \mathbb{R}^n$ or functions $v(\vec{x}) \in \mathbb{R}, \vec{x} \in \Omega$ are

1. L_1 norm:
$$\|\mathbf{u}\|_{L_1} = \frac{1}{n}\sum_{i=1}^{n} |u_i|, \qquad \|v\|_{L_1} = \frac{1}{|\Omega|}\int_\Omega |v(\vec{x})| d\Omega$$

2. L_2 norm:
$$\|\mathbf{u}\|_{L_2} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} u_i^2}, \qquad \|v\|_{L_2} = \sqrt{\frac{1}{|\Omega|}\int_\Omega v(\vec{x})^2 d\Omega}$$

3. L_p norm, $p > 1$:
$$\|\mathbf{u}\|_{L_p} = \left(\frac{1}{n}\sum_{i=1}^{n} u_i^p\right)^{1/p}, \qquad \|v\|_{L_p} = \left(\frac{1}{|\Omega|}\int_\Omega v(\vec{x})^p d\Omega\right)^{1/p}$$

4. L_∞ norm:
$$\|\mathbf{u}\|_{L_\infty} = \max |u_i|, \qquad \|v\|_{L_\infty} = \text{ess} \sup_{\mathbf{x} \in \Omega} |v(\vec{x})|$$

The above integrals are **Lebesque integrals**, which means that they are insensitive to changes in v on a set of zero measure (e.g. at a point or on a line in a 2D domain). Note that "ess sup" is the **essential supremum**, which is similar to "max" but ignores regions of Ω which are of zero measure.

Associated with the function norms defined above are Lebesque spaces, L^p. A function $v(\vec{x})$ lies in the space L^p if it has a bounded L_p norm. Another useful space and norm for

functions is H^1, the space of functions with square integrable values and first derivatives (gradients). The H^1 norm of a function $v(\vec{x})$ is

$$\|v\|_{H^1}^2 = \frac{1}{|\Omega|}\left[\int_\Omega v(\vec{x})^2\, d\Omega + \int_\Omega \nabla v \cdot \nabla v\, d\Omega\right].$$

This is a stricter requirement than L^2, which just requires square-integrable values.

We define the space $C^m(\Omega)$ as the set of functions that have continuous derivatives up to order at least m. For example, $C^0(\Omega)$ is the space of continuous functions on Ω (no jumps), $C^1(\Omega)$ is the space of functions with continuous first derivatives, etc.

A given vector norm $\|\cdot\|$ induces a **matrix norm** of the form

$$\|\mathbf{A}\| = \max_{\mathbf{v}\neq\mathbf{0}} \frac{\|\mathbf{A}\mathbf{v}\|}{\|\mathbf{v}\|}. \tag{1.3.1}$$

For example, the L_1 norm induces the matrix norm $\|\mathbf{A}\|_{L_1} = \max_j \sum_i |A_{ij}|$, which is the maximum absolute-value column sum. The L_∞ norm induces $\|\mathbf{A}\|_{L_\infty} = \max_i \sum_j |A_{ij}|$, the maximum absolute-value row sum. The **Frobenius norm** of matrix $\mathbf{A}$ is $\|\mathbf{A}\|_F = \sqrt{A_{ij}A_{ij}}$ (note the summation on the repeated indices i, j).

For a given norm, the **condition number** is defined as $\kappa(\mathbf{A}) = \|\mathbf{A}^{-1}\|\,\|\mathbf{A}\|$. When solving a linear system $\mathbf{Ax} = \mathbf{b}$, the condition number measures the relative change in $\mathbf{x}$ for a given change in $\mathbf{b}$ (or vice versa). For the L_2 norm, $\kappa(\mathbf{A})$ is the ratio of the largest to smallest singular value. Very high condition numbers are undesirable because they mean high sensitivity to data, and hence inaccurate inverses. The condition number of a singular matrix is infinite.

1.4 The Fourier Transform

The Fourier transform is a technique for converting between spatial/temporal and frequency representations of a function. When the entire function is known, this can be done at a continuous level by integrating the function weighted by sinusoids. When the function is only known at points, as is often the case in numerical representations, the discrete Fourier transform (DFT) is used.

Suppose that on a domain $x \in [0, L)$ we have $N + 1$ samples u_j of a periodic function $u(x)$ at discrete points x_j, where $j = 0, 1, \ldots, N$, N is even, and $x_j = jh$, with $h = L/N$. Note that since $u(x)$ is periodic, $u_N = u_0$. We can represent $u(x)$ using a sum of sinusoidal functions (complex exponentials) of varying frequencies:

$$u(x) = \sum_{k=-N/2}^{N/2-1} f_k e^{2\pi i k x/L}, \tag{1.4.1}$$

where f_k is the amplitude of the k^{th} frequency and $i = \sqrt{-1}$. The chosen form of the exponent, with the factor of 2π, ensures that we are considering functions that are periodic

on the given domain, $0 \leq x < L$. Including both negative and positive frequencies in the range eliminates an aliasing problem with high frequencies. To determine the amplitudes f_k, we require that the representation in Equation 1.4.1 exactly interpolates the state at the nodes x_j,

$$u_j = u(x_j) = \sum_{k=-N/2}^{N/2-1} f_k e^{2\pi i k x_j/L} = \sum_{k=-N/2}^{N/2-1} f_k e^{2\pi i k j/N}, \tag{1.4.2}$$

where we used the fact that $x_j/L = jh/L = j(L/N)/L = j/N$. It turns out we can solve for f_k explicitly by multiplying both sides of the above equation by $e^{-2\pi i l j/N}$ and summing over j,

$$\sum_{j=0}^{N-1} u_j e^{-2\pi i l j/N} = \sum_{j=0}^{N-1} \sum_{k=-N/2}^{N/2-1} f_k e^{2\pi i (k-l) x_j/L} = \sum_{k=-N/2}^{N/2-1} f_k \sum_{j=0}^{N-1} e^{2\pi i (k-l) x_j/L}. \tag{1.4.3}$$

The trick now is to recognize that the last sum is zero for all cases except when $k = l$. First, when $k = l$, the exponent is zero, so that we are just summing ones and the result is N (since j loops over N values). Next, when $k \neq l$, we are summing complex numbers that are evenly spaced around the unit circle. Thinking of the complex numbers as vectors, the vector sum is zero by symmetry. So we have

$$\sum_{j=0}^{N-1} e^{2\pi i (k-l) x_j/L} = N\delta_{lk}, \tag{1.4.4}$$

where δ_{lk} is the Kronecker delta, which is 0 when $k \neq l$ and 1 when $k = l$. Substituting this into Equation 1.4.3, we obtain

$$\begin{aligned} \sum_{j=0}^{N-1} u_j e^{-2\pi i l j/N} &= \sum_{k=-N/2}^{N/2-1} f_k N \delta_{lk} \\ \sum_{j=0}^{N-1} u_j e^{-2\pi i l j/N} &= f_l N \\ f_l &= \frac{1}{N} \sum_{j=0}^{N-1} u_j e^{-2\pi i l j/N} \end{aligned} \tag{1.4.5}$$

This is a closed-form expression for the amplitude of the l^{th} mode in the Fourier expansion, Equation 1.4.1. We can also go the other way: given the N amplitudes f_l, we can determine the values of u a the N nodes, u_j. This constitutes the *inverse* DFT, and the formula has already been given, in Equation 1.4.2.

Example 1.1 (Continuous Fourier transform)**.** In some cases, identifying the Fourier amplitudes is easy. For example, suppose $u(x) = e^{2\pi i x/L} + 2e^{6\pi i x/L}$. Then, from Equation 1.4.1, we immediately obtain $f_1 = 1$, $f_3 = 2$, and all the other f_k are zero.

Example 1.2 (Discrete Fourier transform)**.** Consider now the periodic function

$$u(x) = 1 + \sin^3(2x) + 0.3\cos^2(2x),$$

for a domain of size $L = 2\pi$. We sample $u(x)$ at N equally-spaced points, apply the DFT, and compare the representation using Equation 1.4.1 to the original function. Figure 1.4.1 shows this comparison for different N. We see that for $N = 4$, the data are all the same, so that the Fourier transform cannot distinguish any detail in the function beyond just a constant value. As N increases, the representation of the function improves, and by $N = 16$, all the frequencies have been captured and the interpolation is exact. More general functions will have nonzero amplitudes for all frequencies, though these amplitudes will decay with N.

1.5 Partial Differential Equations

1.5.1 Examples

Let's look at a few prototypical scalar partial differential equations (PDEs).

Advection

$$\frac{\partial u}{\partial t} + \frac{\partial (au)}{\partial x} = 0 \tag{1.5.1}$$

This equation expresses transport of a quantity u in a flow of speed a. The advective flux is au, which indicates that the state just moves with the flow velocity. The equation can be made more interesting by having a depend on location, and/or by extending it to two and three spatial dimensions. An initial condition $u(x,t) = u_0(x)$ is required, and the state u must be specified on inflow boundaries, e.g. on the left of a one-dimensional domain if $a > 0$. Figure 1.5.1 shows an example solution for the case of $a = 1$: from $t = 0$ to $t = 1$, the initial state shifts over to the right by 1, but it does not change shape.

Diffusion

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x}\left(-\nu \frac{\partial u}{\partial x}\right) = 0 \tag{1.5.2}$$

This equation expresses the spreading-out of a quantity u in viscous fluid (or diffusive solid) of viscosity (diffusivity) ν. The state u could be the temperature in a solid or the concentration of a pollutant in a fluid. The diffusive flux is $-\nu\frac{\partial u}{\partial x}$, indicating that the viscous transport occurs *down* the gradient of u. That is, peaks tend to spread out, whereas if the minus sign were not there, peaks would tend to become sharper (anti-diffusion). Figure 1.5.2 shows an example solution for the case of $\nu = 0.01$: from $t = 0$ to $t = 1$, the initial state diffuses into a shorter and wider profile. Note that the total amount of u is still conserved.

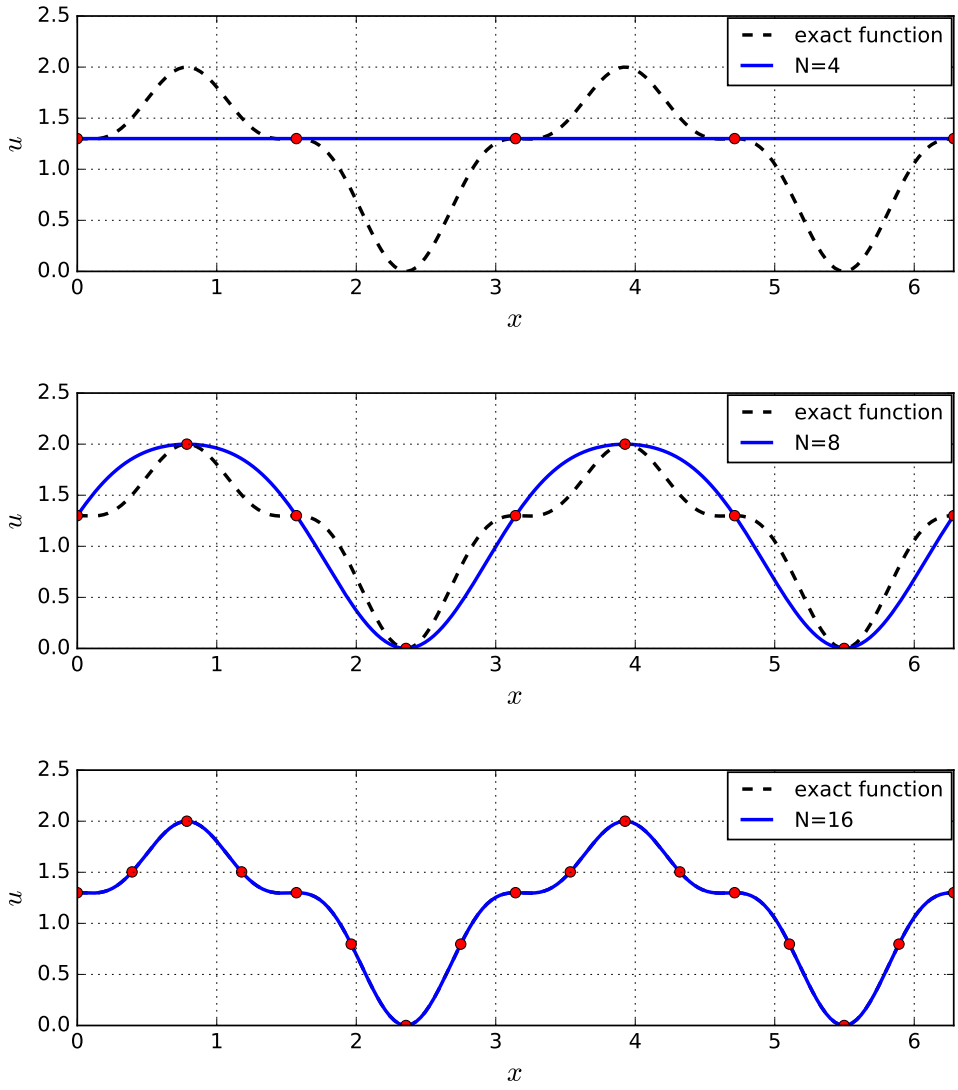


Figure 1.4.1: Discrete Fourier Transform representations of a spatial function $u(x)$, using various numbers of frequencies, N. The red circles indicate the sampled points, which are equally spaced on the domain of length $L = 2\pi$.

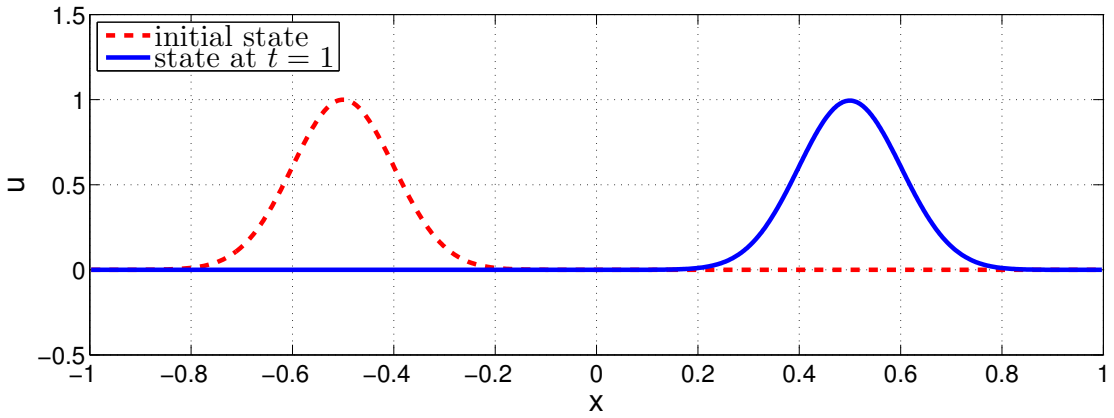


Figure 1.5.1: Sample solution of scalar advection for a Gaussian initial condition and the boundary condition $u(x = -1) = 0$.

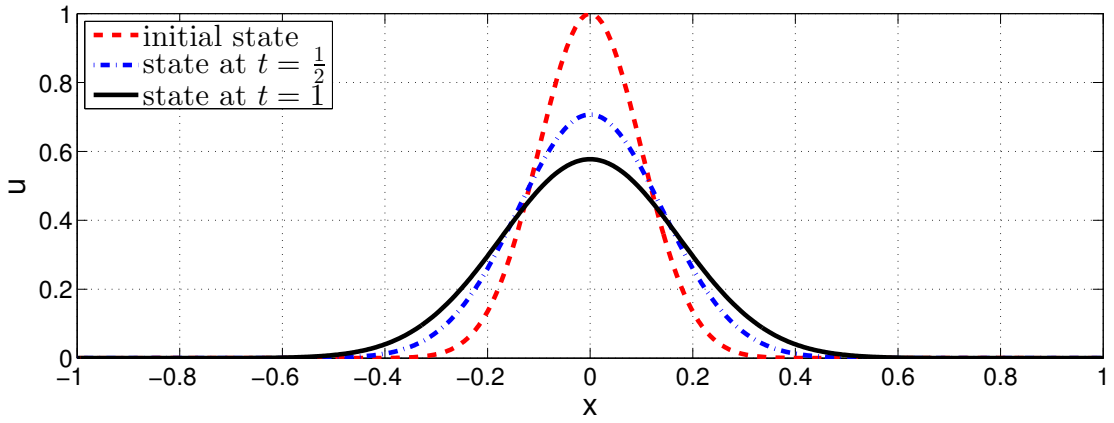


Figure 1.5.2: Sample solution of scalar diffusion for a Gaussian initial condition and boundary conditions $u(x = -1) = u(x = 1) = 0$.

Advection-Diffusion

$$\frac{\partial u}{\partial t} + \frac{\partial (au)}{\partial x} + \frac{\partial}{\partial x}\left(-\nu \frac{\partial u}{\partial x}\right) \quad = \quad 0. \tag{1.5.3}$$

This is just a combination of advection and diffusion. The relative magnitudes of the advection speed a and the viscosity ν determine how much the state spreads for a given distance it is transported. Letting L be such a characteristic distance, the **Peclet number** is defined as $Pe = aL/\nu$. High Peclet numbers (e.g. over 100) indicate dominance of advection, whereas lower ones indicate that diffusion dominates. Figure 1.5.3 shows the solution profiles at different times for an advection problem with $a = 1$ and $\nu = .01$. With a time horizon of 1, the characteristic distance is taken to be $L = 1$, so that $Pe = 100$. We indeed see that the state undergoes visible translation to the right, even though it concurrently diffuses.

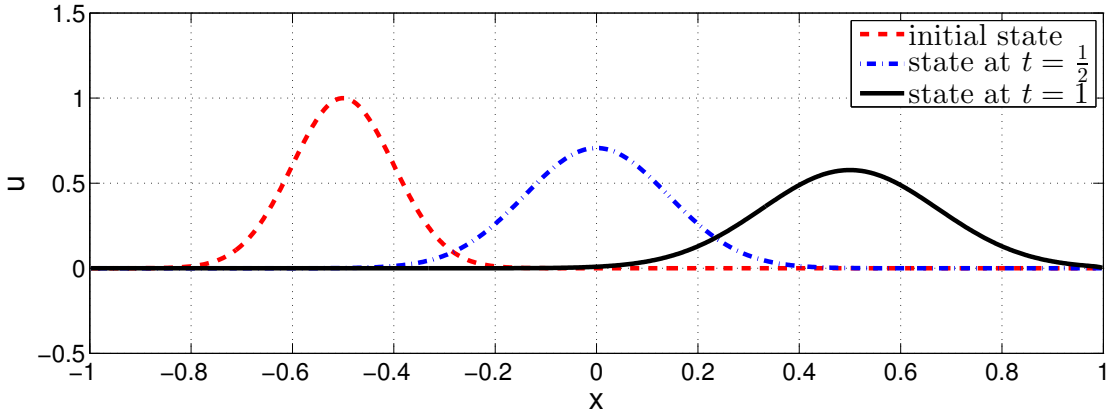


Figure 1.5.3: Sample solution of scalar advection-diffusion for a Gaussian initial condition and boundary conditions $u(x = -1) = u(x = 1) = 0$.

1.5.2 Classification

Consider a quasi-linear, first-order system of PDEs,

$$\frac{\partial \mathbf{u}}{\partial t} + \sum_{j=1}^{\dim} \mathbf{A}_j \frac{\partial \mathbf{u}}{\partial x_j} \quad = \quad \mathbf{0}. \tag{1.5.4}$$

This equation admits a plane wave solution of the form

$$\mathbf{u} \quad = \quad \hat{\mathbf{u}}\, e^{i(\vec{k}\cdot\vec{x} - \omega t)}, \tag{1.5.5}$$

where $\vec{k}$ is the wave number vector, ω is the temporal frequency, and $\hat{\mathbf{u}}$ is the amplitude vector. Substituting this solution form into the PDE system in Equation 1.5.4 and using

$$\frac{\partial \mathbf{u}}{\partial t} = -i\omega \hat{\mathbf{u}} e^{i(\vec{k}\cdot\vec{x} - \omega t)}, \qquad \frac{\partial \mathbf{u}}{\partial x_j} = i k_j \hat{\mathbf{u}} e^{i(\vec{k}\cdot\vec{x} - \omega t)},$$

gives, after canceling out the common $e^{i(\vec{k}\cdot\vec{x}-\omega t)}$,

$$-i\omega\hat{\mathbf{u}} + i\sum_{j=1}^{\dim} \mathbf{A}_j k_j \hat{\mathbf{u}} \quad = \quad \mathbf{0}. \tag{1.5.6}$$

Rearranging gives the following eigenvalue problem:

$$\hat{\mathbf{A}}\hat{\mathbf{u}} = \omega\hat{\mathbf{u}}, \quad \text{where } \hat{\mathbf{A}} \equiv \sum_j \mathbf{A}_j k_j. \tag{1.5.7}$$

The original system in Equation 1.5.4 is then classified according to the eigenvalues of $\hat{\mathbf{A}}$. Specifically, the system is:

- **Hyperbolic** if for all $\vec{k}$, $\hat{\mathbf{A}}$ has all real eigenvalues and a complete set of linearly-independent eigenvectors.
- **Elliptic** if for all $\vec{k}$, the eigenvalues are complex.
- **Parabolic** if the eigenvalues are purely imaginary.
- **Hybrid** if both real and complex eigenvalues are present.

Alternatively, if no time derivative is present in the PDE system, the wave solution takes the form $\mathbf{u} = \hat{\mathbf{u}}\, e^{i\vec{k}\cdot\vec{x}}$, and substituting this into Equation 1.5.4 (no time derivative) results in $\hat{\mathbf{A}}\hat{\mathbf{u}} = \mathbf{0}$, where recall $\hat{\mathbf{A}} \equiv \sum_j \mathbf{A}_j k_j$. This has a non-trivial solution when $\det(\hat{\mathbf{A}}) = 0$, and the character of the equation is determined by aspects of the solution vectors $\vec{k}$ that make this determinant zero. Specifically, the system is

- **Hyperbolic** if the solution vectors $\vec{k}$ that make the determinant of $\hat{\mathbf{A}}$ zero are all real. These are called *characteristics*.
- **Elliptic** if all the characteristics are complex.
- **Parabolic** if the system has an incomplete set of characteristics (i.e. some roots of multiplicity greater than one).
- **Hybrid** if some characteristics are real and some complex.

Example 1.3 (System of two equations)**.** Consider the following first-order system of two equations,

$$\begin{aligned} au_x + cv_y &= f_1 \\ bv_x + du_y &= f_2 \end{aligned}$$

Defining the state vector as $\mathbf{u} = [u, v]^T$, we can write this system in the form of Equation 1.5.4,

$$\underbrace{\begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix}}_{\mathbf{A}_1} \frac{\partial}{\partial x}\begin{bmatrix} u \\ v \end{bmatrix} + \underbrace{\begin{bmatrix} 0 & c \\ d & 0 \end{bmatrix}}_{\mathbf{A}_2} \frac{\partial}{\partial y}\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix}$$

There is no time derivative in this system, so to classify the equation we look for vectors $\vec{k}$ that make the determinant of $\hat{\mathbf{A}}$ zero:

$$\det\left(\hat{\mathbf{A}}\right) = \det\left(\mathbf{A}_1 k_1 + \mathbf{A}_2 k_2\right) = \det\begin{bmatrix} ak_1 & ck_2 \\ dk_2 & bk_1 \end{bmatrix} = 0.$$

Taking the determinant of the 2×2 matrix, we have

$$\begin{aligned} abk_1^2 - cdk_2^2 &= 0 \\ \left(\frac{k_1}{k_2}\right)^2 &= \frac{cd}{ab}, \end{aligned}$$

where we have assumed without loss of generality that $k_2 \neq 0$. Using the classification criteria, the system is hyperbolic if $(cd)/(ab) > 0$ (two real solutions), elliptic if $(cd)/(ab) < 0$ (two imaginary solutions), and parabolic if $(cd)/(ab) = 0$ (one solution of multiplicity 2). Finally, we note that the solutions for $\vec{k}$ are defined only up to a multiplicative constant, as they represent characteristic directions.

Example 1.4 (One-dimensional shallow water equations)**.** Integrating the Navier-Stokes equations over the depth of water in a vast reservoir such as a lake or ocean, where the depth is small relative to the horizontal dimensions, yields the shallow water equations. In one spatial dimension, the unsteady form of the equations reads, in quasi-linear form,

$$\begin{aligned} \frac{\partial h}{\partial t} + u\frac{\partial h}{\partial x} + h\frac{\partial u}{\partial x} &= 0, \\ \frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + g\frac{\partial h}{\partial x} &= 0, \end{aligned}$$

where h is the water height, u is the horizontal velocity, and g is the acceleration due to gravity. Defining the state vector as $\mathbf{u} = [h, u]^T$, we can write this system in the form of Equation 1.5.4,

$$\frac{\partial}{\partial t}\begin{bmatrix} h \\ u \end{bmatrix} + \underbrace{\begin{bmatrix} u & h \\ g & u \end{bmatrix}}_{\mathbf{A}_1} \frac{\partial}{\partial x}\begin{bmatrix} h \\ u \end{bmatrix} = \mathbf{0}.$$

To classify the equations we need to look at the eigenvalues of $\hat{\mathbf{A}} = \mathbf{A}_1 k_1$. For non-trivial wave numbers (i.e. $k_1 \neq 0$), this amounts to looking at the eigenvalues of $\mathbf{A}_1$, which satisfy

$$\begin{aligned} \det\begin{bmatrix} u-\lambda & h \\ g & u-\lambda \end{bmatrix} &= 0 \\ (u-\lambda)^2 - gh &= 0 \\ \lambda &= u \pm \sqrt{gh} \end{aligned}$$

As long as $g, h > 0$, there are two real eigenvalues, and hence the system is classified as hyperbolic.

1.5.3 Boundary Conditions and Well-Posedness

For a well-posed system of equations, the solution exists, is unique, and is stable. Well-posedness depends on the character of the equation and on the data specified, as follows.

Initial Value Problem (IVP) The state is given at time $t = 0$ on an infinite spatial domain. Both hyperbolic and parabolic PDEs are well-posed for such a problem. In the hyperbolic case, the initial condition data propagates in waves along characteristics, and in the parabolic case the data diffuses. An elliptic problem is ill-posed because the presence of complex eigenvalues means that there exist solutions that grow exponentially in time.

Initial Boundary Value Problem (IBVP) For this problem, the initial condition is again specified at $t = 0$, but on a finite domain. On the boundaries of this domain, additional boundary conditions are specified for $t > 0$. A hyperbolic problem is well-posed as long as the boundary data provide enough information on entering waves. A parabolic problem is well-posed as long as state information is given on all boundaries. An elliptic problem is ill-posed for the same reason as the IVP case.

Boundary Value Problem (BVP) For this problem, the data is given on a boundary of a fixed domain. Hyperbolic PDEs are ill-posed in this case as traveling wave solutions are not consistent with the full specification of boundary data. Similarly, parabolic PDEs are ill-posed because too much information is specified. On the other hand, elliptic PDEs are well-posed, as the boundary data pins the solution down.

1.5.4 Advection-Diffusion Eigenvalues

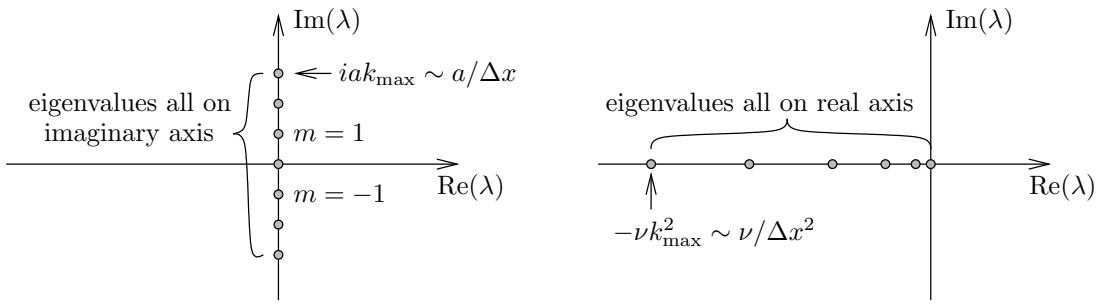
Let's take a closer look at the advection-diffusion PDE in Equation 1.5.3, for constant a and ν. Assuming periodic boundary conditions, the analytical solution takes the form

$$u(x,t) = \sum_{m=-\infty}^{\infty} \hat{u}_m(t) e^{ik_m x}, \tag{1.5.8}$$

where m is an integer and $k_m = 2\pi m$ is the wave number. Substituting this form of the solution into Equation 1.5.3, we find that the $\hat{u}_m$ must satisfy

$$\frac{d\hat{u}_m}{dt} = \lambda_m \hat{U}_m, \qquad \text{where} \quad \lambda_m = iak_m - \nu k_m^2.$$

We consider separately the limits of advection and diffusion-dominated transport. First, suppose that $\nu \to 0$, in which case the eigenvalues are $\lambda_m = -iak_m$, as shown in Figure 1.4(a). These are purely imaginary: physically, from the above solution form, this means that the different waves present in the solution translate undamped at speed a. The maximum eigenvalue magnitude scales linearly with the wave number k_m. On a grid of points spaced



(a) Advection limit (b) Diffusion limit

Figure 1.5.4: Eigenvalues of a scalar advection-diffusion PDE in the advection and diffusion limits.

Δx apart, the maximum resolvable wave number is proportional to the inverse of Δx: $k_m \sim 1/\Delta x$.

Next we consider pure diffusion, i.e. $a \to 0$. In this case the eigenvalues are $\lambda_m = -\nu k_m^2$, as shown in Figure 1.4(b). These are all real and on the negative imaginary axis. The physical interpretation of this is that in pure diffusion, the various solution modes do not translate, but instead dissipate (decay exponentially). Note that the largest eigenvalue magnitude scales as $k_m^2 \sim 1/\Delta x^2$.

1.6 Computation

Computational fluid dynamics is done on computers, which are now very powerful and diverse. Let's first review some key terms:

- **1 bit**: elemental storage unit that can be on (1) or off (0).
- **1 byte**: 8 bits
  - 1 Megabyte (MB) = 10^6 bits
  - 1 Gigabyte (GB) = 10^9 bits
  - 1 Terabyte (TB) = 10^{12} bits
  - 1 Petabyte (PB) = 10^{15} bits
- **int**: integer number, typically stored using 4 bytes.
- **float**: real number stored using 4 bytes. This is called *single precision* and provides about 8 digits of precision.
- **double**: real number stored using 8 bytes. This is called *double precision* and provides about 16 digits of precision. It is the default for many programming languages.

- **FLOP**: floating-point operations (e.g. addition or multiplication) per second. So 1 Gigaflop is 10^9 FLOPs, 1 Teraflop is 10^{12} FLOPs, etc.
- **core**: processing unit that reads and executes instructions.
- **memory**: (RAM) short-term storage of data and programs in execution, quickly accessible by the core(s). The typical size of memory in a personal computer is currently 8GB.
- **cache**: very short-term storage of data close to the core, which the core can access faster than main memory.
- **hard disk**: long-term storage of data and programs, not quickly accessible by the core(s). A typical size in a personal computer is 256-512GB.

Computers have become very powerful due to an increase in the number of transistors per chip. This number has been doubling about every two years, a phenomenon that has been named *Moore's law*, after Gordon Moore, a co-founder of Intel. This is purely an observation, and in fact the period for doubling is currently slowing down to about 2.5 years. Nevertheless, the growth remains rapid and has a direct impact on the performance of today's computers. Up until about 2004, the speed at which computation could be done grew because of increases in processor clock speeds. However, limits on power and thermal management have stifled this growth, and since 2004, computers have become "faster" by employing multiple cores, instead of increasing the speed of a single core. This trend has made critical the practice of *parallel programming*, whereby programs are written to split a problem into tasks that can be done concurrently, ideally with minimal communication between tasks.

Large computations require large computers (many cores), and the most advanced calculations may utilize hundreds of thousands of cores. Such computations are done on supercomputers, which consist of many cores, not necessarily identical, linked together via a communication network. As of June 2016, the most powerful supercomputer in the world is the Sunway TaihuLight, at the National Supercomputing Center in Wuxi, China. It has over 10 million cores and achieves a peak performance of 125 petaflops. Such performance comes with high power requirements: the Sunway TaihuLight consumes 15 megawatts of power![2]

1.7 Algorithms

1.7.1 Design Considerations

Machine Precision When solving systems on a computer, an important consideration is round-off error due to finite *machine precision*. Most programs operate with double precision, which means that 8 bytes (64 bits) are used to encode a real number. This means that many (uncountably infinite) numbers are not representable on a computer. So every time we operate on numbers (e.g. add, multiply, etc.) we potentially introduce a little bit of error.

[2]This is an improvement over the number two supercomputer, China's Tianhe-2, which maxes out at 55 petaflops but consumes almost 18 megawatts of power.

For double-precision operations on $\mathcal{O}(1)$ numbers, these errors are each $\mathcal{O}(10^{-16})$. However, over many operations, these errors can accumulate and be noticeable in the final result. In addition, some (not so good) numerical algorithms can be *unstable*, meaning that these errors are amplified exponentially over the operations.

Complexity Complexity refers to the cost of an algorithm, typically measured in operation counts. The complexity is often expressed in terms of an input number, N, which represents the size of the problem. For example, in a sorting algorithm, this would be the number of items to be sorted. In a matrix operation, this could be the number of rows/columns. In CFD, N may refer to the number of nodes, elements, or edges in a computational mesh, depending on the type of algorithm used.

The complexity is usually given as an order of magnitude estimate. For example, $\mathcal{O}(N)$ refers to an algorithm that scales linearly with the size of the problem. We use order-of-magnitude expressions because the precise factor in front of the cost measure is less important than the dependence of the cost on N. For example, in CFD applications, we often deal with very large N (e.g. meshes with billions of nodes). In such cases, an algorithm of complexity $\mathcal{O}(N^2)$ will be useless relative to an $\mathcal{O}(N)$ algorithm, even if the $\mathcal{O}(N)$ algorithm has constant scaling factor out front that is 10 or 100 times larger than the $\mathcal{O}(N^2)$ algorithm. In general, for CFD applications, we seek algorithms that scale like $\mathcal{O}(N)$ or at most $\mathcal{O}(N \log N)$.

Scalability Since today's computers rely on multi-core architectures, algorithms must be *scalable* beyond just a single processor. This means designing algorithms so that the majority of tasks can be preformed concurrently. Sometimes this is easy, and the term *embarrassingly parallel* is given to algorithms that are perfectly scalable to start. For example, Monte Carlo simulations involve running the same code repeatedly for different inputs, and these can be done independently on each core without any communication. Most algorithms, however, are not perfectly scalable. *Amdahl's law* states that if an un-parallelizable portion of the code consumes a fraction f of the total serial run time, then parallelization is limited to speeding up the code by a factor $1/f$, regardless of the number of processors used.

Communication between processors also affects scalability. The most successful parallel algorithms divide up the work in a way that processors do not need to send a lot of data to each other often, but instead spend the majority of their time doing their own calculations. A simple model for communication is that the cost (in time) of sending a message of size (bytes) n is $L + n/B$, where: L is the latency, which measures the fixed cost of initiating a message transfer; and B is the bandwidth, which measures the throughput of the network connections. The goal in supercomputer networking is to minimize the latency and maximize the bandwidth.

1.7.2 Gaussian Elimination

Gaussian elimination is a standard direct technique for solving the system

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{1.7.1}$$

where $\mathbf{A} \in \mathbb{R}^{N \times N}$ is a given matrix, $\mathbf{b} \in \mathbb{R}^N$ is a given right-hand-side vector, and $\mathbf{x} \in \mathbb{R}^N$ is the vector of unknowns for which we are trying to solve. The technique works as follows: first, solve the first equation of the system for the first unknown in terms of the other unknowns; next, substitute for the first unknown in the remaining equations; next, repeat the process with the second equation, third equation, etc. By the time we reach the final equation, there is only one unknown, and hence the solution for the last unknown is trivial. To solve for the remaining unknowns, proceed backwards, back-substituting the known values into the equations, starting with the second-to-last and continuing until the first.

Let's look at an example of a 3 by 3 system:

$$\begin{bmatrix} 1 & 2 & 2 \\ 1 & 1 & 2 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}.$$

Subtract the first row from the second row:

$$\begin{bmatrix} 1 & 2 & 2 \\ 0 & -1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 3 \end{bmatrix}.$$

Add the second row to the third row:

$$\begin{bmatrix} 1 & 2 & 2 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 4 \end{bmatrix}.$$

Add two times the second row and -2 times the third row to the first row:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -5 \\ 1 \\ 4 \end{bmatrix}.$$

From this last system, we read off that $[x_1, x_2, x_3] = [-5, -1, 4]$.

A systematic and efficient implementation of Gaussian elimination is the pivoted lower-upper (PLU) factorization of the matrix $\mathbf{A}$. Moreover, this factorization can be done in place. For the matrix $\mathbf{A}$ above, the factorization reads

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\mathbf{P}} \underbrace{\begin{bmatrix} 1 & 2 & 2 \\ 1 & 1 & 2 \\ 0 & 1 & 1 \end{bmatrix}}_{\mathbf{A}} = \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix}}_{\mathbf{L}} \underbrace{\begin{bmatrix} 1 & 2 & 2 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\mathbf{U}}$$

The matrix $\mathbf{P}$ is a permutation matrix, which dictates in what order the equations are solved during Gaussian elimination. For instance, if the first entry of the first row of $\mathbf{A}$ were zero, we could not solve the first equation for the first variable, and we would instead exchange the first row with another row in the matrix, in fact one in which the absolute value of the

first entry is largest. This matrix $\mathbf{P}$ of row swaps consists of a single 1 in each row/column and it can be stored as one integer vector. The matrix $\mathbf{L}$ is lower triangular with 1's on the main diagonal, while the matrix $\mathbf{A}$ is upper triangular. By not storing the 1's of $\mathbf{L}$, both of these matrices can be stored rolled into one single matrix, usually overwriting the storage of $\mathbf{A}$. The advantage of this factorization is that we can easily apply both $\mathbf{A}$ and $\mathbf{A}^{-1}$, in the former case by multiplying by $\mathbf{U}$, then $\mathbf{L}$, then $\mathbf{P}^{-1}$, and in the latter case by multiplying by $\mathbf{P}$, then $\mathbf{L}^{-1}$, and finally $\mathbf{U}^{-1}$. Note that the applications of $\mathbf{L}^{-1}$ and $\mathbf{U}^{-1}$ involve just forward or backward substitution.

The steps for constructing the PLU factorization are given in Algorithm 1. Note the two nested `for` loops, and the vector operation on $\mathbf{U}$ in the inner loop, resulting in a complexity that is $\mathcal{O}(N^3)$, in fact $2N^3/3$ additions/multiplications. Again, separate storage for $\mathbf{L}$ and $\mathbf{U}$ is not required when the matrix $\mathbf{A}$ is overwritten.

Algorithm 1 Pivoted Lower-Upper (PLU) factorization

$\mathbf{U} = \mathbf{A}$, $\mathbf{L} = \mathbf{I}$, $\mathbf{P} = \mathbf{I}$
for $k = 1 : (N-1)$ **do**
    Choose $i \geq k$ that maximizes $|U_{ik}|$
    swap$(U_{k,k:N}, U_{i,k:N})$ (interchange rows i and k)
    swap$(L_{k,1:(k-1)}, L_{i,1:(k-1)})$
    swap$(P_{k,:}, P_{i,:})$
    for $j = (k+1) : N$ **do**
        $L_{j,k} = U_{j,k}/U_{kk}$
        $U_{j,k:N} = U_{j,k:N} - L_{j,k}U_{k,k:N}$
    end for
end for

1.7.3 Bisection

The bisection method is a way of iteratively finding a root of the scalar equation $f(x) = 0$ over an interval $x \in [a, b]$, where $f(a)$ and $f(b)$ have opposite signs. The idea is to close in on the root by repeatedly halving the interval, using evaluations of $f(x)$ to determine in which half a root must lie. Algorithm 2 presents this method. Note, that the tolerance is user-specified and depends on the desired accuracy.

Algorithm 2 Bisection

```
Given a, b
while (b − a) > tolerance do
    c = (a + b)/2
    if f(c) = 0 then
        a = b = c
    else if f(c) · f(a) < 0 then
        b = c
    else
        a = c
    end if
end while
```

1.7.4 Newton's Method

Consider a system of N nonlinear equations,

$$\mathbf{R}(\mathbf{U}) = \mathbf{0}, \tag{1.7.2}$$

where $\mathbf{U} \in \mathbb{R}^N$ is the state vector of unknowns and $\mathbf{R} \in \mathbb{R}^N$ is the residual vector, the components of which are nonlinear functions of $\mathbf{U}$. The goal is to find the state solution (assumed unique) that makes the residual zero. The Newton-Raphson method finds such a solution iteratively, by successive linearizations of the problem.

Given a solution guess $\mathbf{U}_k$ for which $\mathbf{R}(\mathbf{U}_k) \neq \mathbf{0}$, we seek an update $\Delta\mathbf{U}_k$ such that

$$\mathbf{R}(\mathbf{U}_k + \Delta\mathbf{U}_k) \approx \mathbf{0}. \tag{1.7.3}$$

Linearizing the left-hand side, we form the system

$$\mathbf{R}(\mathbf{U}_k) + \left.\frac{\partial \mathbf{R}}{\partial \mathbf{U}}\right|_{\mathbf{U}_k} \Delta\mathbf{U}_k = \mathbf{0}. \tag{1.7.4}$$

Solving for the update,

$$\Delta\mathbf{U}_k = -\left[\left.\frac{\partial \mathbf{R}}{\partial \mathbf{U}}\right|_{\mathbf{U}_k}\right]^{-1} \mathbf{R}(\mathbf{U}_k). \tag{1.7.5}$$

$\left.\frac{\partial \mathbf{R}}{\partial \mathbf{U}}\right|_{\mathbf{U}_k}$ is the residual Jacobian matrix about state $\mathbf{U}_k$. For large N, the above linear system is typically solved using an iterative method. Once $\Delta\mathbf{U}_k$ is available, we set $\mathbf{U}_{k+1} = \mathbf{U}_k + \Delta\mathbf{U}_k$ and continue iterating until a residual convergence criterion is met. Figure 1.7.1 illustrates this process for a scalar problem.

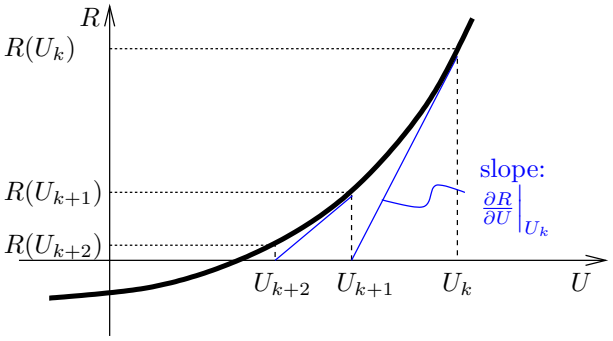


Figure 1.7.1: Illustration of the Newton-Raphson algorithm for a scalar problem.

If $\mathbf{U}_k$ is close to the solution of (1.7.2), which we call $\mathbf{U}_*$, we can write the truncated Taylor-series expansion,

$$\mathbf{0} = \mathbf{R}(\mathbf{U}_*) = \mathbf{R}(\mathbf{U}_k) + \frac{\partial \mathbf{R}}{\partial \mathbf{U}}\Big|_{\mathbf{U}_k} (\mathbf{U}_* - \mathbf{U}_k) + \mathcal{O}(\mathbf{U}_* - \mathbf{U}_k)^2$$

From this we subtract the update equation

$$\mathbf{0} = \mathbf{R}(\mathbf{U}_k) + \frac{\partial \mathbf{R}}{\partial \mathbf{U}}\Big|_{\mathbf{U}_k} (\mathbf{U}_{k+1} - \mathbf{U}_k)$$

and obtain

$$\mathbf{0} = \frac{\partial \mathbf{R}}{\partial \mathbf{U}}\Big|_{\mathbf{U}_k} (\mathbf{U}_* - \mathbf{U}_{k+1}) + \mathcal{O}(\mathbf{U}_* - \mathbf{U}_k)^2 \quad \Rightarrow \quad (\mathbf{U}_* - \mathbf{U}_{k+1}) = \mathcal{O}(\mathbf{U}_* - \mathbf{U}_k)^2$$

This says that the state error at iteration $k+1$ is proportional to the square of the state error at iteration k. That is, Newton-Raphson will converge quadratically when the state is close to the exact solution.

If $\mathbf{U}_k$ is not close to the solution $\mathbf{U}_*$, the linearization may not be very accurate, leading to large and inaccurate $\Delta\mathbf{U}_k$ and failure of Newton-Raphson. Sometimes, especially in CFD, the updated state $\mathbf{U}_k + \Delta\mathbf{U}_k$ might not even be physical. A standard solution is to *limit* the update according to

$$\mathbf{U}_{k+1} = \mathbf{U}_k + \omega\Delta\mathbf{U}_k, \tag{1.7.6}$$

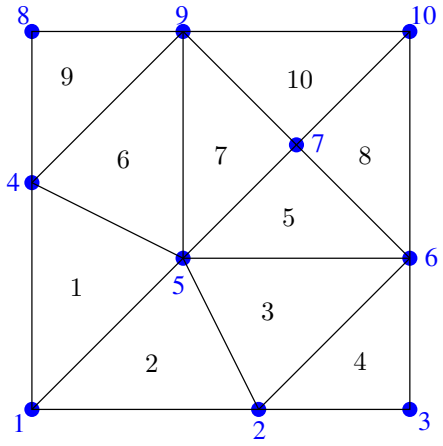
where ω is chosen to prevent excessive changes in the state. For example, in CFD, a reasonable requirement is to keep changes in density and pressure to some small percent (e.g. 10-20%) of the current values. Similar requirements can be formulated for other equations that have physical constraints on variables. A single ω, set by the most restrictive update requirement, is typically used for the entire state.

Continuation methods also address convergence problems associated with $\mathbf{U}_k$ not being close to the solution $\mathbf{U}_*$. However, instead of working only with the update given by $\Delta\mathbf{U}_k$, continuation methods generally alter the equations. Various continuation methods are used in CFD, including parameter continuation, order continuation, boundary-condition continuation, pseudo-time continuation. This list is not exclusive, and some methods are very simple. For example, in order continuation, low-order solutions are used to initialize high-order solves.

Pseudo-time continuation is one of the most-popular techniques for improving robustness of CFD solvers. In a steady-state CFD calculation, an initial guess will usually be far from the solution, and update limiting alone may not be sufficient to converge Newton-Raphson. The idea in pseudo-time continuation is to let physical time evolution guide the state update from the initial condition to the desired steady solution. That is, an artificial time stepping term is included, so that the state evolves physically to from the initial condition to steady state.

1.7.5 Hash Tables

Most CFD codes use meshes to divide a domain into small, manageable pieces, on which the solution can be systematically approximated. In two dimensions, meshes of triangles are easy to generate and flexible in filling the spaces around complicated geometries. Figure 1.7.2 shows a mesh of triangles, with nodes and elements (triangles) numbered. We consider two data structures associated with this mesh. First, the matrix $\mathbf{E}$ gives the node numbers of the nodes adjacent to each element. For example, the first row corresponds to the first element, which is adjacent to nodes 1, 5, and 4. The matrix $\mathbf{N}$ stores the reverse information: the element numbers adjacent to each node. For example, node 4 (fourth row of $\mathbf{N}$) is surrounded by elements 1, 6, and 9.



$$\mathbf{E} = \begin{bmatrix} 1 & 5 & 4 \\ 1 & 2 & 5 \\ 2 & 6 & 5 \\ 2 & 3 & 6 \\ 5 & 6 & 7 \\ 4 & 5 & 9 \\ 5 & 7 & 9 \\ 7 & 6 & 10 \\ 4 & 9 & 8 \\ 7 & 10 & 9 \end{bmatrix} \qquad \mathbf{N} = \begin{bmatrix} 1 & 2 & & & & \\ 2 & 3 & 4 & & & \\ 4 & & & & & \\ 1 & 6 & 9 & & & \\ 1 & 2 & 3 & 5 & 6 & 7 \\ 3 & 4 & 5 & 8 & & \\ 5 & 7 & 8 & 10 & & \\ 9 & & & & & \\ 6 & 7 & 9 & 10 & & \\ 8 & 10 & & & & \end{bmatrix}$$

Figure 1.7.2: Sample mesh of triangles, with node numbers in blue and element numbers in black. The matrices $\mathbf{E}$ and $\mathbf{N}$ give the node numbers adjacent to each element and vice versa, respectively.

Suppose that you are given matrix $\mathbf{E}$ and your task is to construct matrix $\mathbf{N}$. An inefficient way to do this would be to loop over the nodes, and for each node to loop over elements to find, via $\mathbf{E}$, which elements are adjacent to that node. This is inefficient because of the nested loop over elements inside the loop over nodes. The complexity of such an algorithm would be $\mathcal{O}(N^2)$, where N is the number of elements/nodes (they scale in the same fashion). An efficient $\mathcal{O}(N)$ algorithm does exist for this problem, using just a loop over elements. First, we obtain the number of nodes by looking at the largest entry in $\mathbf{E}$. We then obtain the maximum number of elements around a node by looping over the entries in $\mathbf{E}$ and keeping a count of the number of times each node is referenced. We can then use this maximum number for allocation purposes of $\mathbf{N}$, which in fact would be stored as a sparse matrix or a set of linked lists in practice. Finally, the key step is to loop over elements again, and for each element, to add the element number to the three rows of $\mathbf{N}$ that correspond to the three nodes adjacent to the element. Algorithm 3 presents these steps in detail.

Algorithm 3 Identifying elements around nodes: from **E** to **N**

Given **E** with N_e rows
number of nodes: N_n = maximum number in **E**
Set counter vector $\mathbf{c} = \mathbf{0}$, size N_n
Initialize $\mathbf{N} = \mathbf{0}$ (or empty linked list)
for $e = 1 : N_e$ **do**
    for $i = 1 : 3$ **do**
        $n = E_{e,i}$
        $c_n = c_n + 1$
        $N_{n,c_n} = e$
    end for
end for

1.7.6 Gram-Schmidt Orthogonalization

A set of *linearly independent* vectors or functions forms a basis, the span of which is a vector or function space. Sometimes we need an *orthogonal* set of basis vectors/functions, which means that the inner product (see Section 1.3) between any two of them must be zero. The term **orthonormal** adds the requirement that the inner product between each vector/function and itself is unity. The **Gram-Schmidt** algorithm provides a method for systematically constructing such a set, starting from a set of linearly-independent vectors or functions.

Algorithm 4 presents this algorithm in the context of a general inner product $(\cdot, \cdot)$. This is actually called the *modified* Gram-Schmidt method, as the same method with a different ordering of operations is sensitive to machine precision rounding (i.e. not backwards stable). The input is a set of N linearly-independent vectors $\mathbf{u}_i$, and the output is a set of orthonormal vectors $\mathbf{v}_i$ that span the same space.

Algorithm 4 The modified Gram-Schmidt method

Given a set of vectors: $\mathbf{u}_i$, $i \in 1 \ldots N$
for $i = 1 : N$ **do**
    $\mathbf{v}_i = \mathbf{u}_i$
    for $j = 1 : (i - 1)$ **do**
        $\mathbf{v}_i = \mathbf{v}_i - (\mathbf{v}_j, \mathbf{v}_i)\mathbf{v}_j$
    end for
    $\mathbf{v}_i = \mathbf{v}_i / \sqrt{(\mathbf{v}_i, \mathbf{v}_i)}$
end for

Example 1.5 (Generation of the Legendre polynomials)**.** Consider the space of functions of one-dimension, x, over the range $-1 \leq x \leq 1$, and the inner product

$$(u, v) \quad = \quad \int_{-1}^{1} u(x)v(x)\, dx. \tag{1.7.7}$$

We use the modified Gram-Schmidt algorithm to construct a basis for order p polynomials that is orthonormal with respect to this inner product. The starting point is a set of linearly-independent but not necessarily orthogonal/orthonormal functions. The simplest choice is monomials: $u_0(x) = 1$, $u_2(x) = x$, ..., $u_p(x) = x^p$. Note that convenience, to match the order, we have started our counting of functions at 0. Let's begin with the first function, $u_0(x)$. Since

$$(u_0, u_0) = \int_{-1}^{1} u_0 u_0 \, dx = \int_{-1}^{1} 1 \, dx = 2,$$

the first basis vector is

$$v_0(x) = \frac{u_0(x)}{\sqrt{2}} = \frac{1}{\sqrt{2}}.$$

The next basis vector begins with $v_1 = u_1$ and is first orthogonalized against $v_0(x)$:

$$v_1 = v_1 - (v_0, v_1)v_0.$$

Performing the inner product calculation,

$$(v_0, v_1) = \int_{-1}^{1} v_0 v_1 \, dx = \int_{-1}^{1} \frac{1}{\sqrt{2}} x \, dx = 0,$$

So that v_1 remains $v_1(x) = x$. To normalize it, we need

$$(v_1, v_1) = \int_{-1}^{1} v_1 v_1 \, dx = \left[x^3/3\right]_{-1}^{1} = \frac{2}{3}.$$

Normalizing v_1 by the square root of this quantity gives

$$v_1(x) = \sqrt{\frac{3}{2}} x.$$

The third basis vector begins with $v_2 = u_2$ and is first orthogonalized against *both* v_0 and v_1:

$$v_2 = v_2 - (v_0, v_2)v_0 - (v_1, v_2)v_1.$$

The required inner products are

$$\begin{aligned} (v_0, v_2) &= \int_{-1}^{1} \frac{1}{\sqrt{2}} x^2 \, dx = \frac{\sqrt{2}}{3}, \\ (v_1, v_2) &= \int_{-1}^{1} \sqrt{\frac{3}{2}} x^3 \, dx = 0, \end{aligned}$$

so that

$$v_2 = v_2 - \frac{\sqrt{2}}{3} v_0 - 0 v_1 = x^2 - \frac{\sqrt{2}}{3} \frac{1}{\sqrt{2}} = x^2 - \frac{1}{3}.$$

Orthonormalizing v_2 requires the inner product

$$(v_2, v_2) = \int_{-1}^{1} (x^2 - 1/3)^2 \, dx = \frac{8}{45},$$

so that

$$v_2(x) = \sqrt{\frac{45}{8}} \left(x^2 - \frac{1}{3} \right) = \sqrt{\frac{5}{8}} \left(3x^2 - 1 \right).$$

We could continue this process systematically to as high degree polynomial as needed. The resulting functions are the (normalized) Legendre polynomials, which form a useful well-conditioned basis for various high-order methods.

Chapter 2

Finite Difference Approximations

2.1 Examples

Finite difference approximations are widely used in discretizing ordinary and partial differential equations. One of the simplest finite difference formulas is that of a first derivative,

$$\left.\frac{du}{dx}\right|_x \approx \frac{u(x+\Delta x)-u(x)}{\Delta x}, \tag{2.1.1}$$

where in the limit $\Delta x \to 0$ we recover the standard definition of a derivative. We can use this formula to approximate the slope of $u(x)$ at a point of a grid, as shown in Figure 2.1.1. Note that for now we assume a *uniform grid*, which is one in which the points are equally spaced. On this grid, we represent $u(x)$ through u_i, which are values of u at the $N+1$ nodes, $x_i = i\Delta x$, $0 \le i \le N$.

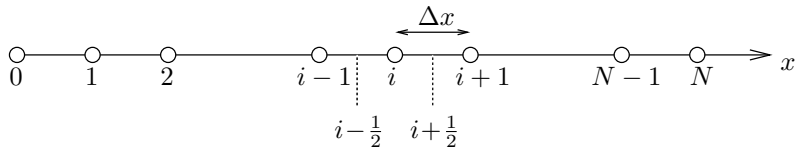


Figure 2.1.1: Uniform grid used for one-dimensional finite difference formulas. The spacing between all consecutive nodes is Δx.

If we're sitting at node i, we can look *forward* to $i+1$ and calculate the first derivative using Equation 2.1.1,

$$\text{forward difference:} \qquad \left.\frac{du}{dx}\right|_i \approx \frac{u_{i+1}-u_i}{\Delta x}. \tag{2.1.2}$$

But we can also look *backward* to node $i-1$,

$$\text{backward difference:} \qquad \left.\frac{du}{dx}\right|_i \approx \frac{u_i-u_{i-1}}{\Delta x}. \tag{2.1.3}$$

Or, for symmetry, we can average the forward and backward formulas to obtain

$$\text{central difference:} \qquad \left.\frac{du}{dx}\right|_i \approx \frac{u_{i+1} - u_{i-1}}{2\Delta x}. \tag{2.1.4}$$

We'll see in the next section that the central difference formula is more accurate than the forward or backward difference formulas. This can be reasoned graphically, as shown in Figure 2.1.2, where the forward and backward difference exhibit bias to one side of x_i, whereas the central difference used data symmetrically around x_i.

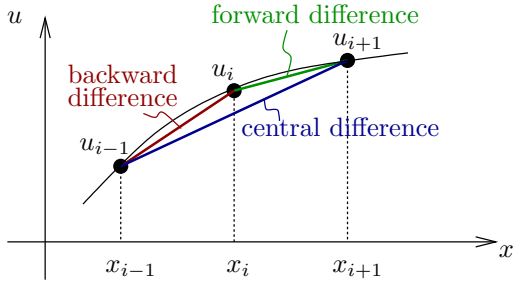


Figure 2.1.2: Calculation of du/dx at x_i from nodal values of u using a forward difference, backward difference, and the central difference.

We can apply the central-difference formula using consecutive nodes to get an approximation of the first derivative half-way between the nodes,

$$\left.\frac{du}{dx}\right|_{i+\frac{1}{2}} \approx \frac{u_{i+1} - u_i}{\Delta x}, \quad \text{and} \quad \left.\frac{du}{dx}\right|_{i-\frac{1}{2}} \approx \frac{u_i - u_{i-1}}{\Delta x}.$$

Note the use of $i \pm \frac{1}{2}$, as defined in Figure 2.1.1. Taking the difference of these two first derivatives gives us an approximation to the second derivative at node i,

$$\left.\frac{d^2u}{dx^2}\right|_i \approx \frac{\left.\frac{du}{dx}\right|_{i+\frac{1}{2}} - \left.\frac{du}{dx}\right|_{i-\frac{1}{2}}}{\Delta x} \approx \frac{\frac{u_{i+1}-u_i}{\Delta x} - \frac{u_i-u_{i-1}}{\Delta x}}{\Delta x} = \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2}. \tag{2.1.5}$$

These are just a few popular finite difference formulas. Before we discuss how to derive general formulas, we first look at how to measure their accuracy.

2.2 Truncation Error Analysis

For a finite Δx, the forward difference formula in Equation 2.1.1 is just an approximation of the true first derivative at x. We identify the *truncation error* as the difference between this approximation when the exact u is used in the finite difference formula and the true derivative,

$$\text{forward difference truncation error} = \tau^{\text{forward}} \equiv \underbrace{\frac{u(x+\Delta x) - u(x)}{\Delta x}}_{\text{approximate derivative}} - \underbrace{\left.\frac{du}{dx}\right|_x}_{\text{true derivative}}. \tag{2.2.1}$$

We can calculate the truncation error, or at least the rate of convergence with Δx, by using Taylor-series expansions. In the above equation, we expand $u(x + \Delta x)$ as

$$u(x + \Delta x) = u(x) + \Delta x \frac{du}{dx} + \frac{1}{2}\Delta x^2 \frac{d^2u}{dx^2} + \frac{1}{6}\Delta x^3 \frac{d^3u}{dx^3} + \mathcal{O}(\Delta x^4).$$

All derivatives above are taken at x. Using this expansion in Equation 2.2.1, the truncation error of our forward difference is

$$\begin{aligned} \tau^{\text{forward}} &= \frac{1}{\Delta x}\left(u(x) + \Delta x \frac{du}{dx} + \frac{1}{2}\Delta x^2 \frac{d^2u}{dx^2} + \frac{1}{6}\Delta x^3 \frac{d^3u}{dx^3} + \mathcal{O}(\Delta x^4) - u(x)\right) - \frac{du}{dx} \\ &= \frac{1}{2}\Delta x \frac{d^2u}{dx^2} + \frac{1}{6}\Delta x^2 \frac{d^3u}{dx^3} + \mathcal{O}(\Delta x^3). \end{aligned}$$

This shows that the forward difference is first-order accurate, meaning that the truncation error has a leading term of order Δx^1. A similar analysis for the backwards difference scheme shows that it is also first-order accurate,

$$\begin{aligned} \tau^{\text{backward}} &= \frac{u(x) - u(x - \Delta x)}{\Delta x} - \left.\frac{du}{dx}\right|_x \\ &= \frac{1}{\Delta x}\left(u(x) - u(x) + \Delta x \frac{du}{dx} - \frac{1}{2}\Delta x^2 \frac{d^2u}{dx^2} + \frac{1}{6}\Delta x^3 \frac{d^3u}{dx^3} + \mathcal{O}(\Delta x^4)\right) - \frac{du}{dx} \\ &= -\frac{1}{2}\Delta x \frac{d^2u}{dx^2} + \frac{1}{6}\Delta x^2 \frac{d^3u}{dx^3} + \mathcal{O}(\Delta x^3). \end{aligned}$$

The central difference, $[u(x + \Delta x) - u(x - \Delta x)]/\Delta x$, is the average of the forward and backward differences, so its truncation error is

$$\begin{aligned} \tau^{\text{central}} &= \frac{1}{2}\left(\tau^{\text{forward}} + \tau^{\text{backward}}\right) \\ &= \frac{1}{6}\Delta x^2 \frac{d^3u}{dx^3} + \mathcal{O}(\Delta x^3). \end{aligned}$$

This shows that the central difference approximates the derivative to second order, as opposed to the first-order accuracy of forward and backward differences.

Example 2.1 (Second derivative difference truncation error)**.** How accurate is the second-derivative difference in Equation 2.1.5?

$$\left.\frac{d^2u}{dx^2}\right|_i \approx \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2}$$

Using a Taylor-series expansion about node i, we have

$$\begin{aligned} u_{i+1} &= u(x_i + \Delta x) = u_i + \Delta x \frac{du}{dx} + \frac{1}{2}\Delta x^2 \frac{d^2u}{dx^2} + \frac{1}{6}\Delta x^3 \frac{d^3u}{dx^3} + \mathcal{O}(\Delta x^4) \\ u_{i-1} &= u(x_i - \Delta x) = u_i - \Delta x \frac{du}{dx} + \frac{1}{2}\Delta x^2 \frac{d^2u}{dx^2} - \frac{1}{6}\Delta x^3 \frac{d^3u}{dx^3} + \mathcal{O}(\Delta x^4) \end{aligned}$$

All derivatives are taken at x_i. The truncation error is the error between the finite difference approximation via Equation 2.1.5 and the actual second derivative,

$$\begin{aligned}\tau &= \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2} - \frac{d^2u}{dx^2} \\ &= \frac{1}{\Delta x^2}\left(0 + 0 + \Delta x^2 \frac{d^2u}{dx^2} + 0 + \mathcal{O}(\Delta x^4)\right) - \frac{d^2u}{dx^2} \\ &= \mathcal{O}(\Delta x^2)\end{aligned}$$

So this finite difference gives a "second-order" accurate approximation of the second derivative.

The order of accuracy tells us how quickly the approximated derivative converges to the true derivative as the mesh is refined, i.e. as Δx decreases. We can verify the order of accuracy numerically by applying the finite differences to a known function, computing the truncation error via Equation 2.2.1, and monitoring how the error changes as Δx decreases. During such a test, we plot the logarithm of the error versus the logarithm of Δx. If the finite difference is implemented correctly, we expect to see a straight line with slope equal to p, the order of accuracy. This is because the truncation error behaves as

$$\tau = \mathcal{O}(\Delta x^p) = C\Delta x^p,$$

for some constant C. Taking the log of both sides, we have

$$\log \tau = \log C + p \log \Delta x.$$

So a plot of the logarithm of the error versus the logarithm of Δx should reduce to a straight line with slope p. In practice, to make plots easier to read, we use a log-log scale instead of plotting the log of the quantities. The plots will not always be exactly straight lines, even if everything is implemented correctly, because the order of magnitude estimates only hold asymptotically for $\Delta x \to 0$. At finite Δx, higher-order error terms may not be negligible.

Example 2.2 (Verifying the order of accuracy of finite differences)**.** Consider approximating the first derivative of the analytic function

$$u(x) = 1 + (\sin(2x))^3$$

at the point $x_0 = 1$. Starting with an initial $\Delta x = 0.1$, we successively reduce Δx by factors of two and compute the truncation errors for the three difference methods. Listing 2.2.1 presents the code that implements these steps, and Figure 2.2.1 shows the resulting convergence plot. The convergence rates (values of p) are shown in the legend entries, and these are measured between the results with the two smallest Δx values. We confirm that the forward and backward difference methods yield approximately first-order slope values, whereas the central difference yields second-order slope values.

Listing 2.2.1: Code for verifying the order of accuracy of the forward, backward, and central difference approximations to the first derivative of an analytic function.

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

def uexact(x):
    return 1 + (np.sin(2*x))**3

def guexact(x):
    return 3*(np.sin(2*x))**2 * np.cos(2*x) * 2

def printfig(f, fname):
    plt.figure(f.number); plt.grid()
    plt.tick_params(axis='both', labelsize=12)
    f.tight_layout(); plt.show(block=False); plt.savefig(fname)

def main():
    x0 = 0.5  # point at which derivative is approximated
    dx = 0.1  # initial delta x
    N = 6     # number of dx values to consider
    E = np.zeros((3,N))
    dxv = np.zeros((N,1));
    tauB = np.zeros((N,1));
    tauF = np.zeros((N,1))
    tauC = np.zeros((N,1))
    gu0 = guexact(x0)
    for n in range(N):
        dxv[n] = dx
        x = np.array([x0-dx, x0, x0+dx])
        u = uexact(x)
        tauB[n] = (u[1]-u[0])/(x[1]-x[0]) - gu0
        tauF[n] = (u[2]-u[1])/(x[2]-x[1]) - gu0
        tauC[n] = (u[2]-u[0])/(x[2]-x[0]) - gu0
        dx /= 2
    f = plt.figure(figsize=(6,4))
    rate = np.log2(tauF[N-2]/tauF[N-1])
    plt.loglog(dxv, abs(tauF), linewidth=2, color='black', label='Forward: %.2f'%(rate))
    rate = np.log2(tauB[N-2]/tauB[N-1])
    plt.loglog(dxv, abs(tauB), linewidth=2, color='blue', label='Backward: %.2f'%(rate))
    rate = np.log2(tauC[N-2]/tauC[N-1])
    plt.loglog(dxv, abs(tauC), linewidth=2, color='red', label='Central: %.2f'%(rate))
    plt.legend(fontsize=12, loc=2, borderaxespad=0.1)
    plt.xlabel(r'$\Delta x$', fontsize=16)
    plt.ylabel(r'truncation error, $\tau$', fontsize=16)
    printfig(f, '../figs/pverify.pdf')
    plt.close(f)

if __name__ == "__main__":
    main()
```

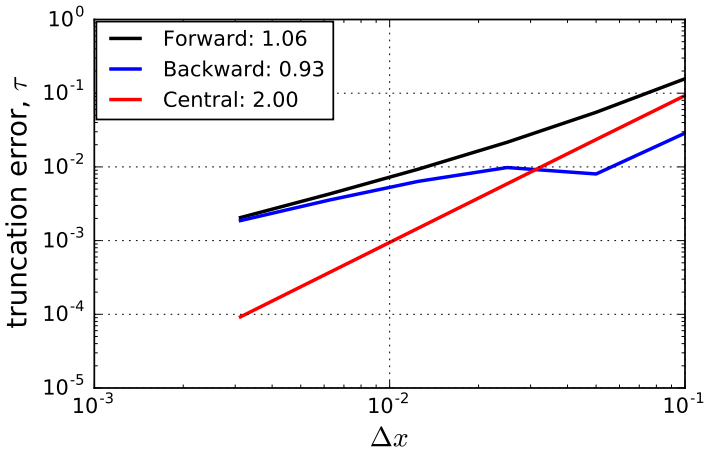


Figure 2.2.1: Convergence of the truncation error for forward, backward, and central difference approximations to the first derivative of a known function.

Example 2.3 (An approximation to the fourth derivative)**.** Let's derive a difference approximation to $\frac{d^4u}{dx^4}$ at node j by applying the second-order central difference approximation twice. The answer will involve nodes $j-2, j-1, j, j+1, j+2$.

At node j in our finite difference grid,

$$\begin{aligned}
\frac{d^4u}{dx^4}(x_j) &\approx \frac{\frac{d^2u}{dx^2}(x_{j+1}) - 2\frac{d^2u}{dx^2}(x_j) + \frac{d^2u}{dx^2}(x_{j-1})}{\Delta x^2} \\
&\approx \frac{(u_{j+2} - 2u_{j+1} + u_j) - 2(u_{j+1} - 2u_j + u_{j-1}) + (u_j - 2u_{j-1} + u_{j-2})}{\Delta x^4} \\
&= \frac{u_{j+2} - 4u_{j+1} + 6u_j - 4u_{j-1} + u_{j-2}}{\Delta x^4}
\end{aligned}$$

To determine the order of accuracy, we use Taylor-series expansions,

$$\begin{aligned}
u_{j-2} &= u_j - 2\Delta x u_x + \frac{1}{2}(2\Delta x)^2 u_{xx} - \frac{1}{6}(2\Delta x)^3 u_{xxx} + \frac{1}{24}(2\Delta x)^4 u_{xxxx} \\
&\quad - \frac{1}{120}(2\Delta x)^5 u_{xxxxx} + \mathcal{O}(\Delta x^6) \\
u_{j-1} &= u_j - 1\Delta x u_x + \frac{1}{2}(1\Delta x)^2 u_{xx} - \frac{1}{6}(1\Delta x)^3 u_{xxx} + \frac{1}{24}(1\Delta x)^4 u_{xxxx} \\
&\quad - \frac{1}{120}(1\Delta x)^5 u_{xxxxx} + \mathcal{O}(\Delta x^6) \\
u_j &= u_j \\
u_{j+1} &= u_j + 1\Delta x u_x + \frac{1}{2}(1\Delta x)^2 u_{xx} + \frac{1}{6}(1\Delta x)^3 u_{xxx} + \frac{1}{24}(1\Delta x)^4 u_{xxxx} \\
&\quad + \frac{1}{120}(1\Delta x)^5 u_{xxxxx} + \mathcal{O}(\Delta x^6) \\
u_{j+2} &= u_j + 2\Delta x u_x + \frac{1}{2}(2\Delta x)^2 u_{xx} + \frac{1}{6}(2\Delta x)^3 u_{xxx} + \frac{1}{24}(2\Delta x)^4 u_{xxxx} \\
&\quad + \frac{1}{120}(2\Delta x)^5 u_{xxxxx} + \mathcal{O}(\Delta x^6)
\end{aligned}$$

Summing these together with the given coefficients yields,

$$\begin{aligned}
u_{j+2} - 4u_{j+1} + 6u_j - 4u_{j-1} + u_{j-2} &= 0u_j + 0u_x + 0u_{xx} + 0u_{xxx} + \\
&\quad (\Delta x)^4 u_{xxxx} + 0u_{xxxxx} + \mathcal{O}(\Delta x^6).
\end{aligned}$$

Since we have to divide by Δx^4, the leading error term that remains will be $\mathcal{O}(\Delta x^2)$ and so the finite difference approximation is second-order accurate.

2.3 General Finite-Difference Formulas

In the previous sections we presented a few common finite difference formulas that had simple interpretations related to the definition of the first derivative. Now we take a look at systematically deriving general finite difference formulas.

The problem statement reads as follows: given $l + r + 1$ consecutive points on a uniform grid, as illustrated in Figure 2.3.1, determine coefficients $a_j, -l \leq j \leq r$, such that the following summation (i.e. finite difference)

$$\sum_{j=-l}^{r} a_j u_j, \tag{2.3.1}$$

approximates a desired derivative quantity, usually at x_0. Note that the finite difference *stencil* consists of the point in question, $j = 0$, as well as l points to the left and r points to the right. For example, for a forward difference, $l = 0, r = 1$; for a backward difference, $l = 1, r = 0$, and for a central difference, $l = r = 1$.

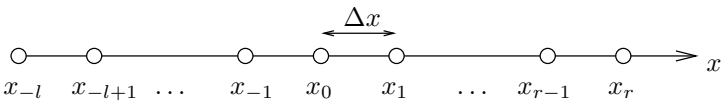


Figure 2.3.1: Points used in the derivation of a general finite-difference formula. The spacing between all consecutive nodes is Δx.

2.3.1 Undetermined Coefficients

In this method we insert into Equation 2.3.1 Taylor-series expansions for u_j about $x = x_0$,

$$u_j = u_0 + (x_j - x_0)\frac{du}{dx} + \frac{1}{2}(x_j - x_0)^2\frac{d^2u}{dx^2} + \dots.$$

We then group powers of Δx and choose a_j so that we obtain an approximation of the desired derivative with the maximum possible accuracy.

Example 2.4 (One-sided difference for $\frac{du}{dx}$)**.** How accurate of an approximation can we obtain for the first derivative, $\frac{du}{dx}$, at node 0 using information from only one side? In particular, let's use two points on the left and no points on the right: $l = 2$, $r = 0$. So the players are u_{-2}, u_{-1}, and u_0. We have (all derivatives are at x_0)

$$\begin{aligned} u_{-2} &= u_0 + (-2\Delta x)\frac{du}{dx} + \frac{1}{2}(-2\Delta x)^2\frac{d^2u}{dx^2} + \frac{1}{6}(-2\Delta x)^3\frac{d^3u}{dx^3} + \dots \\ u_{-1} &= u_0 + (-\Delta x)\frac{du}{dx} + \frac{1}{2}(-\Delta x)^2\frac{d^2u}{dx^2} + \frac{1}{6}(-\Delta x)^3\frac{d^3u}{dx^3} + \dots \end{aligned}$$

Inserting into Equation 2.3.1 and grouping powers of Δx leaves

$$\begin{aligned} \sum_{j=-l}^{r} a_j u_j &= a_{-2}u_{-2} + a_{-1}u_{-1} + a_0 u_0 \\ &= a_{-2}\left(u_0 + (-2\Delta x)\frac{du}{dx} + \frac{1}{2}(-2\Delta x)^2\frac{d^2u}{dx^2} + \frac{1}{6}(-2\Delta x)^3\frac{d^3u}{dx^3} + \dots\right) \\ &\quad + a_{-1}\left(u_0 + (-\Delta x)\frac{du}{dx} + \frac{1}{2}(-\Delta x)^2\frac{d^2u}{dx^2} + \frac{1}{6}(-\Delta x)^3\frac{d^3u}{dx^3} + \dots\right) \\ &\quad + a_0 u_0 \\ &= (a_{-2} + a_{-1} + a_0)u_0 \\ &\quad + (-2a_{-2} - a_{-1})\,\Delta x\frac{du}{dx} \\ &\quad + \left(2a_{-2} + \frac{1}{2}a_{-1}\right)\Delta x^2\frac{d^2u}{dx^2} \\ &\quad + \left(-\frac{4}{3}a_{-2} - \frac{1}{6}a_{-1}\right)\Delta x^3\frac{d^3u}{dx^3} \\ &\quad + \mathcal{O}(\Delta x^4) \end{aligned}$$

We want this sum to approximate the first derivative, $\frac{du}{dx}$, as accurately as possible. But we only have three knobs, a_{-2}, a_{-1}, a_0, so we can't get too ambitious about accuracy. Focusing on the first three terms above, we have the following equations:

$$\begin{aligned} a_{-2} + a_{-1} + a_0 &= 0 \\ -2a_{-2} - a_{-1} &= \frac{1}{\Delta x} \\ 2a_{-2} + \frac{1}{2}a_{-1} &= 0 \end{aligned}$$

Summing the last two of these, we obtain

$$-\frac{1}{2}a_{-1} = \frac{1}{\Delta x} \quad \Rightarrow \quad a_{-1} = -\frac{2}{\Delta x}$$

Substituting this result into the last equation,

$$a_{-2} = -\frac{1}{4}a_{-1} = \frac{1}{2\Delta x}.$$

Finally, substituting both results into the first equation gives

$$a_0 = -a_{-2} - a_{-1} = -\frac{1}{2\Delta x} + \frac{2}{\Delta x} = \frac{3}{2\Delta x}$$

So our one-sided approximation for the first derivative is

$$\frac{du}{dx} \approx \frac{3}{2\Delta x}u_0 + \frac{1}{2\Delta x}u_{-1} - \frac{2}{\Delta x}u_{-2} = \frac{3u_0 - 4u_{-1} + u_{-2}}{2\Delta x}$$

A quick check in the above equation with the Taylor-series expansions shows that the truncation error in this finite difference is $\mathcal{O}(\Delta x^2)$ – i.e. the difference is second-order accurate.

We can use a table to more concisely keep track of the Taylor expansion terms that appear in the method of undetermined coefficients. The following example repeats Example 2.4 with such a table.

Example 2.5 (Taylor-series table for $\frac{du}{dx}$)**.** The goal is to construct $\frac{du}{dx}$ at node 0 using u_{-2}, u_{-1}, and u_0. We perform Taylor-series expansions of these quantities about node 0 and arrange terms into columns based on the power of Δx. Table 2.3.1 shows the result.

The last row in this table is the desired sum of the rows in each column, using the multiplicative factors a_{-2}, a_{-1}, and a_0. Since we are after $\frac{du}{dx}$, we want the sum in this column, s_1, to be $\frac{du}{dx}$. The sums on the lower-order terms (columns to the left) must be zero to ensure a consistent finite-difference formulas. Otherwise, we would not observe convergence as $\Delta x \to 0$. In addition, we would like as many higher-order sums (columns

Table 2.3.1: Taylor-series table for calculating a one-sided finite difference approximation to $\frac{du}{dx}$ at node 0. All derivatives are are calculated about node 0.

Factor	Quantity	u_0	$\Delta x \frac{du}{dx}$	$\Delta x^2 \frac{d^2u}{dx^2}$	$\Delta x^3 \frac{d^3u}{dx^3}$
a_{-2}	u_{-2}	1	-2	2	$-\frac{4}{3}$
a_{-1}	u_{-1}	1	-1	$\frac{1}{2}$	$-\frac{1}{6}$
a_0	u_0	1	0	0	0
Desired sums:		$s_0 = 0$	$s_1 = \frac{du}{dx}$	$s_2 = 0$	s_3

to the right) as possible to be zero, since each additional zero terms increases the order of accuracy by one. The result is precisely the system of equations in the previous example.

To determine the order of accuracy, we check whether or not s_3 is zero. From Table 2.3.1, s_3 is the sum of the first higher-order column we did not explicitly force to be zero. It is possible that this term is zero, in which case we obtain a fortuitous extra order of accuracy. However, in our case,

$$s_3 = \left(-\frac{4}{3}a_{-2} - \frac{1}{6}a_{-1}\right)\Delta x^3 \frac{d^3u}{dx^3} = \left(-\frac{2}{3\Delta x} + \frac{1}{3\Delta x}\right)\Delta x^3 \frac{d^3u}{dx^3} = -\frac{1}{3}\Delta x^2 \frac{d^3u}{dx^3}.$$

The presence of this $\mathcal{O}(\Delta x^2)$ term indicates that the finite difference is second-order accurate.

2.3.2 Lagrange Interpolation

In this method, we derive a finite difference approximation by first fitting a polynomial of order $l+r$ to the u-values at the $l+r+1$ points. We do this by Lagrange interpolation. We then calculate the desired derivative by differentiating the interpolated polynomial.

Specifically, the order $l+r$ polynomial that interpolates values u_j at the $l+r+1$ points x_j is

$$\tilde{u}(x) = \sum_{j=-l}^{r} L_j(x) u_j, \quad \text{where} \quad L_j(x) = \prod_{i \neq j} \frac{x - x_i}{x_j - x_i}. \tag{2.3.2}$$

Note that the Lagrange polynomial $L_j(x)$ is 1 at node x_j and 0 at all of the other nodes, as shown in Figure 2.3.2. Once we have $\tilde{u}(x)$ we can differentiate it at $x = x_0$ to obtain the desired derivative.

Example 2.6 (Derivation of the central difference for $\frac{du}{dx}$)**.** Using $l = r = 1$, let's see if we can use Lagrange interpolation to derive the central difference formula that we presented in

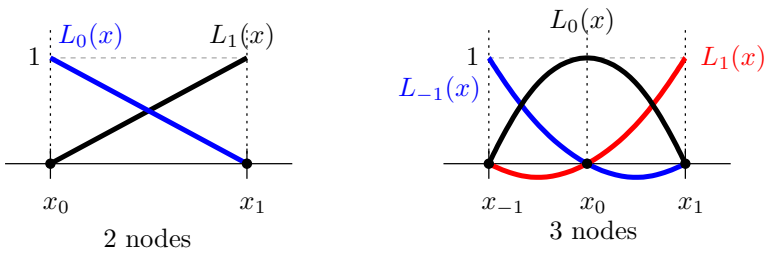


Figure 2.3.2: Lagrange interpolating polynomials for two and three nodes.

Equation 2.1.4. The three Lagrange interpolating functions are

$$\begin{aligned} L_{-1}(x) &= \frac{(x-x_0)(x-x_1)}{(-\Delta x)(-2\Delta x)} \qquad 1 \text{ at node } j=-1 \\ L_0(x) &= \frac{(x-x_{-1})(x-x_1)}{(\Delta x)(-\Delta x)} \qquad 1 \text{ at node } j=0 \\ L_1(x) &= \frac{(x-x_{-1})(x-x_0)}{(2\Delta x)(\Delta x)} \qquad 1 \text{ at node } j=1 \end{aligned}$$

The polynomial

$$\tilde{u}(x) = \sum_{j=-1}^{1} L_j(x) u_j$$

will then interpolate the three states u_{-1}, u_0, u_1. To obtain the first derivative at $x = x_0$ we differentiate this polynomial and evaluate at $x = x_0$,

$$\left.\frac{d\tilde{u}(x)}{dx}\right|_{x_0} = \sum_{j=-1}^{1} \left.\frac{dL_j(x)}{dx}\right|_{x_0} u_j$$

The derivatives of the Lagrange polynomials are

$$\begin{aligned} \left.\frac{dL_{-1}}{dx}\right|_{x_0} &= \left.\frac{2x - x_1 - x_0}{2\Delta x^2}\right|_{x_0} = \frac{-1}{2\Delta x} \\ \left.\frac{dL_0}{dx}\right|_{x_0} &= \left.\frac{2x - x_1 - x_{-1}}{-\Delta x^2}\right|_{x_0} = 0 \\ \left.\frac{dL_1}{dx}\right|_{x_0} &= \left.\frac{2x - x_0 - x_{-1}}{2\Delta x^2}\right|_{x_0} = \frac{1}{2\Delta x} \end{aligned}$$

So our finite difference approximation is

$$\begin{aligned} \left.\frac{du}{dx}\right|_{x_0} \approx \left.\frac{d\tilde{u}(x)}{dx}\right|_{x_0} = \sum_{j=-1}^{1} \left.\frac{dL_j(x)}{dx}\right|_{x_0} u_j &= \frac{-1}{2\Delta x} u_{-1} + 0 u_0 + \frac{1}{2\Delta x} u_1 \\ &= \frac{u_1 - u_{-1}}{2\Delta x}, \end{aligned}$$

and we see that we have indeed recovered our central difference result. We don't directly get the order of accuracy using Lagrange interpolation, but we can reason as follows: we fit a quadratic function of x to our three points so the slope is going to be a linear function of x, which has an error (missing term) that behaves like $\mathcal{O}(\Delta x^2)$. In general, however, getting the order of accuracy is more straightforward when using Taylor-series expansions.

The calculation of Lagrange polynomials by hand can become tedious, and the derivatives are even more complicated. Differentiating Equation 2.3.2 using the product rule, we have that the derivative of the j^{th} Lagrange basis function is

$$\frac{dL_j}{dx} = \sum_{k \neq j} \frac{1}{x_j - x_k} \prod_{i \neq j, i \neq k} \frac{x - x_i}{x_j - x_i}. \tag{2.3.3}$$

When working with many Lagrange nodes (i.e. high-order polynomials), a computer program is helpful in evaluating the basis functions and their derivatives. Listing 2.3.1 presents two functions that do this, taking as input a vector of Lagrange node coordinates (`xn`) and a scalar value of x at which the basis or derivatives should be evaluated. The output of each function is a vector.

Listing 2.3.1: Functions for computing Lagrange basis values and derivatives.

```
1  import numpy as np
2
3  # Computes Lagrange basis function values
4  def basis(xn, x):
5      N = len(xn)
6      phi = np.zeros(N)
7      for j in range(N):
8          pj = 1.0
9          for i in range(N):
10             if (i==j): continue
11             pj *= (x-xn[i])/(xn[j]-xn[i])
12         phi[j] = pj
13     return phi
14
15 # Computes Lagrange basis gradients
16 def gbasis(xn, x):
17     N = len(xn)
18     gphi = np.zeros(N)
19     for j in range(N):
20         gphi[j] = 0.
21         for k in range(N):
22             if (k==j): continue
23             gj = 1.0/(xn[j]-xn[k])
24             for i in range(N):
25                 if (i==j) or (i==k): continue
26                 gj *= (x-xn[i])/(xn[j]-xn[i])
```

```
            gphi[j] += gj
    return gphi
```

Listing 2.3.2 shows how the first of these functions, `basis`, can be used to make a plot of the Lagrange basis functions for an arbitrary order. This function loops over a large number of points in the range over which the basis functions are sought, evaluates the function values, stores them in an array, and then plots them using different colors. Figure 2.3.3 shows the resulting plot. We could similarly plot basis gradients, or evaluate the gradients at specific points to design finite difference formulas.

Listing 2.3.2: Code for plotting Lagrange basis functions.

```
import numpy as np
import matplotlib.pyplot as plt
from lagrange import basis,gbasis

def printfig(f, fname):
    plt.figure(f.number); plt.grid()
    plt.tick_params(axis='both', labelsize=12)
    f.tight_layout(); plt.show(block=False); plt.savefig(fname)

def main():
    L = 1; N = 6; Np = 200  # length of domain, num nodes and plot nodes
    xn = np.linspace(0,L,N)  # node locations
    xp = np.linspace(0,L,Np) # plotting points
    Phi = np.zeros((Np,N))
    for n in range(Np): Phi[n,:] = basis(xn, xp[n])  # basis values
    f = plt.figure( figsize=(8,3))
    colors = ['black', 'red', 'blue', 'green', 'magenta', 'cyan']
    for i in range(N): plt.plot(xp, Phi[:,i], linewidth=2, color=colors[i%6])
    plt.xlabel('$x$', fontsize=16)
    plt.ylabel('$L(x)$', fontsize=16)
    printfig(f, '../figs/plotlagrange.pdf')
    plt.close(f)

if __name__ == "__main__":
    main()
```

2.3.3 Non-Uniform Spacing

Grids used in CFD often have nodes that are not spaced uniformly. That is, the spacing between nodes is not the same for all pairs of adjacent nodes. Finite difference formulas for such grids will differ from those derived so far, as now Δx is not constant. However, the techniques introduced in this section for deriving finite-difference formulas still apply.

As an example, consider approximating the second derivative of u with respect to x using the value of u at three nodes, as shown in Figure 2.3.4. We are interested in the second derivative measured at the middle node, x_2: $\left.\frac{d^2u}{dx^2}\right|_{x_2}$.

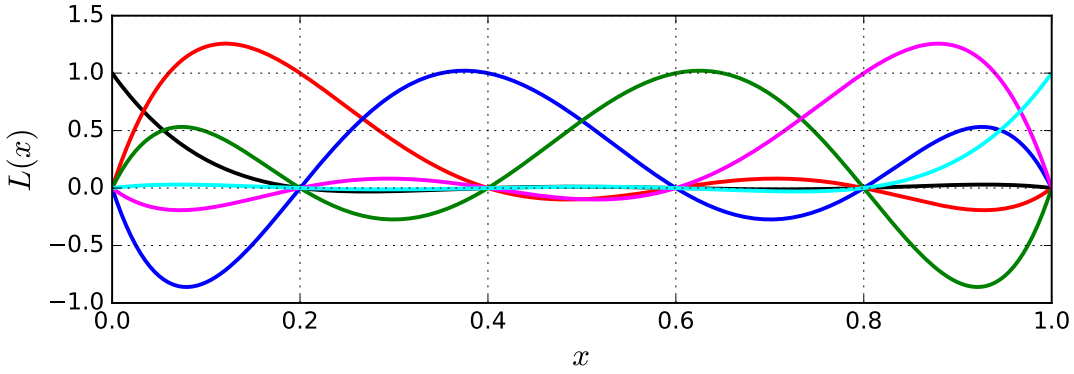


Figure 2.3.3: Lagrange interpolating polynomials for two and three nodes.

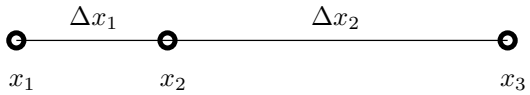


Figure 2.3.4: Three non-uniformly spaced nodes.

To obtain the most-accurate 3-point finite-difference stencil for the second derivative, we perform a Taylor-series expansion about x_2. In the following formulas, all derivatives are taken at x_2.

$$\begin{aligned} u_1 &= u_2 - \Delta x_1 u_x + \frac{1}{2}(\Delta x_1)^2 u_{xx} - \frac{1}{6}(\Delta x_1)^3 u_{xxx} + \mathcal{O}(\Delta x_1^4) \\ u_2 &= u_2 \\ u_3 &= u_2 + \Delta x_2 u_x + \frac{1}{2}(\Delta x_2)^2 u_{xx} + \frac{1}{6}(\Delta x_2)^3 u_{xxx} + \mathcal{O}(\Delta x_2^4) \end{aligned}$$

Consider now a finite difference formula for $\left.\frac{d^2u}{dx^2}\right|_{x_2}$ using the combination $au_1 + bu_2 + cu_3$. Substituting the above expansions and collecting terms,

$$\begin{aligned} au_1 + bu_2 + cu_3 &= (a+b+c)u_2 + \\ &\quad (-a\Delta x_1 + c\Delta x_2)u_x + \\ &\quad (\frac{1}{2}a\Delta x_1^2 + \frac{1}{2}c\Delta x_2^2)u_{xx} + \\ &\quad (-\frac{1}{6}a\Delta x_1^3 + \frac{1}{6}c\Delta x_2^3)u_{xxx} + \\ &\quad \mathcal{O}(\Delta x_1^4) + \mathcal{O}(\Delta x_2^4) \end{aligned}$$

Isolating the second derivative gives the following linear system

$$\begin{aligned} a+b+c &= 0 \\ -a\Delta x_1 + c\Delta x_2 &= 0 \\ \frac{1}{2}a\Delta x_1^2 + \frac{1}{2}c\Delta x_2^2 &= 1 \end{aligned}$$

The second equation gives $c = a\Delta x_1/\Delta x_2$, from which the first gives $b = -a - c = -a - a\Delta x_1/\Delta x_2$, and the third equation gives,

$$\begin{aligned} \frac{1}{2}a\Delta x_1^2 + \frac{1}{2}a\frac{\Delta x_1}{\Delta x_2}\Delta x_2^2 &= 1 \\ a(\Delta x_1^2 + \Delta x_1\Delta x_2) &= 2 \\ a &= \frac{2}{\Delta x_1\left(\Delta x_1 + \Delta x_2\right)} \\ \Rightarrow c &= \frac{2}{\Delta x_2\left(\Delta x_1 + \Delta x_2\right)} \\ \Rightarrow b &= -\frac{2\Delta x_1 + 2\Delta x_2}{\Delta x_1\Delta x_2\left(\Delta x_1 + \Delta x_2\right)} = -\frac{2}{\Delta x_1\Delta x_2} \end{aligned}$$

So the finite difference approximation is

$$\begin{aligned} \left.\frac{d^2u}{dx^2}\right|_{x_2} &\approx au_1 + bu_2 + cu_3 \\ &= \frac{2}{\Delta x_1\left(\Delta x_1 + \Delta x_2\right)}u_1 - \frac{2}{\Delta x_1\Delta x_2}u_2 + \frac{2}{\Delta x_2\left(\Delta x_1 + \Delta x_2\right)}u_3 \\ &= \frac{\Delta x_2 u_1 - (\Delta x_1 + \Delta x_2)u_2 + \Delta x_2 u_3}{\Delta x_1\Delta x_2\left(\Delta x_1 + \Delta x_2\right)/2} \end{aligned}$$

When $\Delta x_1 = \Delta x_2$ we recover the standard second-order accurate central-difference formula in Equation 2.1.5.

2.4 Multiple Dimensions

Both the method of undetermined coefficients and Lagrange interpolation extend to multiple dimensions. In two dimensions, a finite difference formula will typically involve a rectangular grid of points, as illustrated in Figure 2.4.1. Note that the center point is $i = 0, j = 0$ – there are l columns of points to the left, r columns to the right, d rows below, and u rows above. The total number of points is $(l + r + 1)(u + d + 1)$. We then use a weighted sum over these points to approximate a desired derivative quantity, such as the following mixed derivative,

$$\left.\frac{\partial^{m+n}u}{\partial x^m\partial y^n}\right|_{x_0,y_0} \approx \sum_{i=-l}^{r}\sum_{j=-d}^{u} a_{ij}u_{ij}. \tag{2.4.1}$$

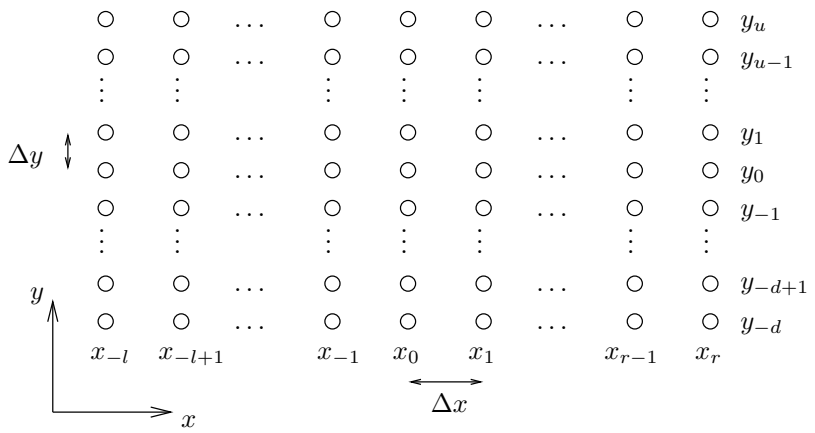


Figure 2.4.1: Points used in the derivation of a general finite-difference formula in two dimensions. The spacing between all consecutive nodes is Δx in the x direction and Δy in the y direction.

We now use either the method of undetermined coefficients or Lagrange interpolation in order to calculate the coefficients a_{ij}. When using undetermined coefficients, the Taylor-series expansion in multiple dimensions reads

$$\begin{aligned} u_{ij} &= u_{00} + (x_i - x_0)\frac{\partial u}{\partial x} + (y_j - y_0)\frac{\partial u}{\partial y} + \frac{1}{2}(x_i - x_0)^2\frac{\partial^2 u}{\partial x^2} \\ &\quad + \frac{1}{2}(y_j - y_0)^2\frac{\partial^2 u}{\partial y^2} + (x_i - x_0)(y_j - y_0)\frac{\partial^2 u}{\partial x \partial y} + \dots \end{aligned}$$

Note that all derivatives are computed at x_0, y_0. When using Lagrange interpolants, the Lagrange polynomial at each node is given by the product of one-dimensional Lagrange interpolants in x and y,

$$\begin{aligned} \tilde{u}(x,y) &= \sum_{j=-l}^{r}\sum_{k=-d}^{u} L_j^x(x) L_k^y(y) u_{jk} \\ L_i^x(x) &= \prod_{k\neq i} \frac{x - x_k}{x_i - x_k} \\ L_j^y(x) &= \prod_{k\neq j} \frac{y - y_k}{y_j - y_k} \\ a_{jk} &= \frac{\partial^{m+n} u}{\partial x^m \partial y^n}\left[L_i^x(x) L_j^y(x)\right] \end{aligned}$$

Two-dimensional problems require more points and hence are more expensive to set up and solve compared to one-dimensional problems. However, the basic procedure for setting up the linear system remains unchanged: each node contributes one unknown and one equation.

Such finite differences are useful when we have a uniform distribution of points, i.e. when the domain is rectangular. For non-rectangular domains, more general finite difference formulas that do not rely on equal spacing of the points can be used, as discussed in Section 2.5.

Example 2.7 (Mixed derivative in two dimensions)**.** Consider approximating the mixed derivative $\frac{\partial^2 u}{\partial x \partial y}$ in 2D. We will use the 4-point stencil shown in Figure 2.4.2 (the 4 shaded corner points), and we are interested in the mixed derivative at the middle node (0,0). Note, the integer pairs are indices that identify the nodes. The spacing between nodes is Δx horizontally and Δy vertically. The desired form for the mixed derivative is

$$\left.\frac{\partial^2 u}{\partial x \partial y}\right|_{0,0} = a u_{1,1} + b u_{-1,-1} + c u_{1,-1} + d u_{-1,1}.$$

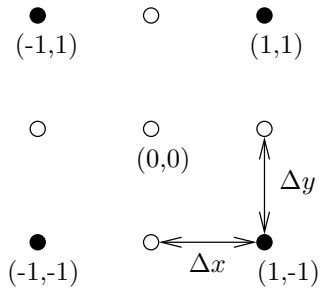


Figure 2.4.2: Stencil for approximating a mixed derivative in two dimensions.

A Taylor-series expansion of u at node (i, j), about node (0,0), is

$$\begin{aligned} u_{ij} \;=\; & u_{00} + (x_i - x_0)u_x + (y_j - y_0)u_y + \frac{1}{2}(x_i - x_0)^2 u_{xx} + \frac{1}{2}(y_j - y_0)^2 u_{yy} \\ & + (x_i - x_0)(y_j - y_0)u_{xy} + \frac{1}{6}(x_i - x_0)^3 u_{xxx} + \frac{1}{2}(x_i - x_0)^2(y_j - y_0)u_{xxy} \\ & + \frac{1}{2}(x_i - x_0)(y_j - y_0)^2 u_{xyy} + \frac{1}{6}(y_j - y_0)^3 u_{yyy} + \ldots \end{aligned}$$

Using this expansion, we have

$$\begin{aligned}
u_{1,1} &= u_{0,0} + \Delta x u_x + \Delta y u_y + \frac{1}{2}\Delta x^2 u_{xx} + \frac{1}{2}\Delta y^2 u_{yy} + \Delta x \Delta y u_{xy} \\
&\quad + \frac{1}{6}\Delta x^3 u_{xxx} + \frac{1}{6}\Delta y^3 u_{yyy} + \frac{1}{2}\Delta x^2 \Delta y u_{xxy} + \frac{1}{2}\Delta x \Delta y^2 u_{xyy} \\
u_{-1,-1} &= u_{00} - \Delta x u_x - \Delta y u_y + \frac{1}{2}\Delta x^2 u_{xx} + \frac{1}{2}\Delta y^2 u_{yy} + \Delta x \Delta y u_{xy} \\
&\quad - \frac{1}{6}\Delta x^3 u_{xxx} - \frac{1}{6}\Delta y^3 u_{yyy} - \frac{1}{2}\Delta x^2 \Delta y u_{xxy} - \frac{1}{2}\Delta x \Delta y^2 u_{xyy} \\
u_{1,-1} &= u_{0,0} + \Delta x u_x - \Delta y u_y + \frac{1}{2}\Delta x^2 u_{xx} + \frac{1}{2}\Delta y^2 u_{yy} - \Delta x \Delta y u_{xy} \\
&\quad + \frac{1}{6}\Delta x^3 u_{xxx} - \frac{1}{6}\Delta y^3 u_{yyy} - \frac{1}{2}\Delta x^2 \Delta y u_{xxy} + \frac{1}{2}\Delta x \Delta y^2 u_{xyy} \\
u_{-1,1} &= u_{00} - \Delta x u_x + \Delta y u_y + \frac{1}{2}\Delta x^2 u_{xx} + \frac{1}{2}\Delta y^2 u_{yy} - \Delta x \Delta y u_{xy} \\
&\quad - \frac{1}{6}\Delta x^3 u_{xxx} + \frac{1}{6}\Delta y^3 u_{yyy} + \frac{1}{2}\Delta x^2 \Delta y u_{xxy} - \frac{1}{2}\Delta x \Delta y^2 u_{xyy}
\end{aligned}$$

We want to find constants a, b, c, d such that

$$a u_{1,1} + b u_{-1,-1} + c u_{1,-1} + d u_{-1,1} = u_{xy}$$

to the maximum possible accuracy (i.e. we want to cancel as many terms in the Taylor series as possible). Using the above equations, we have:

$$\begin{aligned}
1: \quad & a + b + c + d &&= 0 \\
\Delta x: \quad & a - b + c - d &&= 0 \\
\Delta y: \quad & a - b - c + d &&= 0 \\
\Delta x^2: \quad & a + b + c + d &&= 0 \\
\Delta y^2: \quad & a + b + c + d &&= 0 \\
\Delta x \Delta y: \quad & a + b - c - d &&= \frac{1}{\Delta x \Delta y}
\end{aligned}$$

We didn't go any further because we already have enough equations. It actually looks like we have too many equations (6 for 4 unknowns), but fortunately 3 equations are the same ($a + b + c + d = 0$). Solving the 4 independent equations (e.g. a matrix solve), we obtain,

$$a = \frac{1}{4\Delta x \Delta y}, \quad b = \frac{1}{4\Delta x \Delta y}, \quad c = \frac{-1}{4\Delta x \Delta y}, \quad d = \frac{-1}{4\Delta x \Delta y}$$

To determine the order of accuracy, we need to find the first terms in the Taylor series that do not cancel. Using the calculated a, b, c, d we see that all of the additional terms we have written in the above Taylor-series expansions cancel. Writing more terms in a multi-dimensional Taylor series expansion gets messy, but we can reason as follows: terms that

do *not* cancel will be ones where Δx and Δy appear to odd powers, since the sum of these involves $a+b-c-d$, which by construction is not zero (see the above equations for a, b, c, d). The first of these terms was $\Delta x \Delta y$, which had the u_{xy} term we were after. The next terms will be $\Delta x^3 \Delta y$ and $\Delta x \Delta y^3$, and these will give the leading-error terms. Since a, b, c, d have $\Delta x \Delta y$ in the denominator, the leading error terms are

$$\text{first un-canceled terms: } \mathcal{O}\left(\frac{\Delta x^3 \Delta y}{\Delta x \Delta y}\right) = \mathcal{O}\left(\Delta x^2\right) \quad \text{and} \quad \mathcal{O}\left(\frac{\Delta x \Delta y^3}{\Delta x \Delta y}\right) = \mathcal{O}\left(\Delta y^2\right)$$

So the calculated finite-difference approximates the mixed derivative to second-order accuracy in the x and y spacing.

2.5 Non-Rectangular Domains

Practical computational domains of interest are rarely rectangular, and on these domains an irregular spacing of points prevents the direct application of standard finite difference formulas. One option is to derive formulas for non-uniform grids, as done in one spatial dimension in Section 2.3.3. However, in two and three spatial dimensions, this becomes a tedious process. A powerful alternative exists when one can map the irregular domain to a regular domain, such as a unit square. Figure 2.5.1 illustrates such a mapping. In

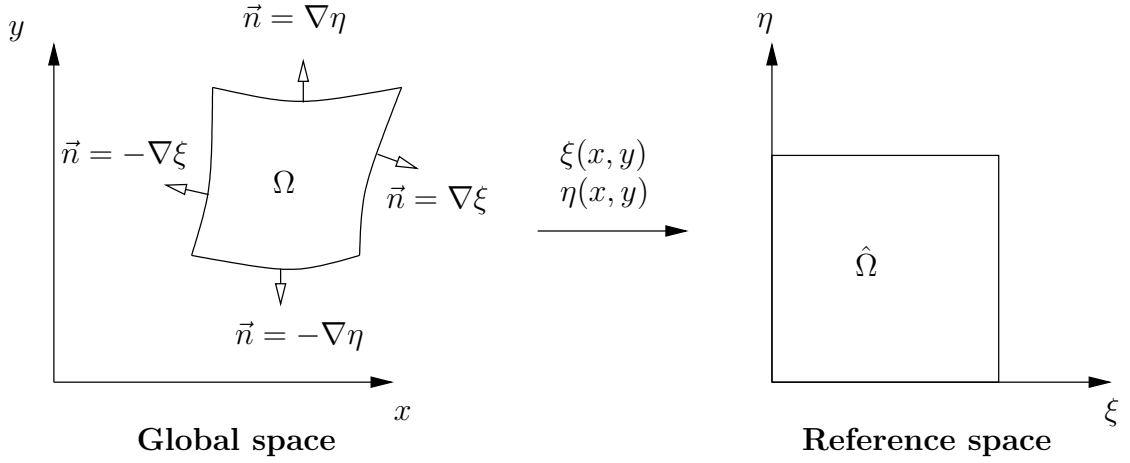


Figure 2.5.1: Mapping between global and reference spaces for transforming finite differences.

this mapping, x, y are the coordinate in *global space*, which is the space that contains our irregular domain on which we want to approximate finite differences. Rather than putting points down arbitrarily in this irregular domain (which may be difficult in itself), we instead put down a uniform grid of points on the unit square in *reference space*. In this space, the coordinates are ξ, η.

We use the chain rule to transform derivatives from global space to reference space. For example, derivatives with respect to x and y transform as

$$\frac{\partial}{\partial x} = \xi_x \frac{\partial}{\partial \xi} + \eta_x \frac{\partial}{\partial \eta} \qquad \frac{\partial}{\partial y} = \xi_y \frac{\partial}{\partial \xi} + \eta_y \frac{\partial}{\partial \eta}$$

This means that if we have some scalar quantity $u(x, y)$, we can express global-space derivatives of u in terms of local-space derivatives via,

$$\begin{aligned}
u_x &= u_\xi \xi_x + u_\eta \eta_x \\
u_y &= u_\xi \xi_y + u_\eta \eta_y \\
u_{xx} &= \frac{\partial}{\partial x}(u_x) = \frac{\partial}{\partial x}(u_\xi \xi_x + u_\eta \eta_x) \\
&= u_{\xi x}\xi_x + u_\xi \xi_{xx} + u_{\eta x}\eta_x + u_\eta \eta_{xx} \\
&= (\xi_x u_{\xi\xi} + \eta_x u_{\xi\eta})\,\xi_x + u_\xi \xi_{xx} + (\xi_x u_{\eta\xi} + \eta_x u_{\eta\eta})\,\eta_x + u_\eta \eta_{xx} \\
&= u_{\xi\xi}\xi_x^2 + 2u_{\xi\eta}\xi_x\eta_x + u_{\eta\eta}\eta_x^2 + u_\xi \xi_{xx} + u_\eta \eta_{xx} \\
u_{yy} &= \frac{\partial}{\partial y}(u_y) = \frac{\partial}{\partial y}(u_\xi \xi_y + u_\eta \eta_y) \\
&= u_{\xi y}\xi_y + u_\xi \xi_{yy} + u_{\eta y}\eta_y + u_\eta \eta_{yy} \\
&= (\xi_y u_{\xi\xi} + \eta_y u_{\xi\eta})\,\xi_y + u_\xi \xi_{yy} + (\xi_y u_{\eta\xi} + \eta_y u_{\eta\eta})\,\eta_y + u_\eta \eta_{yy} \\
&= u_{\xi\xi}\xi_y^2 + 2u_{\xi\eta}\xi_y\eta_y + u_{\eta\eta}\eta_y^2 + u_\xi \xi_{yy} + u_\eta \eta_{yy}
\end{aligned}$$

Note that these expressions no longer have derivatives of u with respect to x or y. Instead, all differentiation with respect to x and y is offloaded to the mapping, $\xi(x, y)$ and $\eta(x, y)$. This mapping is assumed known, so its derivatives can be calculated. Then, for the finite difference approximation of the derivatives with respect to ξ and η, e.g. $u_{\xi\xi}, u_{\xi\eta}$, etc., standard formulas can be used on the regular and uniform node spacing in reference space.

In some cases, we do not have explicit expressions for $\xi(x, y)$ or $\eta(x, y)$, but instead have the inverse $x(\xi, \eta)$ and $y(\xi, \eta)$. In these cases we need to invert the mapping, and this can be done locally, using the chain rule. Given an infinitesimal displacement $(d\xi, d\eta)$ in reference space, the corresponding displacement in global space is

$$dx = x_\xi d\xi + x_\eta d\eta \tag{2.5.1}$$

$$dy = y_\xi d\xi + y_\eta d\eta \tag{2.5.2}$$

We can write this transformation, and the corresponding one in the opposite direction, in matrix form,

$$\begin{bmatrix} dx \\ dy \end{bmatrix} = \underbrace{\begin{bmatrix} x_\xi & x_\eta \\ y_\xi & y_\eta \end{bmatrix}}_{\text{Jacobian matrix}} \begin{bmatrix} d\xi \\ d\eta \end{bmatrix} \qquad \begin{bmatrix} d\xi \\ d\eta \end{bmatrix} = \underbrace{\begin{bmatrix} \xi_x & \xi_y \\ \eta_x & \eta_y \end{bmatrix}}_{\text{inverse Jacobian matrix}} \begin{bmatrix} dx \\ dy \end{bmatrix}$$

The underbraces define the mapping *Jacobian* matrix, $\mathbf{J}$, and its inverse, $\mathbf{J}^{-1}$. Taking the inverse of a two by two matrix, we have

$$\Rightarrow \begin{bmatrix} \xi_x & \xi_y \\ \eta_x & \eta_y \end{bmatrix} = \begin{bmatrix} x_\xi & x_\eta \\ y_\xi & y_\eta \end{bmatrix}^{-1} = \frac{1}{J}\begin{bmatrix} y_\eta & -x_\eta \\ -y_\xi & x_\xi \end{bmatrix}, \qquad J \equiv \det(\mathbf{J}) = x_\xi y_\eta - y_\xi x_\eta$$

This expression can be used to compute derivatives of ξ and η with respect to x and y, using derivatives of x and y with respect to ξ and η.

On non-rectangular domains, expressions for the normal vector on domain boundaries are often required to properly enforce certain boundary conditions. To obtain an expression for the normal vector, we recognize that the normal vector is perpendicular to the physical domain boundary. Now, a physical domain boundary is level curve of constant η or constant ξ, since these are constant on the boundaries of the unit square in reference space. To obtain the vector that is perpendicular to the level curve of any function, we take the gradient of that function. So in our case, we need to take the gradient (with respect to x, y) of the functions $\xi(x, y)$ and $\eta(x, y)$ on the domain boundaries. For example, the direction of the normal vector on the top boundary, where $\eta(x, y) = 1$, is $\nabla\eta$. On the right boundary, where $\xi(x, y) = 1$, the normal vector is $\nabla\xi$. Figure 2.5.1 gives these expressions for all four boundaries. Note that $\nabla\eta = [\eta_x, \eta_y]^T$, and that these vectors should be normalized to give a unit normal vector.

We can integrate over the global domain by using the transformation from the reference domain. The integral of some function $f(\vec{x})$ transforms as

$$\int_{\Omega} f(\vec{x})\, dx\, dy \quad = \quad \int_{\hat{\Omega}} f(\vec{x}(\vec{\xi}))J\, d\xi\, d\eta. \tag{2.5.3}$$

Note the presence of the Jacobian matrix determinant, J, in the reference-space integral. J is the ratio of the elemental volume in global space to the elemental volume in reference space. This can be derived by considering how an infinitesimal rectangle of sides $d\xi$ and $d\eta$ in reference space transforms into global space. The result is a parallelogram between two vectors: the first is the mapping of $(d\xi, 0)$, which is $(x_\xi, y_\xi)d\xi$; the second is the mapping of $(0, d\eta)$, which is $(x_\eta, y_\eta)d\eta$. Taking the cross product of these global-space vectors gives the area of the parallelogram, $(x_\xi y_\eta - y_\xi x_\eta)d\xi d\eta = Jd\xi d\eta$.

Example 2.8 (A shear mapping to a parallelogram)**.** Consider the reference-to-global mapping given by

$$x = \xi + \frac{1}{2}\eta, \quad y = \eta, \tag{2.5.4}$$

as illustrated in Figure 2.5.2. The inverse of this mapping is

$$\xi = x - \frac{1}{2}y, \quad \eta = y. \tag{2.5.5}$$

The Jacobian matrix of the mapping is

$$\mathbf{J} = \begin{bmatrix} x_\xi & x_\eta \\ y_\xi & y_\eta \end{bmatrix} = \begin{bmatrix} 1 & \frac{1}{2} \\ 0 & 1 \end{bmatrix} \tag{2.5.6}$$

The determinant is $J = \det(\mathbf{J}) = 1$, which says that the mapping preserves areas, even though the shape gets distorted. The normal vector on the top of the global-space domain, where η is constant, is proportional to

$$\vec{n}_{\text{top}} \propto \nabla\eta = 0\hat{x} + 1\hat{y} \tag{2.5.7}$$

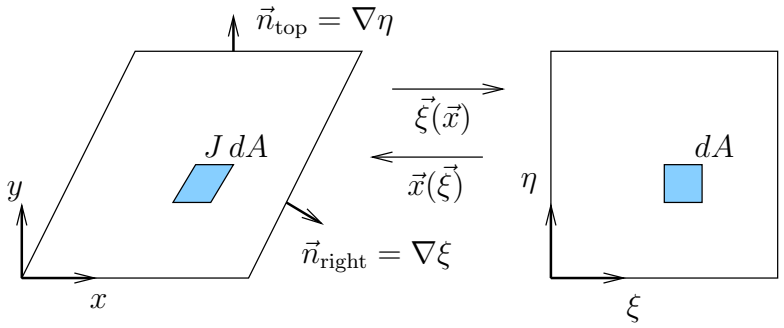


Figure 2.5.2: A simple shear mapping from reference to global space.

In this case, the proportionality constant is unity, since $\nabla\eta$ is already a unit vector. On the right boundary of the global-space domain, the normal is $\nabla\xi$,

$$\vec{n}_{\text{right}} \propto \nabla\xi = 1\hat{x} - \frac{1}{2}\hat{y}. \tag{2.5.8}$$

2.6 Compact Differences

To increase the order of accuracy of a finite difference approximation of the form in Equation 2.3.1, we need to use more points. This can be problematic on domain boundaries, where we only have points to one side. An alternative is to abandon the explicit formulas for the derivatives and instead to form a system of equations that we solve for the derivatives at all of the points at once. That is, we express the derivative at the point of interest in terms of not just the values at nearby points, but also in terms of the derivatives at the nearby points – this results in a system of equations because the derivatives at nearby points are not known. Formulas that give rise to such systems are called **compact differences**.

A popular compact difference in one dimension, on a uniform grid, involves u and du/dx at the two adjacent points. Let's call these points $-1, 0, 1$. The formula for the derivative at node 0 is then

$$\left.\frac{du}{dx}\right|_0 = a_{-1}u_{-1} + a_0u_0 + a_1u_1 + b_{-1}\left.\frac{du}{dx}\right|_{-1} + b_1\left.\frac{du}{dx}\right|_1 \tag{2.6.1}$$

We write Taylor-series expansions for all of the terms on the right hand side, resulting in Table 2.6.1

With four knobs $(a_{-1}, a_0, a_1, b_{-1}, b_1)$, we can enforce four constraints, which will be the

Table 2.6.1: Taylor-series expansions of terms appearing in the compact difference approximation to $du/dx|_0$.

Factor	Quantity	u_0	$\Delta x \left.\frac{du}{dx}\right\|_0$	$\Delta x^2 \left.\frac{d^2u}{dx^2}\right\|_0$	$\Delta x^3 \left.\frac{d^3u}{dx^3}\right\|_0$	$\Delta x^4 \left.\frac{d^4u}{dx^4}\right\|_0$	$\Delta x^5 \left.\frac{d^5u}{dx^5}\right\|_0$
a_{-1}	u_{-1}	1	-1	$\frac{1}{2}$	$-\frac{1}{6}$	$\frac{1}{24}$	$-\frac{1}{120}$
a_0	u_0	1	0	0	0	0	0
a_1	u_1	1	1	$\frac{1}{2}$	$\frac{1}{6}$	$\frac{1}{24}$	$\frac{1}{120}$
$b_{-1}/\Delta x$	$\Delta x \left.\frac{du}{dx}\right\|_{-1}$	0	1	-1	$\frac{1}{2}$	$-\frac{1}{6}$	$\frac{1}{24}$
$b_1/\Delta x$	$\Delta x \left.\frac{du}{dx}\right\|_1$	0	1	1	$\frac{1}{2}$	$\frac{1}{6}$	$\frac{1}{24}$
Desired sums:		$s_0 = 0$	$s_1 = \frac{du}{dx}$	$s_2 = 0$	$s_3 = 0$	$s_4 = 0$	s_5

first four sums in the table. The corresponding equations are, in matrix form

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ -1 & 0 & 1 & 1 & 1 \\ 1/2 & 0 & 1/2 & -1 & 1 \\ -1/6 & 0 & 1/6 & 1/2 & 1/2 \\ 1/24 & 0 & 1/24 & -1/6 & 1/6 \end{bmatrix} \begin{bmatrix} a_{-1} \\ a_0 \\ a_1 \\ b_{-1}/\Delta x \\ b_1/\Delta x \end{bmatrix} = \begin{bmatrix} 0 \\ 1/\Delta x \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

The solution to this system is

$$a_{-1} = -\frac{3}{4\Delta x}, \quad a_0 = 0, \quad a_1 = \frac{3}{4\Delta x}, \quad b_{-1} = -\frac{1}{4}, \quad b_1 = -\frac{1}{4}$$

So the compact difference expression is, from Equation 2.6.1,

$$\begin{aligned} \left.\frac{du}{dx}\right|_0 &= \frac{3}{4\Delta x}(-u_{-1} + u_1) + -\frac{1}{4}\left.\frac{du}{dx}\right|_{-1} - \frac{1}{4}\left.\frac{du}{dx}\right|_1 \\ \frac{1}{4}\left.\frac{du}{dx}\right|_{-1} + \left.\frac{du}{dx}\right|_0 + \frac{1}{4}\left.\frac{du}{dx}\right|_1 &= \frac{3}{4\Delta x}(-u_{-1} + u_1) \end{aligned}$$

When written in this last form, with all of the unknown derivatives on the left-hand side, we see that we have a system of equations to solve for the derivatives. Fortunately, this system is tridiagonal, and the solution requires only one forward and one backward sweep through the matrix.

Chapter 3

Boundary Value Problems

The finite difference approximations derived in Section 2 can be used to numerically solve partial differential equations. The resulting method is the **finite difference method** and is one of the commonly used techniques for solving general PDEs. In the finite difference method, the unknowns are the values of the solution at points, or nodes, in the domain. The equations come from enforcing the PDE at these points, using finite differences to approximate the derivatives. In this chapter we show how the method works for elliptic boundary value problems.

3.1 Examples

3.1.1 Heat Conduction in a One-Dimensional Bar

The equation governing heat conduction in a one-dimensional bar with a heat source term is

$$-\kappa \frac{d^2T}{dx^2} = q(x), \tag{3.1.1}$$

where $T(x)$ is the temperature distribution, κ is the (constant) thermal conductivity, and $q(x)$ is the heat source. The domain is $x \in [0, L]$. We construct a uniform grid of N intervals, $N+1$ points, with spacing $\Delta x = L/N$.

The unknowns are the temperature values at the nodes, T_i, where i indexes the nodes. The range of i depends on the boundary conditions. For now, we assume simple boundary conditions where the temperature is known at the bar endpoints, which are at nodes $i = 0$ and $i = N$. So the values of T_0 and T_N are known, and we are left with $N-1$ unknowns: $T_1, T_2, \ldots, T_{N-1}$.

To obtain a solvable system, we need the same number of equations as unknowns: $N-1$. In the finite difference method, these equations arise from a finite difference approximation of the PDE, Equation 3.1.1, at each *interior node*, $i = 1, 2, \ldots, N-1$. We saw in the previous chapter that finite difference approximations are not unique, and so we have to make a choice of which difference to use. A popular method arises from using a second-order

difference, i.e. Equation 2.1.5, to discretize the second derivative $\frac{d^2T}{dx^2}$. So the discretized form of Equation 3.1.1 becomes, at node x_i,

$$-\kappa \frac{T_{i-1} - 2T_i + T_{i+1}}{\Delta x^2} = q(x_i).$$

Rearranging the terms,

$$-T_{i-1} + 2T_i - T_{i+1} = \frac{1}{\kappa}\Delta x^2 q(x_i). \tag{3.1.2}$$

This equation holds at all interior nodes. The unknowns appear on the left-hand side, whereas the source term on the right-hand side is assumed known. A caveat is that near the boundaries, i.e. $i = 1$ or $i = N - 1$, the equations need to be rearranged slightly to put the known values of T_0 and T_N on the right-hand side:

$$\begin{aligned} \text{at } i = 1: \quad 2T_1 - T_2 &= \frac{1}{\kappa}\Delta x^2 q(x_1) + T_0 \\ \text{at } i = N-1: \quad -T_{N-2} + 2T_{N-1} &= \frac{1}{\kappa}\Delta x^2 q(x_{N-1}) + T_N \end{aligned}$$

Writing Equation 3.1.2 for all interior points yields a system of equations, which in matrix form reads

$$\begin{bmatrix} 2 & -1 & 0 & 0 & 0 & \dots & 0 \\ -1 & 2 & -1 & 0 & 0 & \dots & 0 \\ 0 & -1 & 2 & -1 & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & -1 & 2 & -1 & 0 \\ 0 & \dots & 0 & 0 & -1 & 2 & -1 \\ 0 & \dots & 0 & 0 & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \\ T_3 \\ \vdots \\ T_{N-3} \\ T_{N-2} \\ T_{N-1} \end{bmatrix} = \begin{bmatrix} q(x_1)\Delta x^2/\kappa + T_0 \\ q(x_2)\Delta x^2/\kappa \\ q(x_3)\Delta x^2/\kappa \\ \vdots \\ q(x_{N-3})\Delta x^2/\kappa \\ q(x_{N-2})\Delta x^2/\kappa \\ q(x_{N-1})\Delta x^2/\kappa + T_N \end{bmatrix} \tag{3.1.3}$$

Listing 3.1.1 shows a Python code that sets up and solves this system for a source term $q(x) = \sin(\pi x/L)$, and Figure 3.1.1 shows the temperature distribution for one set of parameters, using the driver code in Listing 3.1.2.

Listing 3.1.1: Computes the temperature in a heated 1D bar with Dirichlet boundary conditions.

```
import numpy as np
from scipy import sparse
from scipy.sparse import linalg

def heat1d(T0, TN, L, kappa, N):
    dx = float(L)/N                # spacing
    x = np.linspace(0, L, N+1)  # all nodes
    data = np.ones((3,N-1))     # diagonals for making a sparse matrix
    data[0,:] *= -1; data[1,:] *= 2; data[2,:] *= -1
    diags = np.array([-1, 0, 1])
    A = sparse.spdiags(data, diags, N-1, N-1, 'csr')
```

```
    q = np.sin(np.pi*x/L)          # source term
    b = q[1:N]*dx*dx/kappa         # right-hand side
    b[0] += T0; b[N-2] += TN       # Dirichlet boundary conditions
    Tt = linalg.spsolve(A,b)       # solution at interior points
    T = np.zeros(N+1)              # solution at all points
    T[0] = T0; T[1:N] = Tt; T[N] = TN
    return x, T

def main():
    x, T = heat1d(3, 4, 2.0, .5, 10)

if __name__ == "__main__":
    main()
```

Listing 3.1.2: Driver code for the function `heat1d`.

```
import numpy as np
import matplotlib.pyplot as plt
from heat1d import heat1d

def plotsol(x, T, fname):
    f = plt.figure( figsize =(8,3))
    plt.plot(x, T, 'o-', linewidth=2, color='blue')
    plt.xlabel(r'position, $x$', fontsize=16)
    plt.ylabel(r'Temperature, $T$', fontsize=16)
    plt.figure(f.number); plt.grid()
    plt.tick_params(axis='both', labelsize=12)
    f.tight_layout(); plt.show(block=False); plt.savefig(fname)
    plt.close(f)

def main():
    T0, TN, L, kappa = 1, 4, 2, 0.5
    x, T = heat1d(T0, TN, L, kappa, 10)
    plotsol(x, T, '../figs/heat1d_N10.pdf')

if __name__ == "__main__":
    main()
```

How good is our solution? The answer depends on the number of points we use. This is a simple problem and we know the exact solution. So let's measure accuracy by an L_2 error norm,

$$\epsilon = \sqrt{\sum_{i=0}^{N} \Delta x (T_i - T^{\text{exact}}(x_i))^2}, \tag{3.1.4}$$

where $T^{\text{exact}}(x)$ is the exact solution, given by

$$T^{\text{exact}}(x) \quad = \quad \frac{1}{\kappa \pi^2} \sin(\pi x) + T_0 + (T_N - T_0)x.$$

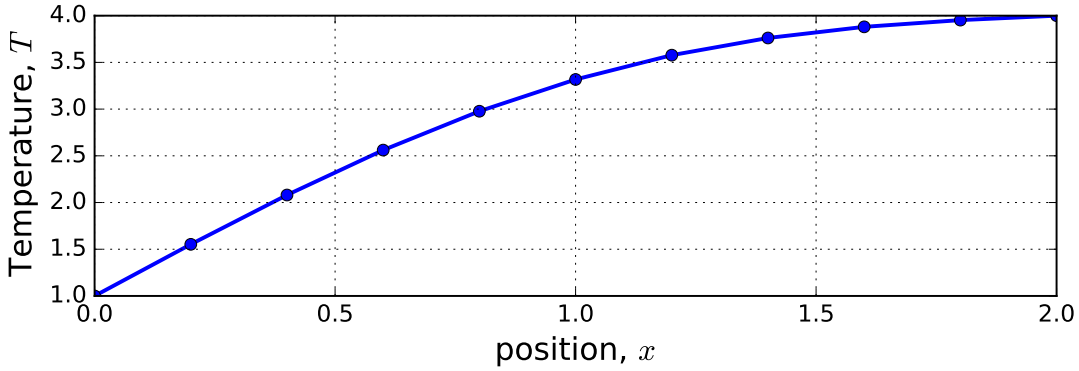


Figure 3.1.1: Sample finite-difference result for the temperature in a heated 1D bar, with $L = 2$, $q(x) = \sin(\pi x/L)$, $\kappa = 0.5$, $T_0 = 1$, $T_N = 4$, and $N = 10$.

Figure 3.1.2 shows the convergence of the L_2 error norm with mesh refinement, generated by the code in Listing 3.1.3. As indicated by the slope, second-order accuracy is achieved, i.e. $\epsilon \sim \mathcal{O}(\Delta x^2)$. This supports our analysis in Example 2.1.

Listing 3.1.3: Performs a convergence study for the one-dimensional heat conduction problem.

```
import numpy as np
import matplotlib.pyplot as plt
from heat1d import heat1d

def main():
    T0, TN, L, kappa, N = 1, 4, 2,  0.5,  4
    Nref = 5
    dxv = np.zeros(Nref); ev = np.zeros(Nref)
    for i in range(Nref):
        x, T = heat1d(T0, TN, L, kappa, N)
        dxv[i] = x[2]-x[1]
        Te = 1./(kappa*(np.pi/L)**2)*np.sin(np.pi*x/L) + T0 + (TN-T0)*x/L
        ev[i] = np.sqrt(dxv[i]*np.sum((T-Te)**2))
        N *= 2
    f = plt.figure( figsize =(8,4))
    rate = np.log2(ev[Nref-2]/ev[Nref-1])
    plt.loglog(dxv, ev, 'o-', linewidth=2, color='blue', label='rate = %.2f'%(rate))
    plt.xlabel(r'mesh spacing, $\Delta x$', fontsize=16)
    plt.ylabel(r'$L_2$ error norm', fontsize=16)
    plt.legend( fontsize =16, borderaxespad=0.1, loc=2)
    plt.figure(f.number); plt.grid()
    plt.tick_params(axis='both', labelsize =12)
    f.tight_layout (); plt.show(block=False)
    plt.savefig('../figs/heat1dconv.pdf')
    plt.close(f)

if __name__ == "__main__":
```

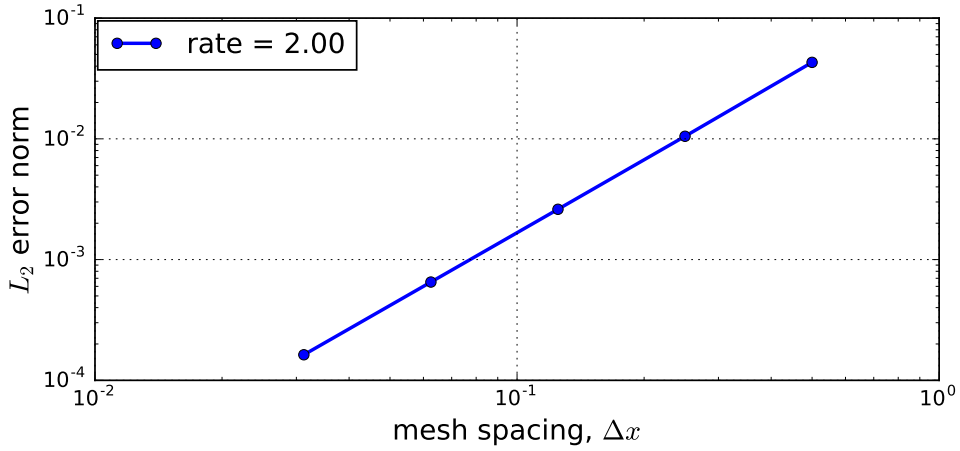


Figure 3.1.2: Convergence of the discrete L_2 error norm of the computed temperature distribution in a one-dimensional bar, relative to the actual temperature distribution.

3.1.2 Heat Conduction in a Two-Dimensional Plate

The equation governing heat conduction in two dimensions is

$$-\kappa\nabla^2 T = q(\vec{x}), \tag{3.1.5}$$

where T is the temperature, $\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$ is the Laplace operator, κ is the constant thermal conductivity, and $q(\vec{x})$ is a heat source term. The domain is a rectangle of dimensions $L_x \times L_y$, as shown in Figure 3.1.3.

We use a uniform grid of $N_x + 1$ points in the x direction and $N_y + 1$ points in the y direction. The interval spacings are then

$$\Delta x = \frac{L_x}{N_x}, \qquad \Delta y = \frac{L_y}{N_y}.$$

The unknowns are the temperature values at the nodes, which we will denote with two indices as $T_{i,j}$, with $i \in [0, \ldots, N_x]$, and $j \in [0, \ldots, N_y]$. In this example, for flexibility in applying the boundary conditions, we treat *all* of the nodes in the mesh as unknowns, including those on the boundary. So the total number of unknowns is $N \equiv (N_x + 1)(N_y + 1)$.

To obtain a solvable system, we need N equations, and these come from enforcing a finite difference approximation of the PDE in Equation 3.1.5 at interior nodes, and boundary conditions at boundary nodes. We consider the interior nodes first, $(i, j) \in (1 \ldots N_x - 1, 1 \ldots N_y - 1)$, where ∇^2 must be approximated using temperatures at surrounding nodes.

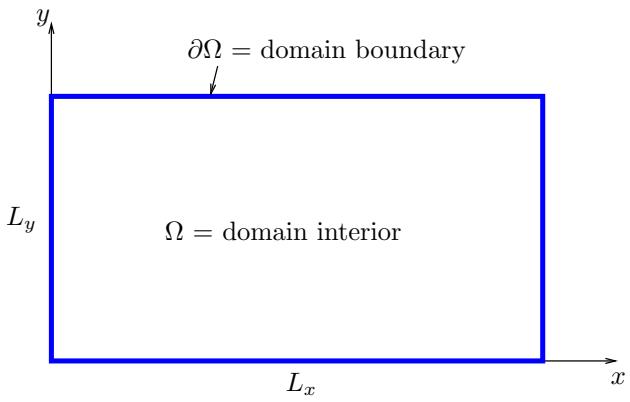


Figure 3.1.3: Rectangular domain for a two-dimensional Poisson problem.

Since $\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$, we can use the second-order discretization in Equation 2.1.5 for each second derivative,

$$\begin{aligned} \frac{\partial^2 T}{\partial x^2} &\approx \frac{T_{i-1,j} - 2T_{i,j} + T_{i+1,j}}{\Delta x^2}, \\ \frac{\partial^2 T}{\partial y^2} &\approx \frac{T_{i,j-1} - 2T_{i,j} + T_{i,j+1}}{\Delta y^2}. \end{aligned}$$

Substituting these formulas into Equation 3.1.5 gives the following system at interior nodes,

$$\frac{-T_{i-1,j} + 2T_{i,j} - T_{i+1,j}}{\Delta x^2} + \frac{-T_{i,j-1} + 2T_{i,j} - T_{i,j+1}}{\Delta y^2} = \frac{1}{\kappa} q_{i,j}.$$

Note, $q_{i,j}$ refers to the source term $q(\vec{x})$ evaluated at node (i, j) in our grid.

To complete the system, we need to specify boundary conditions. For this example, we choose a simple Dirichlet boundary condition of zero temperature on the boundary: $T = 0$ on $\partial\Omega$. Translating this into equations gives

$$\begin{aligned} T_{0,j} = T_{N_x,j} &= 0, \\ T_{i,0} = T_{i,N_y} &= 0, \end{aligned}$$

where i ranges from 0 to N_x, and j range from 0 to N_y. We can write this system in matrix form as

$$\mathbf{AT} = \mathbf{q},$$

where $\mathbf{T}$ is a column vector of all N unknown temperatures. The ordering of unknowns is up to us to decide, and typically some logical ordering that is easy to program is used. We choose to stack the unknowns row by row, so that the temperature vector looks like

$$\mathbf{T} = \left[T_{0,0}, T_{1,0}, \ldots, T_{N_x,0}, T_{0,1}, T_{1,1}, \ldots, T_{N_x,1}, \ldots, T_{0,N_y}, T_{1,N_y}, \ldots, T_{N_x,N_y}\right]^T.$$

q is a column vector of the right-hand side $q(\vec{x})$ source term evaluated at the nodes and multiplied by $1/\kappa$. Finally, **A** is an $N \times N$ matrix where each row contains coefficients that multiply entries in the **T** vector to give the equation at one node. This matrix is sparse because because in each equation, each unknown only "talks to" its four immediate neighbors (above, below, left, right).

Listing 3.1.4 shows a Python code that sets up and solves this system for a hard-coded source term $q(\vec{x})$. Note the use of a sparse matrix structure to store the matrix **A**. The matrix gets populated "on the fly" in the loops over nodes, which is not efficient from a memory management perspective, and Python issues a warning about this. Listing 8.1.1 in Section 8.1 presents a more efficient implementation using vectors that describe the locations and values of the nonzero entries.

Figure 3.1.4 shows the temperature distribution for one set of parameters, using the driver code in Listing 3.1.5.

Listing 3.1.4: Computes the temperature in a heated 2D plate with Dirichlet boundary conditions.

```
import numpy as np
from scipy import sparse
from scipy.sparse import linalg

def source(x, y, Lx, Ly, kappa):
    return np.sin(np.pi*x/Lx)*(-(np.pi/Lx)**2*y/Ly*(1-y/Ly) - 2/Ly**2)*(-kappa)

def heat2d(Lx, Ly, Nx, Ny, kappa):
    dx = float(Lx)/Nx;  dy = float(Ly)/Ny;   # spacing
    x = np.linspace(0, Lx, Nx+1)  # x nodes
    y = np.linspace(0, Ly, Ny+1)  # y nodes
    Y, X = np.meshgrid(y, x)      # matrices of all nodes
    N = (Nx+1)*(Ny+1) # total number of unknowns
    A = sparse.csr_matrix((N,N))  # empty sparse matrix
    q = np.zeros(N)               # empty rhs vector

    # fill in interior contributions
    for iy in range(1,Ny):
        for ix in range(1,Nx):
            i = iy*(Nx+1)+ix; x = ix*dx; y = iy*dy
            iL = i-1; iR = i+1; iD = i-(Nx+1); iU = i+(Nx+1)
            q[i] = source(x,y,Lx,Ly,kappa)/kappa
            A[i,i] = 2./dx**2 + 2./dy**2
            A[i,iL] = A[i,iR] = -1./dx**2
            A[i,iD] = A[i,iU] = -1./dy**2

    # enforce Dirichlet boundary conditions
    for ix in range(Nx+1):
        i = ix; A[i,i] = 1.; q[i] = 0.
        i = Ny*(Nx+1)+ix; A[i,i] = 1.; q[i] = 0.
    for iy in range(Ny+1):
        i = iy*(Nx+1); A[i,i] = 1.; q[i] = 0.
        i = iy*(Nx+1)+Nx; A[i,i] = 1.; q[i] = 0.

```

```
    # solve system
    Tv = linalg.spsolve(A,q)     # solution at all points
    T = np.reshape(Tv, (Nx+1,Ny+1), order='F') # reshape into matrix

    return X, Y, T

def main():
    X, Y, T = heat2d(2.0, 1.0, 4, 3, 0.5)

if __name__ == "__main__":
    main()
```

Listing 3.1.5: Driver code for the function `heat2d`.

```
import numpy as np
import matplotlib.pyplot as plt
from heat2d import heat2d

def plotsol(X, Y, T, fname):
    f = plt.figure( figsize =(8,6))
    plt.contourf(X, Y, T)
    plt.xlabel(r'$x$', fontsize=16)
    plt.ylabel(r'$y$', fontsize=16)
    plt.figure(f.number); plt.grid()
    plt.tick_params(axis='both', labelsize=12)
    f.tight_layout (); plt.show(block=False); plt.savefig(fname)
    plt.close(f)

def main():
    Lx, Ly, Nx, Ny, kappa = 2., 1., 40, 20, 0.5
    X, Y, T = heat2d(Lx, Ly, Nx, Ny, kappa)
    plotsol(X, Y, T, '../figs/heat2d_N861.pdf')

if __name__ == "__main__":
    main()
```

For the source term $q(\vec{x})$ we choose a convenient function that will make our error calculation easier. Rather than arbitrarily picking q and having to solve Equation 3.1.5 for $T(\vec{x})$, we instead work backwards: first choose $T(\vec{x})$, and then substitute into Equation 3.1.5 to compute $q(\vec{x}) = q(x, y)$. For $T(\vec{x})$ we pick a smooth function that satisfies the Dirichlet boundary condition $T = 0$ on a rectangle with $L_x = 2$ and $L_y = 1$:

$$T(\vec{x}) = T(x, y) = \sin\left(\pi \frac{x}{L_x}\right) \frac{y}{L_y} \left(1 - \frac{y}{L_y}\right). \tag{3.1.6}$$

The source term is then $q = -\kappa \nabla^2 T$, which is

$$q(\vec{x}) = q(x, y) = -\kappa \sin\left(\pi \frac{x}{L_x}\right) \left[-\left(\frac{\pi}{L_x}\right)^2 \frac{y}{L_y} \left(1 - \frac{y}{L_y}\right) - \frac{2}{L_y^2} \right]. \tag{3.1.7}$$

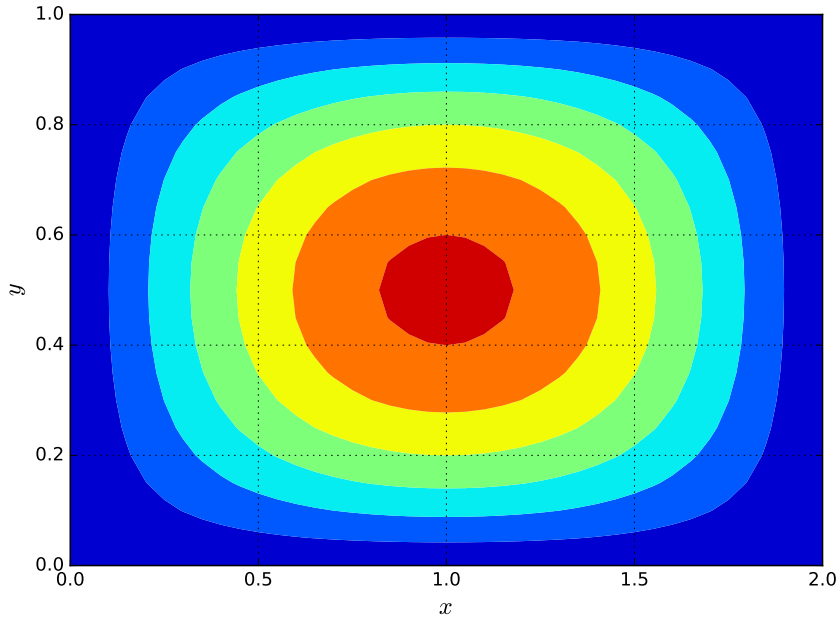


Figure 3.1.4: Sample finite-difference result for the temperature in a heated 2D plate, with $L_x = 2$, $L_y = 1$, $q(x)$ as in Equation 3.1.7, $\kappa = 0.5$, $N_x = 40$, $N_y = 20$.

Using the exact solution, we can measure accuracy with an L_2 error norm that approximates the integral over the domain,

$$\epsilon = \sqrt{\sum_{i=0}^{N_x} \sum_{j=0}^{N_y} \Delta x \Delta y (T_{i,j} - T^{\text{exact}}(\vec{x}_{i,j}))^2},$$

where $T^{\text{exact}}(\vec{x})$ is the exact solution from Equation 3.1.6. Figure 3.1.5 shows the convergence of the L_2 error norm with mesh refinement, generated by the code in Listing 3.1.6. As indicated by the slope, second-order accuracy is achieved, i.e. $\epsilon \sim \mathcal{O}(h^2)$, where h could be either Δx, Δy, or (in our case) the geometric mean of the two. This second-order rate is expected as we used second-order differences for $\frac{\partial^2}{\partial x^2}$ and $\frac{\partial^2}{\partial y^2}$.

Listing 3.1.6: Performs a convergence study for the two-dimensional heat conduction problem.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from heat2d import heat2d
4
5 def main():
6     Lx, Ly, Nx, Ny, kappa = 2., 1., 4, 2, 0.5
7     Nref = 4
8     h = np.zeros(Nref); ev = np.zeros(Nref)
9     for i in range(Nref):
```

```
            X, Y, T = heat2d(Lx, Ly, Nx, Ny, kappa)
            dA = Lx*Ly/(Nx*Ny); h[i] = np.sqrt(dA)
            Te = np.sin(np.pi*X/Lx)*Y/Ly*(1-Y/Ly)
            ev[i] = h[i]*np.sqrt(np.sum(np.sum((T-Te)**2)))
            Nx *= 2; Ny*=2
        f = plt.figure( figsize =(8,4))
        rate = np.log2(ev[Nref-2]/ev[Nref-1])
        plt.loglog(h, ev, 'o-', linewidth=2, color='blue', label='rate = %.2f'%(rate))
        plt.xlabel(r'mesh spacing, $h \equiv \sqrt{\Delta x \Delta y}$', fontsize=16)
        plt.ylabel(r'$L_2$ error norm', fontsize=16)
        plt.legend( fontsize =16, borderaxespad=0.1, loc=2)
        plt.figure(f.number); plt.grid()
        plt.tick_params(axis='both', labelsize =12)
        f.tight_layout (); plt.show(block=False)
        plt.savefig('../figs/heat2dconv.pdf')
        plt.close(f)

if __name__ == "__main__":
    main()
```

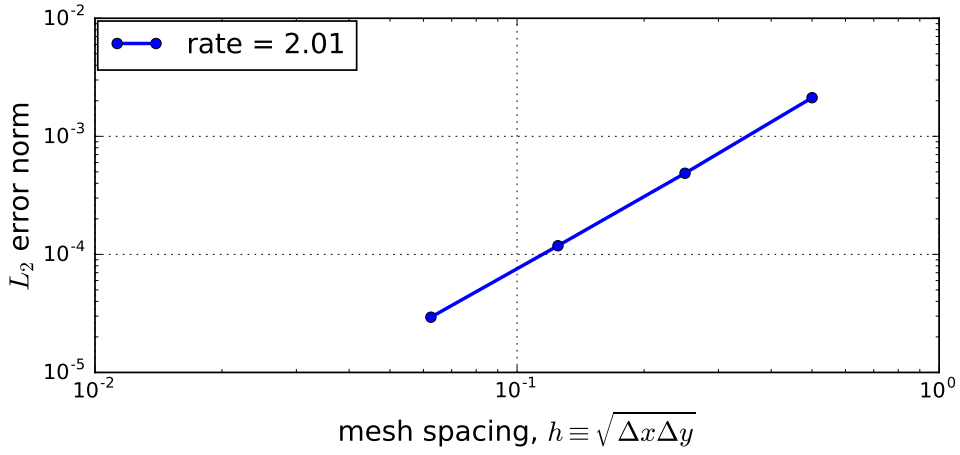
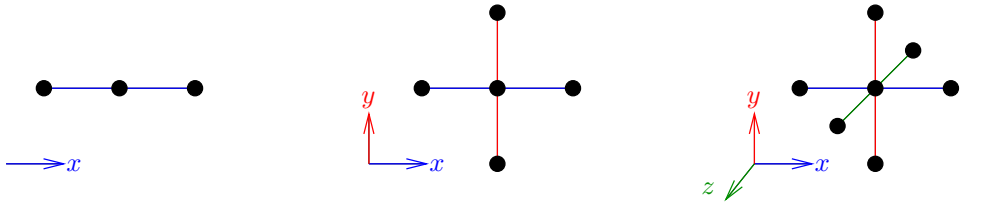


Figure 3.1.5: Convergence of the discrete L_2 error norm of the computed temperature distribution in a two-dimensional plate, relative to the actual temperature distribution.

3.2 Stencils and Matrices

Finite difference approximations are not unique, and they vary in cost and accuracy. In the previous section we saw examples of using second-order differences to approximate the second derivative. For each point at which we wanted to approximate the second derivative, we had to use temperature values at neighboring points. For the chosen second-order differences,

we only needed temperatures at the immediate neighbors (left/right and up/down in two dimensions). If we wanted more accuracy, we would need to use more neighbors. The nodes involved in approximating a derivative constitute the **stencil** of the method. In the examples of the previous sections, we used three (1D) and five (2d) point stencils, while in three dimensions the equivalent second-order discretization would require a nine-point stencil, as shown in Figure 3.2.1.

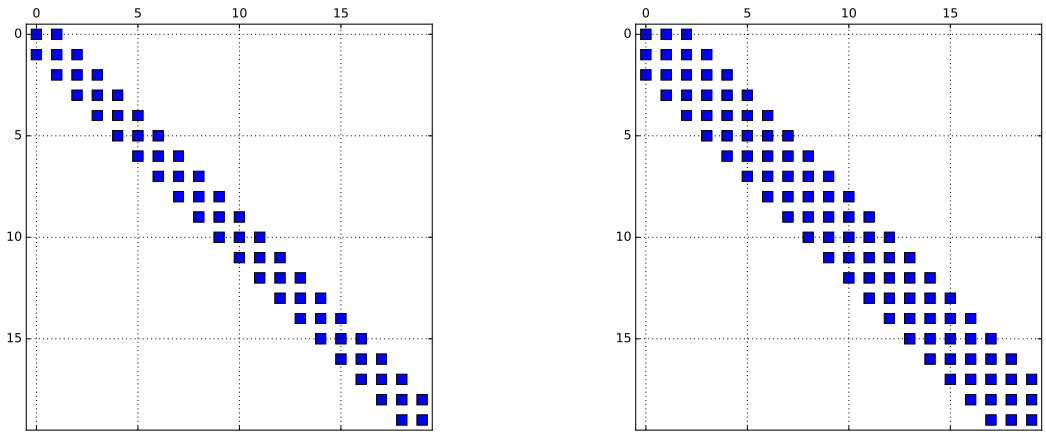


3-point stencil in 1D 5-point stencil in 2D 9-point stencil in 3D

Figure 3.2.1: Nearest-neighbor stencils in one, two, and three dimensions.

The more points in a stencil, the more expensive the resulting finite-difference method. This is because more operations are required to compute each difference when measuring the derivative, and because the matrix becomes higher in **bandwidth**. The bandwidth of a matrix can be defined as the average number of nonzero entries in a matrix row. Higher bandwidth means that the matrix is less sparse, so that the system is generally harder to solve (there is more coupling between the unknowns).

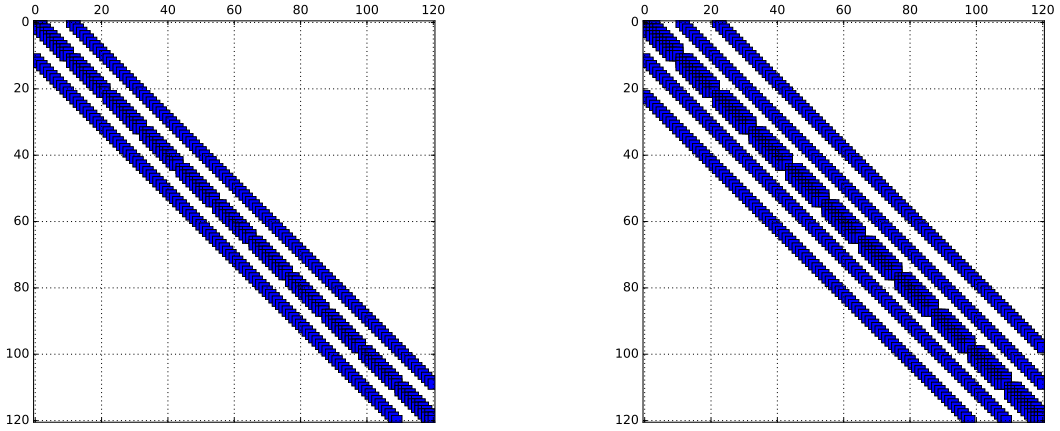
Figure 3.2.2 shows the sparsity patterns for a one-dimensional discretization, using a 3-point (immediate neighbors) or 5-point (neighbors of neighbors) stencils. We see higher bandwidth (more nonzero entries) for the 5-point stencil.



(a) 3-point stencil (b) 5-point stencil

Figure 3.2.2: Nonzero entries for one-dimensional 3-point and 5-point stencils, with 20 unknowns.

Figure 3.2.3 shows the sparsity patterns for a two-dimensional discretization, using a 5-point (immediate neighbors) or 7-point (neighbors of neighbors) stencils. As expected, we see a higher bandwidth for the 7-point stencil. Note that the non-contiguous entries are due to the coupling of unknowns in the y-direction when the unknowns are ordered along the x direction first.



(a) 5-point stencil

(b) 7-point stencil

Figure 3.2.3: Nonzero entries for two-dimensional 5-point and 7-point stencils, for a mesh with 10 nodes in each direction.

When working with extended stencils, care must be taken near domain boundaries, where, for example, neighbors of neighbors may not be available. In these cases, the stencils can be modified to use more information from the other side.

3.3 Boundary Conditions

In the one-dimensional example of Section 3.1, we used Dirichlet boundary conditions, so that only the interior nodes were unknown. The system size was then reduced by the two boundary nodes, at which the temperature was known. On the other hand, in the two-dimensional example, we kept all of the temperature nodes as unknowns, even though we also used Dirichlet boundary conditions. This made for a larger system, but the boundary conditions were easily enforced via equations that stated "temperature = known value" for these nodes.

We now consider how to impose other boundary conditions, which fall into three general categories:

1. **Dirichlet**: the state value on the boundary is specified,

$$T = T_b.$$

For example, the boundary may represent the edge of an adjacent reservoir, in which the temperature is approximately constant.

2. **Neumann**: the derivative of the state in the normal direction (g_b, possibly a function of position) is specified,

$$\nabla T \cdot \vec{n} = g_b.$$

For example, an adiabatic (no gradient) boundary condition would have $g_b = 0$.

3. **Robin**: a weighted combination of the state and its normal derivative is specified,

$$aT + b\nabla T \cdot \vec{n} = g_b,$$

where a and b are known constants, and g_b could be a function of position on the boundary. For example, on a convective boundary, the heat flux is given by a combination of the boundary temperature and its gradient.

In practice, different boundary conditions could be imposed on different parts of the domain boundary, $\partial\Omega$. For flexibility in imposing the different boundary conditions, we treat all boundary nodes as unknowns.

3.3.1 Neumann Boundary Conditions for a 1D Bar

For the one-dimensional heat conduction problem in Section 3.1.1, we used Dirichlet boundary conditions. We now show how to impose Neumann and Robin boundary conditions. Suppose that the problem setup remains as in Section 3.1.1, except that on the right end of the bar we impose the Neumann condition

$$\frac{dT}{dx} = g_R, \tag{3.3.1}$$

where g_R is a constant. The discretized form of this condition requires a finite difference approximation for $\frac{dT}{dx}$ at the right end of the domain. For this we try a simple two-point difference,

$$\frac{T_N - T_{N-1}}{\Delta x} = g_R \quad \Rightarrow \quad T_N - T_{N-1} = g_R \Delta x. \tag{3.3.2}$$

The modified version of Equation 3.1.3 now reads

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & \dots & 0 \\ -1 & 2 & -1 & 0 & 0 & \dots & 0 \\ 0 & -1 & 2 & -1 & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & -1 & 2 & -1 & 0 \\ 0 & \dots & 0 & 0 & -1 & 2 & -1 \\ 0 & \dots & 0 & 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} T_0 \\ T_1 \\ T_2 \\ \vdots \\ T_{N-2} \\ T_{N-1} \\ T_N \end{bmatrix} = \begin{bmatrix} T_0 \\ q(x_1)\Delta x^2/\kappa \\ q(x_2)\Delta x^2/\kappa \\ \vdots \\ q(x_{N-2})\Delta x^2/\kappa \\ q(x_{N-1})\Delta x^2/\kappa \\ g_R \Delta x \end{bmatrix} \tag{3.3.3}$$

Listing 3.3.1 shows a Python code that sets up and solves this system for the source term $q(x) = \sin(\pi x/L)$, and Figure 3.3.1 shows the temperature distribution for one set of parameters.

Listing 3.3.1: Computes the temperature in a heated 1D bar with a Dirichlet boundary condition on the left end and a Neumann boundary condition on the right end.

```
import numpy as np
from scipy import sparse
from scipy.sparse import linalg

def heat1d(T0, gR, L, kappa, N):
    dx = float(L)/N                # spacing
    x = np.linspace(0, L, N+1)  # all nodes
    data = np.ones((3,N+1))     # diagonals for making a sparse matrix
    data[0,:] *= -1; data[1,:] *= 2; data[2,:] *= -1
    diags = np.array([-1, 0, 1])
    A = sparse.spdiags(data, diags, N+1, N+1, 'csr')
    q = np.sin(np.pi*x/L)*dx*dx/kappa # right-hand side
    A[0,0] = 1; A[0,1] = 0; q[0] = T0 # Dirichlet BC on left
    A[N,N-1] = -1; A[N,N] = 1; q[N] = gR*dx # Neumann BC on right
    T = linalg.spsolve(A,q)     # solution at all points
    return x, T

def main():
    x, T = heat1d(3, 1.0, 2.0, .5, 10)

if __name__ == "__main__":
    main()
```

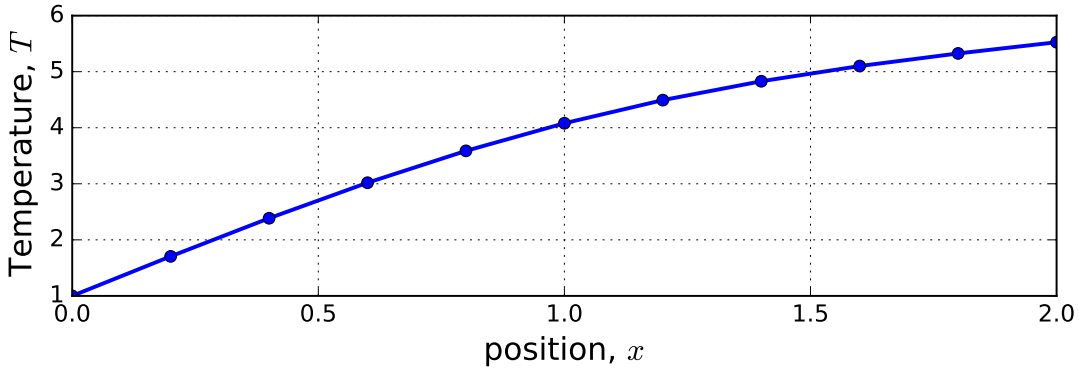


Figure 3.3.1: Sample finite-difference result for the temperature in a heated 1D bar, with $L = 2$, $q(x) = \sin(\pi x/L)$, $\kappa = 0.5$, $T_0 = 1$, $T_N = 4$, $N = 10$, and $g_R = 1$ for the Neumann condition on the right end.

The finite difference used to discretize the boundary condition can affect the global rate of convergence of the solution. Using the known exact solution to this problem we monitor

the convergence rate of the discrete L_2 error norm, Equation 3.1.4, with mesh refinement. Figure 3.3.2 shows that with our choice of a simple 2-point difference for the Neumann boundary condition, which is first-order accurate, the solution still converges at a global rate of 2. The reason for this is that the first-order error only affects a small region of the bar that shrinks as $\Delta x \to 0$, so that the majority of the solution converges at second-order. Such behavior also depends on the character of the equation: for example, in hyperbolic systems, a boundary condition error on an inflow could pollute a large region of the interior solution. To impose Robin boundary conditions, we would modify Equation 3.3.2 to include

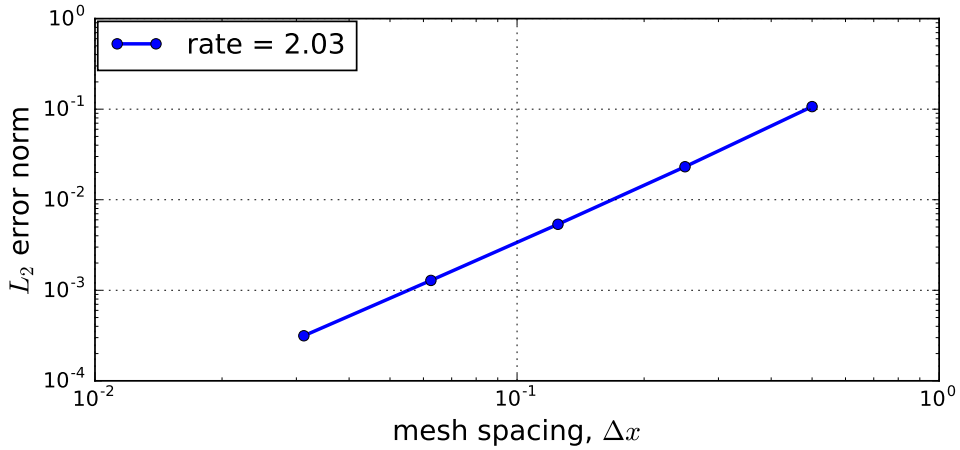


Figure 3.3.2: Convergence of the discrete L_2 error norm of the computed temperature distribution in a one-dimensional bar, using a Neumann boundary condition on the right end.

the temperature itself, T_N, in addition to the discretized difference. This would only affect the last equation of the system.

3.4 Direct Solvers

In the previous section we reduced a linear boundary value problem to a system, $\mathbf{Ax} = \mathbf{b}$, and then solved it using a **direct solver**. This term refers to a set of deterministic operations on the matrix system that, after a certain number of steps, yields the solution to machine precision accuracy. One example of a direct solver is the Gaussian elimination algorithm, presented in Section 1.7.2. In the form of the PLU factorization, this is a commonly-used method to solve matrix systems that are not large. Due to its $\mathcal{O}(N^3)$ complexity, as N grows, the algorithm quickly becomes expensive.

When the matrix has certain structure, the cost of a direct solver can be reduced. One example is a tridiagonal matrix, which occurs in the 3-point discretization of the heat conduction equation (Section 3.1.1). Such a matrix has entries only on the main diagonal and

the two off-diagonals adjacent to the main diagonal, as shown in Figure 3.2(a). This system can be solved in $\mathcal{O}(N)$ operations using two sweeps, in a method called the Thomas algorithm. This method is just Gaussian elimination, made simpler by the fact that a lot of entries are zero. It can be implemented in-place via the following LU factorization,

$$\underbrace{\begin{bmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & \ddots & \\ & & \ddots & \ddots & c_{N-1} \\ & & & a_N & b_N \end{bmatrix}}_{\mathbf{A}} = \underbrace{\begin{bmatrix} 1 & & & & \\ \frac{a_2}{b'_1} & 1 & & & \\ & \frac{a_3}{b'_2} & 1 & & \\ & & \ddots & \ddots & \\ & & & \frac{a_N}{b'_{N-1}} & 1 \end{bmatrix}}_{\mathbf{L}} \underbrace{\begin{bmatrix} b'_1 & c_1 & & & \\ & b'_2 & c_2 & & \\ & & b'_3 & \ddots & \\ & & & \ddots & c_{N-1} \\ & & & & b'_N \end{bmatrix}}_{\mathbf{U}} \tag{3.4.1}$$

where blanks refer to zero entries, and

$$\begin{aligned} b'_1 &= b_1 \\ b'_{i+1} &= b_{i+1} - a_{i+1}c_i/b'_i \end{aligned}$$

Applying $\mathbf{A}^{-1}$ to a vector $\mathbf{f}$ then consists of two steps,

$$\begin{aligned} \text{(1) Solve } \mathbf{Ly} &= \mathbf{f}, \\ \text{(2) Solve } \mathbf{Uu} &= \mathbf{y}. \end{aligned}$$

These are done via forward and backward substitution, respectively. $\mathbf{y}$ is a temporary vector, and $\mathbf{u}$ is the desired solution to $\mathbf{Au} = \mathbf{f}$. The complexity of this algorithm is $\mathcal{O}(N)$ because each step (LU factorization, application of $\mathbf{L}^{-1}$, and $\mathbf{U}^{-1}$) requires just one pass through the diagonals of the matrix. Storage-wise, no extra matrices need to be created, since the b'_i values overwrite the b_i values in the matrix, and a_{i+1}/b'_i overwrites a_{i+1}, per Equation 3.4.1. The original matrix $\mathbf{A}$ can also be applied to a vector $\mathbf{x}$, either through successive application of $\mathbf{U}$ followed by $\mathbf{L}$ or in one loop using Algorithm 5.

Algorithm 5 Single-loop application of $\mathbf{y} = \mathbf{Ax}$ using a factored tridiagonal matrix.

```
for i = N : −1 : 1 do
    t = b'_i x_i
    if i < N then
        t = t + c_i x_{i+1}
        y_{i+1} = y_{i+1} + (a_{i+1}/b'_i) t
    end if
    y_i = y_i + t
end for
```

For larger stencils, especially in higher dimensions, the bandwidth of the matrix grows and the Thomas algorithm cannot be used directly. However, it can be used in *block* form

when the mesh nodes are ordered in a structured way, e.g. first along x, then along y in two dimensions. This is the case in Figure 3.3(a), where the matrix is block tridiagonal, with block size of 10, the number of unknowns along one dimension.

In general, consider a two dimensional finite difference problem on a mesh with $m \times m$ nodes. The number of unknowns is $N = m^2$, and the matrix $\mathbf{A}$ is of size $N \times N = m^2 \times m^2$. However, when the unknowns are ordered systematically along one dimension first, the matrix becomes block tridiagonal, with matrix blocks of size $m \times m$. These blocks represent how one horizontal row of unknowns interacts with the adjacent rows above and below. In the Thomas algorithm, these blocks must be factored/inverted (the equivalent of dividing by a scalar), and the cost of this with Gaussian elimination is $\mathcal{O}(m^3)$ per block. As there are $\mathcal{O}(m)$ blocks, the complexity of the Thomas algorithm is $\mathcal{O}(m^4)$.

In three spatial dimensions, for a mesh of $m \times m \times m$ nodes, the number of unknowns is $N = m^3$. The matrix can still be viewed as block tridiagonal, but now with block size of $m^2 \times m^2$. These blocks represent how one *plane* of unknowns interacts with its neighbors. The cost of factoring these blocks is $\mathcal{O}((m^2)^3) = \mathcal{O}(m^6)$, and since there are $\mathcal{O}(m)$ blocks (planes), the complexity of the Thomas algorithm is $\mathcal{O}(m^7)$. This means that if we double the resolution (number of nodes) in each dimension, the system becomes $2^7 = 128$ times more expensive to solve. Note that this is an overestimate because one would not typically use Gaussian elimination to invert the $m^2 \times m^2$ blocks, which are themselves sparse matrices.

3.5 Iterative Smoothers

When solving a problem with a large number of unknowns, especially in multiple dimensions, a direct solver may not be cost-effective or feasible. Whereas the matrix $\mathbf{A}$ may be sparse, the LU factorization of $\mathbf{A}$ resulting from Gaussian elimination may become dense, or may contain large dense blocks that consume prohibitively many operations to calculate and/or are too large to store in computer memory. An alternative lies in **iterative** methods, in which an *approximate* solution is sought through the application of multiple inexpensive steps. The advantage is that at each iteration, a solution approximation exists that (ideally) improves with additional iterations.

We center the presentation of iterative methods on a prototypical one-dimensional diffusion PDE,

$$-u_{xx} = f, \tag{3.5.1}$$

discretized using a second-order difference on a uniform grid of $N+1$ nodes,

$$\frac{-u_{i-1} + 2u_i - u_{i+1}}{\Delta x^2} = f_i. \tag{3.5.2}$$

The unknown states u_i are indexed by i, $0 \leq i \leq N$. In an iterative solution method, we begin with a guess of the solution and successively improve it through iterations. We therefore now add a superscript n to indicate the iteration number: u_i^n is the value of the state at node i on iteration n. The initial guess is denoted by $n = 0$.

3.5.1 The Jacobi Iteration

At each node i, Equation 3.5.2 expresses a relationship that must be satisfied among the states at node i and its neighbors. Presumably, our initial guess of the solution, $\mathbf{u}^0$, will not satisfy this equation at every node (if it did we would be done!). Taking a local point of view of just the one equation at node i, we could satisfy the finite difference equation by adjusting the value of the state at node i, u_i. That is, we solve Equation 3.5.2 for u_i, holding the neighboring values fixed at the present iteration n, and call the resulting solution the state at iteration $n+1$:

$$u_i^{n+1} = \frac{1}{2}\left(u_{i-1}^n + u_{i+1}^n + \Delta x^2 f_i\right). \tag{3.5.3}$$

Doing this at every node i constitutes the **Jacobi** iteration: each unknown is updated according to the discretized PDE, while holding the neighboring unknowns constant. Since in general every unknown ends up changing, treating neighbor values as constant is an approximation, and we should not expect to arrive at the global solution with one step of this iteration. However, with multiple applications of the Jacobi iteration, the state will (may) converge to the global solution, albeit eventually very slowly. Such an iterative method is called a "smoother" because at each step, it is effective at smoothing out highly-oscillatory errors from the solution.

The Jacobi iteration can also be derived from physical motivations. Consider augmenting Equation 3.5.1 with an unsteady term to read

$$u_t - u_{xx} = f. \tag{3.5.4}$$

This equation now governs the temporal *evolution* of the state. Starting from some initial condition $\mathbf{u}^0$, as time progresses, the state should evolve to the steady-state solution to Equation 3.5.1. This observation motivates the interpretation of the iterative method as a time-marching scheme. Using a simple forward difference for the unsteady term, with n indexing the time nodes, and Δt representing the time step, the discretized version of Equation 3.5.4 reads

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} - \frac{u_{i-1}^n - 2u_i^n + u_{i+1}^n}{\Delta x^2} = f_i. \tag{3.5.5}$$

Solving for u_i^{n+1} yields,

$$u_i^{n+1} = u_i^n + \frac{\Delta t}{\Delta x^2}\left(u_{i-1}^n - 2u_i^n + u_{i+1}^n\right) + \Delta t f_i \tag{3.5.6}$$

We will see later that the time step Δt is limited by $\Delta x^2/2$ for the diffusion problem, and using this maximum value (to arrive at steady state the fastest) gives

$$\begin{aligned} u_i^{n+1} &= u_i^n + \frac{1}{2}\left(u_{i-1}^n - 2u_i^n + u_{i+1}^n\right) + \frac{\Delta x^2}{2} f_i \\ &= \frac{1}{2}\left(u_{i-1}^n + u_{i+1}^n + \Delta x^2 f_i\right). \end{aligned}$$

This equation is the same as that obtained from the local solves in Equation 3.5.3.

3.5.2 The Gauss-Seidel Iteration

In the Jacobi iteration introduced in the previous section, the state at each node was modified to satisfy Equation 3.5.2, while keeping the neighboring states constant at the previous iteration value. The **Gauss-Seidel** iteration is very similar, except that once states are modified, they are used for subsequent updates of neighboring nodes. That is, when updating u_i^n to u_i^{n+1}, the neighbor states may be either at iteration n or at $n+1$, depending on whether or not they were already visited. The Gauss-Seidel iteration therefore depends on the order in which the nodes are updated.

Consider a simple left-to-right ordering of nodes for our one-dimensional diffusion problem. In this case, the Gauss-Seidel update of node i is

$$u_i^{n+1} = \frac{1}{2}\left(u_{i-1}^{n+1} + u_{i+1}^{n} + \Delta x^2 f_i\right). \tag{3.5.7}$$

This is very similar to the Jacobi iteration, except that the state at node $i-1$ is already at iteration $n+1$, since it was already updated in the left-to-right ordering. On the other hand, if we performed Gauss-Seidel with a right-to-left ordering, the update formula would read

$$u_i^{n+1} = \frac{1}{2}\left(u_{i-1}^{n} + u_{i+1}^{n+1} + \Delta x^2 f_i\right). \tag{3.5.8}$$

Alternatively, we could perform an odd/even ordering, where first all the odd nodes are updated (all neighbors at iteration n) and then the even nodes are updated (all neighbors at iteration $n+1$),

$$\text{Step 1: } u_i^{n+1} = \frac{1}{2}\left(u_{i-1}^{n} + u_{i+1}^{n} + \Delta x^2 f_i\right), \quad \forall \text{ odd } i, \tag{3.5.9}$$

$$\text{Step 2: } u_i^{n+1} = \frac{1}{2}\left(u_{i-1}^{n+1} + u_{i+1}^{n+1} + \Delta x^2 f_i\right), \quad \forall \text{ even } i, \tag{3.5.10}$$

In multiple dimensions, there are more ways to order the unknowns. A popular approach in two dimensions is **red-black** Gauss-Seidel, in which the nodes of a grid are colored like a checkerboard, and first all the red nodes are updated, followed by all the black nodes.

3.5.3 Matrix Representation

We can represent the Jacobi and Gauss-Seidel iterations in matrix form in order to gain insight into the methods and to perform analysis. In practice, the matrices are not formed and one uses the above equations instead.

The matrix form of Equation 3.5.2 is $\mathbf{Au} = \mathbf{f}$, where $\mathbf{u}$ is the vector of unknowns and $\mathbf{A}$ is the linear operator with nonzero entries on three diagonals. We split $\mathbf{A}$ as

$$\mathbf{A} = \mathbf{D} + \mathbf{L} + \mathbf{U},$$

where $\mathbf{D}$ is a matrix of only the main diagonal, and $\mathbf{L}/\mathbf{U}$ are the lower/upper triangular portions of $\mathbf{A}$, not including the diagonal.

The Jacobi iteration in Equation 3.5.3 can then be written as

$$\begin{aligned} \mathbf{D}\mathbf{u}^{n+1} &= -(\mathbf{L}+\mathbf{U})\mathbf{u}^n + \mathbf{f} \\ \mathbf{u}^{n+1} &= \underbrace{-\mathbf{D}^{-1}(\mathbf{L}+\mathbf{U})}_{\mathbf{S}_J}\mathbf{u}^n + \mathbf{D}^{-1}\mathbf{f}, \end{aligned}$$

where $\mathbf{S}_J$ is the Jacobi iteration matrix. The Gauss-Seidel iteration in Equation 3.5.7, using the left-to-right ordering, becomes

$$\begin{aligned} (\mathbf{D}+\mathbf{L})\mathbf{u}^{n+1} &= -\mathbf{U}\mathbf{u}^n + \mathbf{f} \\ \mathbf{u}^{n+1} &= \underbrace{-(\mathbf{D}+\mathbf{L})^{-1}\mathbf{U}}_{\mathbf{S}_{GS}}\mathbf{u}^n + (\mathbf{D}+\mathbf{L})^{-1}\mathbf{f}, \end{aligned}$$

where $\mathbf{S}_{GS}$ is the Gauss-Seidel iteration matrix. The presence of $\mathbf{L}$ on the left-hand side in the first equation above accounts for the forward-substitution nature of Gauss-Seidel: the unknowns are updated in a sweep, always using the latest values when available. This dependence of an unknown on the ones prior to it is represented in $\mathbf{L}$. Different Gauss-Seidel orderings can be easily incorporated into the above matrix form by simply re-ordering the unknowns.

In general, an iterative method can be expressed in matrix form by first splitting the matrix as $\mathbf{A} = \mathbf{M} + \mathbf{N}$, where the choice of splitting defines the iterative method (e.g. $\mathbf{M} = \mathbf{D}$ for Jacobi). The update equation then reads

$$\begin{aligned} \mathbf{M}\mathbf{u}^{n+1} &= -\mathbf{N}\mathbf{u}^n + \mathbf{f} \\ \mathbf{u}^{n+1} &= \underbrace{-\mathbf{M}^{-1}\mathbf{N}}_{\mathbf{S}}\mathbf{u}^n + \mathbf{M}^{-1}\mathbf{f}, \end{aligned} \tag{3.5.11}$$

where $\mathbf{S}$ is the iteration matrix, which is important for convergence analysis.

3.5.4 Convergence

One figure of merit for an iterative smoother is how quickly it converges the solution. Let's begin with an example,

$$-u_{xx} = 1, \quad u(0) = u(1) = 0, \quad u^0(x) = 0, \tag{3.5.12}$$

where $u^0(x)$ refers to the initial guess for the iterations. Using the second-order discretization in Equation 3.5.2, we write a code that compares the performance of Jacobi and Gauss-Seidel iterations, two values of N, the number of intervals in the mesh. Figure 3.5.1 shows this comparison by plotting the error in the solution versus number of iterations. Note that this is the *iterative error*, which is the difference between the solution computed using the iterative method at some given iteration and that computed using a direct solver. With many iterations, the iterative error (hopefully) decreases. Separate is the fact that at any finite N (number of grid points), we still have the *discretization error* which is the difference between the direct-solver solution and the exact continuous solution.

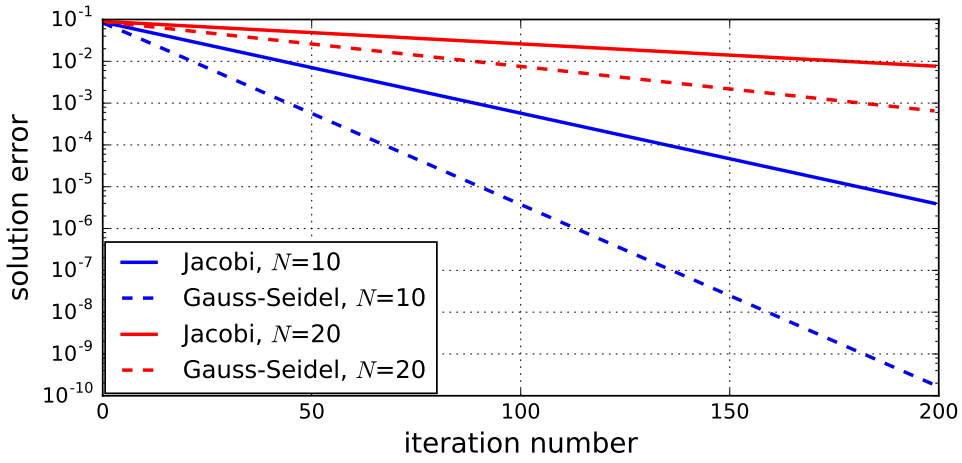


Figure 3.5.1: Convergence of the discrete L_2 error norm of Jacobi and Gauss-Seidel iterations applied to Equation 3.5.12. The exact solution at each N is computed using a direct solver.

From Figure 3.5.1, we see that Gauss-Seidel converges about twice as quickly as Jacobi: for a given error level, it requires half as many iterations. However, as the number of intervals, N, increases (i.e. more unknowns), the convergence of both iterative methods deteriorates. Specifically, the number of iterations required to achieve a certain error level grows as $\mathcal{O}(N^2)$!

We can analyze the convergence of an iterative method by looking at the eigenvalues of the iteration matrix, $\mathbf{S}$ in Equation 3.5.11. Writing this equation for both iteration n and for the exact ($n \to \infty$) solution, we have

$$\begin{aligned} \mathbf{u}^{n+1} &= \mathbf{S}\mathbf{u}^n + \mathbf{M}^{-1}\mathbf{f}, \\ \mathbf{u}^{\infty} &= \mathbf{S}\mathbf{u}^{\infty} + \mathbf{M}^{-1}\mathbf{f}. \end{aligned}$$

Taking the difference of these two equations and defining the iterative error at iteration n as $\mathbf{e}^n \equiv \mathbf{u}^n - \mathbf{u}^{\infty}$ gives

$$\mathbf{e}^{n+1} = \mathbf{S}\mathbf{e}^n = \mathbf{S}^n\mathbf{e}^0.$$

So the iteration matrix $\mathbf{S}$ determines how the error changes from one iteration to the next.

Consider the Jacobi iteration, where

$$\mathbf{S}_J = -\mathbf{D}^{-1}(\mathbf{L}+\mathbf{U}) = -\mathbf{D}^{-1}(\mathbf{A}-\mathbf{D}) = \mathbf{I} - \mathbf{D}^{-1}\mathbf{A}.$$

For the model one-dimensional diffusion problem, $\mathbf{D} = (2/\Delta x^2)\mathbf{I}$, so that

$$\mathbf{S}_J = \mathbf{I} - \frac{\Delta x^2}{2}\mathbf{A}. \tag{3.5.13}$$

From this equation we see that the eigenvectors of $\mathbf{S}_J$ are the same as the eigenvectors of $\mathbf{A}$, and that the eigenvalues are related by

$$\lambda(\mathbf{S}_J) = 1 - \frac{\Delta x^2}{2}\lambda(\mathbf{A}). \tag{3.5.14}$$

These eigenvalues determine the rate at which the error is reduced at each iteration. We need $\lambda(\mathbf{S}_J) < 1$ for convergence, and ideally not close to 1 for fast convergence. To gain insight into actual values, we consider the case of periodic boundary conditions, for which (from Fourier analysis) the eigenvalues of $\mathbf{A}$ are

$$\begin{aligned} \lambda_k(\mathbf{A}) &= \frac{2}{\Delta x^2}\left(1-\cos(k\pi/N)\right), \\ \Rightarrow \quad \lambda_k(\mathbf{S}_J) &= \cos(k\pi/N), \end{aligned} \tag{3.5.15}$$

where k is the **mode number**, $k = 1 \ldots (N-1)$. Note, $k = 0$ and $k = N$ are the constant modes, which are not relevant when Dirichlet boundary conditions are used. We therefore see that all eigenvalues are stable, so that the Jacobi iteration is convergent. The eigenvalue analysis of $\mathbf{A}$ also shows that the corresponding eigenvectors are $v_k(x) = \sin(k\pi x/L)$. Together with the eigenvalue formula, we see that the very low-frequency modes (low k) and high-frequency modes (k close to N) are slowest to converge using the Jacobi iteration, since their eigenvalues have magnitudes close to 1.

Example 3.1 (Convergence of the Jacobi iteration for different modes)**.** Consider a slightly-modified version of the problem in Equation 3.5.12,

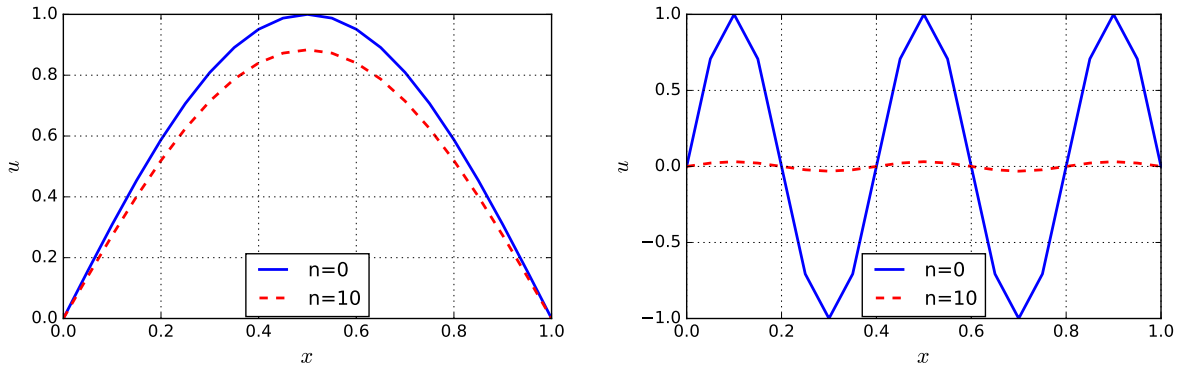
$$-u_{xx} = 0, \quad u(0) = u(1) = 0, \quad u^0(x) = \sin(k\pi x). \tag{3.5.16}$$

As the exact solution is $u(x) = 0$, this is effectively an error equation, where the initial condition defines the starting error. k is the mode number of the error, and we consider two values: $k = 1$ and $k = 5$ for a finite-difference discretization with $N = 20$ intervals. Figure 3.5.2 shows how these two modes look after 10 Jacobi iterations. We see that the error for the more oscillatory $k = 5$ mode decreases much faster than that for $k = 1$, as expected from the eigenvalue analysis: $\cos(1\pi/20) > \cos(5\pi/20)$.

In general, the solution error will contain a mix of all modes that can be represented on the grid. The rate of convergence will therefore be dominated by the largest-magnitude eigenvalue of $\mathbf{S}$. For the Jacobi method applied to the one-dimensional diffusion problem, the eigenvalues are given in Equation 3.5.15. The largest magnitude eigenvalue (slowest converging mode) occurs at $k = 1$, for which

$$\lambda_{\max}(\mathbf{S}_J) = \cos(1\pi/N) \approx 1 - \frac{\pi^2}{2N^2}, \tag{3.5.17}$$

where we have used the Taylor-series approximation for the cosine of a small number. From this equation, we corroborate the result shown in Figure 3.5.1, that the convergence rate depends quadratically on N: doubling N reduces the rate of error decrease by a factor of four. This estimate holds in higher spatial dimensions, where the analysis can be repeated on modes that can be represented in each direction. The number of iterations, n, required



(a) $k = 1$ (b) $k = 5$

Figure 3.5.2: Convergence of the Jacobi iterative method for error modes of two frequencies after $n = 10$ iterations.

to achieve a certain error reduction factor, ϵ, can then be estimated from the requirement that $\lambda_{\max}^n = \epsilon$. From this equation, we have

$$n = \frac{\log \epsilon}{\log \lambda_{\max}} = \frac{\log \epsilon}{\log(1 - \pi^2/(2N^2))} \approx \frac{\log \epsilon}{-\pi^2/(2N^2)} = \frac{-2 \log \epsilon}{\pi^2} N^2,$$

where we have used $\log(1 + x) \approx x$ for small x. Using this equation, we can compare the relative cost between direct and iterative solvers, as shown in Table 3.5.1. The cost is estimated in terms of operations, using the result of Section 3.4 for the direct solver via the (block) Thomas algorithm. The iterative solver cost is the number of iterations $\mathcal{O}(N^2)$ times the cost per iteration, which is just the number of nodes in the mesh ($\mathcal{O}(N^{\text{dim}})$), since each node is visited once. We see from these results that the iterative smoothers we have considered, when used alone, only become advantageous in three dimensions. However, we will soon see that when combined with multigrid, iterative smoothers become much more powerful.

Table 3.5.1: Comparison of the computing cost, measured in operations, of an iterative smoother to a direct solver for the Poisson equation in 1,2,3 dimensions.

Dimension	Matrix size	Direct	Iterative Smoother
1D	$N \times N$	$\mathcal{O}(N)$	$\mathcal{O}(N^2 N) = \mathcal{O}(N^3)$
2D	$N^2 \times N^2$	$\mathcal{O}(N^4)$	$\mathcal{O}(N^2 N^2) = \mathcal{O}(N^4)$
3D	$N^3 \times N^3$	$\mathcal{O}(N^7)$	$\mathcal{O}(N^2 N^3) = \mathcal{O}(N^5)$

3.5.5 Over and Under Relaxation

The general update formula for a smoother, Equation 3.5.11, is

$$\mathbf{u}^{n+1} \quad = \quad \mathbf{S}\mathbf{u}^n + \mathbf{M}^{-1}\mathbf{f}.$$

In certain cases there will be an advantage to not taking this update exactly, but rather performing a weighted average between $\mathbf{u}^n$ and the above update formula. Using ω to denote the weighting factor, the modified update equation is

$$\mathbf{u}^{n+1} \quad = \quad \omega\left[\mathbf{S}\mathbf{u}^n + \mathbf{M}^{-1}\mathbf{f}\right] + (1-\omega)\mathbf{u}^n. \tag{3.5.18}$$

The weighting is called under-relaxation when $0 < \omega < 1$, and over-relaxation when $\omega > 1$. This weighting modifies the eigenvalues of the iteration matrix, since Equation 3.5.18 can be written as

$$\begin{aligned} \mathbf{u}^{n+1} \quad &= \quad \underbrace{\left[\omega\mathbf{S} + (1-\omega)\mathbf{I}\right]}_{\mathbf{S}_\omega}\mathbf{u}^n + \omega\mathbf{f}, \\ \Rightarrow \quad \lambda(\mathbf{S}_\omega) \quad &= \quad \omega\lambda(\mathbf{S}) + (1-\omega). \end{aligned} \tag{3.5.19}$$

Under-relaxation is useful to make the Jacobi method more suitable for multigrid, discussed in the next section. Over-relaxation of Jacobi makes it unstable. On the other hand, over-relaxation can be used for Gauss Seidel, in a method called **successive over-relaxation** (SOR). In this method the over-relaxation is performed successively for each node, always using the latest (over-relaxed) values from neighbors when available. This method can lead to higher convergence rates, as shown in Example 3.2.

Example 3.2 (Over-relaxation of Gauss Seidel)**.** We apply the successive over-relaxation method to the one-dimensional problem in Equation 3.5.12, using a mesh of size $N = 10$. Figure 3.5.3 shows the convergence results of the discrete L_2 error norm. We see that increasing ω improves convergence until about $\omega = 1.6$, beyond which the convergence rate deteriorates. The limit of convergence is $\omega = 2$, at which the error stagnates at $\mathcal{O}(1)$.

3.6 Multigrid

3.6.1 Motivation

In Section 3.5.4 we saw that the number of smoothing iterations required to achieve a fixed error level depends quadratically on the mesh size, measured by N = number of intervals in one direction[1]. This means that for large mesh sizes, iterative smoothers by themselves consume too many iterations to be practical. A solution to this problem is the **multigrid**

[1]For simplicity we assume that in higher dimensions we have N^{dim} areas/volumes.

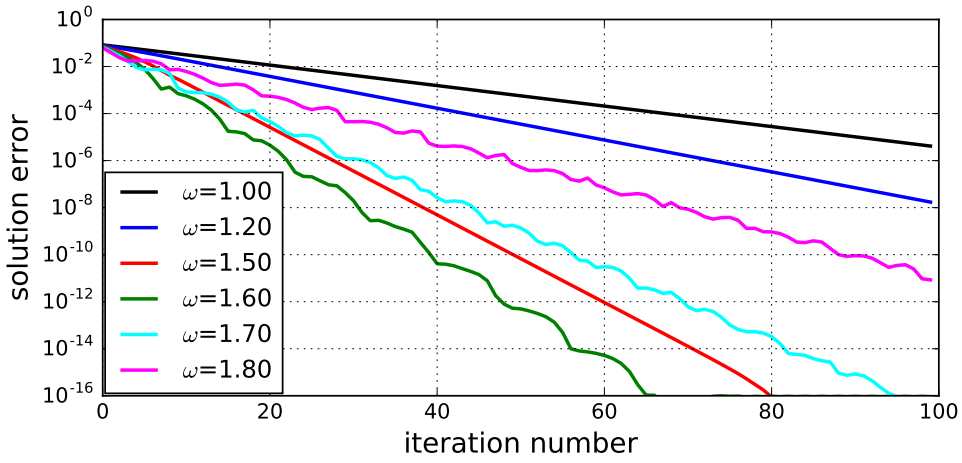


Figure 3.5.3: Performance of over-relaxed Gauss-Seidel (SOR) with different over-relaxation factors, ω, for $N = 10$ intervals.

technique, in which the work and storage is directly proportional to the number of unknowns. This means a cost of $\mathcal{O}(N^{\text{dim}})$, which is much better than the direct and iterative smoother costs in Table 3.5.1.

Multigrid is an iterative algorithm that uses multiple grids with different numbers of points. The term "coarser" refers to fewer N, whereas "finer" refers to larger N. We use a subscript to denote the grid size, measured by the interval spacing. So $\mathbf{u}_h$ refers to the solution on a grid where the nodes are spaced (in 1D) $\Delta x = h$ apart. We assume a linear system, written as

$$\mathbf{A}_h \mathbf{u}_h = \mathbf{f}_h. \tag{3.6.1}$$

On a grid twice as coarse, we would use the subscript $2h$ and the system would become $\mathbf{A}_{2h}\mathbf{u}_{2h} = \mathbf{f}_{2h}$.

Two key observations underlie multigrid:

1. A good initial guess reduces the cost of iteratively solving the system.

2. A low-frequency error mode on grid h appears as higher frequency on a coarser grid.

A consequence of the first observation is that we should use solutions from coarser grids, which are inexpensive to obtain, to initialize the solutions on finer grids. This means that a method of transferring solutions between grids needs to be specified. Note, however, that even with a good guess, iterative smoothers still require $\mathcal{O}(N^2)$ iterations. The second observation is even more powerful, as it provides a means to avoid the stalling out of iterative smoothers: using coarser grids. This is the central idea of multigrid.

3.6.2 Smoothing

Section 3.5 introduced the Jacobi and Gauss-Seidel iterative smoothers. A **smoother** is an iterative method that damps out high-frequency error components faster than low-frequency error components.

The Jacobi iteration by itself is not a smoother, but *under-relaxed* Jacobi can be a smoother, depending on the choice of ω. Recall from Equation 3.5.19 that the eigenvalues of the under-relaxed iteration matrix are given by

$$\lambda(\mathbf{S}_\omega) \quad = \quad \omega\lambda(\mathbf{S}) + (1-\omega).$$

Figure 3.6.1 plots these eigenvalues for the model one-dimensional Poisson problem, Equation 3.5.1, for various ω. For all ω, Jacobi does a poor job at damping the low frequencies. When $\omega = 1$, Jacobi also doesn't damp the high frequencies, since the eigenvalues approach -1 (unity magnitude). For $\omega = 0.5$, the highest frequencies are damped to zero and Jacobi becomes a smoother. However, better performance occurs with intermediate ω that also better damp some of the lower frequencies: e.g. $\omega = 2/3$ damps the $\pi/2$ and π modes equally. Finally, Gauss-Seidel is a good smoother stand-alone, without under/over-relaxation.

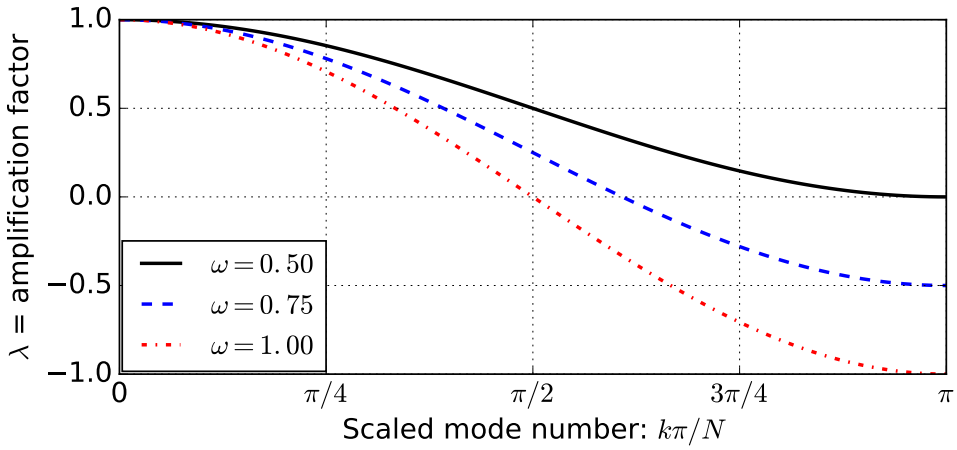


Figure 3.6.1: Eigenvalues of the under-relaxed Jacobi iteration matrix for three values of ω.

An initial solution guess typically has a mix of low and high frequency error components. When an iterative smoother is used by itself, the error then drops quickly at first, as the high-frequency error modes are damped rapidly. After this drop however, convergence stagnates as the low-frequency error modes are damped slowly. The idea in multigrid is to remedy this problem by damping the lower-frequency error components on coarser grids, where they appear as higher-frequency.

3.6.3 Two-Grid Correction

In this section we present the simplest multigrid cycle, which involves two grids. Algorithm 6 and Figure 3.6.2 present this method. The starting point is the state at the n^{th} multigrid iteration on grid h, $\mathbf{u}_h^n$, and the end result is the state at the $(n+1)^{\text{th}}$ iteration.

Algorithm 6 A two-grid correction scheme.

1: ν_1 smoothing iterations on $\mathbf{A}_h\mathbf{u}_h = \mathbf{f}_h$: $\mathbf{u}_h^n \to \mathbf{u}_h^{n+1/3}$.
2: Compute the residual $\mathbf{r}_h = \mathbf{f}_h - \mathbf{A}_h\mathbf{u}_h^{n+1/3}$, **restrict** it: $\mathbf{r}_{2h} = \mathbf{I}_{2h}^h\mathbf{r}_h$.
3: Solve $\mathbf{A}_{2h}\mathbf{e}_{2h} = \mathbf{r}_{2h}$.
4: **Prolongate** $\mathbf{e}_h = \mathbf{I}_h^{2h}\mathbf{e}_{2h}$ and correct $\mathbf{u}_h^{n+2/3} = \mathbf{u}_h^{n+1/3} + \mathbf{e}_h$.
5: ν_2 smoothing iterations on $\mathbf{A}_h\mathbf{u}_h = \mathbf{f}_h$: $\mathbf{u}_h^{n+2/3} \to \mathbf{u}_h^{n+1}$.

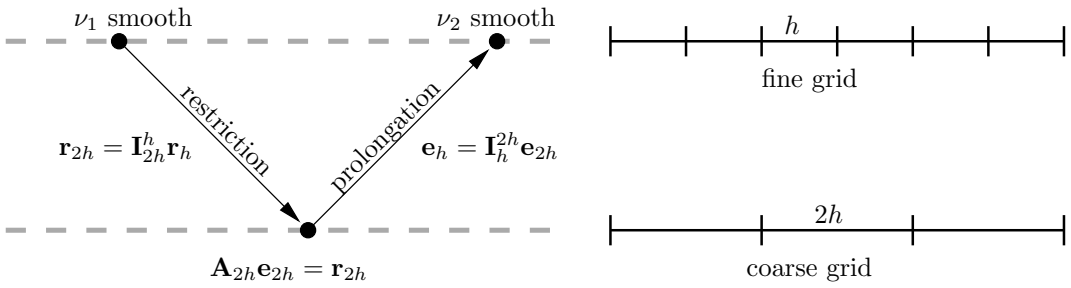


Figure 3.6.2: Two-grid correction scheme.

The first step is ν_1 smoothing iterations on the fine grid, the goal of which is to damp high-frequency error components. Next, we transfer the problem to the coarse grid in a process called *restriction*. To avoid having to transfer the state, we work with the *error equation*, driven by the *residual*, which is defined as

$$\begin{aligned} \mathbf{r}_h &\equiv \mathbf{f}_h - \mathbf{A}_h\mathbf{u}_h^n && (3.6.2)\\ &= \mathbf{f}_h - \mathbf{A}_h\mathbf{u}_h^n - \underbrace{(\mathbf{f}_h - \mathbf{A}_h\mathbf{u}_h^\infty)}_{\mathbf{0}} = \mathbf{A}_h\underbrace{(\mathbf{u}_h^\infty - \mathbf{u}_h^n)}_{\mathbf{e}_h^n} = \mathbf{A}_h\mathbf{e}_h^n, && (3.6.3)\end{aligned}$$

where $\mathbf{u}_h^\infty$ is the exact solution on the grid, and $\mathbf{e}_h^n$ is the error at the n^{th} iteration. We solve this error equation approximately on the coarse grid, by first restricting the residual from grid h to grid $2h$. This requires transferring residual values from the nodes of the fine mesh to the nodes of the coarse mesh. Figure 3.6.3 shows two possible ways of accomplishing this in one dimension, assuming the coarse-grid nodes exactly overlap every other fine one. In the case of simple injection, only the residuals at the overlapping nodes are used, copied over directly, while the in-between nodes are ignored. On the other hand, for full weighting, each fine-space node contributes, with the non-overlapping nodes splitting their contributions equally

Figure 3.6.3: Residual restriction using simple injection and full weighting in one dimension.

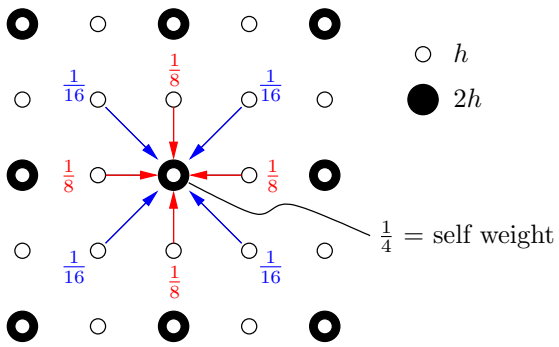


Figure 3.6.4: Full-weighting residual restriction in two dimensions.

among the two adjacent coarse neighbors. Figure 3.6.4 shows the extension of full-weighting restriction to two dimensions.

We denote the residual restriction operator by $\mathbf{I}_{2h}^{h}$. Following residual restriction, $\mathbf{r}_{2h} = \mathbf{I}_{2h}^{h}\mathbf{r}_h$, we solve the coarse-grid error equation: $\mathbf{A}_{2h}\mathbf{e}_{2h} = \mathbf{r}_{2h}$. For now we assume an exact solution of this equation. The next step is to transfer the error $\mathbf{e}_{2h}$ back to the fine grid, to correct the state. This step is called *prolongation*, and this is typically done using interpolation, as shown in Figure 3.6.6. The overlapping nodes get the error exactly, whereas the non-overlapping nodes take the average of the two coarse neighbors. Figure 3.6.5 shows the extension of prolongation to two dimensions.

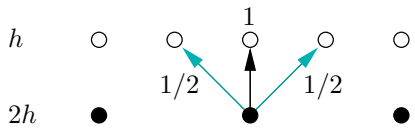


Figure 3.6.5: State/error prolongation using interpolation.

We denote the prolongation operator by $\mathbf{I}_{h}^{2h}$. Following error prolongation, $\mathbf{e}_h = \mathbf{I}_{h}^{2h}\mathbf{e}_{2h}$, we add this error to the fine state and perform ν_2 smoothing iterations. These are necessary because interpolation of the error from the coarse to the fine mesh creates additional errors. Fortunately these are high frequency and can be eliminated quickly by smoothing.

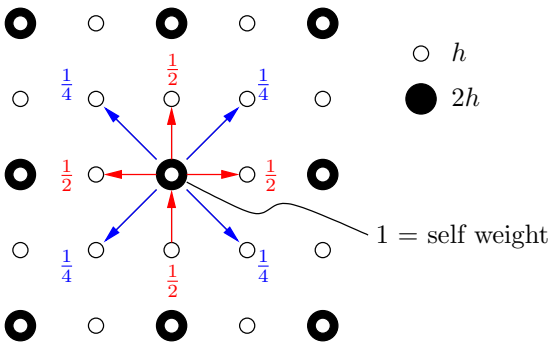


Figure 3.6.6: State/error prolongation in two dimensions. The numbers indicate the fraction of the error at the coarse grid node that is added to each of the fine grid nodes, indicated by the arrows.

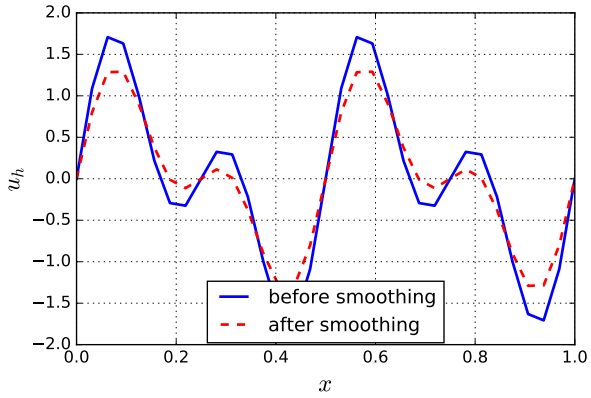
Example 3.3 (Two-grid correction for 1D Poisson)**.** We apply the two-grid correction scheme to a modified version of the one-dimensional problem in Equation 3.5.12,

$$-u_{xx} = 0, \quad u(0) = u(1) = 0, \quad u^0(x) = \sin(5\pi x) + \sin(15\pi x).$$

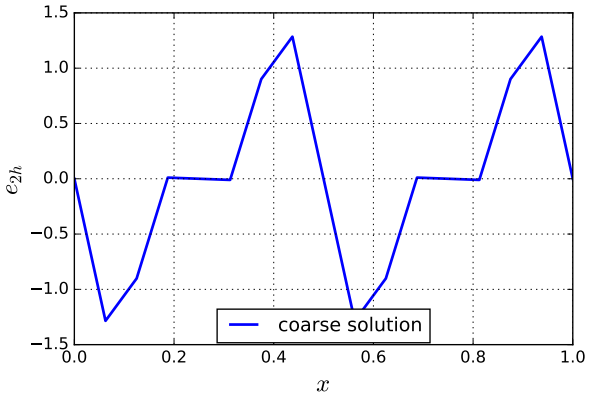
The fine mesh has $N_h = 32$ and $h = 1/32$, and the coarse mesh has $N_{2h} = 16$ and $2h = 1/16$. We use under-relaxed Jacobi as a smoother, with $\omega = 2/3$. Figure 3.6.7 shows the distribution of the solution and error on the fine and coarse grids at various stages of the two-grid correction scheme. Starting with subfigure (a), we see the initial condition, $u_h^0(x)$, and the effect of the ν_1 pre-smoothing iterations. With $\nu_1 = 2$, the error does not drop significantly in this step. The problem is then transferred to the coarse mesh and solved exactly. Subfigure (b) shows the resulting coarse solution, which is the error that when added to u_h should cancel much of the low-frequency content in u_h. We see from subfigure (c) that this is indeed the case (note the y axis range). The error before post-smoothing is oscillatory, due to high-frequency interpolation errors. However, the post-smoothing iterations take care of this and reduce the error to smooth components. This completes one two-grid correction cycle, and the process repeats with the next cycle in subfigures (d)-(f). Note that after just two cycles, the error has dropped dramatically, with the maximum amplitude below .005, whereas the starting value was 1.7. Figure 3.6.8 shows this quantitatively in a plot of the L_2 norm of the fine-grid solution/error versus two-grid correction cycles.

3.6.4 Multiple Grids

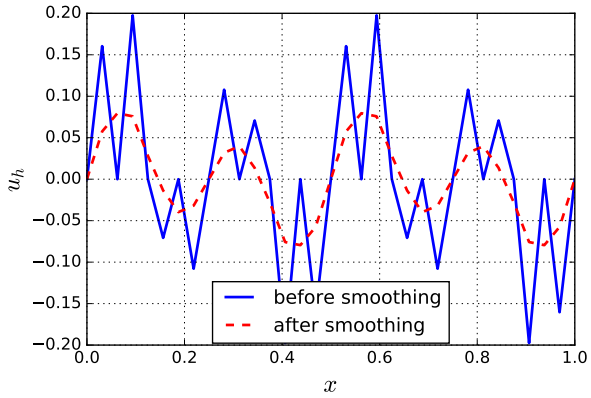
The two-grid scheme of the previous section requires an exact solve on the coarse grid. But the exact solution of this problem is generally still quite expensive. The remedy is to apply the coarse-grid correction recursively, using even coarser grids, until the coarsest grid has so few degrees of freedom that it is trivial to solve exactly. This is the idea underlying the



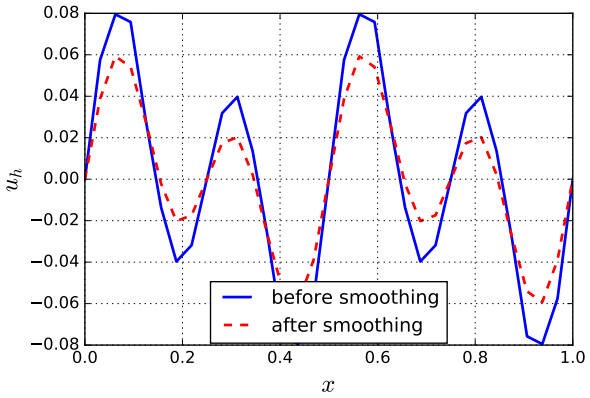
(a) Iteration 1: Pre-smoothing



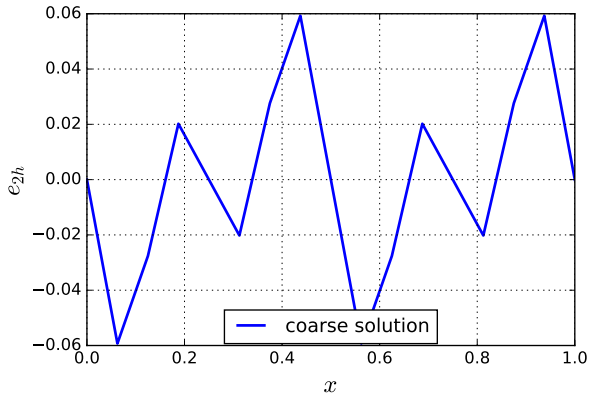
(b) Iteration 1: Coarse solve



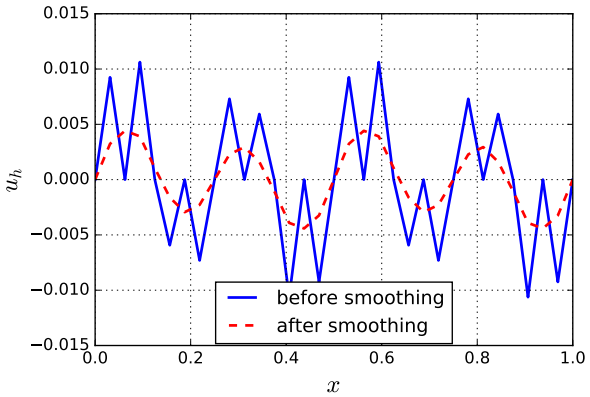
(c) Iteration 1: Post-smoothing



(d) Iteration 2: Pre-smoothing



(e) Iteration 2: Coarse solve



(f) Iteration 2: Post-smoothing

Figure 3.6.7: Error distribution at various stages of the two-grid correction scheme applied to the 1D Poisson equation. Note the changes in the range of the y axis.

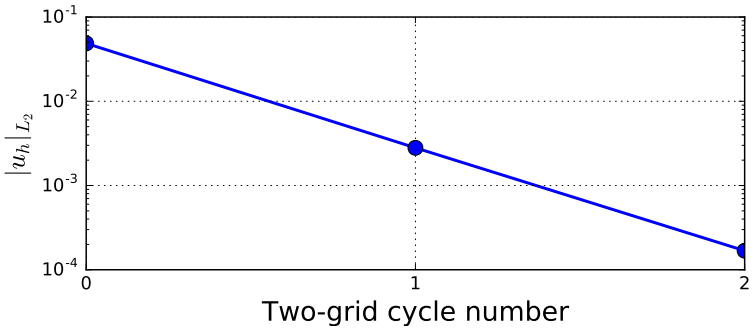


Figure 3.6.8: Convergence of the L_2 norm of the solution error (equal to the solution in this case) for the two-grid correction scheme applied to the 1D Poisson equation.

V-cycle, which is illustrated in Figure 3.6.9 – the "V" refers to the shape formed by the order in which the grids are visited.

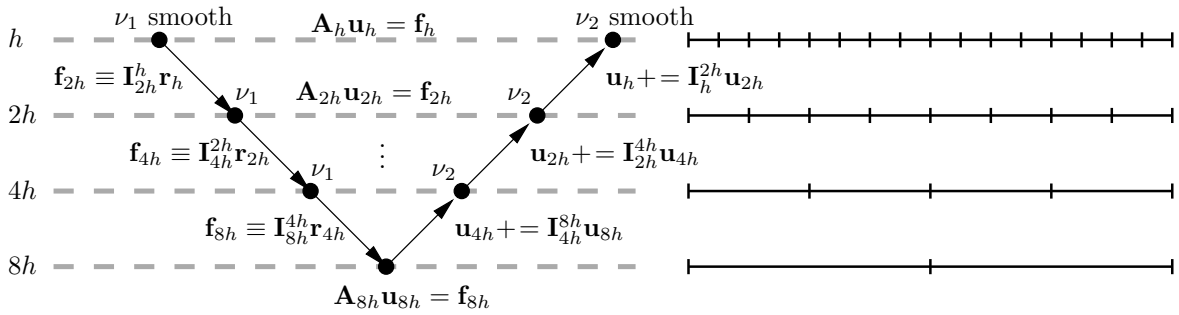


Figure 3.6.9: Extension of the two-grid correction scheme to the V-cycle. The notation has changed from the two-grid scheme in that the error calculated on the coarser levels is denoted by $\mathbf{u}$, and the restricted residual is denoted by $\mathbf{f}$. This makes the problems on the various grids of similar form. However, note that each successively coarser grid solves an *error equation* to correct the solution on the finer grid.

The V-cycle is not the only option for multiple grids. Two other popular cycles are the W-cycle and full multigrid (FMG). Figure 3.6.10 illustrates the order in which the grids are visited for these cycles. The white dots indicate application of an iterative smoother on a grid. The number of V-cycles on the coarser levels of the W-cycle and full-multigrid (FMG) cycles can be increased to do more work on the coarser grids, which can be advantageous when the smoother is particularly weak at damping low frequency errors. When a direct solve is not possible on the coarsest grid, a large number of smoothing iterations can be applied at little cost.

The total cost of each multigrid iteration (i.e. cycle) is often given in terms of *work units*. This measure is based on the cost of one smoothing iteration on the fine grid; that is, one **work unit (WU)** is equal to the cost of one iteration of the smoother on grid h. In

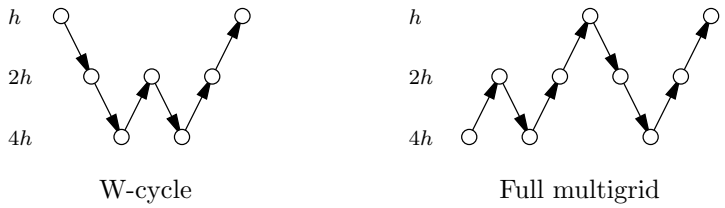


Figure 3.6.10: W-cycle and full multigrid (FMG), shown for three grid levels.

general, the cost of grid transfer is not included in the calculation of the total work units. For example, in a one-dimensional problem with two grids, if a two-grid V cycle consisted of 2 pre-smoothing iterations on h, 10 coarse-smoothing iterations on $2h$, and 2 post-smoothing iterations on h, the total number of work units would be $2 + 0.5(10) + 2 = 9$. Note that smoothing on $2h$ is half as expensive as smoothing on h when in one dimension. In two dimensions, smoothing on $2h$ would be four times less expensive; in three dimensions, the reduction would be a factor of 8!

As mentioned in the motivation portion of this section, multigrid can converge to the solution at a cost that is directly proportional to the number of unknowns. Since smoothing is already directly proportional to the number of unknowns, this means that the number of multigrid iterations required to achieve a certain error must not increase as the mesh is refined. Of course, the number of multigrid levels can and will increase, but this has a fixed, limited effect on the cost, since each coarser grid is half the cost of its finer predecessor. This mesh-independence property of multigrid can be proven by induction, and in the following example, we demonstrate it using a V-cycle in one dimension.

Example 3.4 (V-cycle convergence and mesh-independence)**.** We apply V-cycle multigrid to the same one-dimensional problem considered in Example 3.3. The coarsest mesh has two intervals, and the restricted problem on this mesh, with just 3 unknowns, is always solved exactly. We then vary the refinement of the finest mesh, which changes the number of multigrid levels. For example, for two levels, the finest mesh has $N = 4$; for three levels, the finest mesh has $N = 8$; etc. On each level we perform $\nu_1 = 2$ pre-smoothing and $\nu_2 = 2$ post-smoothing iterations, with $\omega = 2/3$ Jacobi.

Figure 3.6.11 shows the convergence of the L_2 error in the solution as a function of multigrid iterations for various numbers of levels. In this plot, N refers to the number of intervals on the finest mesh. We see that when the number of intervals increases from 8 to 16, the number of V-cycle iterations also increases. However, beyond this jump, the number of multigrid iterations required to converge the solution remains nearly constant with subsequent increases in N and the number of levels. This demonstrates the mesh independence of the V-cycle multigrid iteration.

Example 3.5 (Work unit comparison of multigrid)**.** The previous example showed how the V cycle converges with multigrid iterations. For a fair comparison with smoothing alone,

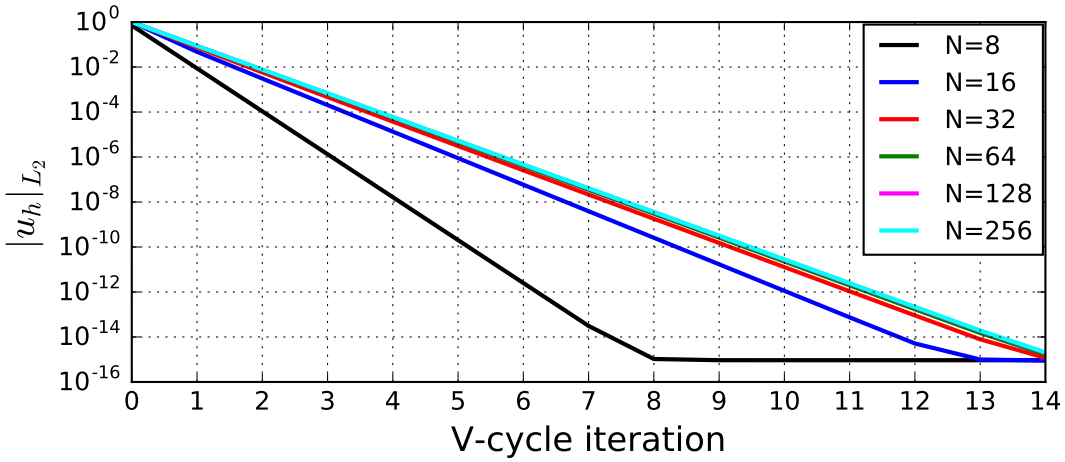


Figure 3.6.11: Convergence of V-cycle multigrid for the 1D Poisson equation, using various numbers of levels and a fixed coarse mesh of two intervals.

the cost measure should be work units, which measures the equivalent number of fine-space smoothing iterations. For a V-cycle on n_{level} levels, with ν_1 pre-smoothing and ν_2 post-smoothing iterations, the number of work units per multigrid cycle is

$$1 \text{ V cycle} = \sum_{l=0}^{n_{\text{level}}-1} (\nu_1 + \nu_2) 2^{-l} \text{ work units.}$$

In this estimate we replace the cost of the coarse-grid exact solve by the cost of $\nu_1 + \nu_2$ smoothing iterations on the coarse grid – both values are negligible when n_{level} is large.

Figure 3.6.12 presents the convergence of V-cycle multigrid as a function of work units, together with the convergence of stand-alone Jacobi iterations. We see that multigrid exhibits mesh-independent convergence with work units, whereas the convergence of Jacobi alone deteriorates as N increases.

3.6.5 Nonlinear Systems

We have presented multigrid for the linear system in Equation 3.6.1. Nonlinear systems cannot be written in the same form, and hence the application of multigrid must be modified. Here, we briefly describe the modifications necessary for the two-grid correction scheme, which can be generalized to V cycles and beyond.

A nonlinear system on the fine grid h can be written generally as

$$\mathbf{R}_h(\mathbf{u}_h) = \mathbf{f}_h. \tag{3.6.4}$$

Algorithm 7 and Figure 3.6.13 present the two-grid correction scheme for such a system. As in the linear case, the starting point for this method is the state at the n^{th} multigrid iteration

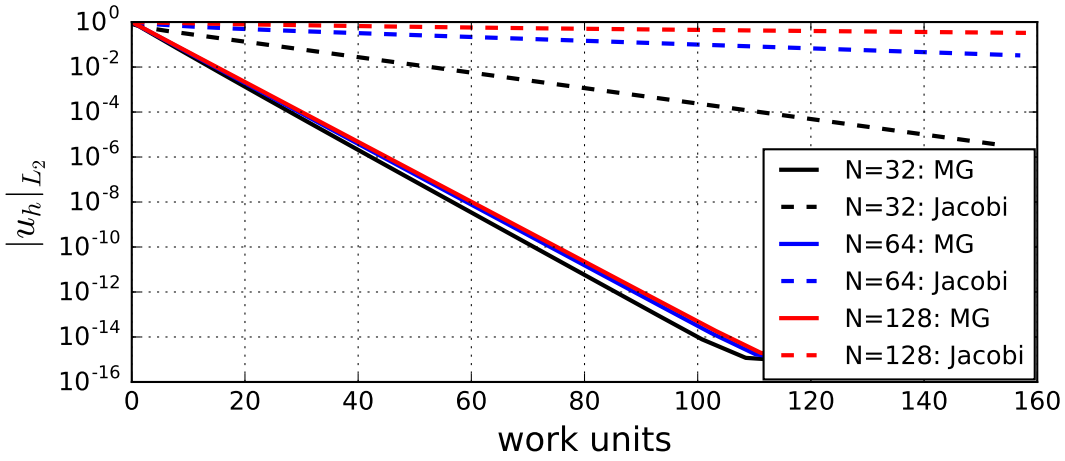


Figure 3.6.12: Comparison of the convergence of V-cycle multigrid and pure Jacobi iterations, as a function of work units.

on grid h, $\mathbf{u}_h^n$, and the end result is the state at the $(n+1)^{\text{th}}$ iteration. However, the primary difference between the linear and nonlinear case is that we cannot form an equation for just the error on the coarse grid when the equations are nonlinear. Instead, in the nonlinear case, we use a technique called **full-approximation storage** (FAS), in which the state is stored at all levels, in contrast to the linear case where the error replaced the state on the coarser levels. Full-approximation storage simulates solving the error equation in the context of a

Algorithm 7 A two-grid correction scheme for a nonlinear problem, using FAS.

1: Smooth $\mathbf{u}_h$ ν_1 times on the fine grid.
2: Compute residual $\mathbf{r}_h = \mathbf{f}_h - \mathbf{R}_h(\mathbf{u}_h)$, and restrict it: $\mathbf{r}_H = \mathbf{I}_H^h \mathbf{r}_h$.
3: Restrict the state to the coarse grid, $\mathbf{u}_H^0 = \widetilde{\mathbf{I}}_H^h \mathbf{u}_h$.
4: Solve for the error on the coarse grid, $\mathbf{R}_H(\mathbf{u}_H) = \mathbf{R}_H(\mathbf{u}_H^0) + \mathbf{r}_H$.
5: Prolongate $\mathbf{e}_h = \mathbf{I}_h^H(\mathbf{u}_H - \mathbf{u}_H^0)$ and correct $\mathbf{u}_h = \mathbf{u}_h + \mathbf{e}_h$.
6: Smooth $\mathbf{u}_h$ ν_2 times on the fine grid

modified problem on the coarse mesh. Note that when the fine grid residual (defined as $\mathbf{r}_h = \mathbf{f}_h - \mathbf{R}_h(\mathbf{u}_H)$) is zero, i.e. when the fine-space problem is already solved exactly, then $\mathbf{r}_H = \mathbf{0}$, so that the coarse solution is $\mathbf{u}_H = \mathbf{u}_H^0$ giving a zero correction, $\mathbf{e}_h = \mathbf{0}$, as expected. The key is including $\mathbf{R}(\mathbf{u}_H^0)$ on the right-hand side in the coarse-grid problem. Finally, we note that FAS requires the restriction of the state from the fine grid to the coarse grid, $\mathbf{u}_H^0 = \widetilde{\mathbf{I}}_H^h \mathbf{u}_h$. In our finite-difference setting, this transfer can be done using simple injection or full weighting, as shown in Figure 3.6.3. For other discretizations, such as finite elements, the state restriction can differ from the residual restriction, with the former performed via least-squares approximation.

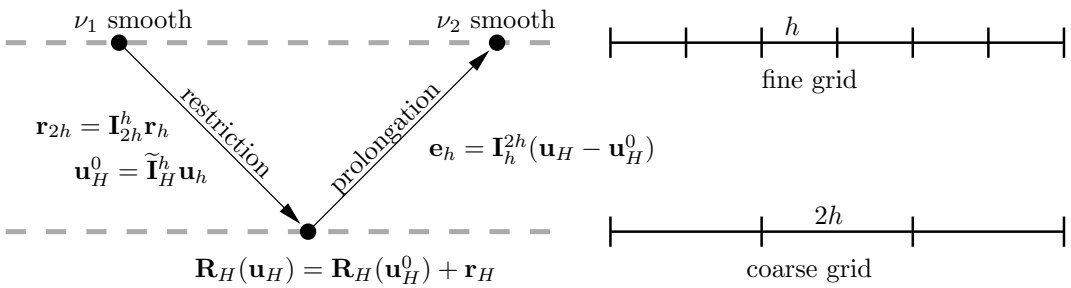


Figure 3.6.13: Two-grid correction scheme for a nonlinear problem, using full-approximation storage (FAS).

3.7 Krylov Subspace Methods

Multigrid works well for certain classes of problems, in particular elliptic boundary value problems on structured meshes. When other equations or other meshes are considered, the performance of multigrid generally deteriorates. In such cases a better alternative may lie in a **Krylov subspace method**.

Krylov subspace methods are among the most powerful techniques for solving large linear systems, including those arising in CFD discretizations. When solving the system $\mathbf{Au} = \mathbf{f}$, these methods seek the best solution in the vector space,

$$\mathcal{K}^n = \text{span}\left\{\mathbf{f},\ \mathbf{Af},\ (\mathbf{A})^2\mathbf{f},\ \ldots,\ (\mathbf{A})^n\mathbf{f}\right\}. \tag{3.7.1}$$

The vector $(\mathbf{A})^2\mathbf{f}$ consists of two applications of $\mathbf{A}$ to $\mathbf{f}$: $\mathbf{A}(\mathbf{Af})$. Hence, these vectors that span the space can be built efficiently by repeated applications of the operator $\mathbf{A}$

A popular Krylov subspace method in CFD is the generalized minimal residual method, GMRES, which picks the vector from $\mathcal{K}^n$, call it $\mathbf{u}_*$, that minimizes the L_2 norm of the residual, $||\mathbf{f} - \mathbf{Au}_*||$. Conceptually, we could just form the $n+1$ vectors in (3.7.1) and solve a least-squares minimization problem. In practice, such an approach would be extremely ill-conditioned, which means sensitive to round-off errors. Instead, we use a variant that employs Gram-Schmidt orthonormalization based on the Arnoldi iteration. A version of this algorithm that uses up to n_{outer} restarts is given in Algorithm 8. In this algorithm, $Givens(a, b)$ is a 2×2 counter-clockwise rotation matrix by $\tan^{-1}(b/a)$. The convergence rate of GMRES depends on the eigenvalue distribution of $\mathbf{A}$, where clustered eigenvalues are good. The largest storage requirement is the $n+1$ vectors $\mathbf{v}_j$.

GMRES is applicable to the asymmetric linear systems typically found in CFD. Another popular Krylov subspace method is the conjugate gradients (CG) method, which is applicable to symmetric systems. A competitive extension of CG for asymmetric systems is biconjugate gradients (BCG) and an improved variant, Bi-CGSTAB. Unlike GMRES, these methods do not minimize the residual norm, but, with a three-term recurrence relation, they obviate the need to store the entire Krylov subspace basis.

Finally, we mention that for most problems, Krylov methods like GMRES, CG, BCG, etc. need to be preconditioned to obtain satisfactory performance. A **preconditioner** is a

Algorithm 8 The generalized minimal residual method (GMRES).

$\mathbf{u} = 0$
for $i = 1 : n_{\text{outer}}$ **do**
  $\mathbf{v}_1 = \mathbf{A}\mathbf{u} - \mathbf{b}$, $\beta = ||\mathbf{v}_1||$, $\mathbf{v}_1 = \mathbf{v}_1/\beta$
  $\mathbf{g} = [\beta, 0, \ldots, 0] \in \mathbb{R}^{n+1}$
  for $j = 1 : n$ **do**
    $\mathbf{w} = \mathbf{A}\mathbf{v}_j$
    for $k = 1 : j$ **do**
      $H(k, j) = \mathbf{w}^T \mathbf{v}_k$
      $\mathbf{w} = \mathbf{w} - H(k, j)\mathbf{v}_k$
    end for
    $H(j + 1, j) = ||\mathbf{w}||$
    $\mathbf{v}_{j+1} = \mathbf{w}/||\mathbf{w}||$
    for $k = 1 : j - 1$ **do**,
      $H([k, k + 1], j) = \mathbf{G}_k H([k, k + 1], j)$
    end for
    $\mathbf{G}_j = Givens(H(j, j), H(j + 1, j))$
    $H([j, j + 1], j) = \mathbf{G}_j H([j, j + 1], j)$
    $g([j, j + 1]) = \mathbf{G}_j g([j, j + 1])$
    $g(j + 1)$ contains the L_2 residual norm; check for convergence
  end for
  $\mathbf{y} = \mathbf{H}^{-1}\mathbf{g}$, $\mathbf{V} = [\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_{n+1}]$, $\mathbf{u} = \mathbf{u} + \mathbf{V}\mathbf{y}$,
end for

pseudo-inverse to the matrix, one that is generally easy to apply. It accelerates the performance of GMRES because the resulting search space of vectors more quickly hones in on the true solution. The better the preconditioner approximates the inverse of $\mathbf{A}$, the faster the convergence.

Chapter 4

Initial Value Problems

4.1 Example

Consider an initial value problem for scalar diffusion, governed by Equation 1.5.2. This problem reads: determine $u(x,t)$ that satisfies

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x}\left(-\nu \frac{\partial u}{\partial x}\right) = 0, \quad x \in [0,L], \quad u(x,0) = u_0(x), \tag{4.1.1}$$

where $u_0(x)$ is a given initial condition on the domain $(0,L)$, and the boundary conditions are periodic: $u(L,t) = u(0,t)$. We solve this equation using the finite difference method, by placing a grid of points in both space and time, and using finite difference approximations for the derivatives. Assume that the points are placed uniformly with spacing Δx in space and Δt in time. Various finite difference approximations are possible, and we pick the following: a forward difference in time (FT), and a central difference in space (CS). This method will therefore be called FTCS. These finite difference approximations read

$$\begin{aligned} (u_t)_j^n &\approx \frac{u_j^{n+1} - u_j^n}{\Delta t}, \\ (u_{xx})_j^n &\approx \frac{u_{j-1}^n - 2u_j^n + u_{j+1}^n}{\Delta x^2}. \end{aligned}$$

The notation u_j^n refers to the state at space node j and time level n (this is *not* an exponent). Substituting these approximations into Equation 4.1.1, and assuming a constant viscosity ν, we have

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} - \nu \frac{u_{j-1}^n - 2u_j^n + u_{j+1}^n}{\Delta x^2} = 0. \tag{4.1.2}$$

From this equation we can explicitly solve for u_j^{n+1},

$$u_j^{n+1} = u_j^n + \underbrace{\frac{\nu \Delta t}{\Delta x^2}}_{\mu}\left(u_{j+1}^n - 2u_j^n + u_{j-1}^n\right), \tag{4.1.3}$$

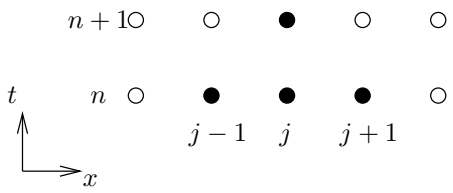


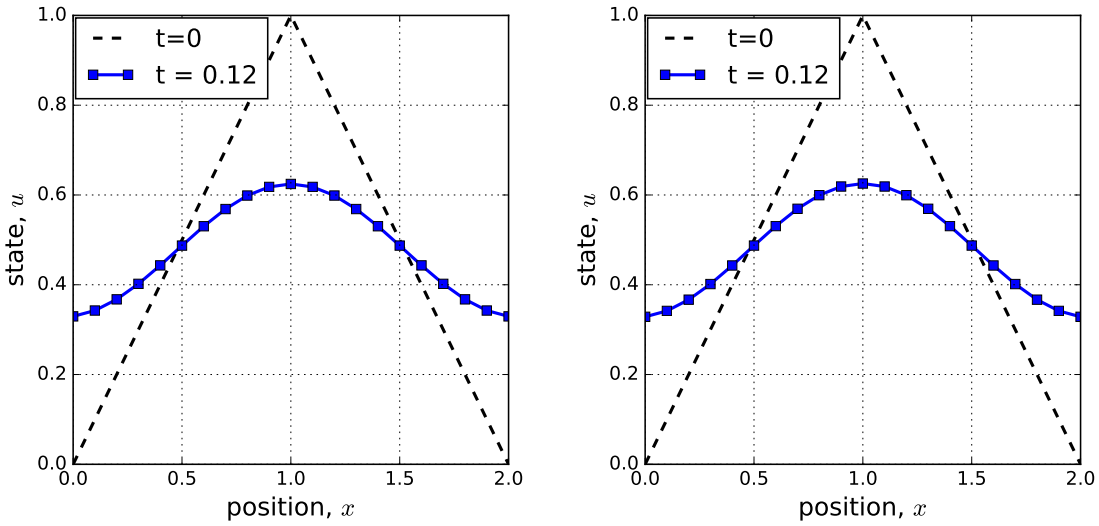
Figure 4.1.1: Stencil for the forward in time, centered in space (FTCS) scheme.

where μ is a non-dimensional parameter called the **heat number**. Figure 4.1.1 shows the stencil for the update in Equation 4.1.3. We see that the updated state depends on the current state at the same node and the two neighbors.

We apply the FTCS method to a case with $\nu = 1$, $L = 2$, initial condition $u_0(x) = 1 - |x - 1|$, and $\Delta x = 0.1$. Figure 4.2(a), generated using the code in Listing 4.1.1, shows how the solution behaves when $\mu = 0.25$. Note that the choice of heat number sets the time step according to $\Delta t = \mu \Delta x^2/\nu$. At $\mu = 0.25$, the solution behaves as expected, with the peak spreading out in time. What if we wanted to obtain the solution faster, i.e. with a larger Δt for a given Δx? The remaining plots in Figure 4.1.2 show the behavior of the solution for $\mu = 0.4$, $\mu = 0.5$, and $\mu = 0.6$. We see that the solution diverges (begins to blow up) for $\mu = 0.6$, and some trial and error shows that $\mu = 0.5$ is the limiting value for stability. We will see later in stability analysis that we can predict this limiting value of μ.

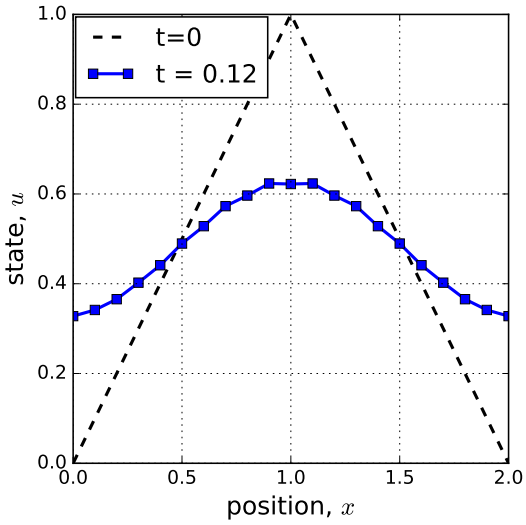
Listing 4.1.1: Code implementing FTCS for the diffusion problem.

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

def runheat(mu):
    Nx, T, L = 21, .12, 2.0
    x = np.linspace(0,L,Nx)
    Nt = int(np.ceil(T/(mu*(x[1]-x[0])**2)))
    U = np.zeros([Nx, Nt])
    U[:,0] = 1-np.abs(x-1.0)  # initial condition
    for n in range(Nt-1):
        for j in range(0,Nx):
            U[j,n+1] = U[j,n] + mu*(U[j-1,n]-2.*U[j,n]+U[np.mod(j+1,Nx),n])
    f = plt.figure( figsize=(5,5))
    plt.plot(x, U[:,0], '--', linewidth=2, color='black', label='t=0')
    plt.plot(x, U[:,Nt-1], 's-', linewidth=2, color='blue', label='t = %.3g'%(T))
    plt.xlabel(r'position, $x$', fontsize=16)
    plt.ylabel(r'state, $u$', fontsize=16)
    plt.legend( fontsize=16, borderaxespad=0.1, loc=2)
    plt.figure(f.number); plt.grid()
    plt.tick_params(axis='both', labelsize=12)
    f.tight_layout(); plt.show(block=False)
    plt.savefig('../figs/heat_ivp_mu%2.0f.pdf'%(mu*100))
    plt.close(f)

```

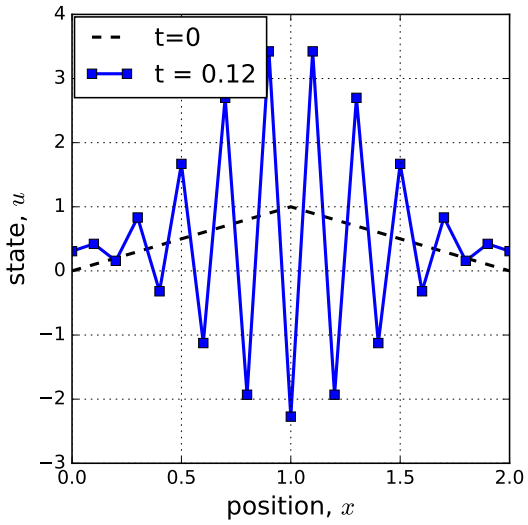


(a) $\mu = 0.25$

(b) $\mu = 0.4$



(c) $\mu = 0.5$



(d) $\mu = 0.6$

Figure 4.1.2: Initial ($t = 0$) and final ($t = 0.12$) time solutions to the scalar diffusion problem in Equation 4.1.1, using FTCS with different heat numbers, μ.

```
25 def main():
26     muv = [.25, .4,  .5,  .6]
27     for mu in muv: runheat(mu)
28
29 if __name__ == "__main__":
30     main()
```

4.2 Fully-Discrete Schemes for Advection

The physical process of fluid transport plays an important role in CFD, and the prototypical equation that models transport is linear advection. For a scalar u propagating at constant speed a, this equation reads

$$\frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} = 0. \tag{4.2.1}$$

In this section we present several classical finite-difference solution techniques for this equation, assuming a constant spatial node spacing of Δx and temporal spacing of Δt. These are called **fully-discrete** schemes because the temporal and spatial derivatives are discretized simultaneously, resulting in an update equation that potentially mixes state values at different spatial and temporal nodes.

FTCS = Forward in Time, Centered in Space

This scheme goes by the same name as the one already introduced for diffusion in the previous section. However since advection involves only a first derivative in space, the method is different. The approximations are

$$u_t \approx \frac{u_j^{n+1} - u_j^n}{\Delta t}, \quad u_x \approx \frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x}, \tag{4.2.2}$$

where again j indicates the spatial node index and n the temporal node index. Substituting into Equation 4.2.1 results in

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} + a\frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x} = 0. \tag{4.2.3}$$

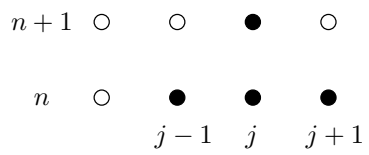
From this equation we can explicitly solve for the updated state at node j,

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

Defining the **CFL number** as $\boxed{\sigma \equiv a\Delta t/\Delta x}$, we rewrite the update in non-dimensional form,

$$\boxed{u_j^{n+1} = u_j^n - \frac{\sigma}{2}(u_{j+1}^n - u_{j-1}^n)} \tag{4.2.4}$$

The stencil of points involved in this update is shown below.



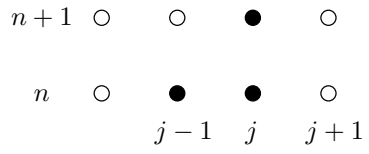
FOU = First Order Upwind

Assuming $a > 0$, the finite-difference approximations in this scheme read

$$u_t \approx \frac{u_j^{n+1} - u_j^n}{\Delta t}, \quad u_x \approx \frac{u_j^n - u_{j-1}^n}{\Delta x}.$$

If $a < 0$, the spatial approximation would involve nodes $j+1$ and j instead of j and $j-1$. Substituting these approximation into Equation 4.2.1 and solving for u_j^{n+1} gives

$$\boxed{u_j^{n+1} = u_j^n - \sigma(u_j^n - u_{j-1}^n)} \tag{4.2.5}$$



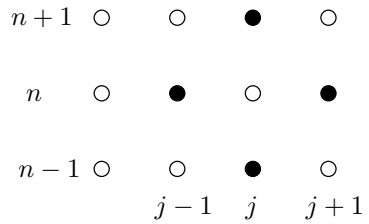
Leapfrog

This scheme uses a centered difference for both the spatial and the temporal derivatives,

$$u_t \approx \frac{u_j^{n+1} - u_j^{n-1}}{2\Delta t}, \quad u_x \approx \frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x}.$$

Substituting these into the advection equation and solving for u_j^{n+1} gives

$$u_j^{n+1} = u_j^{n-1} - \sigma(u_{j+1}^n - u_{j-1}^n). \tag{4.2.6}$$



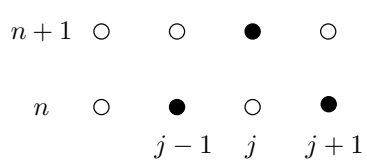
Lax-Friedrichs

This scheme is similar to FTCS, except that the temporal derivative approximation uses the average of the spatial neighbors instead of the state at node j,

$$u_t \approx \frac{u_j^{n+1} - (u_{j-1}^n + u_{j+1}^n)/2}{\Delta t}, \quad u_x \approx \frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x}.$$

We can again solve for u_j^{n+1} explicitly, resulting in

$$u_j^{n+1} = \frac{1}{2}(u_{j-1}^n + u_{j+1}^n) - \frac{\sigma}{2}(u_{j+1}^n - u_{j-1}^n).$$



Lax-Wendroff

This is a classic scheme that is at the heart of many other schemes. It uses a novel "trick" whereby temporal and spatial derivatives are interchanged via the original differential equation. The derivation begins with a Taylor-series expansion of the state in time at time node $n+1$, about time node n,

$$u_j^{n+1} = u_j^n + \Delta t(u_t)_j + \frac{\Delta t^2}{2}(u_{tt})_j + \mathcal{O}(\Delta t^3). \tag{4.2.7}$$

The trick is to now use Equation 4.2.1 to substitute a spatial derivative for u_t at node j,

$$(u_t)_j = -a(u_x)_j.$$

Differentiating this equation with respect to time gives, at node j,

$$(u_{tt})_j = -a(u_{xt})_j = -a((u_t)_x)_j = -a((-au_x)_x)_j = a^2(u_{xx})_j.$$

Substituting both of these expressions into Equation 4.2.7 gives

$$u_j^{n+1} = u_j^n + -a(u_x)_j\Delta t + a^2(u_{xx})_j\frac{\Delta t^2}{2} + \mathcal{O}(\Delta t^3). \tag{4.2.8}$$

Using second-order central difference formulas for the space derivatives,

$$(u_x)_j \approx \frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x}, \quad (u_{xx})_j \approx \frac{u_{j-1}^n - 2u_j^n + u_{j-1}^n}{\Delta x^2},$$

yields

$$u_j^{n+1} = u_j^n - \frac{a\Delta t}{2\Delta x}\left(u_{j+1}^n - u_{j-1}^n\right) + \frac{a^2\Delta t^2}{2\Delta x^2}\left(u_{j-1}^n - 2u_j^n + u_{j+1}^n\right).$$

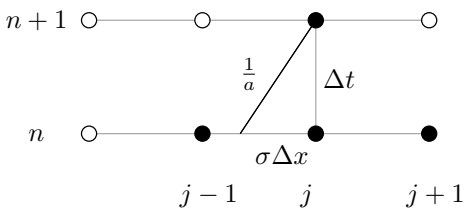
In terms of the non-dimensional CFL number, $\sigma = a\Delta t/\Delta x$, this equation reads

$$\boxed{u_j^{n+1} = u_j^n - \frac{\sigma}{2}(u_{j+1}^n - u_{j-1}^n) + \frac{\sigma^2}{2}(u_{j-1}^n - 2u_j^n + u_{j+1}^n)} \tag{4.2.9}$$

Another interpretation of the Lax-Wendroff method comes from rewriting the above equation as

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

which is a quadratic interpolation of the state between the nodes $j-1$, j, and $j+1$. Specifically, the factors in front of the nodal states in the above equation are the Lagrange quadratic polynomials evaluated at $x = -\sigma\Delta x$ relative to the central node. So the Lax-Wendroff method can be interpreted as tracing the characteristic (slope $1/a$ in the x,t diagram) from node $j, n+1$ back to time level n, where the position is $-\sigma\Delta x$ to the left of node j, and using a quadratic state reconstruction at time level n, based on nodes $j-1$, j, and $j+1$.



Beam-Warming

This scheme is similar to Lax-Wendroff, in that space derivatives replace time derivatives. Thus, the starting point is Equation 4.2.8. However, Beam-Warming uses different finite-difference approximations for u_x and u_{xx}:

$$(u_x)_j \approx \frac{3u_j^n - 4u_{j-1}^n + u_{j-2}^n}{2\Delta x} + \mathcal{O}(\Delta x^2), \quad (u_{xx})_j \approx \frac{u_{j-2}^n - 2u_{j-1}^n + u_j^n}{\Delta x^2} + \mathcal{O}(\Delta x).$$

The first derivative is approximated by a second-order accurate backward difference, with nodes $j-2, j-1, j$. The second derivative is approximated by a central difference ... but centered at $j-1$, not at j. The reason for this mismatch is that the scheme is designed to use only nodes $j-2$, $j-1$, and j, whereas centering the difference about node j would bring in node $j+1$. The mismatch reduces the order of accuracy of the u_{xx} difference by one, to first-order, but this does not affect the scheme's second-order accuracy, because u_{xx} gets multiplied by Δt^2 in Equation 4.2.8. Substituting the above derivatives into Equation 4.2.8 results in

$$u_j^{n+1} \;=\; u_j^n - \frac{a\Delta t}{2\Delta x}\left(3u_j^n - 4u_{j-1}^n + u_{j-2}^n\right) + \frac{a^2\Delta t^2}{2\Delta x^2}\left(u_{j-2}^n - 2u_{j-1}^n + u_j^n\right).$$

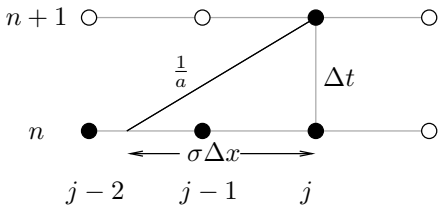
In terms of the CFL number, σ, this equation reads

$$\boxed{u_j^{n+1} = u_j^n - \frac{\sigma}{2}(3u_j^n - 4u_{j-1}^n + u_{j-2}^n) + \frac{\sigma^2}{2}(u_{j-2}^n - 2u_{j-1}^n + u_j^n)} \tag{4.2.10}$$

In a similar fashion to Lax-Wendroff, another interpretation of the Beam-Warming scheme comes from rewriting the above equation as

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

which is a quadratic interpolation of the state between the nodes $j-2$, $j-1$, and j. The factors in front of the nodal states in the above equation are the Lagrange quadratic polynomials evaluated at $x = -\sigma\Delta x$ relative to the central node. So Beam-Warming traces the characteristic (slope $1/a$) from node $j, n+1$ back to time level n, where the position is $-\sigma\Delta x$ to the left of node j, and uses a quadratic state reconstruction at time level n, from nodes $j-2$, $j-1$, and j.



4.3 Semi-Discrete Methods

In the previous section we discretized both space and time at once, using finite differences on a set of points in space and time. A slightly less flexible but more systematic approach is to discretize the PDE in space first, leaving the time derivatives alone. This is then called the **semi-discrete** form of the equation, an ordinary differential equation (ODE) that can be written generally as

$$\frac{d\mathbf{u}}{dt} = \mathbf{f}(\mathbf{u}, t), \tag{4.3.1}$$

where $\mathbf{u}$ is the vector of spatial unknowns, i.e. the solution values at the grid points, and $\mathbf{f}$ is the result of the spatial discretization. Note, no superscript n is present on any of the values because we have not yet discretized time. Rather, time is discretized in the next, separate step, and various time-integration techniques can be used.

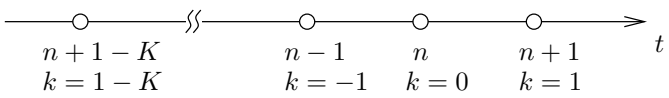
Some of the schemes from the previous section could be written in semi-discrete form, e.g. first-order upwind, but others could not, e.g. Lax-Wendroff. In the Lax-Wendroff case, we replaced temporal derivatives with spatial derivatives, via a reuse of the PDE, and this step could not be done separately through a spatial and temporal discretization. In the present section we look at schemes for time-integrating Equation 4.3.1.

4.3.1 Multi-Step Methods

Multi-step methods relate an unknown state at time node $n+1$ to previous states and slopes, on a temporal grid with constant time steps, Δt. The formula for a K-step method applied

to Equation 4.3.1 is

$$\sum_{k=1-K}^{1} \alpha_k \mathbf{u}^{n+k} = \Delta t \sum_{k=1-K}^{1} \beta_k \mathbf{f}(\mathbf{u}^{n+k}, t^{n+k}), \tag{4.3.2}$$



where α_k, β_k are coefficients that define the method. Note the relationship between k and n in the diagram. Without loss of generality, we set $\alpha_1 = 1$, and our goal is to solve for $\mathbf{u}^{n+1}$. When $\beta_1 = 0$, $\mathbf{u}^{n+1}$ does not appear on the right-hand side and we have an **explicit** formula for $\mathbf{u}^{n+1}$. However, when $\beta_1 \neq 0$, $\mathbf{u}^{n+1}$ appears within $\mathbf{f}$ on the right-hand side and we must solve an **implicit** equation.

One of the simplest multi-step methods is forward Euler, which discretizes the time derivative in Equation 4.3.1 with a forward difference. Table 4.3.1 presents this and several other popular multi-step methods, together with their temporal orders of accuracy. Methods that require more than one previous state (e.g. BDF2), must be started with another method (e.g. BDF1).

Table 4.3.1: Coefficients of several multi-step methods.

Name[a]	Order	α^{-2}	α^{-1}	α^{0}	α^{1}	β^{-2}	β^{-1}	β^{0}	β^{1}
FE	1	0	0	-1	1	0	0	1	0
AB2	1	0	0	-1	1	0	$-\frac{1}{2}$	$\frac{3}{3}$	0
AB3	1	0	0	-1	1	$\frac{5}{12}$	$-\frac{16}{12}$	$\frac{23}{12}$	0
BDF1	1	0	0	-1	1	0	0	0	1
BDF2	2	0	$\frac{1}{2}$	-2	$\frac{3}{2}$	0	0	0	1
BDF3	3	$\frac{11}{6}$	-3	$\frac{3}{2}$	$-\frac{1}{3}$	0	0	0	1
AM2	2	0	0	-1	1	0	0	$\frac{1}{2}$	$\frac{1}{2}$
AM3	3	0	0	-1	1	0	$-\frac{1}{12}$	$\frac{8}{12}$	$\frac{5}{12}$

[a] FE = forward Euler, AB = Adams-Bashforth, BDF = backward difference formula, AM = Adams-Moulton, AM2 = TRAP = trapezoidal rule, BDF1 = BE = backward Euler

4.3.2 Multi-Stage Methods

The accuracy of multi-step methods depends on the number of preceding time nodes used. Since these are known, calculating the next state is inexpensive. However, when a long

history of states is required, the method must be started with a different formula for initial states where the full history is not available. In addition, high-order multi-step methods generally exhibit poor stability, particularly for lightly-damped oscillatory modes. Multi-stage methods, illustrated in Figure 4.3.1, address these issues, albeit with an increased cost per time step.

As illustrated in Figure 4.3.1, multi-stage methods use temporary intermediate states to compute the state at time node $n+1$ using only the previous state at time node n.

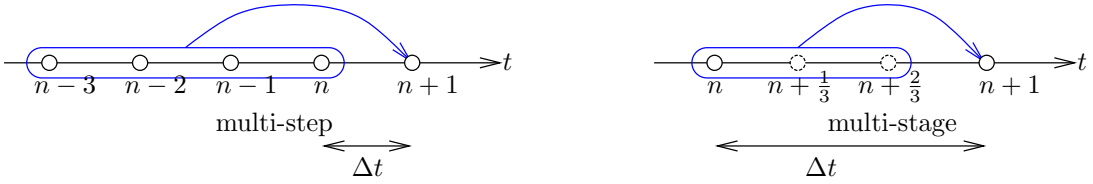


Figure 4.3.1: Schematic difference between multi-step and multi-stage time integration methods. Multi-stage methods use temporary intermediate states in the calculation of the $n+1$ state from the n state.

Two-Stage Runge Kutta

We present two second-order accurate two-stage methods. The first of these, known as a **predictor-corrector** method, is

$$\begin{aligned} \mathbf{f}_0 &= \mathbf{f}(\mathbf{u}^n, t^n) \\ \mathbf{f}_1 &= \mathbf{f}(\mathbf{u}^n + \Delta t \mathbf{f}_0, t^{n+1}) \\ \mathbf{u}^{n+1} &= \mathbf{u}^n + \tfrac{1}{2}\Delta t\,(\mathbf{f}_0 + \mathbf{f}_1) \end{aligned} \tag{4.3.3}$$

The two-stages are the two evaluations of $\mathbf{f}$. The method is second-order accurate because it uses the average of the slope at the current state and at a (rough) prediction of the $n+1$ state.

Another popular two-stage method is **modified Euler**,

$$\begin{aligned} \mathbf{f}_0 &= \mathbf{f}(\mathbf{u}^n, t^n) \\ \mathbf{f}_1 &= \mathbf{f}(\mathbf{u}^n + \tfrac{1}{2}\Delta t \mathbf{f}_0, t^{n+\frac{1}{2}}) \\ \mathbf{u}^{n+1} &= \mathbf{u}^n + \Delta t \mathbf{f}_1 \end{aligned} \tag{4.3.4}$$

In the first stage, forward Euler is used to obtain an approximation to the state at the midpoint of the time step. The slope, $\mathbf{f}$, at this midpoint state is then used to advance the state from node n to $n+1$. Figure 4.3.2 presents these steps graphically for a scalar problem.

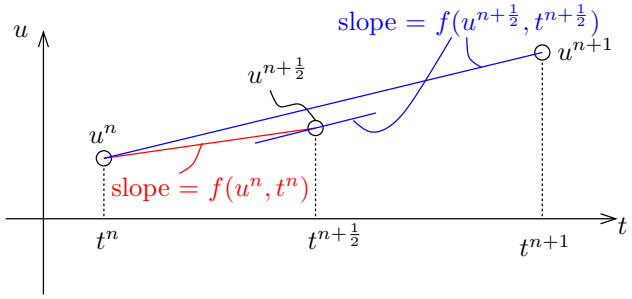


Figure 4.3.2: Graphical representation of the two stages in modified Euler time integration.

Four-Stage Runge Kutta

One of the most popular four-stage integration methods is the following sequence:

$$\begin{aligned} \mathbf{f}_0 &= \mathbf{f}\left(\mathbf{u}^n,\, t^n\right) \\ \mathbf{f}_1 &= \mathbf{f}\left(\mathbf{u}^n + \frac{1}{2}\Delta t \mathbf{f}_0,\, t^n + \frac{\Delta t}{2}\right) \\ \mathbf{f}_2 &= \mathbf{f}\left(\mathbf{u}^n + \frac{1}{2}\Delta t \mathbf{f}_1,\, t^n + \frac{\Delta t}{2}\right) \\ \mathbf{f}_3 &= \mathbf{f}\left(\mathbf{u}^n + \Delta t \mathbf{f}_2,\, t^n + \Delta t\right) \\ \mathbf{u}^{n+1} &= \mathbf{u}^n + \frac{\Delta t}{6}\left(\mathbf{f}_0 + 2\mathbf{f}_1 + 2\mathbf{f}_2 + \mathbf{f}_3\right) \end{aligned} \tag{4.3.5}$$

This method requires four evaluations of $\mathbf{f}$ per time step, and it is fourth-order accurate.

4.3.3 Stiffness and Implicit Methods

A *stiff* system of ODEs exhibits a wide range of time scales, i.e. contains both fast and slow processes. Mathematically, this means that the magnitudes of the eigenvalues of $\partial\mathbf{f}/\partial\mathbf{u}$ are highly-disparate, with ratios of three or more orders of magnitude. Often the large-magnitude eigenvalues represent fast, highly-damped transients that do not affect the rest of the system and hence need not be accurately tracked. However, if a numerical time integration scheme is not stable for these eigenvalues, the transients will diverge and quickly destroy the solution. This happens for explicit methods, which we will soon see have time step restrictions that are inversely proportional to the eigenvalue magnitude, so that stability comes at a hefty price of a very small time step. On the other hand, *implicit methods* generally exhibit much better stability properties and are therefore more suitable for stiff systems.

In an implicit multi-step method, the update formula for $\mathbf{u}^{n+1}$, i.e. Equation 4.3.2, contains the unknown $\mathbf{u}^{n+1}$ in the forcing function $\mathbf{f}$ on the right-hand side. Solving for $\mathbf{u}^{n+1}$ therefore requires more work than in an explicit method, where we could easily write down an expression for $\mathbf{u}^{n+1}$ in terms of the previous states.

The prototypical implicit time integration method is BDF1, or backward Euler,

$$\mathbf{u}^{n+1} - \mathbf{u}^n = \Delta t \mathbf{f}(\mathbf{u}^{n+1}, t^{n+1}). \tag{4.3.6}$$

Let's first consider the case of a linear system, where $\mathbf{f}(\mathbf{u}, t) = \mathbf{A}\mathbf{u}$. To solve Equation 4.3.6 we put terms with the unknown $\mathbf{u}^{n+1}$ on the left-hand side and everything else on the right-hand side,

$$\begin{aligned} \mathbf{u}^{n+1} - \Delta t \mathbf{A}\mathbf{u}^{n+1} &= \mathbf{u}^n \\ (\mathbf{I} - \Delta t \mathbf{A})\,\mathbf{u}^{n+1} &= \mathbf{u}^n \\ \mathbf{u}^{n+1} &= (\mathbf{I} - \Delta t \mathbf{A})^{-1}\,\mathbf{u}^n \end{aligned}$$

The backward Euler update therefore requires solving an $N \times N$ system, and this will be the case for all implicit methods. Fortunately, most systems in CFD will be sparse, and hence the cost will not scale as $\mathcal{O}(N^3)$. Yet, the additional cost of the solve has to be factored into the decision when choosing between explicit and implicit methods.

When $\mathbf{f}$ is a nonlinear function of $\mathbf{u}$, we have to solve a nonlinear system of algebraic equations at every time step. Re-arranging Equation 4.3.6, assuming a time-independent $\mathbf{f}$, gives the update formula,

$$\begin{aligned} \mathbf{u}^{n+1} &= \mathbf{u}^n + \Delta t \mathbf{f}(\mathbf{u}^{n+1}) \\ \mathbf{u}^{n+1} - \Delta t \mathbf{f}(\mathbf{u}^{n+1}) &= \mathbf{u}^n \\ \underbrace{\mathbf{u}^{n+1} - \Delta t \mathbf{f}(\mathbf{u}^{n+1}) - \mathbf{u}^n}_{\mathbf{R}(\mathbf{u}^{n+1})} &= \mathbf{0} \end{aligned}$$

We can solve this system using the Newton-Raphson method, presented in Section 1.7.4, usually with the state at the previous time step, $\mathbf{u}^n$, as the starting guess for $\mathbf{u}^{n+1}$. The quantity $\mathbf{R}(\cdot)$ above is called the unsteady residual, and Newton-Raphson requires its linearization with respect to the state at every iteration. The number of iteration depends on the required accuracy and the accuracy of the starting guess (for large time steps, the guess $\mathbf{u}^n$ may not be that good). However, once close to the solution, Newton-Raphson converges rapidly, so that usually only a few iterations are required.

Moving beyond backward Euler, high-order accurate implicit methods also exist. From Table 4.3.1, these include BDF2 and BDF3, second and third-order backwards difference methods. BDF methods exist for even higher orders, but their stability advantages degrade after BDF3. We also have the Adams-Moulton (AM) family, with AM2, the trapezoidal method, a popular choice. Higher-order AM schemes loose their stability advantages. Finally, in addition to multistep, implicit multi-stage schemes exist, including many flavors of implicit Runge-Kutta schemes. These require multiple nonlinear solves to advance one time step, but they generally retain excellent stability properties at high-order accuracy.

With the multiple matrix solves and residual/Jacobian evaluations *for every time step*, the cost of implicit methods relative to explicit ones certainly goes up for nonlinear problems. This is in addition to the up-front cost of implementing implicit methods with the required residual Jacobian. Yet, in CFD, there exist systems which exhibit such disparate time scales that implicit methods are still desirable over explicit ones.

4.4 Consistency and Order of Accuracy

So far we have presented a variety of numerical schemes for initial value problems. How do we know these methods will work? And if they do, how accurate will they be? Key to the first question is the idea of **consistency**, which refers to the numerical method faithfully representing the partial differential equation as $\Delta x, \Delta t \to 0$. The second question is in fact related to the first, as accuracy depends on how quickly errors in the numerical methods drop with Δx and Δt. We address both of these questions in this section.

4.4.1 The Modified Equation

One technique for determining consistency and accuracy of a numerical method, applicable to general fully-discrete methods, involves analysis of what's called the modified equation. In this section we present this method for a scalar PDE, in the context of a finite difference discretization with spatial index j and temporal index n.

Denote by $D(u)$ the mathematical model, i.e. the governing unsteady PDE, that we are interested in solving. Denote by $N(u_j^n)$ the chosen numerical scheme, which operates on state values stored at spatial nodes j and temporal nodes n. The numerical scheme is consistent with D if for *all* smooth functions u,

$$\tau_j^n \equiv N(u_j^n) - D(u)_j^n \to 0 \quad \text{as} \quad \Delta x, \Delta t \to 0,$$

where j, n are spatial/temporal indices. When u is the exact solution of $D(u) = 0$, the quantity τ_j^n is called the truncation error.

Example 4.1 (Truncation error of FOU for advection)**.** The advection equation is

$$D(u) = u_t + a u_x = 0. \tag{4.4.1}$$

The first-order upwind (FOU) scheme applied to this equation gives

$$N(u_j^n) = \frac{u_j^{n+1} - u_j^n}{\Delta t} + \frac{a(u_j^n - u_{j-1}^n)}{\Delta x}. \tag{4.4.2}$$

Using Taylor-series expansions,

$$\begin{aligned} u_j^{n+1} &= u_j^n + (u_t)_j \Delta t + \frac{1}{2}(u_{tt})_j^n \Delta t^2 + \mathcal{O}(\Delta t^3), \\ u_{j-1}^n &= u_j^n - (u_x)_j \Delta x + \frac{1}{2}(u_{xx})_j^n \Delta x^2 + \mathcal{O}(\Delta x^3). \end{aligned}$$

Equation 4.4.2 becomes

$$N(u_j^n) = \underbrace{(u_t + a u_x)_j^n}_{D(u)_j^n} + \underbrace{\frac{\Delta t}{2}(u_{tt})_j^n - \frac{a \Delta x}{2}(u_{xx})_j^n + \mathcal{O}(\Delta t^2, \Delta x^2)}_{\text{truncation error: } \mathcal{O}(\Delta x, \Delta t)}.$$

This shows that FOU is consistent and first-order accurate, as the powers on both Δx and Δt in the truncation error are 1.

The above analysis can also be used to obtain the **modified equation**, sometimes called the *equivalent* equation. This is the PDE approximated by the numerical scheme to a higher accuracy than the original PDE. Note, for a consistent discretization, the extra terms in the modified PDE go to zero as $\Delta x, \Delta t \to 0$. Determining the modified equation is like reverse engineering: we seek the equation that we are really solving for a given finite $\Delta x, \Delta t$ when we try to solve $D(u) = 0$ numerically.

Let $\bar{u}$ denote the solution to the modified equation. To obtain the modified equation, we write $N(\bar{u}_j^n)$ in terms of spatial derivatives only, using Taylor-series expansions. Removing the j and n indices then leaves the modified equation.

Example 4.2 (Modified Equation for FOU applied to advection)**.** Recall from Example 4.1 that, after Taylor-series expansions, FOU for advection reads (in terms of $\bar{u}$)

$$N(\bar{u}_j^n) = (\bar{u}_t + a\bar{u}_x)_j^n + \frac{\Delta t}{2}(\bar{u}_{tt})_j^n - \frac{a\Delta x}{2}(\bar{u}_{xx})_j^n + \mathcal{O}(\Delta t^2, \Delta x^2). \tag{4.4.3}$$

To obtain the modified equation, we need to express $(\bar{u}_{tt})_j^n$ in terms of spatial derivatives. For this, we use the expression for $N(\bar{u}_j^n)$ itself,

$$\begin{aligned}
(\bar{u}_t)_j^n &= -a(\bar{u}_x)_j^n + \mathcal{O}(\Delta t, \Delta x) \\
(\bar{u}_{tt})_j^n &= -a(\bar{u}_{xt})_j^n + \mathcal{O}(\Delta t, \Delta x) \\
&= -a\left[-a(\bar{u}_{xx})_j^n\right] + \mathcal{O}(\Delta t, \Delta x) = a^2(\bar{u}_{xx})_j^n + \mathcal{O}(\Delta t, \Delta x),
\end{aligned}$$

where we have used $\bar{u}_{xt} = \bar{u}_{tx} = (\bar{u}_t)_x$. Substituting into Equation 4.4.3 gives

$$N(\bar{u}_j^n) = (\bar{u}_t + a\bar{u}_x)_j^n + \frac{\Delta t}{2}a^2(\bar{u}_{xx})_j^n - \frac{a\Delta x}{2}(\bar{u}_{xx})_j^n + \mathcal{O}(\Delta t^2, \Delta x\Delta t, \Delta x^2). \tag{4.4.4}$$

Setting this expression to zero and neglecting the higher (second) order terms, we obtain the modified equation,

$$\bar{u}_t + a\bar{u}_x = -\frac{\Delta t}{2}a^2\bar{u}_{xx} + \frac{a\Delta x}{2}\bar{u}_{xx} = \frac{a\Delta x}{2}\left(1 - \frac{a\Delta t}{\Delta x}\right)\bar{u}_{xx}. \tag{4.4.5}$$

We see that when applied to advection, FOU actually solves a modified PDE that contains a diffusion term. This term vanishes as $\Delta x, \Delta t \to 0$, but for finite $\Delta x, \Delta t$ the effect of the diffusion will be noticeable. Moreover, Equation 4.4.5 provides a necessary condition for stability: the sign on the diffusion term must remain positive, else we obtain non-physical "anti-diffusion". This means that

$$1 - \frac{a\Delta t}{\Delta x} > 0, \quad \Rightarrow \quad \frac{a\Delta t}{\Delta x} < 1.$$

This is called the Courant-Friedrichs-Lewy (CFL) condition, and we will discuss it further in the stability section. Note that when the CFL number, $\sigma \equiv a\Delta t/\Delta x$, is unity, the diffusion term disappears and the modified equation becomes the original advection PDE, at least up to the second-order terms that we dropped.

4.4.2 Consistency of Multi-Step Schemes

For multi-step schemes, the **local error**, $\boldsymbol{\epsilon}_{\text{local}}^{n+1}$, is the quantity that remains in Equation 4.3.2 after the exact solution, $\mathbf{u}_{\text{exact}}(t)$ is plugged in for both $\mathbf{u}^{n+k}$ and $\mathbf{f} = \dot{\mathbf{u}}$. Defining "what remains" by left-hand-side minus the right-hand-side, we obtain,

$$\boldsymbol{\epsilon}_{\text{local}}^{n+1} \equiv \sum_{k=1-K}^{1} \alpha_k \mathbf{u}_{\text{exact}}^{n+k} - \Delta t \sum_{k=1-K}^{1} \beta_k \mathbf{f}(\mathbf{u}_{\text{exact}}^{n+k}, t^{n+k}). \tag{4.4.6}$$

This error directly pollutes the next state, $\mathbf{u}^{n+1}$. We say that a multi-step method is order p accurate if this local error converges as

$$\boldsymbol{\epsilon}_{\text{local}}^{n+1} = \mathcal{O}\left(\Delta t^{p+1}\right). \tag{4.4.7}$$

The reason for $p+1$ instead of p is that the local error only refers to the error made in one time step, whereas the scheme order of accuracy pertains to the *global* error, which is an accumulation of local errors.

We can calculate $\boldsymbol{\epsilon}_{\text{local}}^{n+1}$ through Taylor-series expansions of the state and derivative about time node n,

$$\begin{aligned} \mathbf{u}^{n+k} &= \mathbf{u}^n + k\Delta t(\mathbf{u}_t)^n + \tfrac{1}{2}(k\Delta t)^2(\mathbf{u}_{tt})^n + \mathcal{O}(\Delta t^3), \\ \mathbf{f}(\mathbf{u}^{n+k}) = \mathbf{u}_t^{n+k} &= \mathbf{u}_t^n + k\Delta t(\mathbf{u}_{tt})^n + \tfrac{1}{2}(k\Delta t)^2(\mathbf{u}_{ttt})^n + \mathcal{O}(\Delta t^3). \end{aligned} \tag{4.4.8}$$

Substituting these expressions into the multi-step update equation and collecting terms with the same power of Δt gives the local error.

Example 4.3 (Local Error of Forward Euler)**.** From Table 4.3.1, the forward Euler update involves $\mathbf{u}^{n+1}$ and $\mathbf{u}^n$. Expanding $\mathbf{u}^{n+1}$ using the Taylor series from Equation 4.4.8 gives

$$\underbrace{\mathbf{u}^n + \Delta t \mathbf{u}_t^n + \frac{1}{2}\Delta t^2 \mathbf{u}_{tt}^n + \mathcal{O}(\Delta t^3)}_{\mathbf{u}^{n+1}} = \mathbf{u}^n + \Delta t \underbrace{\mathbf{u}_t^n}_{\mathbf{f}^n}.$$

The local error is the left-hand side minus the right-hand side

$$\boldsymbol{\epsilon}_{\text{local}}^{n+1} = \frac{1}{2}\Delta t^2 \mathbf{u}_{tt}^n + \mathcal{O}(\Delta t^3).$$

Since the leading term is $\mathcal{O}(\Delta t^2)$, forward Euler is first-order accurate, $p = 1$.

For a time integration method to be useful, it must be at least first-order accurate, i.e. $\boldsymbol{\epsilon}_{\text{local}}^{n+1} = \mathcal{O}(\Delta t^2)$, and we call such a method *consistent*. Substituting the Taylor-series expansions from Equation 4.4.8 into Equation 4.4.6, we find

$$\boldsymbol{\epsilon}_{\text{local}}^{n+1} = \left[\sum_{k=1-K}^{1} \alpha_k\right] \mathbf{u}_{\text{exact}}^n + \left[\sum_{k=1-K}^{1} (k\alpha_k - \beta_k)\right] \mathbf{u}_{\text{exact},t}^n \Delta t + \mathcal{O}(\Delta t^2)$$

For consistency, we need the $\mathcal{O}(1)$ and $\mathcal{O}(\Delta t)$ terms to cancel,

$$\sum_{k=1-K}^{1} \alpha_k = 0, \tag{4.4.9}$$

$$\sum_{k=1-K}^{1} k\alpha_k = \sum_{k=1-K}^{1} \beta_k. \tag{4.4.10}$$

Using Equation 4.4.9 and Equation 4.4.10, we can verify that all of the schemes presented in Table 4.3.1 are consistent. The general form of a consistent two-step method is

$$\xi \mathbf{u}^{n-1} - (1+2\xi)\mathbf{u}^n + (1+\xi)\mathbf{u}^{n+1} = \Delta t \left(\phi \mathbf{f}^{n-1} + (1-\theta-\phi)\mathbf{f}^n + \theta \mathbf{f}^{n+1}\right). \tag{4.4.11}$$

We verify this by first checking that Equation 4.4.9 holds,

$$\sum_{k=1-K}^{1} \alpha_k = \xi - (1+2\xi) + (1+\xi) = 0.$$

Equation 4.4.10 also holds since,

$$\sum_{k=1-K}^{1} k\alpha_k = (-1)\xi + 0\,(-(1+2\xi)) + 1(1+\xi) = 1 = \phi + (1-\theta-\phi) + \theta = \sum_{k=1-K}^{1} \beta_k.$$

In addition, by looking at the local error, one can verify that the scheme in Equation 4.4.11 will be second-order accurate if $\xi = \theta - \phi - 1/2$. It will be third-order accurate if also $\xi = -2\phi - 1/6$.

Example 4.4 (Order of Accuracy of Modified Euler)**.** We can also compute the local error for multi-stage schemes, such as the modified Euler method in Equation 4.3.4. Consider a scalar problem in which $f(u,t) = f(u)$ is time-invariant. We compute the local error by writing out the expression for u^{n+1}, using the fact that $f(u^n) = u_t^n$,

$$\begin{aligned} u^{n+1} &= u^n + \Delta t f\left(u^n + \frac{1}{2}\Delta t f(u^n)\right) \\ &= u^n + \Delta t f\left(u^n + \frac{1}{2}\Delta t u_t^n\right) \\ &= u^n + \Delta t (u_t^n + \frac{1}{2}\Delta t u_{tt}^n) \\ &= u^n + \Delta t u_t^n + \frac{1}{2}\Delta t^2 u_{tt}^n. \end{aligned}$$

This last expression is just a truncated Taylor-series expansion of u^{n+1} about t^n. The leading missing term is $\mathcal{O}(\Delta t^3)$, which is then the local error. This means $p+1 = 3$, so that $p = 2$, and the method is second-order accurate.

4.5 Stability

4.5.1 The CFL condition

The Courant-Friedrichs-Lewy (CFL) number, $\sigma = a\Delta t/\Delta x$, has a rich history dating back to 1928 in the **CFL condition**, which states that *the numerical domain of dependence must contain the physical domain of dependence*. Figure 4.5.1 illustrates this condition for advection and a first-order upwind discretization.

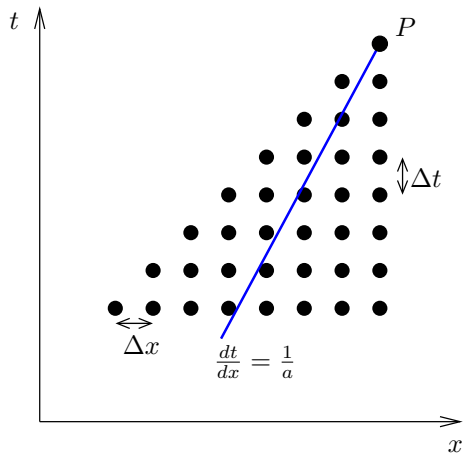


Figure 4.5.1: Illustration of the CFL condition for a first-order upwind discretization of advection in one spatial dimension. The physical domain of dependence of the solution at point P must lie in the numerical domain of dependence (the black dots).

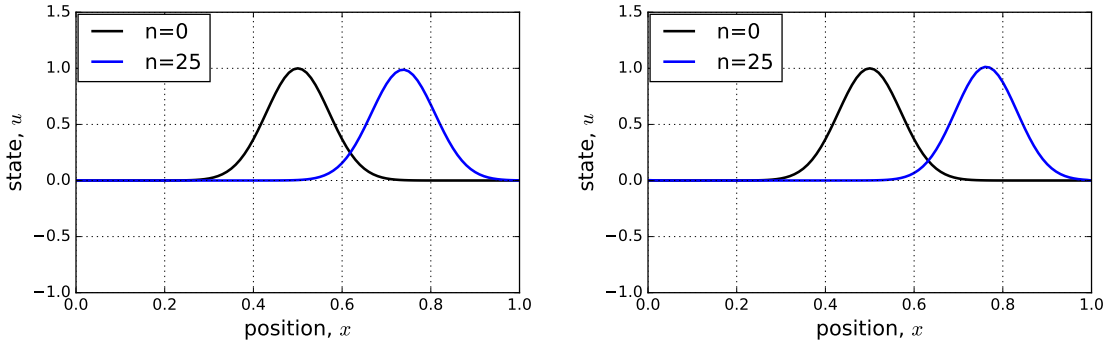
In order for the CFL condition to be satisfied, the line defining the physical domain of dependence must lie within the grid points defining the numerical domain of dependence. This means that

$$\frac{\Delta t}{\Delta x} \leq \frac{1}{a} \quad \Rightarrow \quad \sigma = \frac{a\Delta t}{\Delta x} \leq 1. \tag{4.5.1}$$

The **CFL theorem** states that the CFL condition is *necessary* for the convergence of a numerical approximation of a PDE, linear or nonlinear, and Example 4.5 illustrates this. However, the CFL condition is not *sufficient*: schemes exist that satisfy the CFL condition but do not converge, as illustrated in Example 4.6

Example 4.5 (FOU for advection)**.** Figure 4.5.1 and Equation 4.5.1 indicate that for the first-order upwind method, we need $\sigma \leq 1$ for stability. To test this, we implement FOU for the case $a = 1$, $L = 1$, periodic boundary conditions, and initial condition $u^0(x) = \exp[-(10x/L - 5)^2]$. Figure 4.5.2 shows the initial condition and solution after 25 iterations at two CFL numbers: $\sigma = 0.95$ (stable) and $\sigma = 1.05$ (unstable). We see that after the short time of 25 time steps, the solutions are not very different at the two CFL numbers, though a

close inspection reveals a slight growth of the solution for $\sigma = 1.05$. While $\sigma = 1.05$ should be unstable, the short duration of the simulation does not allow the exponentially-unstable modes to grow to a noticeable size yet.



(a) $\sigma = 0.95$ (b) $\sigma = 1.05$

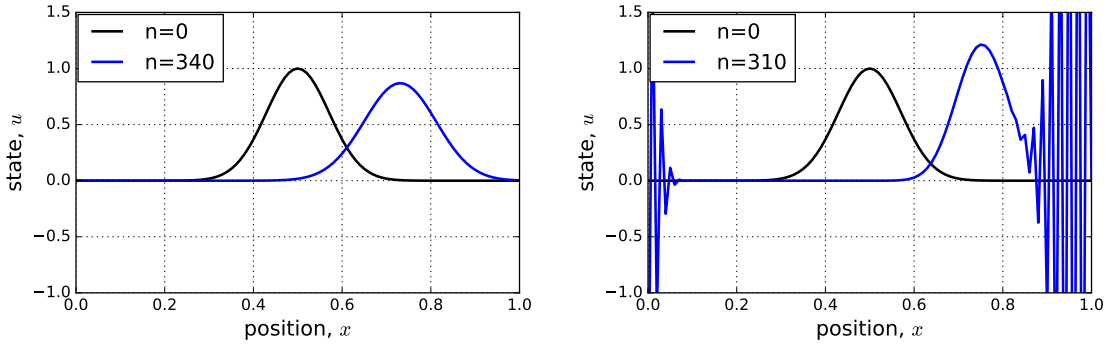
Figure 4.5.2: Short-time solutions of scalar advection in a periodic domain, computed using FOU two CFL numbers, σ, one stable and the other not.

To verify that indeed $\sigma = 0.95$ is stable and $\sigma = 1.05$ is not, we run the simulation for a longer time, over 300 time steps, the exact number chosen to make the final times approximately the same for both solutions. Figure 4.5.3 shows the results. While the $\sigma = 0.95$ case shows a slight attenuation of the solution (stability), the $\sigma = 1.05$ case shows clear growth in the solution. Moreover, in the $\sigma = 1.05$ case we see high-frequency oscillations that will grow exponentially to dominate and destroy the solution. Running other σ values close to 1 reveals numerically that $\sigma = 1$ is the stability limit, as predicted by the CFL theorem.

Example 4.6 (FTCS for advection)**.** Consider the forward in time, centered in space scheme for advection, introduced in Section 4.2. The update formula, Equation 4.2.4, reads

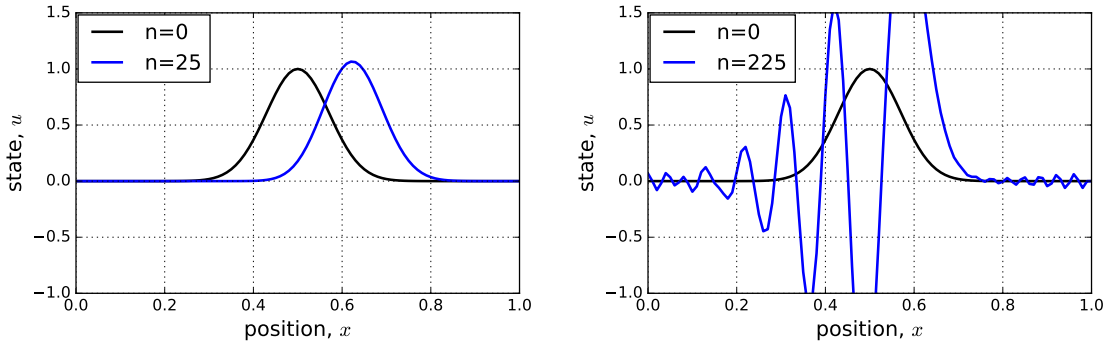
$$u_j^{n+1} = u_j^n - \frac{\sigma}{2}(u_{j+1}^n - u_{j-1}^n). \tag{4.5.2}$$

As the numerical domain of dependence includes both neighbors, including the upwind one, the CFL condition is satisfied as long as $\sigma \leq 1$. However, the method turns out to be unstable for all values of σ! Figure 4.5.4 shows the behavior of the solution for $\sigma = 0.5$, at both short and long time horizons. We see growth of the solution right from the start, with unstable higher-frequency oscillations developing at later times.



(a) $\sigma = 0.95$ (b) $\sigma = 1.05$

Figure 4.5.3: Solutions of scalar advection in a periodic domain, computed using FOU and two CFL numbers, over an extended time horizon.



(a) Short time horizon (b) Long time horizon

Figure 4.5.4: Solutions of scalar advection in a periodic domain, computed using FTCS and $\sigma = 0.5$, at two iteration numbers.

4.5.2 Zero Stability of Time Integration

Suppose that we are interested in the solution of an ODE over a fixed time range of $t \in [0, T]$. Given N time steps of size $\Delta t = T/N$, we say that a multi-step scheme is *zero-stable* if the solution remains bounded as $\Delta t \to 0$, i.e. as $N \to \infty$. Note that this concept does not address stability for a given Δt, which we discuss in the next section.

For multi-step methods, as $\Delta t \to 0$ the right-hand-side of Equation 4.3.2 drops out, leaving a *recurrence relation*, which we write assuming $\mathbf{u} = u$ is a scalar for simplicity,

$$\sum_{k=1-K}^{1} \alpha_k u^{n+k} = 0. \tag{4.5.3}$$

For zero-stability, all solutions of this recurrence relation must remain bounded as $n \to \infty$.

Recurrence relations admit solutions of the form $u^n = u^0 g^n$, where g^n means g raised to the n^{th} power, and where g is a (complex) root of

$$\sum_{k=1-K}^{1} \alpha_k g^k = 0. \tag{4.5.4}$$

For zero stability, we need all roots of this equation to satisfy $|g| < 1$. $|g| = 1$ is allowed as long as the root is not repeated (otherwise $u^n = n u^0 g^n$, which increases without bound, would be a solution).

For the forward Euler scheme, the recurrence relation from Equation 4.5.3 reads

$$\begin{aligned} u^{n+1} - u^n &= 0 \\ u^0 g^{n+1} - u^0 g^n &= 0 \\ u^0 g^n (g - 1) &= 0 \end{aligned}$$

We see that the solutions to this equation are $g = 0$ and a single root $g = 1$. Hence, the forward-Euler scheme is zero-stable. This recurrence relation remains the same for the two and four-stage Runge-Kutta schemes, which only leave $u^{n+1} - u^n = 0$ in the limit $\Delta t \to 0$. Hence, the Runge-Kutta schemes are also zero stable.

4.5.3 Eigenvalue Stability

While zero stability tells us what happens in the limit $\Delta t \to 0$, we often care about the practical case of a fixed nonzero time step. For example, we may want to know how large we can make Δt before the scheme goes unstable. For this stability analysis we will use a very simple but representative scalar system,

$$\frac{du}{dt} = \lambda u, \tag{4.5.5}$$

where λ is a complex scalar. A particular time integration method may give stable solutions for some λ but not others. The term **eigenvalue stability** refers to the fact that, for a

system $\mathbf{u}_t = \mathbf{f}(\mathbf{u})$, if a scheme gives stable solutions for all λ that are the eigenvalues of the matrix $\partial\mathbf{f}/\partial\mathbf{u}$, then the scheme will be stable for the complete system ... nonlinear effects excluded. The analysis of Equation 4.5.5 therefore has broad implications.

We can write the effect of one iteration of a time integration scheme applied to Equation 4.5.5 via an *amplification factor*, g,

$$u^{n+1} = gu^n, \tag{4.5.6}$$

where g is a possibly-complex constant. The justification for this expression is that Equation 4.5.5 is homogeneous and time-independent, so that nothing distinguishes one iteration from another. As a result, $u^n = g^n u^0$, where again g^n means g raised to the n^{th} power. For stability, we require $|g| \leq 1$, and g depends on the physics of the problem, i.e. λ, and on the numerics via the choice of integration scheme and time step, Δt. Let's see how this works through an example.

Example 4.7 (Eigenvalue Stability of Forward Euler)**.** Recall the forward Euler method from Table 4.3.1, which for $f(u) = \lambda u$ is

$$u^{n+1} = u^n + \Delta t\, \lambda u^n.$$

To calculate the amplification factor, we substitute $u^n = g^n u^0$ into this equation,

$$\begin{aligned} g^{n+1}u^0 &= g^n u^0 + \Delta t\, \lambda g^n u^0, \\ g^n u^0(g) &= g^n u^0(1 + \lambda\Delta t), \\ \Rightarrow\ g &= 1 + \lambda\Delta t. \end{aligned}$$

For stability, we'd like the amplification factor to have magnitude less than 1. From the above equation, we see that for real λ, this means $-1 < \lambda\Delta t < 1$. But in general, λ can be complex, and so we need to allow for complex g. We determine the general *stability boundary* by setting $g = e^{i\theta}$, since this is the set of all complex numbers such that $|g| = 1$. Hence we have,

$$\begin{aligned} e^{i\theta} &= 1 + \lambda\Delta t, \\ \Rightarrow\ \lambda\Delta t &= e^{i\theta} - 1, \end{aligned}$$

which is a circle of radius 1 centered at -1 in the complex number plane. This analysis gives us the stability boundary, and all that's left is figuring out which side of the boundary is stable. To do this, we pick a point, i.e. a value of $\lambda\Delta t$, and calculate $|g|$: for example, for forward Euler, choosing the point $\lambda\Delta t = -1$ gives $|g| = 0$, which is stable. This means that the region inside the unit circle centered at $\lambda\Delta t = -1$ is stable, and we obtain the stability diagram shown in Figure 4.5.5.

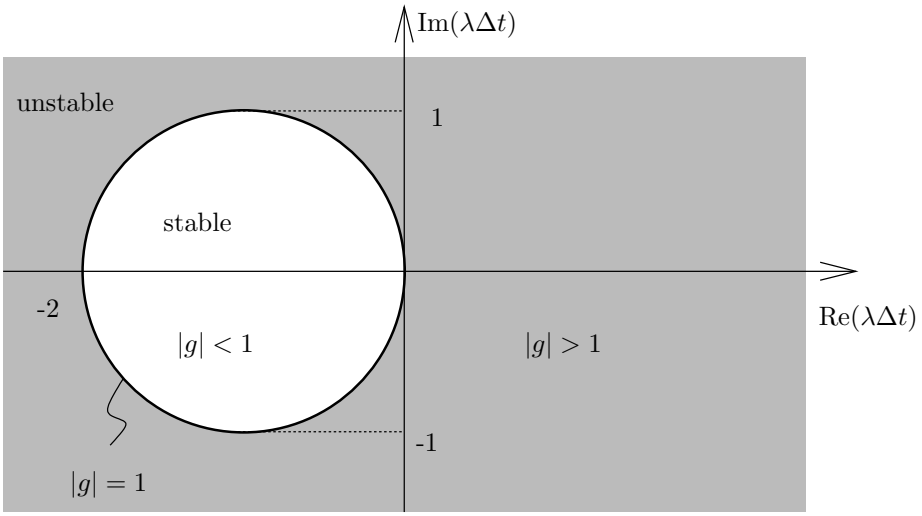


Figure 4.5.5: Eigenvalue stability region for the forward Euler method. The region inside the circle is stable as it yields amplification factors with $|g| < 1$.

In Example 4.7 we introduced the concept of a **stability region**, which is the set of values $\lambda\Delta t$ in the complex number plane for which the time integration scheme remains stable. The **stability boundary** is the edge of this region. While we generally cannot control λ directly (it represents the physics of the problem), we do have control of Δt, and reducing Δt can often stabilize a system by bringing $\lambda\Delta t$ into the stability region. Finally, we note that zero stability manifests itself in the diagram as stability in the left-half-plane as we zoom into the origin by making $\Delta t \to 0$.

Example 4.8 (Eigenvalue Stability of Backward Euler)**.** The backward Euler method, also known as first-order backward differencing (BDF1), is a one-step method similar to forward Euler with the exception that the forcing term is evaluated at $n+1$ instead of at n. Applying this method to our prototypical eigenvalue system in Equation 4.5.5 we have

$$u^{n+1} = u^n + \Delta t\, \lambda u^{n+1}.$$

The stability analysis now proceeds in a way similar to the forward Euler case. Introducing the amplification factor via Equation 4.5.6,

$$\begin{aligned} g^{n+1}u^0 &= g^n u^0 + \Delta t\, \lambda g^{n+1} u^0, \\ g^n u^0(g) &= g^n u^0(1 + g\lambda\Delta t), \\ g &= 1 + g\lambda\Delta t, \\ \Rightarrow\ g &= \frac{1}{1 - \lambda\Delta t}. \end{aligned}$$

To determine the stability boundary, we set $|g| = 1$, i.e. $g = e^{i\theta}$,

$$\begin{aligned} e^{i\theta} &= \frac{1}{1 - \lambda\Delta t}, \\ e^{i\theta} - \lambda\Delta t e^{i\theta} &= 1, \\ \Rightarrow \lambda\Delta t &= 1 - e^{-i\theta}, \end{aligned}$$

which is a circle of radius 1 centered at 1 in the complex number plane. Choosing the circle origin, $\lambda\Delta t = 2$, shows that the inside of the circle is unstable while the outside is stable. Therefore, we obtain the stability region shown in Figure 4.5.6. Note the much larger extent

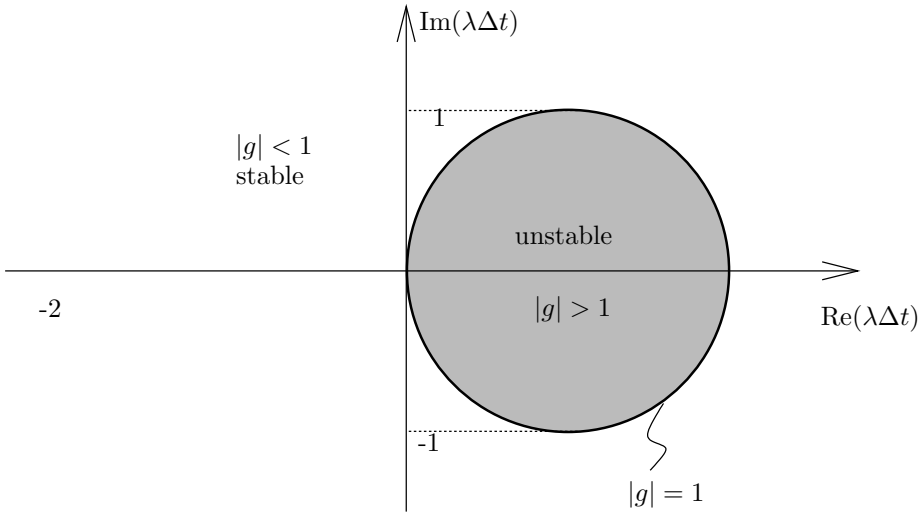


Figure 4.5.6: Eigenvalue stability region for the backward Euler method. The region inside the circle is unstable as it yields amplification factors with $|g| > 1$.

of the backward Euler stability region compared to forward Euler. In particular, the entire left half plane in Figure 4.5.6 is stable. When this happens, we say that the method is **A-stable**. This is a useful property because it means that the numerical scheme is stable for all physically-stable modes (negative real part eigenvalues) of the system.

Example 4.9 (Eigenvalue Stability of Runge-Kutta Methods)**.** For multi-stage methods we can use the same eigenvalue stability analysis procedure that we used for multi-step methods. That is, we consider a homogeneous scalar eigenvalue problem, $f(u) = \lambda u$, and an exponentially-growing/decaying state, $u^n = g^n u^0$. First, let's look at the two-stage (RK2) predictor-corrector method. Combining the two stages of Equation 4.3.3 into one equation,

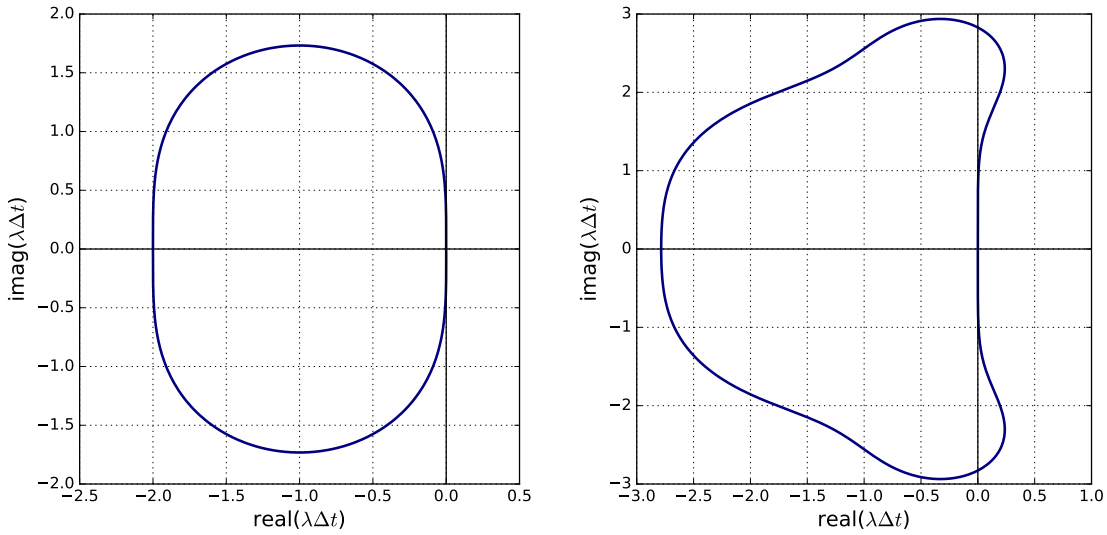
we have

$$\begin{aligned} u^{n+1} &= u^n + \Delta t \frac{1}{2}\left[f(u^n) + f(u^n + \Delta t f(u^n))\right] \\ &= u^n + \Delta t \frac{1}{2}\left[\lambda u^n + \lambda\left(u^n + \Delta t \lambda u^n\right)\right] \\ &= u^n \left[1 + \lambda\Delta t + \frac{1}{2}(\lambda\Delta t)^2\right] \\ \Rightarrow\ g &= 1 + \lambda\Delta t + \frac{1}{2}(\lambda\Delta t)^2 \end{aligned} \tag{4.5.7}$$

Having solved for the amplification factor, we can now determine the region of stability. The stability boundary is given by those $\lambda\Delta t$ values that give $|g| = 1$. All such g can be expressed as $g = e^{i\theta}$ for arbitrary θ. Substituting $e^{i\theta}$ for g above, we obtain a quadratic equation,

$$\begin{aligned} \frac{1}{2}(\lambda\Delta t)^2 + \lambda\Delta t + 1 - e^{i\theta} &= 0 \\ \Rightarrow\ \lambda\Delta t &= -1 \pm \sqrt{2e^{i\theta} - 1} \end{aligned}$$

At this point, we could plot a curve of the stability boundary by considering $\theta \in [0, 2\pi]$. Alternatively, we could use Equation 4.5.7 to make a contour plot of the $|g| = 1$ contour in the complex number plane of $\lambda\Delta t$. We take the latter approach and generate Figure 4.7(a) using the code in Listing 4.5.1.



(a) RK2 (b) RK4

Figure 4.5.7: Stability boundaries for two and four-stage Runge-Kutta methods. The regions inside the boundaries are stable.

Listing 4.5.1: Code for computing the stability boundary of Runge-Kutta methods; predictor-corrector is RK2.

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

def plotstab(Meth):
    meth = Meth.lower()
    f = plt.figure( figsize =(6,6))
    sx,sy = np.linspace(-3,1,201), np.linspace(-3,3,201)
    Y,X = np.meshgrid(sy,sx); Z = X + 1j*Y;
    if (meth == 'rk4'):
        G = 1 + Z + 0.5*Z**2 + (1./6.)*Z**3 + (1./24.)*Z**4;
        A = [-3, 1, -3, 3]
    elif (meth == 'rk2'):
        G = 1 + Z + 0.5*Z**2
        A = [-2.5, 0.5, -2, 2]
    else:
        print('method %s not supported'%(meth))
    plt.contour(X,Y,abs(G), [1], linewidths=2);
    plt.xlabel(r'real($\lambda \Delta t$)', fontsize=16)
    plt.ylabel(r'imag($\lambda \Delta t$)', fontsize=16)
    plt.axhline(0, color='black'); plt.axvline(0, color='black')
    plt.tick_params(axis='both', labelsize=12)
    plt.grid(); plt.axis(A); f.tight_layout(); plt.show(block=False)
    plt.savefig('stab_%s.pdf'%(meth))
    plt.close(f)

def main():
    methods = ['rk4', 'rk2']
    for meth in methods: plotstab(meth)

if __name__ == "__main__":
    main()
```

Note the vertically-oval nature of the stability boundary. We particularly care about the behavior near the origin, where eigenvalues that are close to the imaginary axis now have a chance of being inside the stability boundary (e.g. for a sufficiently-small time step). Compare, for example, to the forward Euler stability boundary in Figure 4.5.5, which, as a circle, turns away from the origin much sooner.

In Figure 4.5.7 and Listing 4.5.1 we also show the stability boundary and generating code for fourth-order Runge-Kutta (RK4). The formula in the code is obtained by first combining all four stages of Equation 4.3.5 into one equation,

$$\mathbf{u}^{n+1} = \mathbf{u}^n \left(1 + \lambda\Delta t + \frac{1}{2}(\lambda\Delta t)^2 + \frac{1}{6}(\lambda\Delta t)^3 + \frac{1}{24}(\lambda\Delta t)^4\right).$$

Substituting $u^n = g^n u^0$ gives us the amplification factor

$$g = 1 + \lambda\Delta t + \frac{1}{2}(\lambda\Delta t)^2 + \frac{1}{6}(\lambda\Delta t)^3 + \frac{1}{24}(\lambda\Delta t)^4.$$

To generate the stability boundary, we plot the $|g| = 1$ contour, with the result in Figure 4.7(b). Testing a point such as $\lambda\Delta t = -1$ shows that the interior of the contour is stable. Note that the stability boundary encloses portions of the imaginary axis – this is good for stability of lightly-damped oscillatory modes. Also, the stability boundary extends further along the negative real axis compared to forward Euler and second-order Runge-Kutta, which means that the time steps could be larger for a given eigenvalue. Finally, the name "four-stage" is given to this method because four function evaluations, $\mathbf{f}(\mathbf{u}, t)$, are required to take one time step.

Example 4.10 (Eigenvalue Stability of FOU for Advection)**.** Let's now look at eigenvalue stability in the context of an ODE system arising from a first-order upwind finite-difference discretization of advection. We consider both periodic and Dirichlet boundary conditions. In the case of Dirichlet boundary conditions, we'll only have to set the state on the inflow (left for $a > 0$) boundary, as the outflow boundary state is never used when the derivative is approximated by an upwind difference. The semi-discrete equation for node j reads

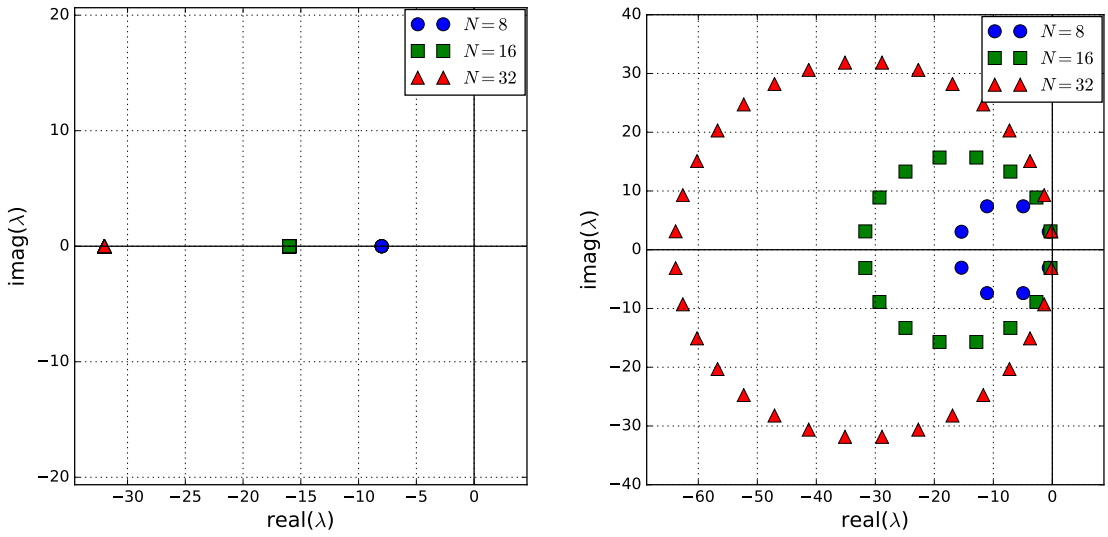
$$\frac{\partial u_j}{\partial t} = -a\frac{u_j - u_{j-1}}{\Delta x}.$$

The boundary condition comes into our equation for the first unknown, $j = 1$, since then $u_{j-1} = u_0$, which is the prescribed Dirichlet boundary condition. So in matrix form our equation with Dirichlet boundary conditions reads

$$\frac{d}{dt}\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{N-2} \\ u_{N-1} \\ u_N \end{bmatrix} = \underbrace{-\frac{a}{\Delta x}\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & \dots & \boxed{0} \\ -1 & 1 & 0 & 0 & 0 & \dots & 0 \\ 0 & -1 & 1 & 0 & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & -1 & 1 & 0 & 0 \\ 0 & \dots & 0 & 0 & -1 & 1 & 0 \\ 0 & \dots & 0 & 0 & 0 & -1 & 1 \end{bmatrix}}_{\mathbf{A}}\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{N-2} \\ u_{N-1} \\ u_N \end{bmatrix} + \begin{bmatrix} \boxed{au_0/\Delta x} \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

The two boxed terms above change when periodic boundary conditions are used. Specifically, in $\mathbf{A}$, the 0 becomes -1, and in the source term, $au_0/\Delta x$ becomes zero.

To determine stability, we are interested in the eigenvalues of $\mathbf{A}$. Figure 4.5.8 shows these eigenvalues for various N, with $a = 1$, $L = 1$, $\Delta x = L/N$, and using both Dirichlet and periodic boundary conditions. Listing 4.5.2 presents the code that generates these plots. Suppose now that we use forward Euler time integration, which has the eigenvalue stability region shown in Figure 4.5.5. In the case of periodic boundary conditions, to get $\lambda\Delta t$ into the circular stability region, we would need $\Delta t \leq 1/N$ – we usually have to be careful with the eigenvalues near the origin, but in this case the eigenvalues are distributed in the form of a circle. The restriction of $1/N$ is specific to our choice of a and L, and a dimensional analysis



(a) Dirichlet BCs

(b) Periodic BCs

Figure 4.5.8: Eigenvalues of the system matrix for a first-order upwind (FOU) discretization of the one-dimensional advection equation, using both Dirichlet and periodic boundary conditions.

shows that the general result is $\Delta t \leq \Delta x/a$, i.e. $\sigma = a\Delta t/\Delta x \leq 1$, which is consistent with the CFL condition in Section 4.5.1.

Listing 4.5.2: Code for computing eigenvalues of the system resulting from an FOU discretization of the advection equation.

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

def ploteigs(Nv, DoPeriodic):
    a, L = 1., 1.
    f = plt.figure( figsize =(6,6))
    smb = ['o', 's', '^']
    for i,N in enumerate(Nv):
        dx = L/N
        e = np.ones(N)*(-a/dx)
        A = np.diag(e,0) + np.diag(e[0:(N-1)], -1)
        if (DoPeriodic): A[0,N-1] = -e[N-1]
        l,v = np.linalg.eig(A)
        plt.plot(l.real,l.imag,smb[i],linewidth=2,markersize=10,label=r'$N=%d$'%(N));
    plt.xlabel(r'real($\lambda$)', fontsize=16)
    plt.ylabel(r'imag($\lambda$)', fontsize=16)
    plt.axhline(0, color='black'); plt.axvline(0, color='black')
    plt.tick_params(axis='both', labelsize=12)
```

```
21      plt.legend(fontsize=14, borderaxespad=0.1, loc=1)
22      plt.grid(); plt.axis('equal'); f.tight_layout(); plt.show(block=False)
23      plt.savefig('../figs/eigstabFOU_%d.pdf'%(DoPeriodic))
24      plt.close(f)
25
26  def main():
27      ploteigs([8,16,32], False)
28      ploteigs([8,16,32], True)
29
30  if __name__ == "__main__":
31      main()
```

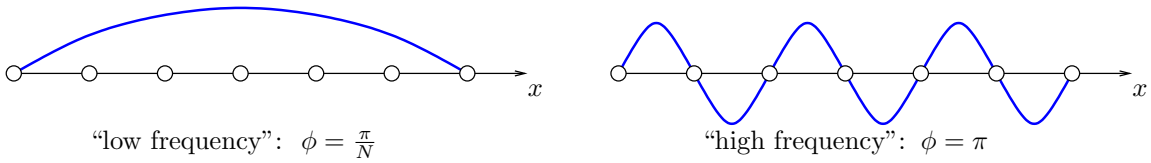
4.5.4 Von-Neumann Analysis

A powerful stability analysis technique for both fully-discrete and semi-discrete schemes is von-Neumann analysis. It applies to linear initial-value problems on uniform grids with periodic boundary conditions. These restrictions may seem limiting, but the implications of von-Neumann analysis are quite broad, since a scheme cannot be used for more complicated cases if it fails for this situation.

The key observation in von-Neumann analysis is that difference equations possess exact solutions of sinusoidal variation,

$$u_j^n = \Re\left[g^n e^{ij\phi}\right], \tag{4.5.8}$$

where g is a (complex) modal amplification factor, $i = \sqrt{-1}$, j is a spatial node index, and ϕ is the mode phase change per mesh interval, as illustrated in Figure 4.5.9. On a finite mesh with N intervals, ϕ takes on discrete values in the set $\left\{\frac{\pi}{N}, \frac{2\pi}{N}, \ldots, \pi\right\}$.



"low frequency": $\phi = \frac{\pi}{N}$ "high frequency": $\phi = \pi$

Figure 4.5.9: In von-Neumann analysis, $\phi \in [0, \pi]$ describes the frequency of a sinusoidal error mode. Specifically, ϕ is the phase change per mesh interval.

Equation 4.5.8 states that if the solution at one time level (n) is a sinusoidal spatial mode with phase change of ϕ per Δx, then the solution at the next time level $(n+1)$ will be that sinusoidal mode multiplied by g. Since g may be complex, both the phase and amplitude of the mode can be affected. For stability, we care about the amplitude, so our requirement is that $|g| \leq 1$ for all $\phi \in [0, \pi]$ – note that g will be a function of the frequency of the mode, i.e. ϕ. This is another way of saying that we do not want any of the modes representable on the grid to "blow up."

To perform von-Neumann analysis on a given finite-difference scheme, we substitute Equation 4.5.8 (without the real part restriction) into the scheme, solve for $g(\phi)$, and determine requirements on the scheme parameters such that $|g(\phi)| < 1$ for all ϕ in $[0, \pi]$.

Example 4.11 (FTCS for Advection)**.** In this example we look at the forward-in-time centered-in-space (FTCS) discretization of linear advection, $u_t + au_x = 0$, which reads

$$\begin{aligned} \frac{u_j^{n+1} - u_j^n}{\Delta t} + a\frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x} &= 0 \\ \Rightarrow \quad u_j^{n+1} &= u_j^n - \frac{a\Delta t}{2\Delta x}\left(u_{j+1}^n - u_{j-1}^n\right) \\ &= u_j^n - \frac{\sigma}{2}\left(u_{j+1}^n - u_{j-1}^n\right) \end{aligned}$$

To apply von-Neumann stability analysis, we substitute $u_j^n = g^n e^{ij\phi}$ into this equation and calculate the amplification factor g,

$$\begin{aligned} g^{n+1}e^{ij\phi} &= g^n e^{ij\phi} - \frac{\sigma}{2}\left(g^n e^{i(j+1)\phi} - g^n e^{i(j-1)\phi}\right) \\ g &= 1 - \frac{\sigma}{2}\left(e^{i\phi} - e^{-i\phi}\right) = 1 - \sigma i \sin\phi \end{aligned}$$

The magnitude of the amplification factor is

$$|g|^2 = 1 + \sigma^2 \sin^2\phi \geq 1.$$

So the FTCS scheme for advection is *unconditionally unstable* because the magnitude of g is always greater or equal to 1 (equality only in the special case when $\phi = \pi$).

Example 4.12 (FOU for advection)**.** Recall the first-order upwind method for advection, Equation 4.2.5, repeated here

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

To apply von-Neumann stability analysis, we substitute $u_j^n = g^n e^{ij\phi}$ and calculate the amplification factor g,

$$\begin{aligned} g^{n+1}e^{ij\phi} &= g^n e^{ij\phi} - \sigma\left(g^n e^{ij\phi} - g^n e^{i(j-1)\phi}\right) \\ g &= 1 - \sigma\left(1 - e^{-i\phi}\right) \\ &= 1 - \sigma + \sigma\cos\phi - i\sigma\sin\phi \end{aligned}$$

This is a complex number and its square magnitude is

$$\begin{aligned} |g|^2 &= (1 - \sigma + \sigma\cos\phi)^2 + (\sigma\sin\phi)^2 \\ &= (1-\sigma)^2 + 2(1-\sigma)\sigma\cos\phi + \sigma^2. \end{aligned}$$

For stability, we need $|g| \leq 1$ for all $\phi \in [0, \pi]$. Since $-1 \leq \cos(\phi) \leq 1$, we have

$$\begin{aligned} |g|^2 &= (1-\sigma)^2 + 2(1-\sigma)\sigma\cos\phi + \sigma^2 \\ &\leq \max\left[(1-\sigma)^2 + 2(1-\sigma)\sigma(-1) + \sigma^2,\ (1-\sigma)^2 + 2(1-\sigma)\sigma(1) + \sigma^2\right] \\ &= \max\left[(1-\sigma-\sigma)^2,\ (1-\sigma+\sigma)^2\right] \\ &= \max\left[(1-2\sigma)^2,\ 1\right] \end{aligned}$$

So we see that $|g| \leq 1$ as long as $(1-2\sigma)^2 \leq 1$. Assuming $a > 0$, σ is positive, and we need $\sigma \leq 1$. This is the same conclusion as the one obtained from the CFL theorem in Section 4.5.1 Note that the requirement on σ can be used to set the time step according to a and Δx.

4.5.5 Dissipation and Dispersion

We have already seen in von-Neumann analysis that numerical schemes can amplify or dampen solution modes differently, depending on the mode frequency. In this section we look with more detail into how modes are affected by numerical schemes. The model problem for this analysis is

$$u_t + au_x = \alpha u_{xx} - \beta u_{xxx}, \quad u(x,0) = u^0(x), \quad \text{periodic domain.} \tag{4.5.9}$$

The solution to this equation takes the general form

$$u(x,t) = \sum_{k=-\infty}^{\infty} U_k^0 \underbrace{e^{-4\pi^2\alpha k^2 t}}_{\text{dissipation}} \underbrace{e^{2\pi i(kx-\omega(k)t)}}_{\text{dispersion}}, \tag{4.5.10}$$

where k is the wave number (spatial frequency) and ω is the temporal frequency. Note that the ratio ω/k determines the speed of the traveling wave. Substituting Equation 4.5.10 into Equation 4.5.9 gives the **dispersion relation**

$$\omega(k) = ak - \beta 4\pi^2 k^3 \quad \Rightarrow \quad \text{wave speed} = \frac{\omega(k)}{k} = a - \beta 4\pi^2 k^2. \tag{4.5.11}$$

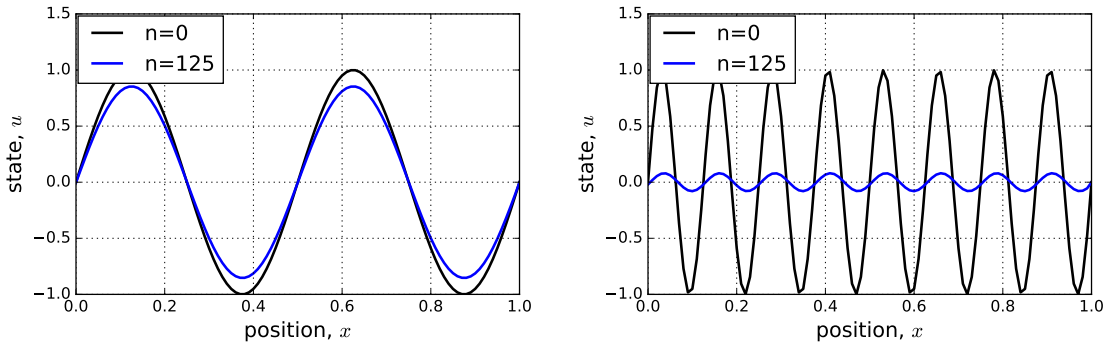
This equation tells us that waves of different spatial frequencies (k) propagate at different speeds when a third-derivative (dispersive) term is present in the PDE, Equation 4.5.9. The exact wave speed should be a for all waves, but Equation 4.5.11 indicates that, especially for high k, the wave speed will not be a if $\beta \neq 0$. In addition, when a second-derivative (diffusive/dissipative) term is present in Equation 4.5.9, the solution form in Equation 4.5.10 tells us to expect an exponential decay of the solution that proceeds more rapidly for higher frequencies, since the exponent is $-4\pi^2\alpha k^2$.

Example 4.13 (Dispersion relation for the first-order upwind scheme)**.** From Example 4.2, the modified equation for first-order upwind reads

$$\bar{u}_t + a\bar{u}_x = \frac{a\Delta x}{2}(1-\sigma)\bar{u}_{xx} - \frac{a\Delta x^2}{6}(1-\sigma^2)\bar{u}_{xxx}, \tag{4.5.12}$$

where we have kept additional high-order terms beyond those in Example 4.2. Though dispersion is present, the leading term ($\mathcal{O}(\Delta x)$ versus $\mathcal{O}(\Delta x^2)$) is dissipative. This means that the dominant effect on traveling waves will be decay of amplitude, the highest spatial frequencies showing the most attenuation by Equation 4.5.10, with wave speed differences secondary.

Figure 4.5.10 shows how FOU treats initial conditions of different frequencies. At a low spatial frequency, the attenuation is not very severe, while at the higher spatial frequency the initial amplitude is almost gone after the same number of iterations. A slight dispersive error is present, but it is difficult to discern from the figure as the solution is dominated by the diffusive error.



(a) Low spatial frequency

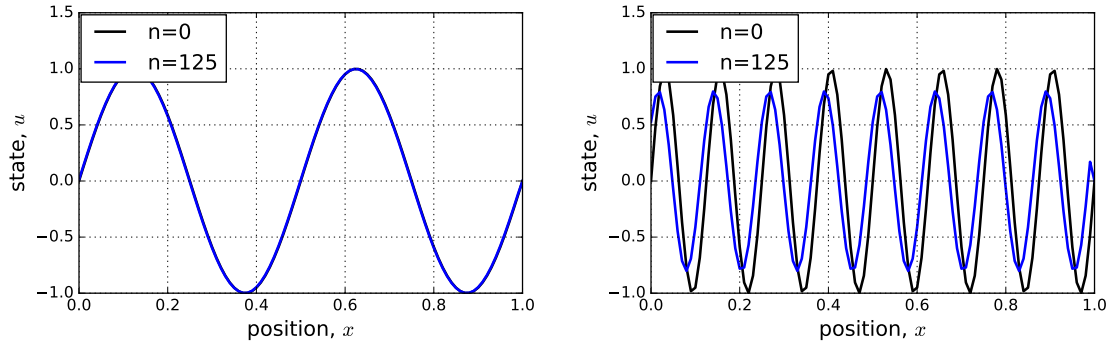
(b) High spatial frequency

Figure 4.5.10: First-order upwind solutions with $\sigma = 0.8$ for two initial conditions: spatial sinusoids of low and high frequency.

Example 4.14 (Dispersion relation for the Lax-Wendroff scheme)**.** The modified equation for Lax-Wendroff is

$$\bar{u}_t + a\bar{u}_x = -\frac{a\Delta x^2}{6}(1-\sigma^2)\bar{u}_{xxx}. \tag{4.5.13}$$

The leading term on the right-hand side is dispersive, with $\beta > 0$ (assuming $\sigma < 1$, which is required for stability). This means that, by the dispersion relation in Equation 4.5.11, the wave propagation speed is too small, with the high frequencies having the lowest wave



(a) Low spatial frequency

(b) High spatial frequency

Figure 4.5.11: Lax-Wendroff solutions with $\sigma = 0.8$ for two initial conditions: spatial sinusoids of low and high frequency.

speeds. Figure 4.5.11 shows this effect for initial conditions at two different frequencies. Note also the less severe attenuation compared to first-order upwind.

For an initial condition that consists of a superposition of frequencies, the high-frequency components will get "left behind"; i.e. they will propagate too slowly when using Lax-Wendroff. Figure 4.5.12 shows this behavior for a box (step) initial condition. Due to the discontinuities, the initial condition possesses significant high-order frequency content. As the solution progresses, these high-frequency modes propagate slower than their low-frequency counterpart, and as they get left behind, "wiggles" form on the *left* side of the propagating initial condition.

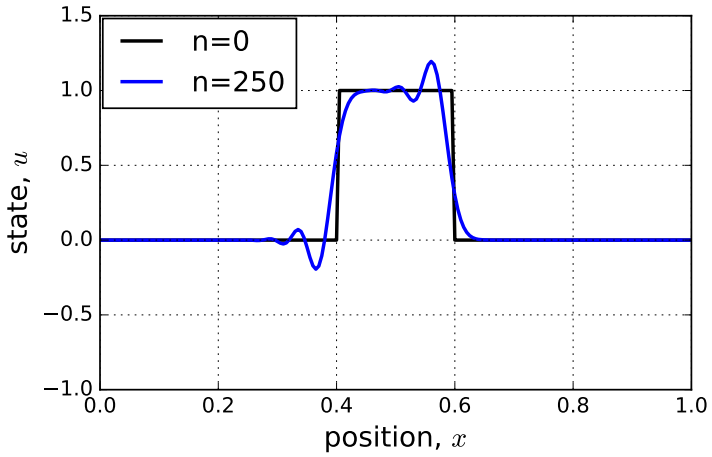


Figure 4.5.12: Lax-Wendroff solution with $\sigma = 0.8$ for a step initial condition.

Rather than using the modified equation, we can instead determine the dispersive behavior of a scheme through von-Neumann analysis. Recall from von-Neumann analysis, e.g. Equation 4.5.8, that we considered initial conditions of the form $u^0 = e^{ij\phi} = e^{ix\phi/\Delta x}$, where ϕ is the phase change per Δx. Comparing to forms with the wave number, where the exponent is $2\pi ikx$, we see that ϕ and k are related through $\phi = 2\pi k\Delta x$. In an advection problem with wave speed a, after a time step Δt, the exact solution at node j is

$$(u_j)_{\text{exact}}(\Delta t) = e^{i(x-a\Delta t)\phi/\Delta x} = e^{ij\phi}e^{-i\sigma\phi}, \tag{4.5.14}$$

where recall $\sigma = a\Delta t/\Delta x$. This equation tells us that, in the exact solution, the amplitude is unaffected whereas the phase is shifted by $-\sigma\phi$. On the other hand, the numerical solution after one time step is

$$(u_j)_{\text{computed}}(\Delta t) = |g|e^{ij\phi}e^{i\arg(g)}, \tag{4.5.15}$$

where $\arg(g) = \tan^{-1}[\Im(g)/\Re(g)]$. The factor $|g|$ out front is the attenuation factor for the amplitude of mode ϕ, whereas the argument of g is the phase change. In the ideal case, we would like $|g| = 1$ and $\arg(g) = -\sigma\phi$. However, a numerical scheme will generally not satisfy these equations for all ϕ, and hence we observe dissipation ($|g| < 1$) and dispersion $\arg(g) \neq -\sigma\phi$. Through von-Neumann analysis, we can calculate g as a function of ϕ, and hence we can discern the solution behavior for different frequencies.

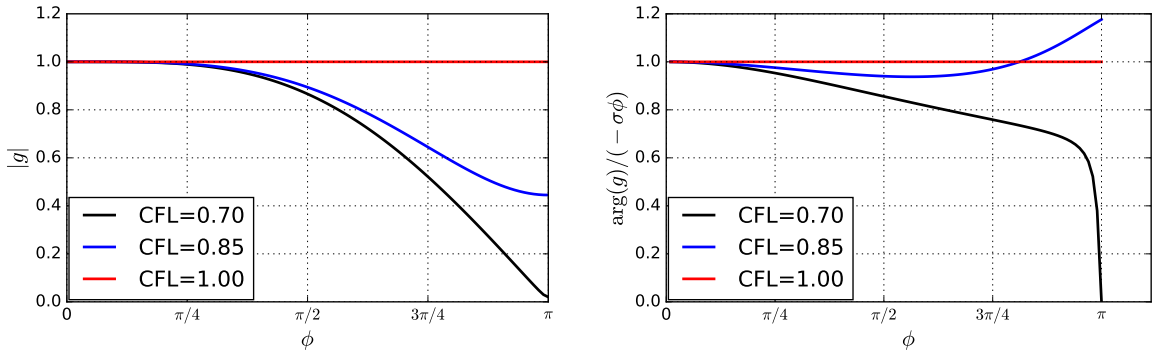
Example 4.15 (Amplification factor for the Lax-Wendroff scheme)**.** Recall the update formula for the Lax-Wendroff scheme in Equation 4.2.9, repeated here,

$$u_j^{n+1} = u_j^n - \frac{\sigma}{2}(u_{j+1}^n - u_{j-1}^n) + \frac{\sigma^2}{2}(u_{j-1}^n - 2u_j^n + u_{j+1}^n).$$

Substituting $u_j^n = g^n e^{ij\phi}$, we obtain

$$\begin{aligned} g^{n+1} &= g^n - \frac{\sigma}{2}g^n(e^{i\phi} - e^{-i\phi}) + \frac{\sigma^2}{2}g^n(e^{-i\phi} - 2 + e^{i\phi}), \\ g &= 1 - \frac{\sigma}{2}(e^{i\phi} - e^{-i\phi}) + \frac{\sigma^2}{2}(e^{-i\phi} - 2 + e^{i\phi}). \end{aligned}$$

Figure 4.5.13 shows the magnitude and argument of g for different CFL numbers. We see that when $\sigma = 1$, the amplification factor has a magnitude of exactly 1, and no phase error (the numerical phase speed matches the physical one). However, for $\sigma < 1$, the magnitude of the amplification factor drops, especially for high spatial frequencies (high ϕ), indicating attenuation of the high frequency modes. In addition, the phase error is generally negative, indicating that Lax-Wendroff propagates waves too slowly, which is consistent with the observations in Example 4.14.



(a) Attenuation: $|g|$

(b) Phase error: $\arg g/(-\sigma\phi)$

Figure 4.5.13: Magnitude and argument of the amplification factor for the Lax-Wendroff scheme, as a function of ϕ: the mode phase change per Δx.

4.6 Convergence

4.6.1 Global Error and Convergence

When we solve an initial value problem with a numerical scheme, we expect to see **convergence**, defined as the reduction of the error to zero as $\Delta x, \Delta t \to 0$. To make this precise, we need to specify what we mean by error. In Equation 4.4.6 we defined the *local error* as that made in one time step when integrating an ODE system. However, when solving a practical problem, we are more concerned with some measure of the *global error*, defined either over an entire time horizon or at the final time.

For a semi-discrete scheme with N time steps, one definition of the global error is an L_∞ norm of the error,

$$\epsilon_{\text{global}} = \max_{n=1\ldots N} \|\mathbf{u}^n - \mathbf{u}^n_{\text{exact}}\|_{L_\infty}.$$

The L_∞ norm at each time node picks out the maximum error over the components of the state, e.g. the spatial distribution. A particular scheme converges if the above error goes to zero as $\Delta x, \Delta t \to 0$. For convergent schemes, we can relate the global error to the local error through

$$\epsilon_{\text{global}} = \mathcal{O}\left(\sum_{n=1}^{N} \|\boldsymbol{\epsilon}^{n+1}_{\text{local}}\|\right). \tag{4.6.1}$$

That is, local errors add up over all time steps to yield the global error. If each local error is $\mathcal{O}(\Delta t^{p+1})$, then

$$\epsilon_{\text{global}} = \mathcal{O}\left(\sum_{n=1}^{N} \|\boldsymbol{\epsilon}^{n+1}_{\text{local}}\|\right) = \mathcal{O}\left(N\Delta t^{p+1}\right) = \mathcal{O}\left(\frac{1}{\Delta t}\Delta t^{p+1}\right) = \mathcal{O}\left(\Delta t^{p}\right). \tag{4.6.2}$$

So we expect the global error to decrease at a rate p that is one order lower than the rate for the local error, $p+1$. Finally, we mention that the local errors do not necessarily sum as indicated in Equation 4.6.2 when the scheme is unstable, since then the errors grow unbounded with iterations.

For general finite-difference methods, including fully-discrete schemes, the global error can be defined as

$$\epsilon_{\text{global}} = \|u(x,t) - u_{\text{exact}}(x,t)\|, \tag{4.6.3}$$

where $\|\cdot\|$ is a user-chosen norm over the fixed space-time domain, $\Omega \times [0,T]$. Alternatively, instead of a norm, the error could be measured just at the final time, or as an integral over just one boundary. The concept of convergence generally applies uniformly to a variety of norms, with the possible exception of the L_∞ norm: for problems with singularities (e.g. shocks) the error may remain high ($\mathcal{O}(1)$) in an isolated part of the domain, whereas the rest of the solution converges at a higher rate.

The **Lax equivalence theorem** states that a numerical scheme will converge if and only if it is consistent and stable. That is

$$\text{stability} \quad \& \quad \text{consistency} \quad \Leftrightarrow \quad \text{convergence}$$

This theorem applies to linear problems on a fixed domain. Stability ensures that the solution does not blow up as $\Delta x, \Delta t \to 0$, whereas consistency ensures that the numerical scheme solves the correct physical problem: i.e. that the truncation error goes to zero as $\Delta x, \Delta t \to 0$. Together, these criteria ensure that the numerical scheme converges to the true solution. In the context of time integration schemes for ODEs, a similar theorem is known as the **Dahlquist equivalence theorem**.

4.6.2 Manufactured Solutions

To test the rate of convergence numerically, we can run a simulation for which we know the exact solution to the PDE and then compare the numerical solution to the exact solution. Alternatively, if an exact solution is not available, we could run with a very fine mesh and use that solution as a surrogate for the exact one. However, this procedure will not guard against all errors made in converting the PDE to the discretized equations. For example, if we accidentally used a second derivative finite difference to discretize a first-derivative, a fine-mesh comparison would not catch the error. A better comparison in such a case is one against a manufactured solution.

A **manufactured solution** is one that, as the name suggests, is "made up". For example, for a scalar PDE, the user would pick some solution,

$$u(\vec{x},t) = u^{\text{MS}}(\vec{x},t). \tag{4.6.4}$$

Now, a made-up solution such as this will generally not satisfy the original PDE, which for the sake of demonstration we will take as advection-diffusion,

$$\frac{\partial u}{\partial t} + \vec{V}\cdot\nabla u - \nu\nabla^2 u \;=\; 0. \tag{4.6.5}$$

Substituting u^{MS} into Equation 4.6.5 results in a remainder of

$$s^{\text{MS}} \equiv \frac{\partial u^{\text{MS}}}{\partial t} + \vec{V} \cdot \nabla u^{\text{MS}} - \nu \nabla^2 u^{\text{MS}}. \tag{4.6.6}$$

If we take this remainder and treat it as a (negative) source term, we can obtain a modified PDE that u^{MS} *does* satisfy,

$$\frac{\partial u^{\text{MS}}}{\partial t} + \vec{V} \cdot \nabla u^{\text{MS}} - \nu \nabla^2 u^{\text{MS}} - s^{\text{MS}} = 0. \tag{4.6.7}$$

So, to test our discretization of the PDE, we have to discretize an additional source term, s^{MS}. Once we have this, we can run a simulation on an arbitrary domain with Dirichlet boundary conditions, $u^{\text{MS}}(\vec{x})$, and test how close the resulting solutions are to the chosen exact solution.

4.6.3 Spatial and Temporal Accuracy

When running unsteady simulations using semi-discrete methods, the solution is polluted by both spatial and temporal discretization errors. Figure 4.6.1 illustrates schematically the relationship between the spatial and temporal error. As a mesh is refined (smaller Δx), the temporal error will eventually dominate unless Δt is decreased as well. For explicit methods,

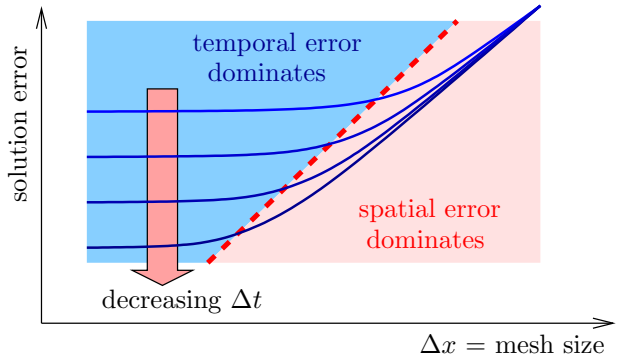


Figure 4.6.1: An imbalance in the spatial and temporal discretization accuracy leads to dominance of the error by one of the discretizations.

we generally cannot control both Δx and Δt independently, since they are tied together by stability restrictions (e.g. on the CFL number). In such cases it is important to match the orders of approximation in space and time, to prevent one of the errors from dominating the other.

As an example, consider the case of one-dimensional advection, on a uniform grid with N_x spatial intervals and periodic boundary conditions. We compute the error after one period of advection, relative to the initial condition, which is $u_0(x) = \exp[-(10x/L - 5)^2]$. In space, we discretize with two methods, first-order upwind (space order = 1), and second-order

upwind (order = 2). In time, we consider forward Euler (FE) and predictor corrector (RK2). Figure 4.6.2 shows how the error converges *with uniform spatial and temporal refinement* for different combinations of the space and time schemes. Note that uniform refinement means that both N_x and N_t are decreased concurrently, keeping the CFL number constant. When

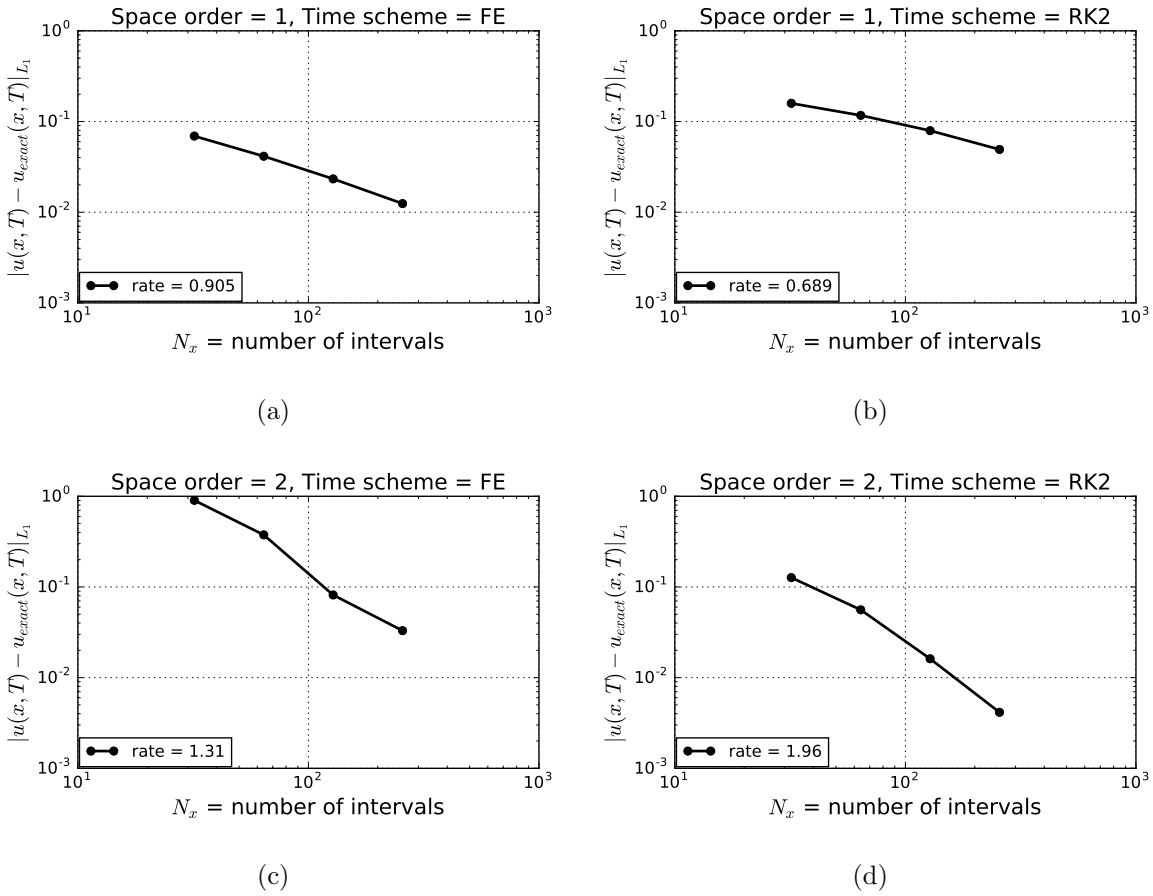


Figure 4.6.2: Error convergence for one-dimensional advection using two spatial orders, first and second order upwind, and two temporal discretizations, forward Euler and the predictor-corrector (RK2) scheme. The runs were performed at a constant CFL number of $\sigma = 0.8$ for the first-order in space, and $\sigma = 0.4$ for second-order in space.

the spatial and temporal orders are matched (e.g. space order 1 and FE, or space order 2 and RK2), then we observe the expected rate of error convergence (1 and 2 respectively). However, when the first-order upwind method is run with RK2, or when the second-order upwind method is run with forward Euler, the rate is limited to 1. So we see that the rate at which the error (compared to exact solution to the PDE) decreases is limited by the lower of the spatial and temporal orders of accuracy.

What if we wanted to compute spatial and temporal orders of accuracy separately? First, to verify the spatial order, we could run a convergence study where we vary N_x but keep N_t

constant at a very large value, and ideally use a high-order time integration method. The reason for keeping N_t high is stability, since we are refining N_x, and to make the temporal error negligible compared to the spatial error. Figure 4.6.3 shows such a comparison using the first-order spatial discretization and an over-refined RK2 temporal discretization. In this case, the spatial error dominates, and we see the expected first-order rate of convergence.

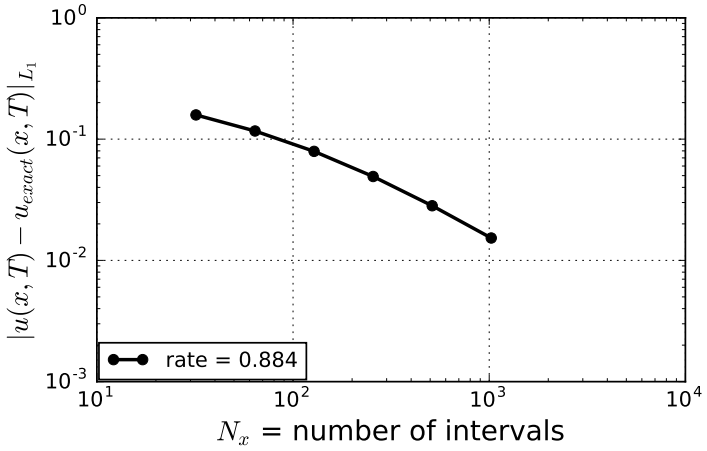


Figure 4.6.3: Error convergence for one-dimensional advection using first-order upwind in space. Each run used an overly-fine time discretization of predictor-corrector with $N_t = 2000$ time steps.

Now what if we wanted to verify the temporal order of accuracy? It is hard to eliminate just the unwanted spatial error, since this would require very fine spatial meshes, and hence necessarily very small time steps (for stability, assuming an explicit method), which would also make the temporal error very small. Instead, another approach is to make a different comparison, using a mesh where N_x is held fixed at some small value. Then, instead of comparing the solution to the exact one, we create a "truth" solution by running the code at the same small N_x, but a very high N_t, ideally with a high-order time integration. Figure 4.6.4 shows the result of such a study for a quite coarse spatial discretization of $N_x = 16$ and first-order upwind. Nevertheless, we see second-order time convergence, since we do not compare to the exact solution, but rather to the result with the same spatial discretization but a very fine time discretization.

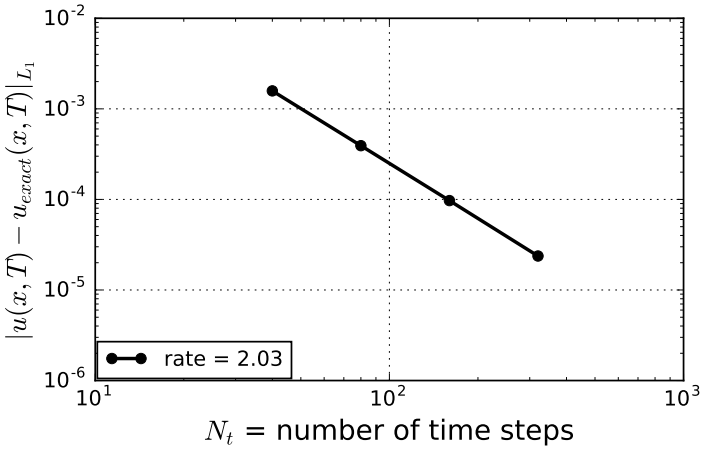


Figure 4.6.4: Error convergence for one-dimensional advection using predictor-corrector in time, with a varying number of time steps. Each run used a fixed grid of $N_x = 16$ intervals, and the error was computed relative to a "truth" solution generated with the same spatial grid and $N_t = 2000$.

Chapter 5

Conservation Laws

5.1 Scalar Conservation Equations

5.1.1 Integral and Differential Form

Let $u(x,t)$ represent a conserved state, with units of some physical quantity *per unit volume*[1]. For example, u could be the density of a fluid, which is mass per unit volume. Let $\vec{f}(u)$ be the flux of the physical quantity, with units of u times velocity. In the case where u is the density, $\vec{f}(u)$ is the amount of mass flowing per unit cross sectional area per unit time, i.e. density times velocity. In one dimension, $\vec{f}$ is just f.

Consider a one-dimensional control volume, i.e. an interval, between $x = x_L$ and $x = x_R$, as shown in Figure 5.1.1. When u is a conserved quantity, it cannot be created or destroyed. This means that the net amount of u entering the control volume via the fluxes on either ends must lead to accumulation of u inside the control volume. This physical reasoning leads

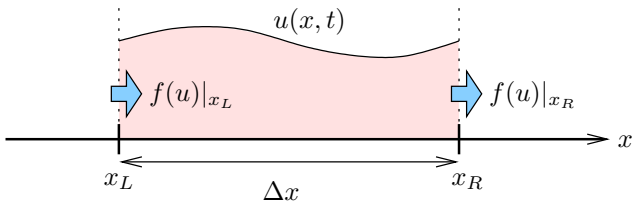


Figure 5.1.1: A one-dimensional control volume with fluxes on the left and right boundaries.

to the **integral form** of the conservation law,

$$\underbrace{\frac{d}{dt}\int_{x_L}^{x_R} u\,dx}_{\text{accumulation of } u} = \underbrace{f(u)|_{x_L} - f(u)|_{x_R}}_{\text{inflow of } u \text{ through } x_L \text{ and } x_R}. \tag{5.1.1}$$

[1]When working in one dimension, volume is just length, and in two dimensions it is area.

Taking the limit of $\Delta x \equiv x_R - x_L \to 0$ in Equation 5.1.1, we obtain

$$\begin{aligned} \Delta x \frac{\partial u}{\partial t} &= f(u)|_{x_L} - f(u)|_{x_L+\Delta x}\,, \\ \frac{\partial u}{\partial t} + \lim_{\Delta x \to 0} \frac{f(u)|_{x_L+\Delta x} - f(u)|_{x_L}}{\Delta x} &= 0. \end{aligned} \tag{5.1.2}$$

Using the definition of the derivative then gives the **differential form** of the conservation law,

$$\frac{\partial u}{\partial t} + \frac{\partial f(u)}{\partial x} = 0, \tag{5.1.3}$$

Linearizing $f(u)$ in Equation 5.1.3,

$$\frac{\partial f(u)}{\partial x} = \underbrace{\frac{df}{du}}_{a(u)} \frac{\partial u}{\partial x} \tag{5.1.4}$$

gives the **quasi-linear form** of the equation,

$$\frac{\partial u}{\partial t} + a(u)\frac{\partial u}{\partial x} = 0. \tag{5.1.5}$$

Note that $a(u)$ has units of velocity, and for scalar problems it is the propagation/transport speed of the state. For systems, a becomes a matrix whose eigenvalues are propagation speeds of different waves in the system.

5.1.2 Examples

Linear Advection

We devoted most of the previous chapter to linear advection, in the quasi-linear form

$$\frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} = 0, \tag{5.1.6}$$

where a is a constant. Taking a inside the derivative, the conservative form of advection reads

$$\frac{\partial u}{\partial t} + \frac{\partial (au)}{\partial x} = 0, \tag{5.1.7}$$

and we see that the flux is $f(u) = au$. Note, when a depends on x, these two equations are not equivalent, since differentiating the product au with respect to x creates not only au_x but also $a_x u$, which is an additional source term that must be included in the quasi-linear form.

Burgers Equation

In Burgers equation, f is no longer a linear function of u. Instead, $f = \frac{1}{2}u^2$, so that the conservative PDE reads

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x}\left(\frac{1}{2}u^2\right) = 0. \tag{5.1.8}$$

Differentiating the flux via the chain rule gives the quasi-linear form,

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0. \tag{5.1.9}$$

This equation is useful in the study conservation laws because it admits shocks, which generally cause difficulties and require special treatment for numerical methods. Note that the flux is **convex**, because $f''(u) \equiv d^2 f/du^2 = 1 > 0$.

Traffic Flow

It may seem odd to think of cars on a road as a fluid, but then again all fluids consist of discrete molecules. The continuum assumption is arguably more valid for fluids at low Knudsen numbers, but we still obtain insight and reasonable results from treating cars on a road as a continuum. In this equation, ρ is the density of cars, defined as the number of cars per unit road length. The flux of cars is

$$f(\rho) = \rho u_{\max}\left(1 - \frac{\rho}{\rho_{\max}}\right), \tag{5.1.10}$$

where $u_{\max}$ is the maximum speed of cars on the road (e.g. the speed limit), and $0 \leq \rho \leq \rho_{\max}$ is the maximum density of cars on the road (e.g. in a bumper-to-bumper traffic jam). The motivation for this flux comes from writing it as $f = \rho u(\rho)$, where

$$u(\rho) = u_{\max}\left(1 - \frac{\rho}{\rho_{\max}}\right) \tag{5.1.11}$$

is the speed of cars as a function of density. If barely any cars are present ($\rho \approx 0$), the cars will travel fast at $u \approx u_{\max}$. However, when the road is congested (ρ high), cars slow down, until $u = 0$ at $\rho = \rho_{\max}$ (stand-still). Note that f is a nonlinear, **concave** function of ρ, $f''(\rho) = -2u_{\max}/\rho_{\max} < 0$ and it also admits shock solutions. The equation in conservative form then reads

$$\frac{\partial \rho}{\partial t} + \frac{\partial f(\rho)}{\partial x} = 0. \tag{5.1.12}$$

Buckley-Leverett

This equation is used as a simple model for two-phase flow, such as oil and water. Let $0 \leq u \leq 1$ represent the saturation (volume fraction) of one of the fluids, e.g. water. So $u = 1$ represents pure water, while $u = 0$ represents pure oil. The flux for this equation is

$$f(u) = \frac{1}{1 + a(1/u^2 - 1)}, \tag{5.1.13}$$

where a is a constant. The conservative form is then the familiar equation

$$\frac{\partial u}{\partial t} + \frac{\partial f(u)}{\partial x} = 0.$$

5.2 The One-Dimensional Finite Volume Method

In the finite volume method, we divide our computational domain into non-overlapping control volumes, or cells. Inside each cell, we store some information about the state, typically just the average value in the cell. At cell interfaces, we define fluxes that provide information on how much of the state flows in and out of the adjacent cells. To obtain a steady-state solution, we seek the cell averages that make the fluxes balanced (zero residual) on each cell. In an unsteady calculation, we simply evolve each cell average (increase it or decrease it) based on the flux imbalance. In this section, we look at how the finite volume method works in one spatial dimension.

5.2.1 Cell Averages and Updates

In one spatial dimension, define the computational domain by $x \in [0, L]$. We divide this domain into N cells, i.e. intervals, which need not be of equal size. For cell j, denote by x_j the middle of the cell, and by $x_{j\pm1/2}$ the left and right boundaries of the cell, as illustrated in Figure 5.2.1.

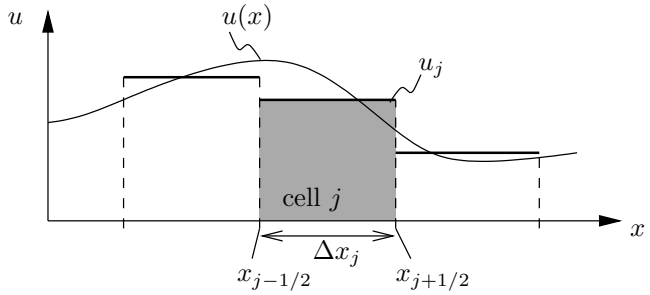


Figure 5.2.1: Cells and cell averages for a 1D finite method.

We'll restrict our attention to scalars for the time being in order to simplify the presentation. Applying the integral form of the conservation law, Equation 5.1.1, on cell j

gives

$$\frac{d}{dt}\int_{x_{j-1/2}}^{x_{j+1/2}} u\,dx \quad = \quad f(u)|_{x_{j-1/2}} - f(u)|_{x_{j+1/2}}\,. \tag{5.2.1}$$

Define the **cell average** of the state inside cell j as

$$u_j \equiv \frac{1}{\Delta x_j}\int_{x_{j-1/2}}^{x_{j+1/2}} u\,dx. \tag{5.2.2}$$

Using this definition and the shorthand $f_{j\pm1/2} = f(u)|_{x_{j\pm1/2}}$, Equation 5.2.1 becomes

$$\Delta x_j \frac{du_j}{dt} \quad = \quad f_{j-1/2} - f_{j+1/2}. \tag{5.2.3}$$

Bringing all terms to the left-hand side then gives

$$\Delta x_j \frac{du_j}{dt} + \underbrace{f_{j+1/2} - f_{j-1/2}}_{R_j=\text{flux residual}} \quad = \quad 0. \tag{5.2.4}$$

Note that we have defined the flux residual, R_j, as the imbalance of the two fluxes for cell j. A positive flux residual means a net outflow of u from the cell and leads to a decrease in the cell average.

So far, we have not made any approximations, as Equation 5.2.4 expresses a physical conservation statement on cell j. However, we cannot turn Equation 5.2.4 into a numerical method without first specifying how we will evaluate the fluxes on cell interfaces, $f_{j\pm1/2}$. Recall that in the finite volume method we will store the cell averages, u_j, as unknowns. This means we do not know the state at the cell interfaces. So we must define **numerical fluxes**, $\hat{F}_{j\pm1/2}$, that express the interface fluxes in terms of the unknowns u_j. Equation 5.2.4 then becomes

$$\Delta x_j \frac{du_j}{dt} + \hat{F}_{j+1/2} - \hat{F}_{j-1/2} \quad = \quad 0. \tag{5.2.5}$$

The choice of the numerical flux is the critical part of a finite volume method, as it determines accuracy, stability, and computational cost.

5.2.2 Numerical Fluxes

In this section we present several methods for computing numerical fluxes at interfaces between cells. Unless otherwise specified, we will consider the scalar advection equation,

$$u_t + au_x \quad = \quad 0, \tag{5.2.6}$$

for which the analytical flux is $f(u) = au$.

Central

The flux at an interface must reconcile two generally different cell averages on either side. One "fair" way of doing this is to just average the fluxes based on these two cell averages:

$$\hat{F}_{j+1/2} = \frac{1}{2}(f_j + f_{j+1}), \tag{5.2.7}$$

where $f_j = f(u_j)$ is the flux computed from the cell average state in cell j. Substituting this formula into Equation 5.2.5, we obtain

$$\begin{aligned} \Delta x_j \frac{du_j}{dt} + \frac{1}{2}(f_j + f_{j+1}) - \frac{1}{2}(f_{j-1} + f_j) &= 0 \\ \Delta x_j \frac{du_j}{dt} + \frac{1}{2}(f_{j+1} - f_{j-1}) &= 0 \end{aligned}$$

At this point, we need a time discretization. The simplest choice is forward Euler; however, this results in an unstable method. This instability can be predicted by noting that the above equation becomes the FTCS method, e.g. from Example 4.11! Granted, that equation was for node states in a finite-difference method, whereas we are now talking about cell averages. Yet, the formulas are identical, which means that we will observe instabilities in our cell averages just like we saw instabilities in the node states. This does not mean that the central discretization is useless in itself: it just has to be paired with either a stronger time-stepping method, with eigenvalue stability along the imaginary axis, or it has to be augmented with additional terms (e.g. diffusion) that stabilize it.

Upwind

A simple flux that builds in the physics of the conservation equation is the upwind flux,

$$\hat{F}_{j+1/2} = \begin{cases} au_j & \text{for } a \geq 0 \\ au_{j+1} & \text{for } a < 0 \end{cases} \tag{5.2.8}$$

This is an asymmetric flux that uses the value of the state from only one side of the interface, as determined by the sign of a. The motivation for this choice comes from considering a piece-wise constant representation of the solution, with the solution in each cell equal to the cell average. What happens when this solution evolves in time? Figure 5.2.2 shows how such a state evolves for $a > 0$: the entire distribution shifts to the right. Looking at the interfaces, we see that the states at the interfaces take on values from the *left* cells for $t > 0$. So the numerical flux in the case of $a > 0$ should use the left states. Conversely, if $a < 0$, the numerical flux should use the right states.

We write this flux, Equation 5.2.8 more compactly, but equivalently, as

$$\hat{F}_{j+1/2} = \frac{1}{2}\left(au_j + au_{j+1}\right) - \frac{1}{2}|a|\left(u_{j+1} - u_j\right). \tag{5.2.9}$$

In this form the numerical flux is the average of the left and right fluxes, corrected by a term that introduces upwinding. Note, the term "upwind" refers to the observation that, at any

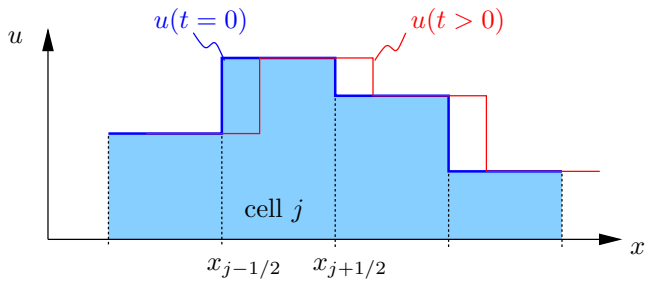


Figure 5.2.2: Advection of a piecewise constant solution for $a > 0$: the entire profile shifts to the right, so that the flux at each interface is determined by the state to the left of that interface.

point in space, we need to look in the direction from which the "wind" (velocity) is coming to determine the state at a future time.

Combined with forward Euler time stepping, Equation 5.2.5 becomes

$$\Delta x_j \frac{u_j^{n+1} - u_j^n}{\Delta t} + \hat{F}_{j+1/2}^n - \hat{F}_{j-1/2}^n = 0. \tag{5.2.10}$$

Substituting Equation 5.2.9 and solving for u_j^{n+1} yields

$$u_j^{n+1} = u_j^n - \underbrace{\frac{a\Delta t}{2\Delta x}(u_{j+1}^n - u_{j-1}^n)}_{\text{centered difference}} + \underbrace{\frac{|a|\Delta t}{2\Delta x}(u_{j+1}^n - 2u_j^n + u_{j-1}^n)}_{\text{diffusive stabilization}}. \tag{5.2.11}$$

As indicated, we can think of this update as a combination of a centered-difference discretization (i.e. FTCS – unstable) and a diffusive stabilization. The upwinding is responsible for the diffusive term, which stabilizes the scheme.

Finally, for nonlinear problems, the equivalent form of Equation 5.2.9 is

$$\hat{F}_{j+1/2} = \frac{1}{2}(f_j + f_{j+1}) - \frac{1}{2}|\hat{a}_{j+1/2}|(u_{j+1} - u_j). \tag{5.2.12}$$

where the speed $\hat{a}_{j+1/2}$ is defined as

$$\hat{a}_{j+1/2} = \begin{cases} \frac{f_{j+1} - f_j}{u_{j+1} - u_j} & \text{when } u_{j+1} \neq u_j, \\ f'(u_j) & \text{when } u_{j+1} = u_j. \end{cases} \tag{5.2.13}$$

Note that this choice of the upwind speed results in pure upwinding of the flux:

$$\hat{F}_{j+\frac{1}{2}} = \begin{cases} f_j & \text{when } \hat{a}_{j+1/2} > 0, \\ f_{j+1} & \text{when } \hat{a}_{j+1/2} \leq 0. \end{cases} \tag{5.2.14}$$

The property that $\hat{a}_{j+1/2}(u_{j+1} - u_j) = f_{j+1} - f_j$ is a condition that underlies the design of the class of numerical fluxes proposed by Philip Roe, and a flux in this class is called a **Roe flux**.

Lax-Wendroff

The Lax-Wendroff flux is

$$\hat{F}_{j+1/2} = \frac{1}{2}(f_j + f_{j+1}) - \frac{1}{2}\hat{a}^2_{j+1/2}\frac{\Delta t}{\Delta x}(u_{j+1} - u_j), \tag{5.2.15}$$

where $\hat{a}_{j+1/2}$ is defined as for the upwind method, in Equation 5.2.13. For a linear problem, $\hat{a}_{j+1/2} = a$, and combined with forward Euler time stepping, Equation 5.2.5 for this flux reduces to the Lax-Wendroff update in Equation 4.2.9,

$$u_j^{n+1} = u_j^n - \frac{\sigma}{2}(u_{j+1}^n - u_{j-1}^n) + \frac{\sigma^2}{2}(u_{j-1}^n - 2u_j^n + u_{j+1}^n).$$

Although this formula was derived in a finite-difference context, with states at nodes, the same formula applies in the finite-volume context with the cell-average as the state. The fact that we can write the Lax-Wendroff method in the form of Equation 5.2.5, with a unique interface flux, Equation 5.2.15, makes this method conservative.

5.3 Multiple Dimensions and Systems

A system of conservation laws in multiple dimensions can be written in the following general, differential form,

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot \vec{\mathbf{F}} = \mathbf{0}, \tag{5.3.1}$$

where

- $\mathbf{u} \in \mathbb{R}^s$ is the conservative state vector with s components.
- $\vec{\mathbf{F}} = \vec{\mathbf{F}}(\mathbf{u}) \in [\mathbb{R}^s]^{\text{dim}}$ is a flux that tells us the direction and rate at which the conserved quantities pass through space. Note that dim is the number of spatial dimensions, and the arrow in $\vec{\mathbf{F}}$ denotes a spatial vector. The units of each flux are conserved quantity per unit area per unit time.

In the previous chapter we already looked at scalar conservation laws in one dimension. Some of these extend to multiple dimensions as well. For example, linear advection for a conserved scalar quantity, u, in a prescribed velocity field $\vec{V}$, reads

$$\frac{\partial u}{\partial t} + \nabla \cdot \underbrace{\left(\vec{V}u\right)}_{\vec{F}} = 0. \tag{5.3.2}$$

That is, the flux in this case is $\vec{F} = \vec{V}u$.

The differential form in Equation 5.3.1 derives from an *integral form*, which we can recover by integrating Equation 5.3.1 over an area A[2]. At present, we restrict our attention

[2] in three dimensions, this would be a volume

to two spatial dimensions. Referring to the definitions in Figure 5.3.1, the integral form of Equation 5.3.1 is

$$\begin{aligned}
\int_A \left[\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot \vec{\mathbf{F}}\right] dA &= \mathbf{0} \\
\int_A \frac{\partial \mathbf{u}}{\partial t} dA + \underbrace{\int_A \nabla \cdot \vec{\mathbf{F}} dA}_{\text{use divergence thm.}} &= \mathbf{0} \\
\int_A \frac{\partial \mathbf{u}}{\partial t} dA + \oint_{\partial A} \vec{\mathbf{F}} \cdot \vec{n}\, dl &= \mathbf{0} \qquad (5.3.3)
\end{aligned}$$

Note that in Equation 5.3.3 the normal vector $\vec{n}$ points outward from the enclosed area.

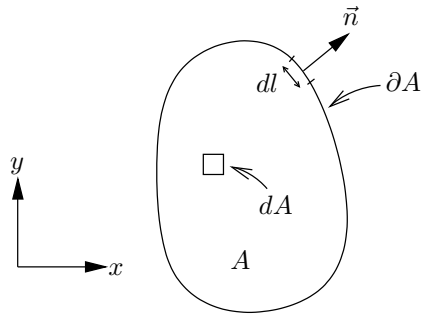


Figure 5.3.1: Definitions for application of the divergence (Gauss) theorem in two dimensions.

5.3.1 The Euler Equations

The compressible Euler equations of gas dynamics govern the flow of a perfect inviscid gas. They derive from the vanishing-viscosity (high-Reynolds number) limit of the compressible Navier-Stokes equations. The equations encode conservation statements for mass, momentum, and energy. In two spatial dimensions, the Euler equations are

$$\begin{aligned}
&\frac{\partial}{\partial t}(\rho) + \frac{\partial}{\partial x}(\rho u) + \frac{\partial}{\partial y}(\rho v) = 0 && \text{(mass)} \\
&\frac{\partial}{\partial t}(\rho u) + \frac{\partial}{\partial x}(\rho u^2 + p) + \frac{\partial}{\partial y}(\rho uv) = 0 && (x\text{-momentum}) \\
&\frac{\partial}{\partial t}(\rho v) + \frac{\partial}{\partial x}(\rho vu) + \frac{\partial}{\partial y}(\rho v^2 + p) = 0 && (y\text{-momentum}) \\
&\frac{\partial}{\partial t}(\rho E) + \frac{\partial}{\partial x}(\rho uH) + \frac{\partial}{\partial y}(\rho vH) = 0 && \text{(energy)}
\end{aligned}$$

The variables appearing in this system of equations are defined as follows:

$$
\begin{array}{lcl}
\rho & = & \text{density} \\
u, v & = & x, y \text{ components of velocity, } \vec{v} \\
E & = & \text{total energy per unit mass} \\
e & = & \text{internal energy per unit mass} \\
H & = & \text{total enthalpy per unit mass} \\
h & = & \text{internal enthalpy per unit mass} \\
p & = & \text{pressure} \\
R & = & \text{gas constant for air, } c_p - c_v \\
\gamma & = & \text{ratio of specific heats, } c_p/c_v = 1.4 \\
c_p & = & \text{specific heat at constant pressure} \\
c_v & = & \text{specific heat at constant volume} \\
c & = & \text{speed of sound} \\
M & = & \text{Mach number}
\end{array}
$$

Calorically-perfect gas:

$$
\begin{aligned}
e &= c_v T \\
h &= c_p T \\
p &= \rho R T \\
&= (\gamma - 1) \underbrace{\left[\rho E - \frac{1}{2} \rho |\vec{v}|^2 \right]}_{\rho e} \\
H &= E + p/\rho \\
M &= |\vec{v}|/c = \sqrt{u^2 + v^2}/c \\
c &= \sqrt{\gamma R T} = \sqrt{\gamma p/\rho}
\end{aligned}
$$

The Euler equations can be written in the compact form of Equation 5.3.1,

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot \vec{\mathbf{F}}(\mathbf{u}) = \mathbf{0},$$

by defining the state and flux vectors as

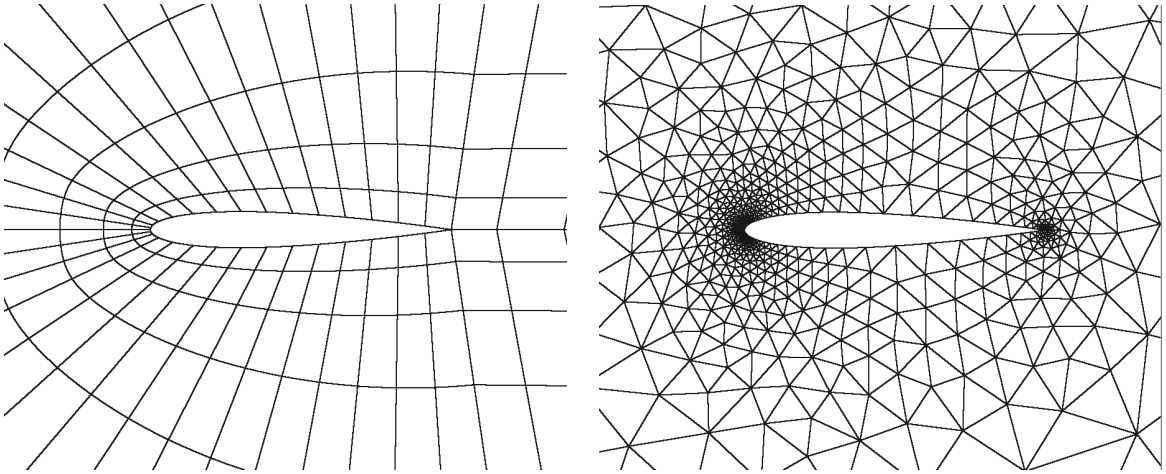
$$\text{state:} \quad \mathbf{u} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho E \end{bmatrix}, \qquad \text{flux:} \quad \vec{\mathbf{F}} = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ \rho uH \end{bmatrix} \hat{x} + \begin{bmatrix} \rho v \\ \rho vu \\ \rho v^2 + p \\ \rho vH \end{bmatrix} \hat{y}.$$

Note that the spatial nature of the vector $\vec{\mathbf{F}}$ is evident from the two components, one along $\hat{x}$ and the other along $\hat{y}$. Each of these components is a rank $s = 4$ vector.

5.4 Cells and Meshes

To solve conservation laws with the finite volume method, we need to use a computational mesh. In one spatial dimension, this was easy, as we could just split the domain into intervals. In general, a **computational mesh**, or **grid**, is a partitioning of the domain into non-overlapping volumes that are simple to work with – these volumes are called **cells**, or **elements**. Figure 5.4.1 shows sample meshes around an airfoil geometry.

In the finite volume method the cells in the meshes can be of almost arbitrary shape. Meshes composed of triangles (2D) and tetrahedra (3D), i.e. **simplex cells**, are usually the easiest to generate. Quadrilaterals (2D) and hexahedra (3D) are also used, although such meshes can be harder to generate for complex geometries. Sometimes meshes contain cells of different shapes, e.g. triangles mixed in with quadrilaterals, and these are called **hybrid meshes**. Figure 5.4.2 shows an example of a hybrid mesh.



(a) Structured mesh (b) Unstructured mesh

Figure 5.4.1: Sample structured and unstructured meshes around a NACA 0012 airfoil.

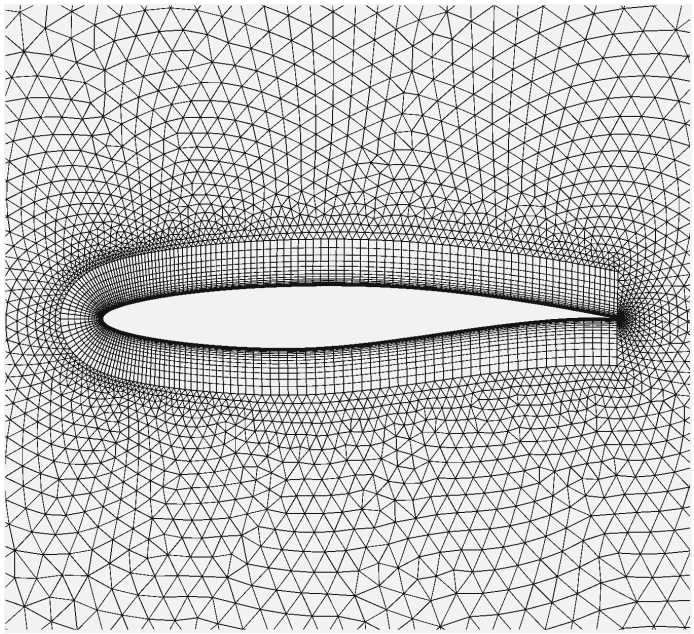


Figure 5.4.2: An example of a hybrid mesh containing both triangles and quadrilaterals.

An important distinction between mesh types is **structured** versus **unstructured**. In a structured mesh, the connectivity between cells is implied and not explicitly stored. An example of a structured mesh is a lattice in a Cartesian coordinate system, e.g. the set of points (i, j) where i and j are integers. The neighbors of each point can be found by just incrementing/decrementing the indices i, j. The mesh in Figure 5.1(a) is an example of a structured mesh. On the other hand, finding the neighbors of a cell in the mesh in Figure 5.1(b) is not as easy. Instead, for such meshes the connectivity between cells is typically stored, and these meshes are called unstructured. Most often, although not always, unstructured meshes will consist of triangles in two dimensions and tetrahedra in three dimensions.

The number of cells in a mesh and local variations in size, orientation, and stretching of cells all affect the quality of the finite-volume solution. The mesh must also resolve the geometry to a sufficient accuracy. Mesh generation is therefore an important but difficult task, best performed adaptively to improve solution accuracy.

5.5 The 2D Finite Volume Method

We now present the two-dimensional finite volume method: a technique for numerically obtaining an approximate solution to the conservation law. The starting point is the integral form in Equation 5.3.3, which we apply repeatedly to cells of the computational mesh. This is the essential part of the finite-volume method. As in one spatial dimension, we define the *cell average* of the state as

$$\text{cell average on cell } i \ = \mathbf{u}_i \equiv \frac{1}{A_i} \int_{A_i} \mathbf{u} \, dA,$$

where A_i refers to the area of cell i. Eqn. 5.3.3 in terms of cell averages becomes

$$\boxed{A_i \frac{d\mathbf{u}_i}{dt} + \oint_{\partial A_i} \vec{\mathbf{F}} \cdot \vec{n} \, dl = \mathbf{0}} \tag{5.5.1}$$

The cell averages are the unknowns in our discretization.

5.5.1 Scalar Flux Residual

Consider a finite volume method for a scalar problem in two dimensions. The PDE is given by Equation 5.3.2, and we use Equation 5.5.1 for our discretization, noting that the cells can be of various shapes. Since the shape of the cells does not change the key concepts, we focus our attention on triangles.

Figure 5.5.1 shows an example of a triangular cell with its nearest neighbors. Equation 5.5.1 applied to this cell, for a scalar problem, yields

$$A_i \frac{du_i}{dt} \underbrace{+ \oint_{\partial A_i} \vec{F} \cdot \vec{n} \, dl}_{R_i} = 0, \tag{5.5.2}$$

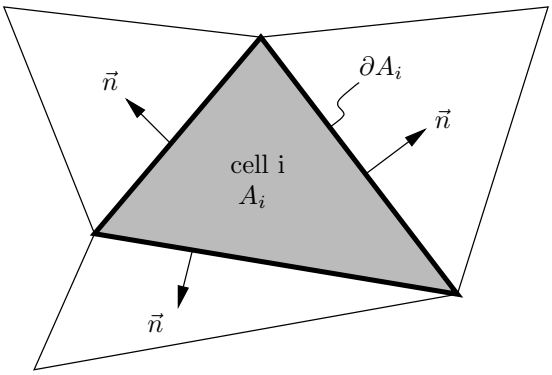


Figure 5.5.1: A triangular cell, indexed by i, in two dimensions. The edges are straight and the normal is outward-pointing and constant on each edge.

where R_i is the flux residual. The calculation of the residual requires knowing the flux on the edges in between cells, and this is not something we have direct access to when all we store is the cell averages. As in the one-dimensional case, this is the point where the approximation comes in: we must define a *numerical flux* that approximates the flux on the edges from an incomplete description of the state – just the cell averages. The residual calculation specifically requires only the flux dotted with the edge normal, so we approximate this directly:

$$\text{normal flux on edge } e \text{ between elements } L \text{ and } R \ = \vec{F} \cdot \vec{n} \approx \hat{F}(u_L, u_R, \vec{n}), \qquad (5.5.3)$$

where L and R denote two adjacent elements with $\vec{n}$ the normal on their common edge, pointing from L to R as shown in Figure 5.5.2. To approximate an integral over the edge,

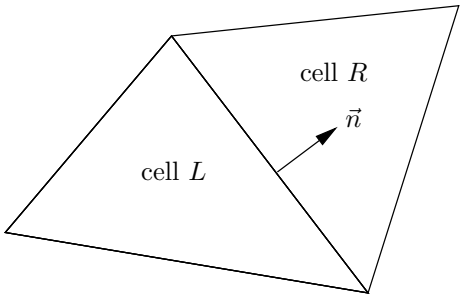


Figure 5.5.2: Two elements sharing a common edge. The L versus R designation is arbitrary, and the only requirement is that $\vec{n}$ points from L to R.

we assume that either (1) the flux is constant along the edge, which is valid for first-order methods, in which the state is also assumed constant inside each cell; or (2) that the flux varies linearly along the edge, which is needed for second-order-accurate methods. In either case, the integral can be approximated as $\hat{F}$ multiplied by the edge length, with $\hat{F}$ evaluated

on the edge midpoint in case (2) for a single-point quadrature formula that handles linear variation. The flux residual in Equation 5.5.2 is then

$$R_i = \sum_{e=1}^{3} \hat{F}(u_i, u_{N(i,e)}, \vec{n}_{i,e})\, \Delta l_{i,e}, \tag{5.5.4}$$

where, referring to Figure 5.5.1, e is an index over the three edges, $N(i,e)$ is the element adjacent to cell i across edge e, $\vec{n}_{i,e}$ is the outward-pointing normal vector from cell i on edge e, and $\Delta l_{i,e}$ is the length of edge e in cell i. Figure 5.5.3 illustrates these definitions.

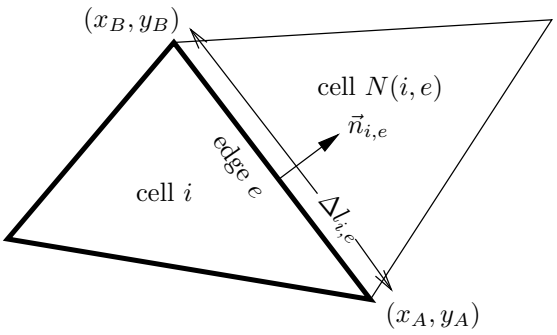


Figure 5.5.3: Definitions of edges and normals for a triangle.

For an edge between two points (x_A, y_A) and (x_B, y_B) the length and normal are

$$\begin{aligned} \Delta l_{i,e} &= \sqrt{(x_A - x_B)^2 + (y_A - y_B)^2}, \\ \vec{n}_{i,e} &= \left[(y_B - y_A)\hat{x} + (x_A - x_B)\hat{y}\right] / \Delta l_{i,e}. \end{aligned}$$

Note that this equation for the normal requires a counter-clockwise ordering of the nodes (i.e. A and B) relative to cell i.

$\hat{F}(u_i, u_{N(i,e)}, \vec{n}_{i,e})$ is the flux function that tells us, in our initial scalar example, how much of u is flowing from cell i to the neighbor cell, $N(i,e)$, through edge e. We saw in the one-dimensional case that the upwind method provides stability, and we use the same idea in two dimensions. The upwinding decision is now made based on the normal component of the velocity field, $\vec{V} \cdot \vec{n}_{i,e}$. That is, we write,

$$\hat{F}(u_i, u_{N(i,e)}, \vec{n}_{i,e}) = \begin{cases} \vec{V} \cdot \vec{n}_{i,e} u_i & \text{for } \vec{V} \cdot \vec{n}_{i,e} \geq 0 \\ \vec{V} \cdot \vec{n}_{i,e} u_{N(i,e)} & \text{for } \vec{V} \cdot \vec{n}_{i,e} < 0 \end{cases}.$$

We can write this as one equation by using absolute values,

$$\hat{F}(u_i, u_{N(i,e)}, \vec{n}_{i,e}) = \frac{1}{2}\vec{V} \cdot \vec{n}_{i,e} \left(u_i + u_{N(i,e)}\right) - \frac{1}{2}|\vec{V} \cdot \vec{n}_{i,e}| \left(u_{N(i,e)} - u_i\right).$$

5.5.2 Time Integration

In a semi-discrete formulation, the last piece is the choice of time integration. For a first-order spatial discretization, the forward Euler method works sufficiently well. Referring to Equation 5.5.2, the complete discrete equation is

$$A_i \frac{u_i^{n+1} - u_i^n}{\Delta t^n} + \underbrace{\sum_{e=1}^{3} \hat{F}(u_i, u_{N(i,e)}, \vec{n}_{i,e})\, \Delta l_{i,e}}_{R_i} = 0. \tag{5.5.5}$$

Note that Δt gets a superscript n to allow for the possibility of a variable time step. This equation can be solved for u_i^{n+1}, the updated state at time node $n+1$ for cell i,

$$u_i^{n+1} = u_i^n - \frac{\Delta t^n}{A_i} R_i. \tag{5.5.6}$$

Note that the update depends only on the states of cells that are immediately adjacent to cell i.

To set the time step, we use the CFL condition. Recall that in one spatial dimension, the CFL number was given by $CFL = |s|\Delta t/\Delta x$, where s is the wave speed (e.g. a in advection). When the cell size is not constant, which is usually the case for unstructured meshes in two dimensions, the CFL number has to be defined separately for every cell. On cell i, the definition is

$$CFL_i = \frac{\Delta t |\bar{s}|_i}{d_i}, \tag{5.5.7}$$

where d_i is a measure of the cell size, and $|\bar{s}_i|$ is the maximum wave speed. For the cell size, the hydraulic radius is typically used,

$$d_i = \frac{2A_i}{P_i}, \tag{5.5.8}$$

where A_i is the area of cell i and P_i is its perimeter. For $|\bar{s}_i|$, an edge-weighted average of wave speeds is used

$$|\bar{s}|_i = \sum_{e=1}^{3} |s|_{i,e} \Delta l_{i,e} / P_i,$$

where $|s|_{i,e}$ is the wave speed on edge e of cell i. This wave speed can be computed during the flux calculation on each edge.

When solving an unsteady problem time-accurately, every cell takes the same time step, which is set to be the smallest (most restrictive) over all of the cells,

$$\text{global } \Delta t = \min_i \left(\frac{\text{CFL}\, d_i}{|\bar{s}|_i} \right). \tag{5.5.9}$$

Note that the CFL number is a constant over all cells, prescribed by the user. When a mesh has very large and very small cells, the small cells will restrict the time step to small values. This is necessary to keep the explicit time marching stable, but it can lead to a hefty computational burden in terms of a large number of time steps. There are at least two ways to address this problem: local time stepping, and implicit methods.

First, suppose that we are not performing a time-accurate simulation, but are instead using time marching as a way to drive the solution to steady state. In such a case, *how* we get to steady state is not important, and so we dispense with time accuracy. Instead of marching every cell with the same time step, we let each cell set its own time step, according to its own local CFL condition. That is, the time step on cell i is

$$\text{local } \Delta t_i = \frac{\text{CFL}\, d_i}{|\bar{s}|_i} = \frac{2A_i\, \text{CFL}}{\sum_{e=1}^{3} |s|_{i,e} \Delta l_{i,e}},$$

where the CFL number is set globally by the user. The above equation gives the Δt_i for **local time stepping**. These element-specific time steps are the ones from which we would pick the minimum for global time stepping. So why does using different local time steps work? It seems non-physical ... and it is. However, if all we want from the solution is steady-state, then the path taken by the solution on the way to steady state is not critical. The above choice of time steps lets big cells advect their errors quickly, which speeds up the rate at which error modes leave the domain and hence accelerates convergence.

Second, suppose that we are actually interested in a time-accurate simulation, but that the time step restriction imposed by the smallest cells is too restrictive, much below the Δt we would need for time accuracy. In such a case, the problem is *stiff*, and we could often benefit from an implicit time marching method. These come at a price of solving a system of potentially nonlinear equations at each time step, e.g. via the Newton-Raphson method.

5.5.3 Implementation

In a finite-volume implementation, the flux is typically the most time-consuming part of the update calculation. Since there is only one flux per edge of the mesh, the fluxes should be calculated by looping over the edges and accumulating residuals on each element. Once all flux residuals are computed, each cell takes an update according to Equation 5.5.5. Then the flux residuals are computed again and the whole process repeats, as shown in Algorithm 9. When driving a problem to steady-state, the initial condition is usually something simple, such as the free-stream conditions, and for large meshes, iterating for thousands of time steps is not uncommon. When marching in a time-accurate manner, the number of time steps is prescribed by the final time of the simulation.

Algorithm 9 The first-order finite-volume method for a steady-state simulation.

Compute/store edge normals, cell areas, etc.
Initialize the state, $\mathbf{U} = \mathbf{U}^0$
for $n = 1 : N_{\text{max iter}}$ **do**
    Initialize residual and wave speed on each cell to zero
    for $f = 1 : N_{\text{face}}$ **do**
        Let L/R be the elements adjacent to face f
        Let $\vec{n}$ be the normal pointing out of L
        Compute the flux on face f using cell-averaged states
        Increment residual on L, decrement it on R
        Add wave speed to tallies on L and R cells
    end for
    If $|\mathbf{R}| <$ tolerance, **break**
    Compute the time step, Δt^n, cell-specific if using local time stepping
    Update the state: $\mathbf{u}_i^{n+1} = \mathbf{u}_i^n - \Delta t^n \mathbf{R}_i / A_i$
end for

Example 5.1 (A scalar finite-volume code on a structured mesh)**.** Consider a square domain, $(x, y) \in [-2, 2]^2$, meshed with N intervals in both the x and y directions, as shown in Figure 5.5.4. In this case, each cell is a square. We solve the time-accurate scalar advection

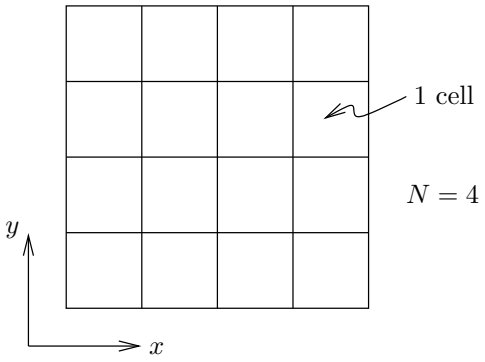


Figure 5.5.4: Mesh used for a sample finite-volume code applied to the 2D scalar advection equation.

equation on this mesh, with velocity $\vec{V} = \hat{x} + \hat{y}$ and periodic boundary conditions. The initial condition is a Gaussian,

$$u_0(x, y) = e^{-4(x^2+y^2)}. \tag{5.5.10}$$

Listing 5.5.1 shows an implementation of the first-order finite volume method for this problem, using an upwind flux and forward Euler time stepping. Note that the CFL definition

for this problem is

$$\mathrm{CFL} = \Delta t \left(\frac{V_x}{\Delta x} + \frac{V_y}{\Delta y} \right) \tag{5.5.11}$$

and we use this definition to set the time step for a prescribed CFL.

Listing 5.5.1: Finite volume method for a 2D scalar advection problem.

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

def plotstate(X, Y, U, fname):
    # Average U to nodes
    N = U.shape[0]; UN = np.zeros([N+1,N+1])
    for iy in range(N+1):
        for ix in range(N+1):
            UN[ix,iy] = 0.25*(U[ix%N,iy%N] + U[ix-1,iy%N] + U[ix%N,iy-1] + U[ix-1,iy-1])
    # Make contour plot
    dosave = not not fname
    f = plt.figure( figsize =(6,6))
    plt.contourf(X,Y,UN, 10)
    plt.xlabel(r'$x$', fontsize=16); plt.ylabel(r'$y$', fontsize=16)
    plt.figure(f.number); plt.grid()
    plt.tick_params(axis='both', labelsize=12)
    f.tight_layout (); plt.show(block=(not dosave))
    if dosave: plt.savefig(fname)
    plt.close(f)

def uexact(X,Y):
    return np.exp(-4.*X*X - 4.*Y*Y)

def flux(uL, uR, Vn):
    return 0.5*Vn*(uL+uR) - 0.5*abs(Vn)*(uR-uL)

def advect(N, CFL):
    # Mesh and initial condition
    s = np.linspace(-2,2,N+1); dx = dy = s[1]-s[0]; A = dx*dy
    [Y,X] = np.meshgrid(s,s)
    sm = 0.5*(s[0:N] + s[1:N+1])
    [Ym,Xm] = np.meshgrid(sm,sm)
    U = uexact(Xm,Ym)
    R = np.zeros(U.shape)

    # Velocity and time stepping
    T = 4.; V = np.array([1.,1.])
    dt = CFL/(abs(V[0])/dx + abs(V[1])/dy)
    Nt = int(np.ceil(T/dt)); dt = T/Nt

    # loop over time steps
    for n in range(Nt):

```

```
44          # zero out residual
45          R *= 0.
46
47          # fluxes on vertical faces -> Residuals
48          Vn = np.dot(V,[1.,0.])
49          for ix in range(N):
50              for iy in range(N):
51                  F = flux(U[ix-1,iy],U[ix,iy], Vn)
52                  R[ix-1,iy] += F*dy; R[ix,iy] -= F*dy
53
54          # fluxes on horizontal faces
55          Vn = np.dot(V,[0.,1.])
56          for iy in range(N):
57              for ix in range(N):
58                  F = flux(U[ix,iy-1],U[ix,iy], Vn)
59                  R[ix,iy-1] += F*dx; R[ix,iy] -= F*dx
60
61          # Forward Euler time step
62          U = U - (dt/A)*R
63
64      return X, Y, U
65
66  def main():
67      X,Y,U = advect(32,1.0)
68      plotstate(X, Y, U, '')
69
70  if __name__ == "__main__":
71      main()
```

Listing 5.5.2: Driver code for the functions in `conv2d`.

```
1   import numpy as np
2   import matplotlib.pyplot as plt
3   from conv2d import advect,uexact,plotstate
4
5   def main():
6       Nref = 6
7       Ev = np.zeros(Nref); hv = np.zeros(Nref)
8       N = 8
9       for i in range(Nref):
10          X,Y,U = advect(N,1.0)
11          dx = X[1,0] - X[0,0]; dy = Y[0,1] - Y[0,0]
12          Xm = 0.5*(X[0:N,0:N] + X[1:N+1,0:N])
13          Ym = 0.5*(Y[0:N,0:N] + Y[0:N,1:N+1])
14          Ue = uexact(Xm,Ym);
15          Ev[i] = np.sqrt(dx*dy*np.sum(np.sum((U-Ue)**2)))
16          hv[i] = np.sqrt(dx*dy)
17          N *= 2;
18      plotstate(X, Y, U, '../figs/conv2dN%d'%(N/2))
19      f = plt.figure(figsize=(6,6))
20      rate = np.log2(Ev[Nref-2]/Ev[Nref-1])
```

```
21      plt.loglog(hv, Ev, 'o-', linewidth=2, color='blue', label='rate = %.2f'%(rate))
22      plt.xlabel(r'mesh spacing, $h \equiv \sqrt{\Delta x \Delta y}$', fontsize=16)
23      plt.ylabel(r'$L_2$ error norm', fontsize=16)
24      plt.legend( fontsize=16, borderaxespad=0.1, loc=2)
25      plt.figure(f.number); plt.grid()
26      plt.tick_params(axis='both', labelsize=12)
27      f.tight_layout(); plt.show(block=False)
28      plt.savefig('../figs/conv2dconv.pdf')
29      plt.close(f)
30
31  if __name__ == "__main__":
32      main()
```

The driver code in Listing 5.5.2 runs multiple simulations at various N (grid size) and keeps track of the L_2 error norm on each grid after advecting one period (i.e. the concentration returns back to the starting position). The L_2 error norm is defined as an integral over the domain, which in terms of the cell averages reads

$$e_{L_2} = \sqrt{\sum_{i=1}^{N}\sum_{j=1}^{N}(u(i,j) - u_{\text{exact}}(i,j))^2 \Delta x \Delta y}, \tag{5.5.12}$$

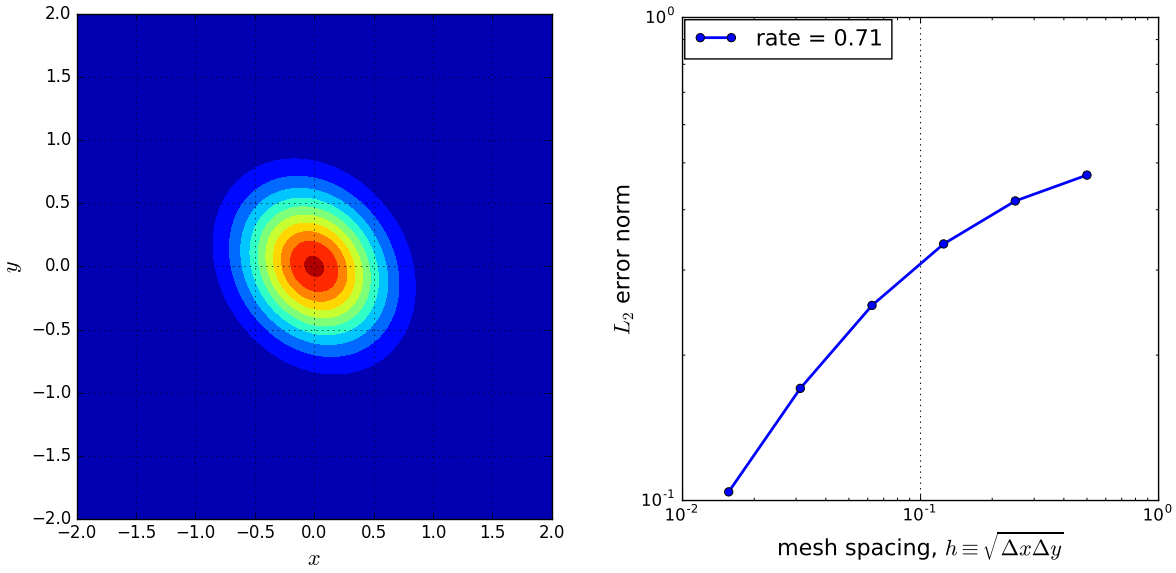
where $\Delta x = \Delta y$ is the square size. Figure 5.5.5 shows a solution on the fine grid and the convergence of the L_2 error. We see that the L_2 error rate is not the expected 1 between the finest grids , but the convergence is not yet asymptotic (the slope on the log-log plot is still increasing at the finest grids). Running even finer grids will eventually yield a convergence rate of 1.

5.6 Euler Fluxes and Boundary Conditions

The finite volume method extends naturally to systems of nonlinear conservation laws, such as the Euler equations of gas dynamics. The key differences compared to our linear scalar equation is in the definition of the flux function at cell interfaces, and in the wider range of possible boundary conditions .

5.6.1 Riemann Solvers

We saw in the scalar case that the flux function must rely on the "upwind" state in order to faithfully model the convection physics. For hyperbolic systems of conservation laws, we must extend this idea to multiple waves: in effect, we have to calculate a flux that for each wave uses the correct upwind state.



(a) Solution for $N = 256$

(b) Convergence rate

Figure 5.5.5: Fine-mesh solution and convergence plot of the L_2 error for the first-order finite volume method applied to a 2D scalar advection problem.

A relatively simple flux function for systems is the **local Lax-Friedrichs**, or **Rusanov** flux. Given two states $\mathbf{u}_L$ and $\mathbf{u}_R$, the flux function is

$$\hat{\mathbf{F}} = \frac{1}{2}\left(\mathbf{F}_L + \mathbf{F}_R\right) - \frac{1}{2} s_{\max}\left(\mathbf{u}_R - \mathbf{u}_L\right), \tag{5.6.1}$$

where $\mathbf{F}_L = \mathbf{F}(\mathbf{u}_L)$, $\mathbf{F}_R = \mathbf{F}(\mathbf{u}_R)$, and $s_{\max}$ is the maximum speed of any wave in the system, considering both $\mathbf{u}_L$ and $\mathbf{u}_R$. The goal of using $s_{\max}$ is to upwind every wave in the system, although for systems with different wave speeds, this flux will upwind too much thereby creating a significant amount of numerical diffusion.

An alternative flux that more carefully upwinds waves one by one is the **Roe** flux, given by

$$\hat{\mathbf{F}} = \frac{1}{2}\left(\mathbf{F}_L + \mathbf{F}_R\right) - \frac{1}{2}\left|\frac{\partial \mathbf{F}}{\partial \mathbf{u}}(\mathbf{u}^*)\right|\left(\mathbf{u}_R - \mathbf{u}_L\right), \tag{5.6.2}$$

where $\left|\frac{\partial \mathbf{F}}{\partial \mathbf{u}}(\mathbf{u}^*)\right|$ refers to taking the absolute values of the eigenvalues, i.e. $\mathbf{R}|\mathbf{\Lambda}|\mathbf{L}$, in the eigenvalue decomposition (see Equation 1.2.3). $\mathbf{u}^*$ is an appropriately-chosen intermediate state based on $\mathbf{u}_L$ and $\mathbf{u}_R$. This choice is only important for nonlinear problems, and the Roe flux uses the Roe-average state, a choice that yields exact single-wave solutions to the

Riemann problem. Specifically, the Roe flux for the Euler equations is

$$\hat{\mathbf{F}} = \frac{1}{2}(\mathbf{F}_L + \mathbf{F}_R) - \frac{1}{2}\begin{bmatrix} |\lambda|_3\Delta\rho & +C_1 & \\ |\lambda|_3\Delta(\rho\vec{v}) & +C_1\vec{v} & +C_2\vec{n} \\ |\lambda|_3\Delta(\rho E) & +C_1 H & +C_2 u \end{bmatrix}$$

where

$$C_1 = \frac{G_1}{c^2}(s_1 - |\lambda|_3) + \frac{G_2}{c}s_2, \qquad C_2 = \frac{G_1}{c}s_2 + (s_1 - |\lambda|_3)G_2$$
$$G_1 = (\gamma - 1)\left(\frac{q^2}{2}\Delta\rho - \vec{v}\cdot\Delta(\rho\vec{v}) + \Delta(\rho E)\right), \qquad G_2 = -u\Delta\rho + \Delta(\rho\vec{v})\cdot\vec{n}$$
$$s_1 = \frac{1}{2}(|\lambda|_1 + |\lambda|_2), \qquad s_2 = \frac{1}{2}(|\lambda|_1 - |\lambda|_2)$$
$$\Delta\mathbf{u} = \mathbf{u}_R - \mathbf{u}_L,$$

and $u = \vec{v}\cdot\vec{n}$. Note, $\vec{v}$ can be defined in any coordinate system; in this general case, $\mathbf{F}_L = \vec{\mathbf{F}}(\mathbf{u}_L)\cdot\vec{n}$ and $\mathbf{F}_R = \vec{\mathbf{F}}(\mathbf{u}_R)\cdot\vec{n}$. The above equations are written in a coordinate-system independent form, which means that in practice, no coordinate system rotation needs to be performed to calculate $\hat{\mathbf{F}}$.

To prevent expansion shocks, an *entropy fix* is needed. One simple choice is to keep all eigenvalues away from zero,

$$\text{if } |\lambda|_i < \epsilon \text{ then } \lambda_i = \frac{\epsilon^2 + \lambda_i^2}{2\epsilon}, \quad \text{for all } i \in [1,4],$$

where ϵ is a small fraction of the Roe-averaged speed of sound, e.g. $\epsilon = 0.05c$. The maximum eigenvalue can be used as the maximum wave speed for time step calculations.

A more dissipative flux than Roe (but not as dissipative as Rusanov) is the **HLLE** flux,

$$\hat{\mathbf{F}} = \frac{1}{2}(\mathbf{F}_L + \mathbf{F}_R) - \frac{1}{2}\frac{s_{\max} + s_{\min}}{s_{\max} - s_{\min}}(\mathbf{F}_R - \mathbf{F}_L) + \frac{s_{\max}s_{\min}}{s_{\max} - s_{\min}}(\mathbf{u}_R - \mathbf{u}_L)$$

where

$$s_{\min} = \min(s_{L,\min}, s_{R,\min})$$
$$s_{\max} = \max(s_{L,\max}, s_{R,\max})$$
$$s_{L,\min} = \min(0, u_L - c_L), \qquad s_{L,\max} = \max(0, u_L + c_L)$$
$$s_{R,\min} = \min(0, u_R - c_R), \qquad s_{R,\max} = \max(0, u_R + c_R)$$

where c_L, c_R are the sound speeds. The maximum wave speed for time step calculations can be taken as $\max(|u_L| + c_L, |u_R| + c_R)$. Note that $u_L = \vec{v}_L\cdot\vec{n}$ and $u_R = \vec{v}_R\cdot\vec{n}$. One advantage of this flux is its simplicity. A disadvantage is that is smears out contact discontinuities.

5.6.2 Boundary Conditions

In the finite-volume method, boundary conditions are generally imposed through fluxes on the boundary faces. These fluxes may depend on data specified on the boundaries and on the state inside the computational domain. For the Euler equations, both are often used, with the amount of information required on the boundary dependent on the type of boundary condition and on the flow state.

Figure 5.6.1 shows the key quantities used in setting a boundary condition. These include:

$$\begin{array}{lcl} \mathbf{u}^+ & = & \text{state inside the computational domain} \\ \mathbf{u}^b & = & \text{state on the boundary, just outside the domain} \\ \vec{n} & = & \text{normal vector pointing out of the domain} \\ \hat{\mathbf{F}}^b & = & \text{numerical flux in direction of } \vec{n} \text{ on boundary} \end{array}$$

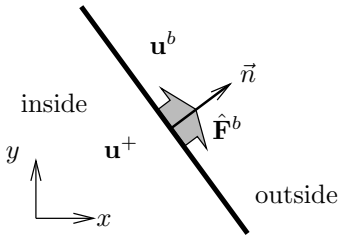


Figure 5.6.1: Interior/boundary state and boundary flux.

We now review some of the commonly used boundary conditions for the Euler equations.

Full State

In this boundary condition, the entire boundary state, $\mathbf{u}^b$, is specified by the user. This does not necessarily mean that the entire state is *necessary* for specifying the boundary condition. For example a full state may be specified at a subsonic inflow, though physically there will be one characteristic (acoustic wave) leaving the domain. The full state (e.g. free-stream condition) will therefore generally be an over-specification of the boundary condition. To account for this, a numerical flux function is used to compute the boundary flux,

$$\hat{\mathbf{F}}^b = \hat{\mathbf{F}}(\mathbf{u}^+, \mathbf{u}^b, \vec{n}). \tag{5.6.3}$$

The choice of flux function is not arbitrary: it must be sufficiently accurate to properly upwind the waves of different speeds: e.g. the Roe flux. The full state boundary condition is typically used at farfield boundaries, where the state is approximately uniform but where the distinction between inflow and outflow may be cumbersome to make.

Inviscid Wall

This boundary condition is used to represent a solid object through which the flow cannot pass (no flow through). It is not a no-slip boundary condition, as such a requirement is only suitable for viscous flows. The boundary flux on the wall is given by

$$\hat{\mathbf{F}}^b = [0, p^b n_x, p^b n_y, 0]^T, \qquad p^b = (\gamma - 1)\left[\rho E^+ - \frac{1}{2}\rho^+ |\vec{v}^b|^2\right], \tag{5.6.4}$$

where $\vec{n} = n_x\hat{x} + n_y\hat{y}$, p^b represents the boundary pressure, and $\vec{v}^b$ is the boundary velocity. The boundary velocity is tangential: the interior velocity with the wall-normal component taken out,

$$\vec{v}^b = \vec{v}^+ - (\vec{v}^+ \cdot \vec{n})\vec{n}. \tag{5.6.5}$$

Note p^b is calculated based on the interior density, energy, and boundary (tangential) velocity.

Inflow T_t, p_t, α

This is an inflow boundary condition for which the flow angle α and two total stagnation quantities are specified: total temperature T_t and total pressure p_t. The boundary flux $\hat{\mathbf{F}}^b$ is determined by constructing the boundary state $\mathbf{u}^b$ from the interior state $\mathbf{u}^+$ – in particular the Riemann invariant J^+ – and the specified quantities. J^+ contains all the information from the interior that is used in constructing the exterior state. It is defined as

$$J^+ = u_n^+ + \frac{2c^+}{(\gamma - 1)}.$$

where $u_n^+ = \vec{v}^+ \cdot \vec{n}$ is the wall-normal velocity component from the interior. The inflow Mach number, M^b, is calculated from J^+ and the specified parameters by solving

$$\left(\gamma R T_t d_n^2 - \frac{\gamma - 1}{2}(J^+)^2\right)(M^b)^2 + \left(\frac{4\gamma R T_t d_n}{\gamma - 1}\right) M^b + \frac{4\gamma R T_t}{(\gamma - 1)^2} - (J^+)^2 = 0,$$

where $d_n = \vec{n}_{\text{in}} \cdot \vec{n}$ and $\vec{n}_{\text{in}} = [\cos(\alpha), \sin(\alpha)]$ is the specified inflow direction. This quadratic is obtained by expressing the boundary speed of sound $c^b = \sqrt{\gamma R T^b}$ in terms of the boundary Mach number M^b and the given total temperature T_t via an isentropic relation,

$$T_t = \left(1 + 0.5(\gamma - 1)(M^b)^2\right) T^b \quad \Rightarrow \quad (c^b)^2 = \frac{\gamma R T_t}{(1 + 0.5(\gamma - 1)(M^b)^2)}.$$

This expression is then substituted for c^b into the definition of J^+,

$$J^+ = M^b c^b d_n + \frac{2c^b}{\gamma - 1}. \tag{5.6.6}$$

The physically relevant solution ($M^b \geq 0$) to this quadratic is used – if both are greater than zero, then the smaller Mach number is used. Using M and the specified stagnation quantities, the exterior state is calculated as follows:

$$\begin{aligned}
\text{exterior static temperature:} \quad T^b &= T_t/[1 + 0.5(\gamma - 1)(M^b)^2] \\
\text{exterior static pressure:} \quad p^b &= p_t(T^b/T_t)^{\gamma/(\gamma-1)} \\
\text{exterior static density:} \quad \rho^b &= p^b/(RT^b) \\
\text{exterior speed of sound:} \quad c^b &= \sqrt{\gamma p^b/\rho^b} \\
\text{exterior velocity:} \quad \vec{v}^b &= M^b c^b \vec{n}_{\text{in}} \\
\text{exterior total energy:} \quad \rho E^b &= p^b/(\gamma - 1) + \frac{1}{2}\rho^b |\vec{v}^b|^2
\end{aligned}$$

The boundary flux is then calculated according to $\hat{\mathbf{F}}^b = \vec{\mathbf{F}}(\mathbf{u}^b) \cdot \vec{n}$.

Inflow ρ_t, c_t, α

In this boundary condition, the total density (ρ_t) and speed of sound (c_t) are given together with the flow angle α. Since $c_t = \sqrt{\gamma p_t/\rho_t}$, we have $p_t = c_t^2 \rho_t/\gamma$ and $T_t = p_t/(\rho_t R)$, and we can use the same procedure as in the previous inflow boundary condition.

Subsonic Outflow

At a subsonic outflow, the boundary static pressure, p^b, is typically specified. The complete exterior state, $\mathbf{u}^b$, is calculated from the Riemann invariant $J^+(\mathbf{U}^+)$, the interior entropy $S^+ = p^+/(\rho^+)^\gamma$, and the interior tangential velocity v^+. The calculation proceeds as follows. First, the exterior density is

$$\rho^b = \left(\frac{p^b}{S^+}\right)^{1/\gamma}. \tag{5.6.7}$$

The boundary normal velocity, u_n^b, is found using J^+ and $c^b = \sqrt{\gamma p^b/\rho^b}$,

$$u_n^b = J^+ - \frac{2c^b}{\gamma - 1}. \tag{5.6.8}$$

Setting the boundary tangential velocity to the interior tangential velocity then fully defines $\vec{v}^b$, according to

$$\vec{v}^b = \vec{v}^+ - (\vec{v}^+ \cdot \vec{n})\vec{n} + (\vec{v}^b \cdot \vec{n})\vec{n}.$$

$(\rho E)^b$ is calculated as $p^b/(\gamma - 1) + \frac{1}{2}\rho^b |\vec{v}^b|^2$. The boundary flux is calculated according to $\hat{\mathbf{F}}^b = \vec{\mathbf{F}}(\mathbf{u}^b) \cdot \vec{n}$.

Supersonic Outflow

The boundary flux is determined based solely on the interior state, $\hat{\mathbf{F}}^b = \vec{\mathbf{F}}(\mathbf{u}^+) \cdot \vec{n}$. No boundary information is required.

Periodic

This is not a true boundary condition. An edge on a periodic BC should be treated as an interior edge with $\mathbf{u}^b$ equal to the state on the corresponding periodic cell.

5.7 High-Order Accuracy

Using cell averages as inputs into the numerical fluxes that we have presented means that we are approximating the solution as constant in each cell. This gives a spatially first-order accurate method, since the discretization error is proportional to the cell size (e.g. diameter) to the first power. We can do better by a high-order extension - in fact the industry standard is a second-order finite-volume method.

In a second-order finite-volume method, we need a linear representation of the solution inside each cell. We cannot construct this using only the cell average. Instead we must look to cell averages on neighboring cells for extra information. For triangular cells, the states on the three adjacent neighbors will give us (more than) enough information to construct a gradient of the state. For example, we can compute the average gradient in cell i, $\nabla u\big|_i$, by using integration by parts,

$$\begin{aligned} \int_{A_i} \nabla \mathbf{u}\, dA &= \int_{\partial A_i} \hat{\mathbf{u}}\vec{n}\, dl \\ \nabla \mathbf{u}\big|_i A_i &= \int_{\partial A_i} \hat{\mathbf{u}}\vec{n}\, dl \\ \Rightarrow \qquad \nabla \mathbf{u}\big|_i &= \frac{1}{A_i}\int_{\partial A_i} \hat{\mathbf{u}}\vec{n}\, dl \end{aligned} \tag{5.7.1}$$

where $\hat{\mathbf{u}}$ is some average state on the boundary of cell i. A simple arithmetic average of the left/right face neighbors will often suffice. We can then use the cell gradient to evaluate the states at each of the edge midpoints using

$$\mathbf{u}(\vec{x}) = \mathbf{u}_i + \nabla \mathbf{u}\big|_i \cdot (\vec{x} - \vec{x}_i), \tag{5.7.2}$$

where $\vec{x}_i$ is the cell centroid and $\vec{x}$ is the location of a desired edge midpoint, as defined in Figure 5.7.1. Performing a similar reconstruction on each cell, one can use the reconstructed solution values, from the left and right cells of a given edge, to compute the fluxes and increment the residuals. The remainder of the algorithm remains the same as in the first-order case. For completeness, we present the second-order finite-volume method in Algorithm 10.

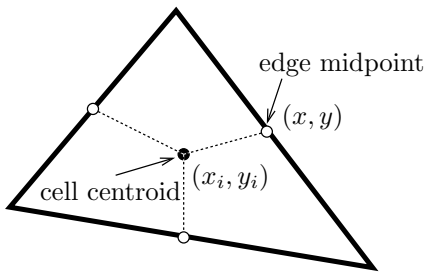


Figure 5.7.1: Centroid and edge midpoints in a triangular mesh cell.

Algorithm 10 The second-order finite-volume method for a steady-state simulation.

Compute/store edge normals, cell areas, etc.
Initialize the state, $\mathbf{U} = \mathbf{U}^0$, e.g. from a first-order solve.
while $|\mathbf{R}| >$ tolerance **do**
    Initialize the gradient, $\nabla u\big|_i$, to zero on each element
    for $f = 1 : N_{\text{face}}$ **do**
        Let L/R be the elements adjacent to face f
        Let $\vec{n}$ be the normal pointing out of L
        Let Δl be the length of face f
        Set $\hat{\mathbf{u}}$ to the average of the L and R cell averages
        Add $\hat{\mathbf{u}}\vec{n}\Delta l$ to $\nabla u\big|_i$ on L and subtract it from R
    end for
    Divide each $\nabla u\big|_i$ by its element area, A_i
    Initialize residual and wave speed on each cell to zero
    for $f = 1 : N_{\text{face}}$ **do**
        Let L/R be the elements adjacent to face f
        Compute the L/R states on the face midpoint using Equation 5.7.2
        Compute the flux on face f using the *edge* states
        Increment residual on L, decrement it on R
        Add wave speed to tallies on L and R cells
    end for
    Compute the time step on each cell
    Update the state: use RK2 or higher-order method
end while

Note that when converging a second-order solution, we generally start with the converged first-order solution as the initial guess. Also, it is useful to encode the residual calculation in a separate function, since the residual must be evaluated more than once per step when using a multi-stage time-marching scheme.

Even higher-order accuracy can be obtained by reconstructing the state at the interfaces using a wider stencil of neighboring cells. In all cases involving reconstruction, shock

capturing techniques are needed to prevent oscillations and nonlinear instabilities.

Chapter 6

Total Variation Diminishing Methods

In this chapter we discuss limiting and total variational diminishing methods, which are required to stabilize certain high-resolution numerical methods in the presence of shocks. To put these methods in context, we begin with a discussion of characteristics and the formation of shocks in nonlinear conservation laws.

6.1 Method of Characteristics

The method of characteristics is an analytical, graphical technique for obtaining solutions to scalar conservation laws. Consider the quasi-linear form in Equation 5.1.5,

$$\frac{\partial u}{\partial t} + a(u)\frac{\partial u}{\partial x} = 0,$$

where recall that $a = \frac{\partial f}{\partial u}$. Suppose we have a sensor that measures u, and whose position is given by $x_{\text{sensor}}(t)$. The instantaneous time rate of change of u measured by the sensor is given by the total derivative,

$$\frac{Du}{Dt} = \frac{\partial u}{\partial t} + \frac{dx_{\text{sensor}}}{dt}\frac{\partial u}{\partial x}. \tag{6.1.1}$$

Note that $\frac{dx_{\text{sensor}}}{dt}$ is the (instantaneous) speed of the sensor. What if we set this to $a(u)$, where u is the state at the sensor location? Then, by Equation 5.1.5,

$$\frac{Du}{Dt} = \frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} = 0. \tag{6.1.2}$$

This means that if the sensor moves at speed $a(u)$, it does not register any change in u. But if u does not change at the sensor's location, neither does $a(u)$, so the sensor speed remains constant! So if at some time t_0 the sensor was at position x_0, where the solution was u_0, the sensor location at later times is

$$x_{\text{sensor}}(t) = x_0 + a(u_0)(t - t_0). \tag{6.1.3}$$

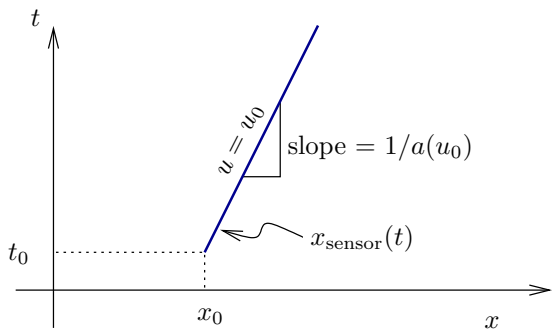


Figure 6.1.1: A sensor moving at speed $a(u_0)$ measures the same value $u = u_0$, and its path is called a characteristic.

This equation represents a *straight line* on a plot of space (horizontal) versus time (vertical), as shown in Figure 6.1.1. Such a line is called a **characteristic**, which is defined as a line along which u does not change.

If a depends explicitly on x, the characteristics are no longer straight, since as the sensor moves in x, the value of a will change. If the problem is linear and a is constant, then all characteristics will have the same slope, independent of the starting x_0, u_0 values. In this case, the parallel characteristics just represent translation of the entire initial condition at speed a (the solution to linear advection). Finally, for a nonlinear problem, when a is a function of u, if the initial condition is not uniform ($u_0(x)$ varies), characteristics emanating from different x_0 will have different slopes. What happens when the characteristics cross? As we discuss next, this corresponds to the formation of a shock.

We now present some examples of using characteristics to obtain solutions.

Linear Advection

The solution to linear advection, Equation 5.1.6, in an infinite or periodic domain is

$$u(x,t) = u_0(x - at),$$

where $u_0(x)$ is the initial condition at $t = t_0$. We can verify this by substituting this form into the PDE and observing that it is identically satisfied, regardless of $u_0(x)$. Figure 6.1.2 shows the characteristics, which are all parallel, and the resulting undisturbed propagation of the initial condition.

Burgers Equation

Consider now Burger's equation, Equation 5.1.9. Given an initial condition at $t = 0$ of $u_0(x)$, on an infinite or periodic domain, the solution at later times is given by

$$u(x,t) = u_0(x - ut). \tag{6.1.4}$$

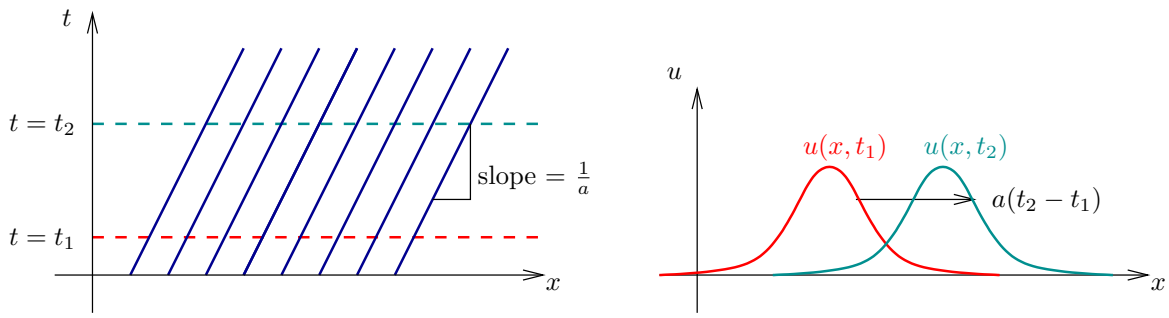


Figure 6.1.2: Characteristics and solution propagation for linear advection.

Note that u appears implicitly in the solution. To verify that this form indeed solves the PDE, we use the chain rule to compute the derivatives,

$$u_t = (-u_t t - u)u_0' \quad \Rightarrow \quad u_t = \frac{-u_0' u}{1 + u_0' t},$$
$$u_x = (1 - u_x t)u_0' \quad \Rightarrow \quad u_x = \frac{u_0'}{1 + u_0' t}.$$

This shows that $u_t = -uu_x$, so that Equation 5.1.9 is identically satisfied. Equation 6.1.4 shows that the solution is constant along straight characteristics defined by $x = x_0 + ut$, and this also means that the solution on a characteristic is $u = u_0(x_0)$.

Example 6.1 (Solution of Burger's equation using the method of characteristics)**.** Suppose that at $t = 0$ the initial condition for Burgers equation is

$$u(x, 0) = u_0(x) = \begin{cases} 1 & x < 0, \\ 1 - x & 0 \le x \le 1, \\ 0 & x > 1. \end{cases} \tag{6.1.5}$$

We can obtain the solution at later times by considering the characteristics emanating from this initial condition. Recall that at (x_0, t_0), the (inverse) slope of the characteristic is $\frac{dx}{dt} = u_0(x_0)$. Figure 6.1.3 shows these characteristic lines. Note that the slope of each characteristic depends on the value of the solution at $t = 0$. We distinguish between three groups of characteristics, based on the three pieces of the u_0 definition in Equation 6.1.5 and these are color-coded in Figure 6.1.3.

We restrict our attention to $t \le 1$, which is before the characteristics cross. In the first region, which lies to the left of the line $x = t$, all characteristics trace back to $x_0 < 0$, for which $u_0 = 1$, so we have

$$\text{Region 1:} \quad x < t, \quad u(x, t) = 1.$$

In the second region, which lies to the right of $x = t$ and to the left of $x = 1$, the characteristics trace back to $0 \le x_0 \le 1$, and the slope of each characteristic is $dx/dt = 1 - x_0$. This means

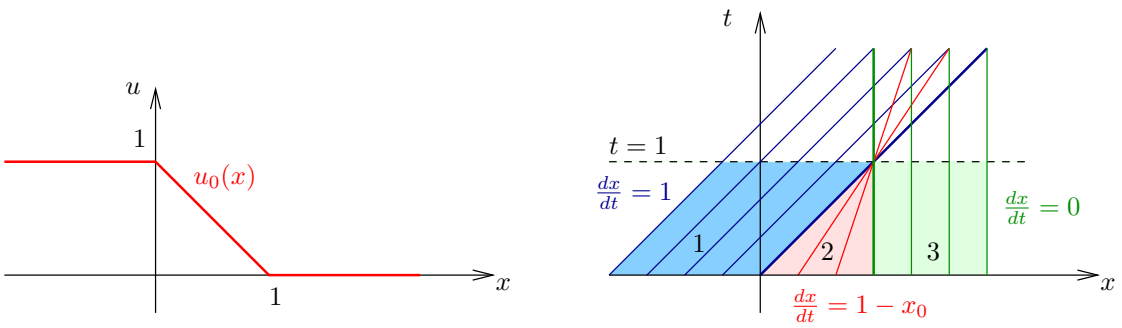


Figure 6.1.3: Initial condition and corresponding characteristics for an example with Burgers equation.

that the characteristic emanating from $0 \le x_0 \le 1$, along which $u = u_0(x_0) = 1 - x_0$, follows the line $x = x_0 + tdx/dt = x_0 + t(1 - x_0)$. We invert this relationship to obtain $x_0 = (x - t)/(1 - t)$, so that the solution in terms of x, t in this region is

$$\text{Region 2:} \quad t \le x \le 1, \quad u(x,t) = 1 - x_0 = 1 - \frac{x - t}{1 - t}.$$

Finally, in the third region, which lies to the right of $x = 1$, the characteristics are vertical since $dx/dt = 0$. These all trace back to the initial condition $u_0 = 0$, so we have

$$\text{Region 3:} \quad 1 < x, \quad u(x,t) = 0.$$

Figure 6.1.4 shows snapshots of this solution at $t = 0, 0.5, 1$. Note that by $t = 1$ a shock forms. For $t > 1$, it would appear from the method of characteristics that the solution is multi-valued at the shock location. However, since the shock is a discontinuity, the differential form of the equation that gave rise to the characteristic solution is no longer valid, since the derivatives (e.g. u_x) no longer exist.

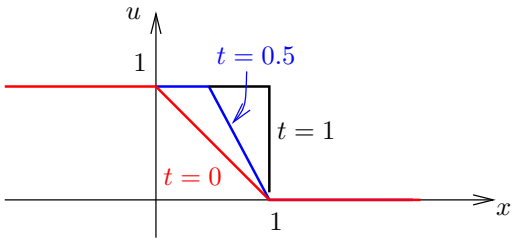


Figure 6.1.4: Solution to the Burgers equation example at $t = 0.5$ and $t = 1$.

6.1.1 Shocks

In the previous section we saw that when characteristics intersect, a shock forms in the solution. A **shock** is a discontinuity in the state, a point at which the differential form of the conservation law no longer holds because the derivatives are not defined. As the method of characteristics is based on this differential form, it can no longer be used once a shock forms.

Is a shock a true discontinuity? For all practical conservation laws, the answer is no. The equations we have encountered thus far are in fact approximations of reality. A key piece of physics missing from these equations as we have written them is *viscosity*, which can be modeled in the conservation equation through a diffusion term, i.e. a second derivative of the state. In many applications, especially aerodynamics, such viscous terms are negligible in most of the domain ... except in isolated regions such as boundary layers and shocks. In these regions, large gradients in the state make the diffusive fluxes non-negligible even when the viscosity is very small. This viscosity then prevents the solution from ever developing true discontinuities. Nevertheless, we often find it convenient to neglect viscosity to simplify our analysis. In this case, consideration of the diffusion term in the limit of vanishing viscosity can provide physical insight into the correct behavior of the solution near shocks.

Since the differential form of the conservation law ceases to be valid at discontinuities, to analyze a solution with shocks, we must revert to the integral form of the equation. We do this first for a prototypical case of a shock formed by discontinuous initial data. This is called a **Riemann problem** and is illustrated in Figure 6.1.5 for a general conservation law. The integral form of the conservation law, Equation 5.1.1, applied to this case with (x_L, x_R)

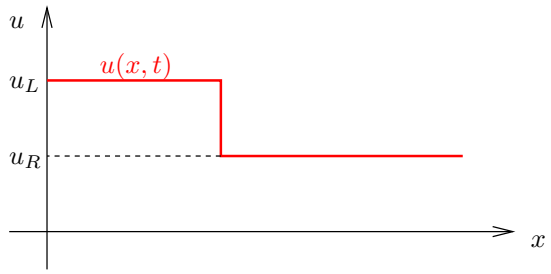


Figure 6.1.5: Setup for the Riemann problem.

straddling the shock gives

$$\frac{d}{dt}\int_{x_L}^{x_R} u\,dx \quad = \quad f(u_L) - f(u_R), \tag{6.1.6}$$

where u_L and u_R are the states to the left and right of the discontinuity. As the system evolves in time, the shock will move. Let $x_s(t)$ be the shock position as a function of time. Figure 6.1.6 shows the shock at two different times, t and $t + \delta t$. In this time, the amount of u inside our control volume changes by $(u_L - u_R)\delta x_s$, so that the time rate of change is

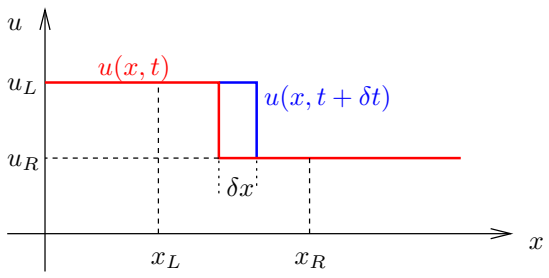


Figure 6.1.6: Evolution of a shock: the Rankine-Hugoniot jump condition determines the shock speed.

$(u_L - u_R)\delta x_s/\delta t$. This is the left-hand-side of Equation 6.1.6, which now gives

$$\begin{aligned}(u_L - u_R)\frac{\delta x_s}{\delta t} &= f(u_L) - f(u_R),\\ s \equiv \frac{\delta x_s}{\delta t} &= \frac{f(u_L) - f(u_R)}{u_L - u_R},\end{aligned} \tag{6.1.7}$$

where we have denoted by s the shock speed $\delta x_s/\delta t$. Equation 6.1.7 is known as the **Rankine-Hugoniot** (RH) jump condition. It states that the shock speed is determined by the states on either side of the shock.

Example 6.2 (Shock speed in Burgers equation)**.** Recall the Burgers equation example in the previous section. The method of characteristics lets us get the solution up until $t = 1$, at which time a shock forms. After $t = 1$, the formed shock moves at speed s, which we can calculate from the RH jump condition in Equation 6.1.7,

$$s = \frac{f(u_L) - f(u_R)}{u_L - u_R} = \frac{\frac{1}{2}u_L^2 - \frac{1}{2}u_R^2}{u_L - u_R} = \frac{1}{2}(u_L + u_R).$$

Since in this case $u_L = 1$ and $u_R = 0$, the shock speed is $s = \frac{1}{2}$. Figure 6.1.7 shows a modified version of the characteristics, with the shock for $t > 1$.

Example 6.3 (Variable shock speed)**.** The shock path is not necessarily a straight line in the x, t plane. Consider Burgers equation with a discontinuous initial condition,

$$u(x, 0) = u_0(x) = \begin{cases} 0 & x < 0,\\ x & 0 \le x \le 1,\\ 0 & x > 1.\end{cases} \tag{6.1.8}$$

The initial shock at $x = 1$ moves to the right for $t > 0$, but its speed is not constant because the states on either side of the shock, u_L and u_R, are not constant with t.

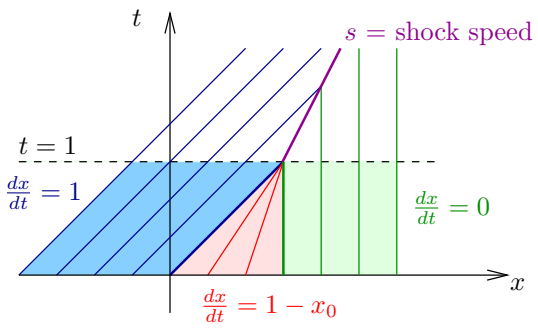


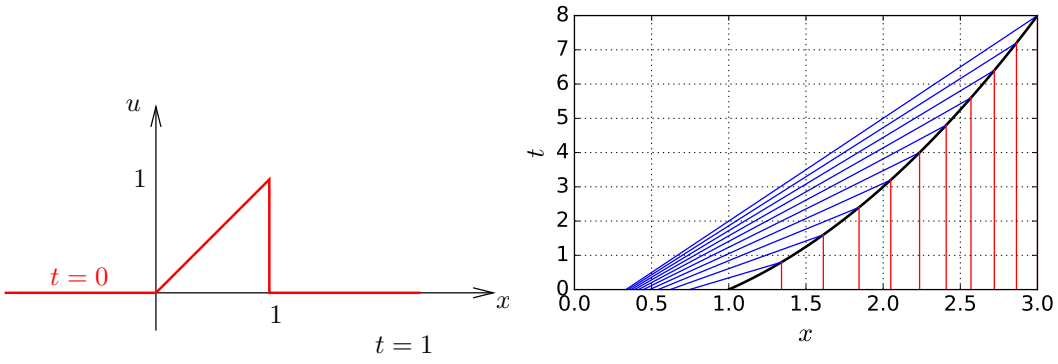
Figure 6.1.7: Characteristics for Burgers example with a shock.

On the left side of the shock, the characteristics emanate from the region $0 \le x \le 1$. In this region, the slope of the characteristics is $dx/dt = x_0$, so that the characteristics follow the line $x = x_0 t + x_0$. The state on one of these characteristics is $u_L = u = x_0 = x/(1+t)$.

On the right side of the shock, we have vertical characteristics bringing in the state $u_R = 0$. So the shock speed, as a function of x, t is

$$s = \frac{dx_s}{dt} = \frac{\frac{1}{2}u_L^2 - \frac{1}{2}u_R^2}{u_L - u_R} = \frac{1}{2}(u_L + u_R) = \frac{x_s}{2(1+t)}.$$

Solving this differential equation, subject to $x_s(0) = 1$, gives the position of the shock as $x_s = \sqrt{1+t}$.



(a) Initial condition (b) Characteristics

Figure 6.1.8: Variable shock speed for discontinuous initial data in Burgers equation.

Listing 6.1.1: Code for plotting characteristics and shock position for Burgers equation with discontinuous initial data.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
```

```
3
4  def printfig(f, fname):
5      plt.figure(f.number); plt.grid()
6      plt.tick_params(axis='both', labelsize=16)
7      f.tight_layout(); plt.show(block=False); plt.savefig(fname)
8
9  def main():
10     t = np.linspace(0,8,51)
11     xs = np.sqrt(1+t)
12     f = plt.figure( figsize=(6,4))
13     plt.plot(xs,t, '-', linewidth=2, color='black')
14     t = np.linspace(0,8,11)
15     xs = np.sqrt(1+t)
16     x0 = xs/(1.+t)
17     for k in range(t.size):
18         plt.plot([x0[k], xs[k]], [0, t[k]], '-', linewidth=1, color='blue')
19         plt.plot([xs[k], xs[k]], [0, t[k]], '-', linewidth=1, color='red')
20     plt.xlabel(r'$x$', fontsize=20)
21     plt.ylabel(r'$t$', fontsize=20)
22     printfig(f, '../figs/RH-variable.pdf')
23     plt.close(f)
24
25 if __name__ == "__main__":
26     main()
```

Note that the conservation law must be in unaltered form to get the correct shock speed. For example, Burgers equation reads

$$u_t + \left(\frac{1}{2}u^2\right)_x = 0, \tag{6.1.9}$$

and we have already seen that the shock speed in a Riemann problem is $s = (u_L + u_R)/2$. If we multiply Burgers equation by u, we obtain

$$\begin{aligned} \left(\frac{1}{2}u^2\right)_t + \left(\frac{1}{3}u^3\right)_x &= 0, \\ v_t + \left(\frac{\sqrt{8}}{3}v^{3/2}\right)_x &= 0, \end{aligned} \tag{6.1.10}$$

where $v = \frac{1}{2}u^2$. Note that this equation is in the form of a conservation law for a different variable, v instead of u. Applying the Rankine-Hugoniot jump condition, Equation 6.1.7, we have

$$s = \frac{f(v_L) - f(v_R)}{v_L - v_R} = \frac{\sqrt{8}}{3}\frac{v_L^{3/2} - v_R^{3/2}}{v_L - v_R} = \frac{2}{3}\frac{u_L^3 - u_R^3}{u_L^2 - u_R^2} \neq \frac{1}{2}(u_L + u_R).$$

So the shock speed in the altered conservation law is not the same as in the original one. The reason for the discrepancy comes from the fact that once we multiply the equation by u, we are working with a different conservation law. In this case, rather than conserving u, we instead conserve $v = \frac{1}{2}u^2$. Although the differential forms of the two equation are equivalent, so that smooth solutions remain the same, the integral forms are different, which accounts for the difference in solutions in the presence of a shock. Therefore, to obtain correct shock speeds, we must use conservation laws for quantities that are actually physically conserved.

6.2 Weak Solutions

In the previous section we saw that the differential form of the PDE is not valid within a shock, and we turned to the integral form in order to compute the solution behavior in the presence of a shock, i.e. the shock speed. More desirable than two equations would be one equation that we could use for both smooth and discontinuous solutions. The weak form of the PDE is such an equation.

A **weak form** of a PDE is a set of integral equations obtained by weighting the PDE with test functions from a certain space and integrating by parts. Denote the test functions by

$$\phi(x,t) \in \mathcal{C}_0^1\left(\mathbb{R} \times \mathbb{R}_+\right), \tag{6.2.1}$$

where $\mathcal{C}_0^1$ is the space of functions that are continuous, have continuous first derivative, and have compact support (vanish at infinity). $\mathbb{R}$ is the set of all real numbers while $\mathbb{R}_+$ is the set of all positive real numbers. Multiplying the differential form of the conservation law, $u_t + f_x = 0$ by ϕ and integrating over all of space and time gives

$$\begin{aligned}
\int_0^\infty \int_{-\infty}^\infty \phi\left[u_t + f_x\right]\, dx\, dt &= 0,\\
\int_{-\infty}^\infty \int_0^\infty \phi u_t\, dt\, dx + \int_0^\infty \int_{-\infty}^\infty \phi f_x\, dx\, dt &= 0,\\
\int_{-\infty}^\infty \left[-\int_0^\infty \phi_t u\, dt + \phi u\Big|_0^\infty\right] dx + \int_0^\infty \left[-\int_{-\infty}^\infty \phi_x f\, dx + \phi f\Big|_{-\infty}^\infty\right] dt &= 0,\\
-\int_{-\infty}^\infty \int_0^\infty \phi_t u\, dt\, dx - \int_{-\infty}^\infty \phi u_0(x)\, dx - \int_0^\infty \int_{-\infty}^\infty \phi_x f\, dx\, dt &= 0,\\
\int_{-\infty}^\infty \int_0^\infty \left[\phi_t u + \phi_x f\right]\, dt\, dx + \int_{-\infty}^\infty \phi(x,0) u_0(x)\, dx &= 0.
\end{aligned} \tag{6.2.2}$$

Note that we have integrated by parts separately in space and time and used the fact that $\phi(x,t)$ vanishes for $x \to \infty$, $x \to -\infty$, and $t \to \infty$. Equation 6.2.2 is the weak form of the PDE, and a $u(x,t)$ that satisfies it is called a **weak solution**. These solutions satisfy the PDE where the solution is smooth and the integral jump conditions where the solution is discontinuous. Note that there are no derivatives on u or f in Equation 6.2.2, which means

that there is no ambiguity in the equation at discontinuities. The downside, however, is that weak solutions are *not unique*.

Example 6.4 (Non-unique weak solution to Burgers equation)**.** Consider Burgers equation with the initial condition $u_0(x) = -1$ for $x < 0$ and $u_0(x) = 1$ for $x \geq 0$, as shown in Figure 6.2.1. One possible weak solution is that the state remains at the initial condition for

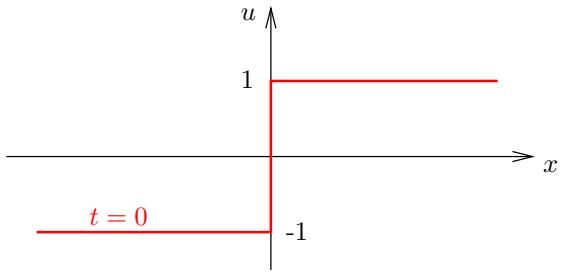


Figure 6.2.1: Initial condition for demonstrating a non-unique weak solution.

all time, $u(x,t) = u_0(x)$. We can verify this by substituting $u_0(x)$ for u in Equation 6.2.2, which results in (note $f = \frac{1}{2}u^2 = \frac{1}{2}$ in this case),

$$\begin{aligned}
\int_{-\infty}^{\infty}\int_{0}^{\infty}\left[\phi_t u_0 + \frac{1}{2}\phi_x\right]\,dt\,dx + \int_{-\infty}^{\infty}\phi(x,0)u_0\,dx &= \\
\int_{-\infty}^{\infty}\left[\phi u_0\right]_{t=0}^{t=\infty}\,dx + \frac{1}{2}\int_{0}^{\infty}\left[\phi\right]_{x=-\infty}^{x=\infty}\,dt + \int_{-\infty}^{\infty}\phi(x,0)u_0\,dx &= \\
-\int_{-\infty}^{\infty}\phi(x,0)u_0\,dx + \frac{1}{2}\int_{0}^{\infty}0\,dt + \int_{-\infty}^{\infty}\phi(x,0)u_0\,dx &= 0.
\end{aligned}$$

In the last step we see that the expression reduces identically to zero, regardless of $\phi(x,t)$, which means that $u(x,t) = u_0(x)$ is a weak solution.

However, another possible solution is a **rarefaction wave**, shown in Figure 6.2.2. One can verify this by substituting this $u(x,t)$ into Equation 6.2.2 and showing that the result is identically zero. In fact, for this problem there are an infinite number of solutions, since we could have $u = u_0$ for part of the time and then the rarefaction solution.

As weak solutions can be non-unique, how do we know which solution is physical? One option is to return to the viscous equation and take the limit of vanishing viscosity. That is, the physical solution satisfies

$$\lim_{\epsilon \to 0}\left(u_t + f_x = \epsilon u_{xx}\right). \tag{6.2.3}$$

However, solving such a system is not always easy, and there exist simpler techniques: conditions that we could test for a given weak solution to identify it as physical.

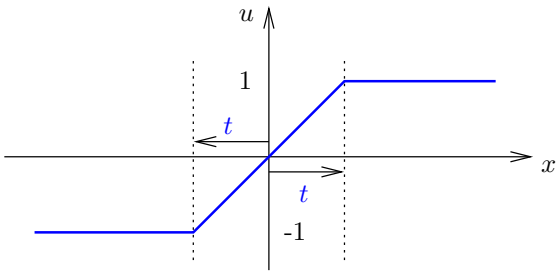


Figure 6.2.2: A rarefaction is also a weak solution of Burgers equation with the initial condition in Figure 6.2.1.

For a convex or concave flux function $f(u)$, a sufficient condition for the solution to be physical is that *characteristics run into the shock*, not out of it. Figure 6.2.3 illustrates this schematically, and the mathematical condition is

$$f'(u_L) > s > f'(u_R), \tag{6.2.4}$$

where $f'(u_L)$, $f'(u_R)$ are the characteristic speeds on the left/right, and s is the shock speed.

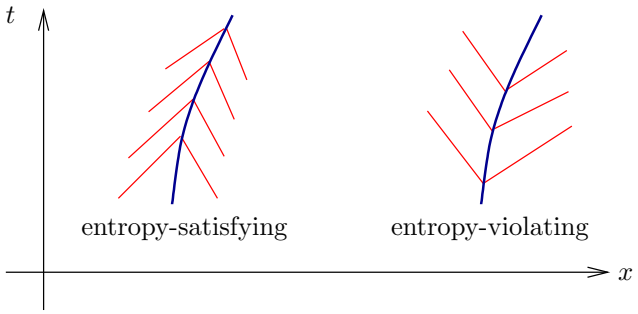


Figure 6.2.3: A physical solution for a concave or convex flux function must exhibit characteristics that run into the shock (entropy-satisfying) not out of it (entropy-violating).

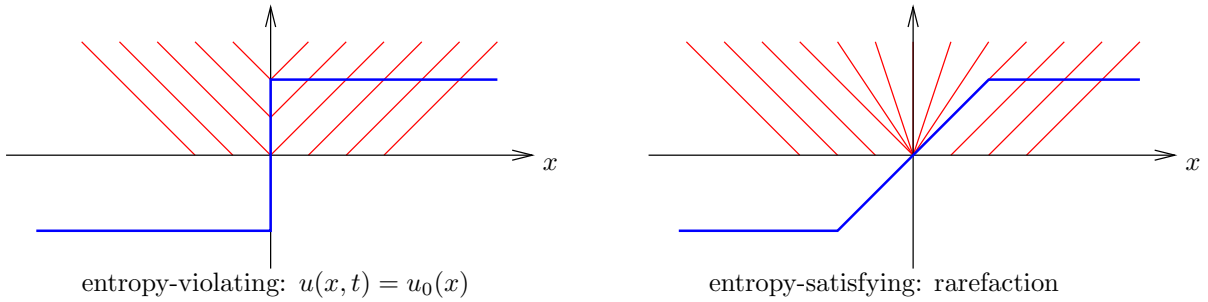
Returning to Example 6.4, we see that the case of $u(x,t) = u_0$, a stationary discontinuity violates this condition, whereas the rarefaction does not. Figure 6.2.4 shows the characteristics in each case.

For general flux functions, we have **Oleinik's condition**, which states

$$\frac{f(u) - f(u_L)}{u - u_L} \geq s \geq \frac{f(u) - f(u_R)}{u - u_R}, \quad \forall u \in [u_L, u_R]. \tag{6.2.5}$$

Alternatively, an even more general condition that extends to systems of equation is based on an **entropy function**. This is a positive, convex function, $\eta(u)$, that comes with a corresponding **entropy flux**, $\psi(u)$, which satisfies

$$\psi'(u) = \eta'(u) f'(u). \tag{6.2.6}$$



entropy-violating: $u(x,t) = u_0(x)$ entropy-satisfying: rarefaction

Figure 6.2.4: A physical solution for a concave or convex flux function must exhibit characteristics that run into the shock (entropy-satisfying) not out of it (entropy-violating).

This equation ensures that for smooth solutions, the entropy function and flux pair satisfy

$$\frac{\partial \eta}{\partial t} + \frac{\partial \psi}{\partial x} = 0. \tag{6.2.7}$$

In the presence of shocks, this equality turns into an inequality, and we have the condition that $u(x,t)$ is an entropy-satisfying (physical) solution of the PDE if, for all entropy function/flux pairs, the inequality

$$\frac{\partial \eta}{\partial t} + \frac{\partial \psi}{\partial x} \leq 0 \tag{6.2.8}$$

is satisfied in the weak sense. When $f(u)$ is a convex flux, we only need to check one entropy function $\eta(u)$.

Example 6.5 (Entropy function for Burgers equation)**.** Consider Burgers equation, $u_t + f_x = 0$, with $f = \frac{1}{2}u^2$. One entropy function for this equation is $\eta(u) = \frac{1}{2}u^2$, and the corresponding entropy flux is $\psi(u) = \frac{1}{3}u^3$. The entropy inequality states that

$$\frac{\partial(\frac{1}{2}u^2)}{\partial t} + \frac{\partial}{\partial x}\left(\frac{1}{3}u^3\right) \leq 0$$

must be satisfied in a weak sense. With the equality, this is equivalent to Equation 6.1.10, which we obtained from the differential form and which is valid for smooth solutions. However, for non-smooth solutions, the inequality above ensures that we pick off the entropy satisfying solution. One can verify that the entropy-violating solution in Example 6.4 does not satisfy the above inequality in a weak sense.

Example 6.6 (Traffic flow solutions)**.** Recall the traffic flow equation, $\rho_t + f_x = 0$, with $f(\rho)$ as given in Equation 5.1.10,

$$f(\rho) \quad = \quad \rho u_{\max}\left(1 - \frac{\rho}{\rho_{\max}}\right),$$

Consider a discontinuous state: a shock with density ρ_L on the left side and ρ_R on the right side. The shock speed is given by Equation 6.1.7,

$$s = \frac{f(u_L) - f(u_R)}{u_L - u_R} = u_{\max}\left(1 - \frac{\rho_L + \rho_R}{\rho_{\max}}\right). \tag{6.2.9}$$

Suppose that there is a traffic jam on the road, with $\rho_R = \rho_{\max}$. As cars approach the traffic jam from the left, they go through a shock (slam on the brakes!), and the shock (start of the jam) moves to the left. The above equation tells us that the shock speed is

$$s = -u_{\max}\left(\frac{\rho_L}{\rho_{\max}}\right) < 0.$$

Figure 6.2.5 illustrates this initial condition and the corresponding characteristics. Note that

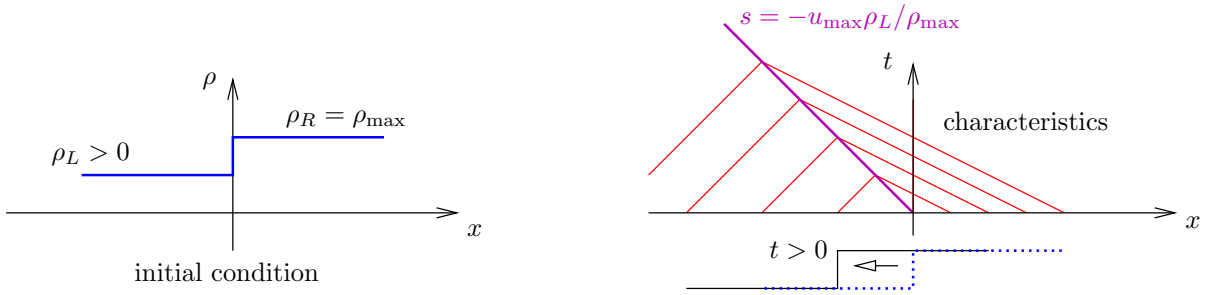


Figure 6.2.5: Solution and characteristics for a traffic jam.

the slope of the characteristics is calculated from the derivative of the flux,

$$a = f'(\rho) = u_{\max}\left(1 - \frac{2\rho}{\rho_{\max}}\right),$$

so that for the region in which $\rho = \rho_R = \rho_{\max}$, the slope is negative and the characteristics run to the left. It is important to distinguish this speed, which is the speed of a wave/disturbance in the density, from the speed of the vehicles ... which are not moving when $\rho = \rho_{\max}$!

Another prototypical traffic flow problem is that of cars stopped at an intersection, reacting to a light that just turned green. The initial condition consists of $\rho_L = \rho_{\max}$ and $\rho_R = 0$. After the light turns green, what happens to the cars? The physical solution is a *rarefaction*: the cars in front accelerate and the ones behind them gradually follow. Another solution, however, is that the cars remain in their places: the Rankine-Hugoniot jump condition is satisfied, but the characteristics run out of the preserved discontinuity (shock), so that the solution violates the entropy condition. Figure 6.2.6 illustrates both of these solutions.

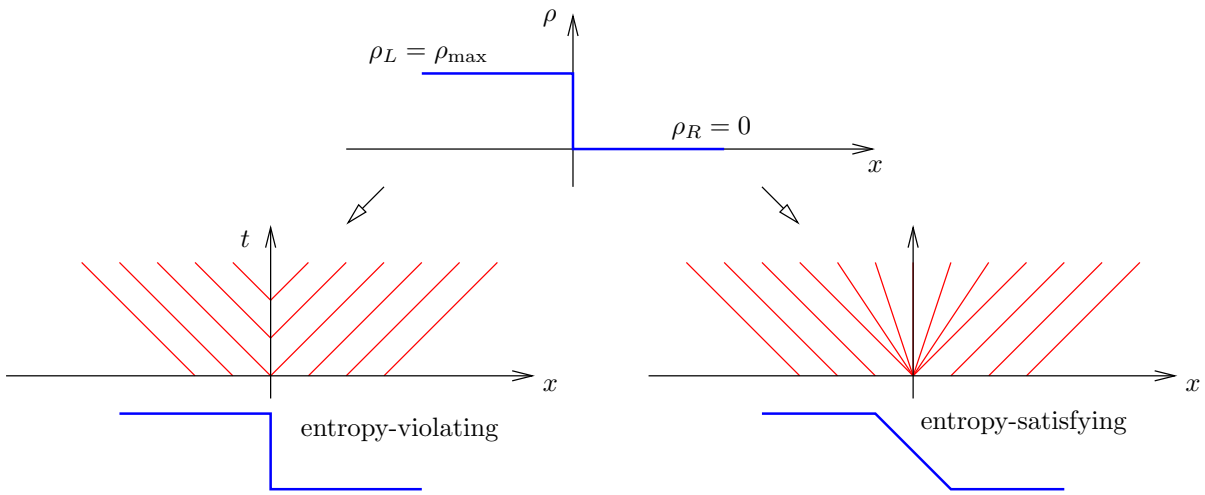


Figure 6.2.6: Solution and characteristics for a green-light problem. Both the entropy-satisfying (rarefaction) and the entropy-violating (shock) solutions are shown.

Example 6.7 (Upwind and Lax-Wendroff for a rarefaction)**.** Recall Example 6.4, in which the initial condition was $u_0(x) = -1$ for $x < 0$ and $u_0(x) = 1$ for $x \geq 0$. We showed two weak solutions: one with the initial discontinuous solution preserved (non-physical, entropy-violating) and one with the discontinuity broken into a rarefaction (physical, entropy-satisfying). Before we discuss the performance of the upwind and Lax-Wendroff methods, we introduce the **Lax-Wendroff Theorem**, which states that: *if a conservative numerical scheme converges, it converges to a weak solution.* Note that this theorem does not guarantee stability, nor does it guarantee that the weak solution that we attain will be the entropy-satisfying one.

So, how do the upwind and Lax-Wendroff fluxes, introduced in Section 5.2.2, fare for the rarefaction problem? Do they yield the entropy-satisfying rarefaction? It turns out that they do not. Let's first look at the upwind method on a grid in which there is a cell interface at $x = 0$ (the discontinuity). For every cell, the cell average is $u_j = \pm 1$, which means that the analytical flux is $f_j = \frac{1}{2}u_j^2 = \frac{1}{2}$ for all cells. The numerical flux expression, Equation 5.2.12, then becomes

$$\begin{aligned} \hat{F}_{j+1/2} &= \frac{1}{2}\left(f_j + f_{j+1}\right) - \frac{1}{2}|\hat{a}_{j+1/2}|\left(u_{j+1} - u_j\right) \\ &= \frac{1}{2} - 0 = \frac{1}{2}, \end{aligned}$$

where the stabilization term is always zero because either $u_{j+1} = u_j$ or $f_{j+1} = f_j$ in the expression for $\hat{a}_{j+1/2}$, Equation 5.2.13. Therefore, $\hat{F}_{j+1/2} = 1/2$ for all j, which means that the flux residual in the update formula, $\hat{F}_{j+1/2} - \hat{F}_{j-1/2}$ in Equation 5.2.5, is identically zero! So the solution does not change and the initial condition is preserved – i.e. we have the entropy-violating solution. Similar reasoning shows that the Lax-Wendroff method also

yields the same entropy-violating solution. The problem with these methods is that they cannot distinguish between shocks and expansion fans. We now consider methods that can guarantee entropy-satisfying solutions.

6.3 Total Variation Diminishing Methods

6.3.1 Total Variation

A useful concept for developing numerical schemes for conservation laws is the **total variation** of a solution. Over an interval $x \in [0, L]$, this is defined as

$$\text{total variation } = \text{TV}(u) = \int_0^L \left| \frac{\partial u}{\partial x} \right| \, dx. \tag{6.3.1}$$

This integral can be simplified to a summation of differences of extrema. For example, for the solution illustrated in Figure 6.3.1, the total variation is

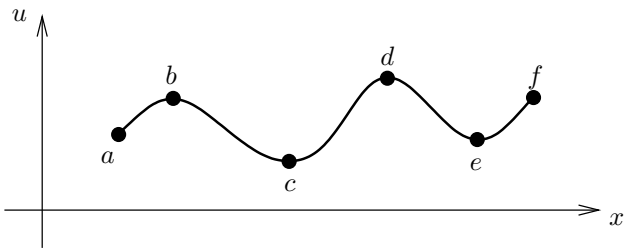


Figure 6.3.1: The extrema of a solution u define its total variation.

$$\begin{aligned} \text{TV}(u) &= |u_b - u_a| + |u_c - u_b| + |u_d - u_c| + ... \\ &= 2\left(\sum \text{maxima} - \sum \text{minima} \right). \end{aligned}$$

For smooth solutions, without shocks, the total variation remains constant. However, for solutions with shocks, the total variation can change. When the solution is entropy-satisfying, the total variation will be non-increasing:

$$\begin{aligned} \text{TV}(u(x,t)) &\leq \text{TV}(u(x,0)), \\ \Rightarrow \quad \frac{d}{dt} \int_0^L \left| \frac{\partial u}{\partial x} \right| \, dx &\leq 0. \end{aligned} \tag{6.3.2}$$

This requirement will serve as a basis for the design of numerical schemes that provide physical solutions to problems with shocks.

6.3.2 Entropy-Satisfying Schemes

One class of entropy-satisfying methods is the set of **monotone schemes**. A monotone scheme is one that can be expressed as

$$u_j^{n+1} = H(u_{j-l}^n, \ldots, u_{j+r}^n), \qquad \text{with} \quad \frac{\partial H}{\partial u_k} \geq 0, \;\; k = j-l, \ldots, j+r. \tag{6.3.3}$$

Such a scheme yields entropy-satisfying solutions, but is at most only first-order accurate.

The best-known monotone method is **Godunov's method**. In this method, the interface flux $\hat{\mathbf{F}}_{j+1/2}$ between the cells j and $j+1$ is computed by solving exactly the Riemann problem with u_j and u_{j+1} on either side. It is sufficient that this solution is valid only instantaneously, as this will ensure convergence in the limit of small time steps. For scalar problems, this **Godunov flux** is

$$\hat{F}_{j+1/2} = \begin{cases} \min\limits_{u_j \leq u \leq u_{j+1}} f(u) & \text{when } u_j \leq u_{j+1} \\ \max\limits_{u_{j+1} \leq u \leq u_j} f(u) & \text{when } u_j > u_{j+1} \end{cases} \tag{6.3.4}$$

The Godunov method then follows from a forward Euler time discretization of Equation 5.2.5,

$$u_j^{n+1} \;\; = \;\; u_j^n - \frac{\Delta t}{\Delta x}\left(\hat{F}_{j+1/2} - \hat{F}_{j-1/2}\right). \tag{6.3.5}$$

The numerical flux in Equation 6.3.4 is valid for any scalar conservation law with convex or non-convex $f(u)$. It satisfies Oleinik's condition, Equation 6.2.5, which means that it should yield entropy-satisfying solutions. Note that the flux is valid only for a short time, until waves from neighboring Riemann problems start interacting with each other.

For purely convex or concave flux functions, Equation 6.3.4 reduces to

$$\hat{F}_{j+\frac{1}{2}} = \begin{cases} f(u_j) & a_j, a_{j+1} \;\geq 0 & \\ f(u_{j+1}) & a_j, a_{j+1} \;< 0 & \\ f(u_j) & a_j > 0 > a_{j+1} & s \geq 0 \\ f(u_{j+1}) & a_j > 0 > a_{j+1} & s < 0 \\ f(u_s) & a_j \leq 0 \leq a_{j+1} & \text{(rarefaction)} \end{cases}, \tag{6.3.6}$$

where $s = (f_j - f_{j+1})/(u_j - u_{j+1})$ is the shock speed, and u_s is the **sonic state** defined by the u value for which $f'(u_s) = 0$. Figure 6.3.2 illustrates the different cases appearing in Equation 6.3.6.

Applied to Burgers equation, Equation 6.3.6 becomes

$$\hat{F}_{j+\frac{1}{2}} = \begin{cases} \frac{1}{2}u_j^2 & u_j, u_{j+1} \;> 0 & \\ \frac{1}{2}u_{j+1}^2 & u_j, u_{j+1} \;< 0 & \\ \frac{1}{2}u_j^2 & u_j > 0 > u_{j+1} & \frac{1}{2}(u_{j+1} + u_j) > 0 \\ \frac{1}{2}u_{j+1}^2 & u_j > 0 > u_{j+1} & \frac{1}{2}(u_{j+1} + u_j) < 0 \\ 0 & u_j < 0 < u_{j+1} & \text{(rarefaction)} \end{cases}. \tag{6.3.7}$$

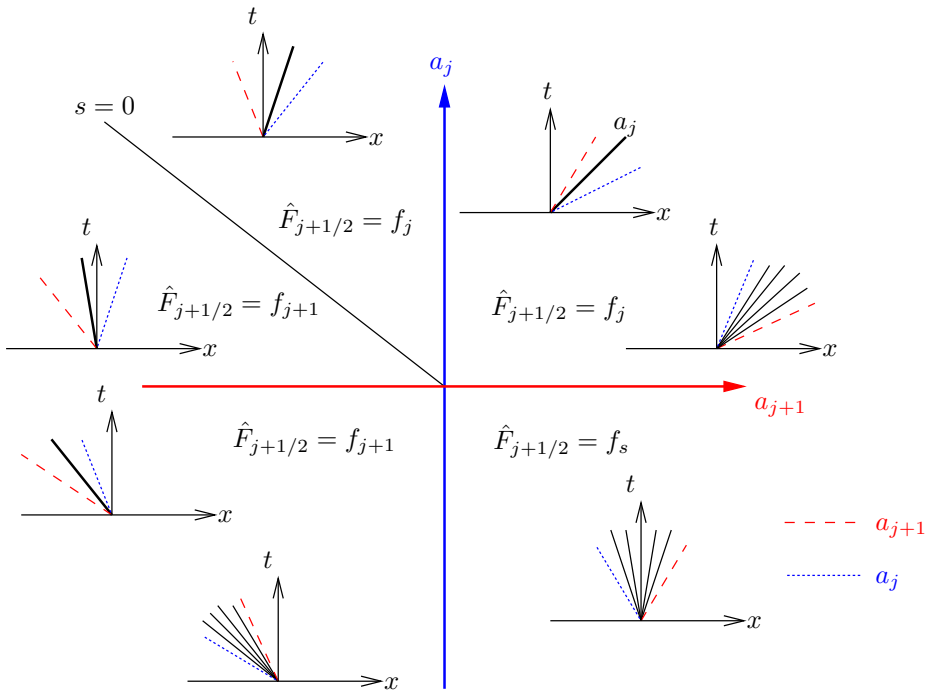


Figure 6.3.2: Illustration of the different cases comprising the Godunov flux. Note, s is the shock speed given by the Rankine-Hugoniot jump condition.

This equation produces a flux which is very similar to the upwind flux but with the key difference that the flux is equal to zero in the case of a rarefaction (expansion). This, apparently, minor modification allows the scheme to distinguish between shocks and rarefactions, and thus to converge to the entropy satisfying solution. The primary disadvantage of monotone schemes is their accuracy, or lack thereof: they are only first-order accurate. However, they provide the building blocks for higher-order accurate schemes.

6.3.3 TVD Definition and Conditions

We saw in the previous section that first-order schemes can be constructed to produce entropy-satisfying solutions. High-order schemes improve accuracy in smooth regions, but they yield solutions that are oscillatory near discontinuities. This is due to dispersion errors and the high-frequency content associated with discontinuities. Such oscillations may be benign for linear problems, but they can lead to non-linear instability and non-physical states in practical, nonlinear systems such as the Euler equations. In addition, the high-order methods we have seen thus far (e.g. Lax-Wendroff) are not guaranteed to produce entropy-satisfying solutions. For example, they can preserve an initial expansion shock instead of converging to the rarefaction solution. In this section we look at **total variation diminishing (TVD)** methods, which can achieve both accurate and entropy-satisfying solutions.

Definition

Recall from Equation 6.3.1 that the total variation measures how oscillatory a solution is. In discrete form, for a finite-difference method, the definition reads

$$\text{total variation } = \text{TV}(u) = \sum_j |u_{j+1} - u_j| \,, \tag{6.3.8}$$

where j indexes the grid nodes. The more "wiggles" the higher the total variation, as shown in Figure 6.3.3 The generation of new extrema leads to an increase in the total variation. On

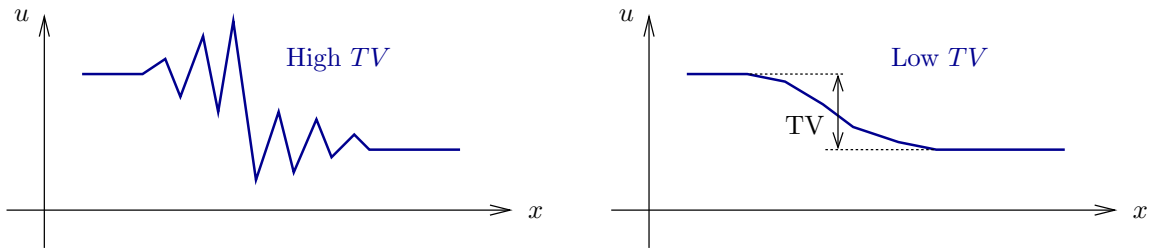


Figure 6.3.3: High and low total-variation solutions for the same net drop in value.

the other hand, total variation diminishing schemes are ones that do not increase the total variation of the solution. The TVD requirement is that

$$TV(u^{n+1}) \leq TV(u^n), \tag{6.3.9}$$

where n is the time-step index of the solution. We note that there exists another, similar, family of methods, TVB, which stands for total-variation bounded. For these methods, there exists a constant C such that $TV(u^{n+1}) \leq C\, TV(u^n)$ for all n as $\Delta t \to 0$ for a fixed total time T. The Lax-Wendroff method is not TVD – see for example the solution in Figure 4.5.12, where wiggles appear at the points of discontinuity.

TVD schemes are **monotonicity preserving**. This means that they do not introduce any new extrema into a solution. However, monotonicity preserving does not mean *monotone*. Instead, the set of monotone schemes, as defined in Sections:TVD:Entropy-Satisfying, is a subset of monotonicity preserving schemes. For linear schemes, they are the same sets, but nonlinear schemes can be made monotonicity-preserving *and high-order accurate*.

Figure 6.3.4 illustrates a Venn diagram of the different sets of methods mentioned thus far. We see that, as discussed before, monotone schemes are entropy-satisfying but first-order accurate. TVD schemes include this set, but also other schemes. Note that not all TVD schemes are entropy-satisfying, but also that not all TVD schemes are first-order accurate – that is, the accuracy of TVD schemes could be higher. The shaded region of entropy-satisfying but not only first-order accurate schemes is where we would like to be. Finally,

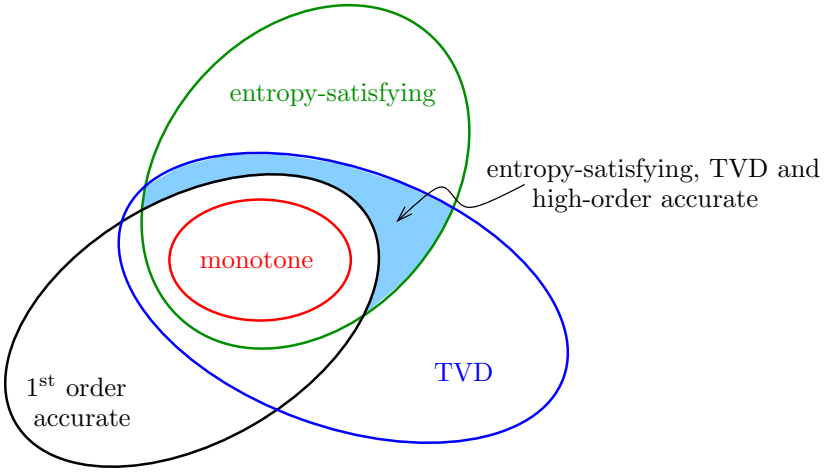


Figure 6.3.4: Venn diagram of various numerical methods.

we note that TVD methods are a subset of stable methods. This means that the TVD condition is a non-linear stability condition, which makes sense since an unstable solution grows without bound and hence necessarily leads to an increasing total variation. Together with consistency, this nonlinear stability ensures convergence for TVD schemes.

Conditions

Consider a scheme of the form

$$u_j^{n+1} = u_j^n + D_{j+\frac{1}{2}} \Delta u_{j+\frac{1}{2}}^n - C_{j-\frac{1}{2}} \Delta u_{j-\frac{1}{2}}^n, \tag{6.3.10}$$

where

$$\Delta u^n_{j+1/2} = u^n_{j+1} - u^n_j, \qquad \Delta u^n_{j-1/2} = u^n_j - u^n_{j-1}. \tag{6.3.11}$$

Such a scheme is TVD if and only if

$$C_{j+\frac{1}{2}} \geq 0, \quad D_{j+\frac{1}{2}} \geq 0, \quad C_{j+\frac{1}{2}} + D_{j+\frac{1}{2}} \leq 1. \tag{6.3.12}$$

Note that the C and D coefficients can depend on the state, so that nonlinear methods are included in this form.

The proof of the condition in Equation 6.3.12 proceeds as follows, with the critical step obtained from the triangle inequality.

$$\begin{aligned}
TV(u^{n+1}) &= \sum_j \left|u^{n+1}_{j+1} - u^{n+1}_j\right| \\
&= \sum_j \left|\Delta u^n_{j+\frac{1}{2}} + D_{j+\frac{3}{2}}\Delta u^n_{j+\frac{3}{2}} - C_{j+\frac{1}{2}}\Delta u^n_{j+\frac{1}{2}} - D_{j+\frac{1}{2}}\Delta u^n_{j+\frac{1}{2}} + C_{j-\frac{1}{2}}\Delta u^n_{j-\frac{1}{2}}\right| \\
&\leq \sum_j \left|D_{j+\frac{3}{2}}\Delta u^n_{j+\frac{3}{2}}\right| + \left|\left(1 - C_{j+\frac{1}{2}} - D_{j+\frac{1}{2}}\right)\Delta u^n_{j+\frac{1}{2}}\right| + \left|C_{j-\frac{1}{2}}\Delta u^n_{j-\frac{1}{2}}\right| \\
&= \sum_j \left|D_{j+\frac{1}{2}}\right|\left|\Delta u^n_{j+\frac{1}{2}}\right| + \left|1 - C_{j+\frac{1}{2}} - D_{j+\frac{1}{2}}\right|\left|\Delta u^n_{j+\frac{1}{2}}\right| + \left|C_{j+\frac{1}{2}}\right|\left|\Delta u^n_{j+\frac{1}{2}}\right| \\
&\leq TV(u^n) \qquad \boxed{\text{if} \quad \left|D_{j+\frac{1}{2}}\right| + \left|1 - C_{j+\frac{1}{2}} - D_{j+\frac{1}{2}}\right| + \left|C_{j+\frac{1}{2}}\right| \leq 1}
\end{aligned}$$

The inequalities can be made equalities for a particular choice of the data u_j, which means that the conditions in Equation 6.3.12 are necessary and sufficient if the scheme is to be TVD for all data.

Example 6.8 (First-order upwind is TVD)**.** Consider the upwind method in Equation 5.2.11, repeated below

$$u^{n+1}_j = u^n_j - \frac{a\Delta t}{2\Delta x}(u^n_{j+1} - u^n_{j-1}) + \frac{|a|\Delta t}{2\Delta x}(u^n_{j+1} - 2u^n_j + u^n_{j-1})\Delta x_j \frac{u^{n+1}_j - u^n_j}{\Delta t}. \tag{6.3.13}$$

Comparing to Equation 6.3.10, we see that the C and D coefficients are

$$C_{j-\frac{1}{2}} = \frac{\Delta t}{2\Delta x}(|a| + a), \qquad D_{j+\frac{1}{2}} = \frac{\Delta t}{2\Delta x}(|a| - a).$$

Clearly, both of these quantities are non-negative, and the condition

$$C_{j-\frac{1}{2}} + D_{j+\frac{1}{2}} = \frac{|a|\Delta t}{\Delta x} \leq 1 \tag{6.3.14}$$

is the same as the CFL condition. Therefore, as long as the CFL condition is satisfied, the first-order upwind method is TVD.

Example 6.9 (Lax-Wendroff is not TVD)**.** Recall the Lax-Wendroff method in Equation 4.2.9, repeated below.

$$\begin{aligned} u_j^{n+1} &= u_j^n - \frac{\sigma}{2}(u_{j+1}^n - u_{j-1}^n) + \frac{\sigma^2}{2}(u_{j-1}^n - 2u_j^n + u_{j+1}^n) \\ &= u_j^n + \underbrace{\frac{1}{2}(\sigma^2 - \sigma)}_{D_{j+\frac{1}{2}}}(u_{j+1}^n - u_j^n) - \underbrace{\frac{1}{2}(\sigma^2 + \sigma)}_{C_{j-\frac{1}{2}}}(u_j^n - u_{j-1}^n) \end{aligned}$$

The TVD condition

$$C_{j+\frac{1}{2}} + D_{j+\frac{1}{2}} = \sigma^2 \leq 1$$

is met as long as $\sigma \leq 1$, which is the same as the CFL condition. However, we also need both coefficients non-negative, and this is not the case for $D_{j+\frac{1}{2}}$, which is

$$D_{j+\frac{1}{2}} = \frac{1}{2}(\sigma^2 - \sigma) = \frac{1}{2}\sigma(\sigma - 1) \leq 0 \quad \text{when} \quad \sigma \leq 1.$$

Therefore, the Lax-Wendroff method is not TVD.

Godunov's Theorem

How accurate of a scheme can we make and still be TVD? **Godunov's theorem** states that linear TVD schemes are at most first-order accurate. This means that high-order TVD schemes must be nonlinear (even for linear problems!). Note that a general scheme that is expressed as

$$u_j^{n+1} = \sum_k c_k u_{j+k}^n$$

is linear if the c_k are constant and nonlinear if the c_k depend on the state u.

We now outline the proof of Godunov's theorem for advection. Recall that TVD schemes are monotonicity preserving. For a linear scheme to be monotonicity preserving, the coefficients c_k must all be non-negative. To see that this is sufficient, evaluate

$$u_{j+1}^{n+1} - u_j^{n+1} = \sum_k c_k(u_{j+k+1}^n - u_{j+k}^n).$$

For non-negative c_k and monotone data at n, every term in the sum is of one sign, so the left-hand-side must also be of that sign $\Rightarrow$ monotonicity is preserved. To see that non-negative c_k are necessary, assume that $c_l < 0$ and consider monotonically decreasing

data $u_j^n = 1$ for $j \le l$, and $u_j^n = 0$ for $j > l$. The above formula at $j = 0$ reduces to $u_1^{n+1} - u_0^{n+1} = c_l(u_{l+1} - u_l) > 0$ and hence monotonicity is not preserved. Thus, all c_k must be positive for a linear scheme to preserve monotonicity. To determine the order of accuracy, we perform a Taylor-series expansion about $x = j\Delta x$, resulting in

$$\begin{aligned}\tau = u_j^{n+1} - u_{j-\sigma}^n &= \sum_k c_k \left[u_j + k\Delta x(u_x)_j + k^2\Delta x^2(u_{xx})_j/2 + \ldots\right] \\ &\quad - \left[u_j - \sigma\Delta x(u_x)_j + \sigma^2\Delta x^2(u_{xx})_j/2 + \ldots\right].\end{aligned}$$

To achieve second order we need

$$\sum_k c_k = 1, \quad \sum_k k c_k = -\sigma, \quad \sum_k k^2 c_k = \sigma^2.$$

Set $c_k = e_k^2$ to indicate that all c_k are positive. Then the above equations require

$$\sum_k (e_k)^2 \sum_k (ke_k)^2 = \left[\sum_k k(e_k)^2\right]^2,$$

which is not possible because it violates *Cauchy's Inequality*: $(\mathbf{a}\cdot\mathbf{a})(\mathbf{b}\cdot\mathbf{b}) \ge (\mathbf{a}\cdot\mathbf{b})^2$, with equality only if the two sequences are proportional (not the case here). Thus, second-order accurate (or higher) linear schemes cannot be monotonicity-preserving, and hence cannot be TVD.

6.3.4 High-Resolution Schemes

To obtain a high-accuracy scheme, we will need to turn to nonlinear methods. However, building blocks for these are linear methods, and we consider one category: **3-point linear schemes**, which can be expressed as

$$u_j^{n+1} = c_{-1}u_{j-1}^n + c_0^n + c_1 u_{j+1}^n. \tag{6.3.15}$$

Consistency requirements (from Taylor-series expansions) for a conservation law require

$$c_{-1} + c_0 + c_1 = 1, \tag{6.3.16}$$

$$-c_{-1} + c_1 = -\sigma, \tag{6.3.17}$$

where σ is the CFL number. With three parameters, these leave one unknown, which we call q, and we write the above update formula as

$$u_j^{n+1} = u_j^n - \frac{\sigma}{2}\left(u_{j+1}^n - u_{j-1}^n\right) + \frac{q}{2}\left(u_{j+1}^n - 2u_j^n + u_{j-1}^n\right). \tag{6.3.18}$$

Note that this looks like a central scheme with an added diffusion term. The role of this diffusion term is to stabilize the central scheme. Specifically, von-Neumann stability analysis gives the requirement that

$$\sigma^2 \le q \le 1.$$

Furthermore, the condition that the scheme be monotonicity-preserving gives

$$|\sigma| \leq q(\sigma) \leq 1.$$

The various methods that we have encountered so far can be written in this form:

$q(\sigma) = 1$	Lax-Friedrichs	Monotonicity preserving
$q(\sigma) = \|\sigma\|$	First-order Upwind	Monotonicity preserving
$q(\sigma) = \sigma^2$	Lax-Wendroff	lower stability bound
$q(\sigma) = 0$	FTCS	unstable

Figure 6.3.5 illustrates these relationships between q and σ for the various schemes. Note, the **low phase-error (LPE)** scheme is given by

$$q(\sigma) = \frac{1 + 2\sigma^2}{3}. \tag{6.3.19}$$

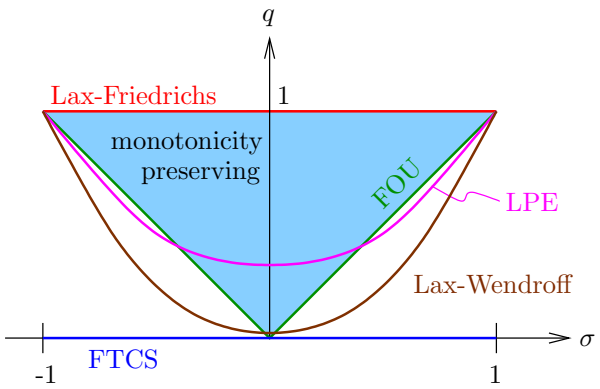


Figure 6.3.5: A $q - \sigma$ diagram of three-point schemes as given in Equation 6.3.18.

Our starting point in the design of high-resolution schemes is the Lax-Wendroff method, Equation 4.2.9,

$$u_j^{n+1} = u_j^n - \frac{\sigma}{2}(u_{j+1}^n - u_{j-1}^n) + \frac{\sigma^2}{2}(u_{j-1}^n - 2u_j^n + u_{j+1}^n).$$

As shown in Example 6.9, Lax-Wendroff is not TVD. On the other hand, the first-order upwind method is TVD, as shown in Example 6.8. Motivated by this observation, we rewrite the Lax-Wendroff method as

$$u_j^{n+1} = \underbrace{u_j^n - \sigma(u_j^n - u_{j-1}^n)}_{\text{first-order upwind}} \underbrace{-\frac{1}{2}\sigma(1-\sigma)(u_{j+1}^n - 2u_j^n + u_{j-1}^n)}_{\text{anti-diffusion}}. \tag{6.3.20}$$

That is, Lax-Wendroff has a term that *removes* (note the sign) diffusion from first-order upwind. However, removing this diffusion is also what makes Lax-Wendroff not TVD, and hence we would like to *limit* the amount of diffusion removed.

Although it is tempting to just put a multiplicative factor on the anti-diffusion term in Equation 6.3.20, doing so would violate conservation, since we would be adding or subtracting from an otherwise conservative update. Instead, we must limit the update through the flux, which for the update in Equation 6.3.20 reads

$$\hat{F}^{LW}_{j+\frac{1}{2}} = au_j + \frac{a}{2}(1-\sigma)(u_{j+1} - u_j). \tag{6.3.21}$$

We introduce a **flux limiter**, $\phi_{j+\frac{1}{2}}$, into this flux expression, to read

$$\hat{F}^{LW}_{j+\frac{1}{2}} = au_j + \frac{a}{2}(1-\sigma)\phi_{j+\frac{1}{2}}(u_{j+1} - u_j). \tag{6.3.22}$$

With this modified flux, the update formula in Equation 6.3.20 becomes

$$u_j^{n+1} \;=\; u_j^n - \sigma(u_j^n - u_{j-1}^n) - \frac{1}{2}\sigma(1-\sigma)\left[\phi_{j+\frac{1}{2}}(u_{j+1}^n - u_j^n) - \phi_{j-\frac{1}{2}}(u_j^n - u_{j-1}^n)\right]$$

If $\phi_{j+\frac{1}{2}} = 1$ at all interfaces, we recover the non-TVD Lax-Wendroff method. On the other hand, if $\phi_{j+\frac{1}{2}} = 0$, we recover the TVD (but inaccurate) first-order upwind method. Our goal is to choose the limiter, $\phi_{j+\frac{1}{2}}$, as close to 1 as possible while enforcing the TVD conditions in Equation 6.3.12. To do this, we must rewrite the update equation in the form of Equation 6.3.10. This gives

$$u_j^{n+1} \;=\; u_j^n + \underbrace{0}_{D_{j+\frac{1}{2}}}\Delta u^n_{j+\frac{1}{2}} - \underbrace{\sigma\left[1 + \frac{1-\sigma}{2}\left(\frac{\phi_{j+\frac{1}{2}}}{r_{j+\frac{1}{2}}} - \phi_{j-\frac{1}{2}}\right)\right]}_{C_{j-\frac{1}{2}}}\Delta u_{j-\frac{1}{2}}, \tag{6.3.23}$$

where $r_{j+\frac{1}{2}}$ is the ratio of adjacent differences (slopes),

$$r_{j+\frac{1}{2}} = \frac{\Delta u_{j-\frac{1}{2}}}{\Delta u_{j+\frac{1}{2}}} = \frac{u_j - u_{j-1}}{u_{j+1} - u_j}. \tag{6.3.24}$$

Figure 6.3.6 illustrates several qualitative state variations with different values of r. Since $D_{j+\frac{1}{2}} = 0$, the TVD conditions in Equation 6.3.12 reduce to the requirement that

$$0 \le C_{j-\frac{1}{2}} \le 1. \tag{6.3.25}$$

This condition translates to

$$\phi(r) = 0 \;\text{ for }\; r \le 0, \qquad 0 \le \frac{\phi(r)}{r} \le 2, \qquad 0 \le \phi(r) \le 2. \tag{6.3.26}$$

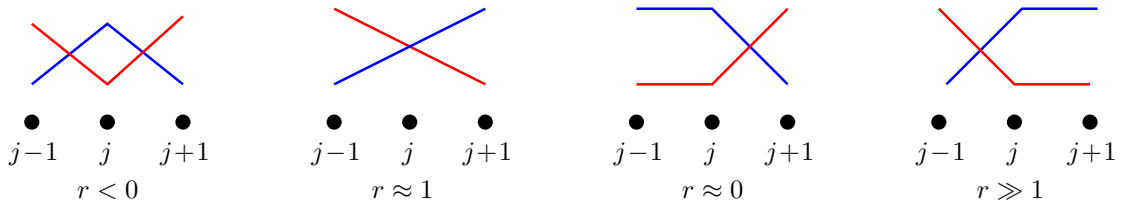


Figure 6.3.6: Ratio of consecutive slopes for the purpose of limiter definition.

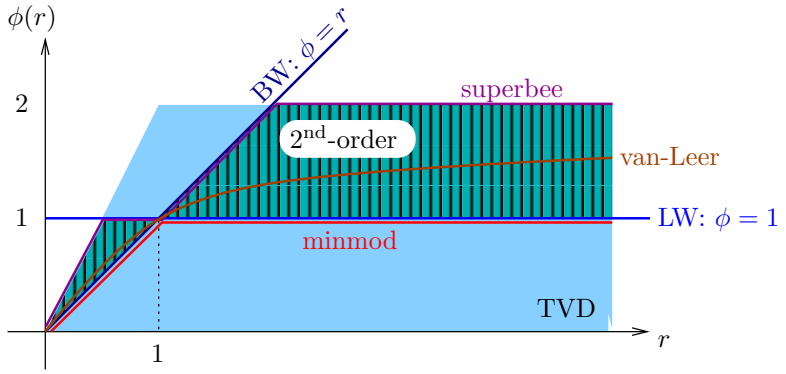


Figure 6.3.7: TVD and second-order requirements on the flux limiter, $\phi(r)$.

In addition, for second-order accuracy, we need $\phi(r = 1) = 1$, which ensures that the method flips to Lax-Wendroff when the solution is not oscillatory. Figure 6.3.7 shows these requirements graphically, with $\phi(r)$ plotted versus r and the TVD region shaded. Note in Figure 6.3.7 that the region of second-order TVD methods is a subset of the TVD region. On the edges of second-order accuracy are the Lax-Wendroff (LW) method and the Beam-Warming (BW) method (see Section 4.2). To stay within this region, ϕ cannot be a linear function of r but must instead curve: this means that the limiter must be nonlinear. Some popular choices for the limiter include:

1. **Minmod**: $\phi(r) = \max(0, \min(1, r))$. This is the most "diffussive" limiter. It corresponds to the lower boundary of the TVD region.

2. **Superbee**: $\phi(r) = \max(0, \min(2r, 1), \min(r, 2))$ This is the least "diffussive" limiter. It corresponds to the upper boundary of the TVD region.

3. **Van Leer (Harmonic)**: $\phi(r) = \dfrac{r + |r|}{1 + |r|}$.

All of these limiters produce second order schemes when the solution is smooth, and reduce to the upwind method at discontinuities. In addition, the limiter possess the symmetry property $\frac{\phi(r)}{r} = \phi\left(\frac{1}{r}\right)$, which ensures that the top corner of a discontinuity is treated symmetrically to a bottom corner.

Figure 6.3.8 shows numerical results using these methods for a linear advection problem. The initial condition consists of a "basket" of wave-forms, in order to assess the performance of the methods in smooth, slope-discontinuous, and value-discontinuous regions. The boundary is periodic, and the simulation runs for one period with $N = 120$ intervals in x and a CFL of $\sigma = 0.5$. As expected, the upwind method is very dissipative, as it is only first-order accurate. The Lax-Wendroff method is second-order accurate but suffers from oscillations due to dispersion errors. The Fromm scheme is the average of Lax-Wendroff and Beam-Warming, which have opposite-sign dispersion errors, so that these cancel; as a result, Fromm actually performs reasonably well out of the methods, even though it is not TVD (see Figure 6.3.7). Out of the TVD methods, minmod is the most dissipative, whereas superbee tends to sharpen all of the data into rectangle-shaped wave-forms. The van-Leer limiter performs in between minmod and superbee, as expected.

6.4 Slope Limiting

Another interpretation of limiting, common in finite-volume methods, is geometric. Here we consider just linear advection. The procedure consists of three operations at each time step:

- *Reconstruct* a linear solution (Ru) in each control volume

$$(Ru)_j(x) = u_j + s_j(x - x_j)$$

  where s_j is the slope to be defined

- *Evolve* the linear profiles forward in time so that flux is

$$\begin{aligned} \hat{F}_{j+\frac{1}{2}} &= \frac{1}{\Delta t}\int_{x_{j+\frac{1}{2}}-a\Delta t}^{x_{j+\frac{1}{2}}} (Ru)(x)dx \\ &= \begin{cases} a(u_j + \dfrac{1-\sigma}{2}\Delta x s_j) & a > 0 \\ a(u_{j+1} - \dfrac{1-\sigma}{2}\Delta x s_{j+1}) & a < 0, \text{ (note } \sigma < 0) \end{cases} \end{aligned}$$

- *Project* the evolved linear profiles by computing new cell averages (automatic if flux defined as above)

Limiting of oscillations comes in through the reconstruction step. To prevent undershoots and overshoots, one way to choose s_j is the harmonic mean,

$$s_j = L(s_{j-\frac{1}{2}}, s_{j+\frac{1}{2}}) = \begin{cases} 0 & s_{j-\frac{1}{2}} s_{j+\frac{1}{2}} < 0 \\ \dfrac{2 s_{j-\frac{1}{2}} s_{j+\frac{1}{2}}}{s_{j-\frac{1}{2}} + s_{j+\frac{1}{2}}} & \text{otherwise} \end{cases},$$

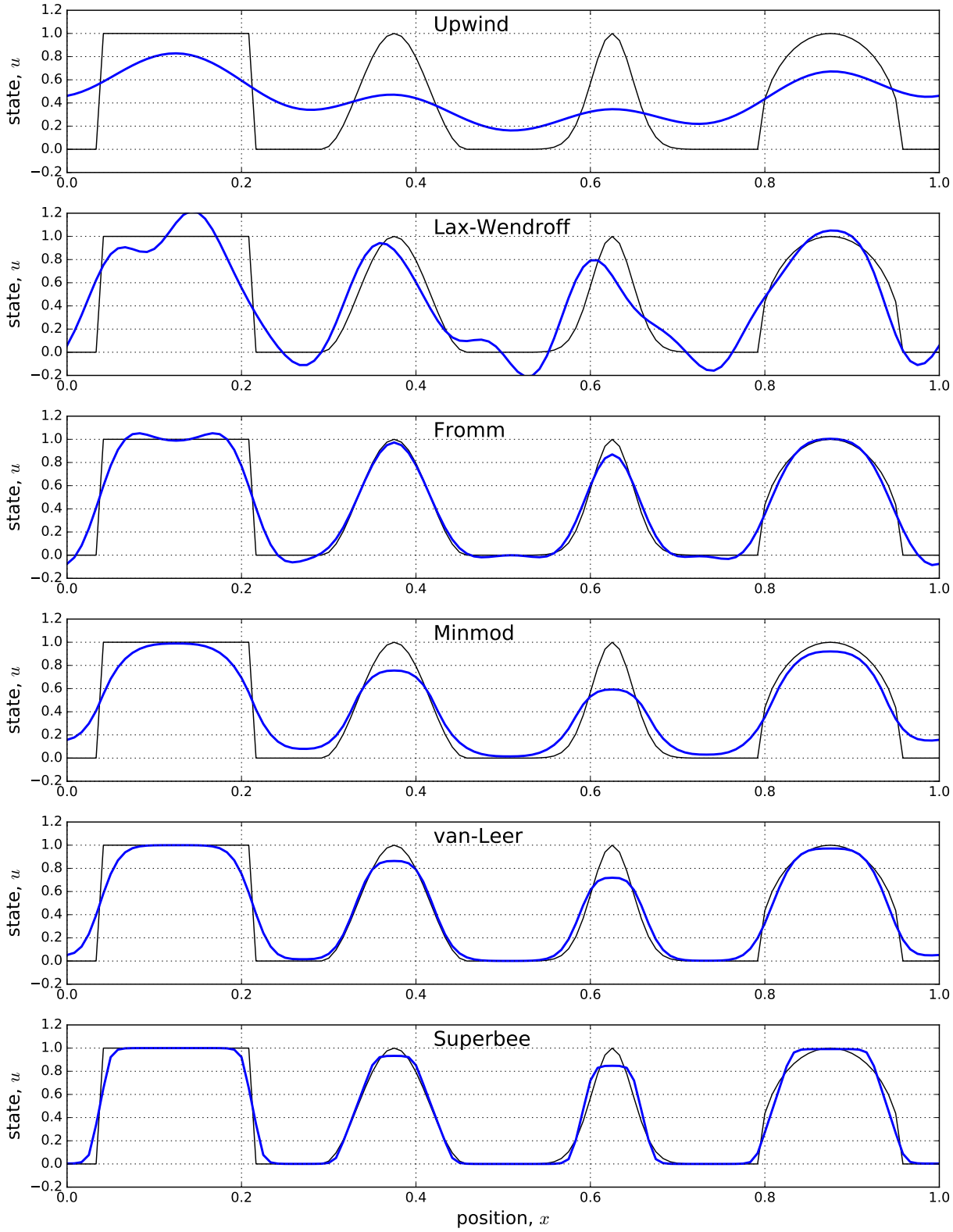


Figure 6.3.8: One period linear advection solutions using various methods (IC in black).

where

$$s_{j-\frac{1}{2}} = \frac{u_j - u_{j-1}}{\Delta x}, \quad s_{j+\frac{1}{2}} = \frac{u_{j+1} - u_j}{\Delta x}.$$

$L(s_{j-\frac{1}{2}}, s_{j+\frac{1}{2}})$ is the slope limiter, and in the above form it ensures that s_j never exceeds the smaller argument by a factor of 2. This means no new data is introduced: the reconstructed solution remains between the neighbors, unless the cell average is an extremum, in which case no slope is introduced.

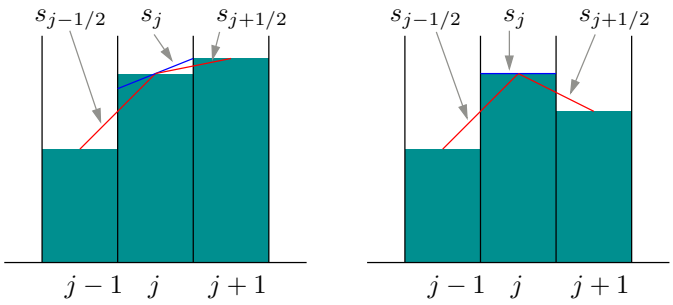


Figure 6.4.1: Slopes used in one-dimensional slope limiting.

The slope limiter is equivalent to the flux limiter provided that

$$\phi_{j+\frac{1}{2}} = \frac{L(s_{j-\frac{1}{2}}, s_{j+\frac{1}{2}})}{s_{j+\frac{1}{2}}}.$$

When applied near solution extrema, these limiters degrade the accuracy. A simple fix is to only apply the limiters when one or both of $s_{j-\frac{1}{2}}, s_{j+\frac{1}{2}}$ are large. The downside is a new problem-dependent parameter: what is a "large" slope?

Chapter 7

The Incompressible Navier Stokes Equations

7.1 Incompressibility

7.1.1 The Incompressible Limit

All fluids exhibit some compressibility; hence the incompressible limit is a simplification. In a fluid, information propagates at speeds related to the fluid velocity, $|\vec{v}|$, and the speed of sound, a. Since $a^2 = \partial p/\partial \rho$ at constant entropy, we see that the speed of sound measures the pressure changes required to produce density changes. In a truly incompressible fluid, the density cannot change, which means that the speed of sound is infinite, and this is impossible. It is more meaningful to talk about nearly incompressible flow, in which $|\vec{v}| \ll a$. In such flows, acoustic waves travel much faster than the fluid and achieve equilibrium in a very short time scale. Variations in density are proportional to $1/a^2$ the variations in pressure, so we can safely assume $\rho =$ **constant**.

Water is the prototypical incompressible fluid, but gas flows at low Mach numbers, M, can also be treated as incompressible. Typically, an incompressible assumption remains valid for M up to 0.3.

Mathematically, in the incompressible limit, the equations change character from hyperbolic to elliptic: infinite propagation speed means that all parts of the flow can influence each other. Thus, numerical treatment of incompressible flows is inherently different from that of compressible flows.

7.1.2 Equations

A constant density, ρ, decouples the energy equation from the Navier-Stokes system, so that all we need to solve for p and $\vec{v}$ are the continuity and momentum equations. First, the **continuity** or **conservation of mass** equation is

$$\begin{aligned} \partial_t \rho + \nabla \cdot (\rho \vec{v}) &= 0, \\ \Rightarrow \nabla \cdot \vec{v} &= 0 \qquad (\rho \text{ constant}). \end{aligned} \tag{7.1.1}$$

For multiple non-mixed fluids, ρ will not be globally constant, but the above continuity equation still holds for each fluid. As a result of Equation 7.1.1, the constitutive relation for

a Newtonian fluid,

$$\underline{\tau} = \mu(\nabla\vec{v} + \nabla\vec{v}^T) - \frac{2}{3}\mu\nabla\cdot\vec{v}\underline{I} \qquad (\underline{I} = \text{identity tensor}), \tag{7.1.2}$$

also simplifies to $\nabla\cdot\underline{\tau} = \mu\nabla^2\vec{v}$. Next, the conservation of momentum equation reads

$$\partial_t\vec{v} + (\vec{v}\cdot\nabla)\vec{v} + \frac{1}{\rho}\nabla p = \nu\nabla^2\vec{v}, \qquad \nu = \mu/\rho. \tag{7.1.3}$$

The left-hand side is not in conservation form, but this is not so important as we do not have shocks. Incompressible flows are often studied at lower Reynolds numbers than compressible flows, so an inviscid assumption is rare. When necessary, inviscid incompressible flow can be analyzed by simplified numerical techniques.

7.1.3 Pressure Poisson Equation (PPE)

Taking the divergence of the momentum equation, $\nabla\cdot$ (Equation 7.1.3), gives the **pressure Poisson equation**,

$$\nabla^2 p = -\rho\nabla\cdot(\vec{v}\cdot\nabla\vec{v}). \tag{7.1.4}$$

In two dimensions, with $\vec{v} = [u, v]$, this equation becomes (after several applications of $\nabla\cdot\vec{v} = 0$),

$$\nabla^2 p = 2\rho(u_x v_y - u_y v_x). \tag{7.1.5}$$

This is a Poisson equation for the pressure. The lack of a p_t term makes the equation elliptic: pressure waves propagate at infinite speed in incompressible flow. For initial conditions, the above says that pressure cannot be specified independently of the velocity field. This is physical, since a truly incompressible fluid cannot be brought into a state that violates this equation.

At a solid wall, the no-slip boundary condition applies, so that Equation 7.1.3 yields

$$\vec{v} = 0 \Rightarrow \vec{v}_t = 0 \Rightarrow \nabla p = \mu\nabla^2\vec{v}. \tag{7.1.6}$$

This Neumann boundary condition needs a compatibility condition to make the equation well-posed.

Numerically, the pressure Poisson equation gives insight into one possible solution procedure. At each time step, we can advance the flow velocities using the momentum equation, and then solve for the new pressure using the pressure Poisson equation. In between the time steps, we can think of the acoustic waves traveling infinitely many times throughout the domain to bring the pressure in equilibrium with the velocity field.

7.1.4 Kinetic Energy Equation

Taking the dot product of the momentum equation with the specific momentum, $\rho\vec{v}$, gives

$$KE_t + \nabla \cdot [(KE + p)\vec{v}] = \mu[\nabla^2 KE - |\nabla^2 \vec{v}|^2], \tag{7.1.7}$$

where the kinetic energy is $KE = \frac{1}{2}\rho\vec{v} \cdot \vec{v}$. The continuity equation, Equation 7.1.1, is again used in the manipulations to arrive at the above equation.

Integrating over a closed domain where $\vec{v} = 0$ on the boundary gives

$$\int_\Omega KE_t \, d\Omega = -\mu \int_\Omega |\nabla^2 \vec{v}|^2 \, d\Omega, \tag{7.1.8}$$

where we have used the divergence theorem, and where two terms have dropped out because $\vec{v} = 0$ on the boundary. This equation says that the kinetic energy in a bounded domain decreases, only due to viscous dissipation.

This observation could be used as a test for a numerical solution procedure. One could run the code for a certain finite time and measure the loss of kinetic energy (through a discrete integration). Repeating the run for smaller values of viscosity and plotting the kinetic energy lost versus viscosity should yield a line that passes through the origin. If it does not, extra dissipation is likely being added through discretization errors.

7.2 Vorticity-Stream Function Approach

7.2.1 Equations

In two dimensions, the incompressible Navier-Stokes equations read

$$u_x + v_y = 0, \tag{7.2.1}$$

$$u_t + uu_x + vu_y + p_x/\rho = \nu\nabla^2 u, \tag{7.2.2}$$

$$v_t + uv_x + vv_y + p_y/\rho = \nu\nabla^2 v. \tag{7.2.3}$$

These can be described by two independent unknowns:

$$\text{vorticity} \qquad \omega = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} \tag{7.2.4}$$

$$\text{stream function} \qquad \frac{\partial \psi}{\partial y} = u, \frac{\partial \psi}{\partial x} = -v \tag{7.2.5}$$

By construction, defining a stream function enforces continuity, since we automatically obtain $u_x + v_y = 0$. The x and y momentum equations can be combined to eliminate the pressure:

$$\frac{\partial}{\partial x}(\text{y-mom}) - \frac{\partial}{\partial y}(\text{x-mom}) \rightarrow \underbrace{\omega_t + u\omega_x + v\omega_y = \nu(\omega_{xx} + \omega_{yy})}_{\text{vorticity transport equation}} \tag{7.2.6}$$

Substituting ψ for u and v in the vorticity definition and the vorticity transport equations gives

$$\begin{aligned} \omega_t + \psi_y \omega_x - \psi_x \omega_y &= \nu(\omega_{xx} + \omega_{yy}) \quad \text{(Parabolic vorticity transport)} && (7.2.7) \\ \psi_{xx} + \psi_{yy} &= -\omega \quad \text{(Poisson equation)} && (7.2.8) \end{aligned}$$

Thus, the three equations for u, v, p have been converted into two equations for ω, ψ.

7.2.2 Boundary Conditions

On a general moving wall, with wall-aligned coordinates s, n, as shown in Figure 7.2.1, the stream function definition gives

$$\psi_n(P) = \frac{\partial \psi}{\partial n}(P) = U_{\text{wall}} \qquad \text{no slip} \tag{7.2.9}$$

$$\psi_s(P) = \frac{\partial \psi}{\partial s}(P) = 0 \qquad \text{zero flow-through} \tag{7.2.10}$$

For a closed domain with no flow-through, the second equation means that ψ is constant on the boundary, and it's simplest to set $\psi(P) = 0$. The vorticity on the boundary can be

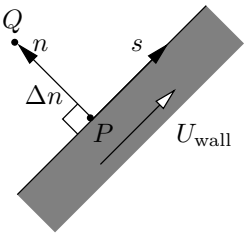


Figure 7.2.1: Quantities used in enforcing a moving wall boundary condition.

related to the stream function

$$\omega(P) = -\nabla^2 \psi(P) = -\psi_{nn}(P), \tag{7.2.11}$$

where we have used the fact that ψ is constant along the wall, so that $\Rightarrow \psi_{ss}(P) = 0$.

A Taylor expansion for ψ to a point Q on the interior gives

$$\begin{aligned} \psi(Q) &= \psi(P) + \psi_n(P)\Delta n + \frac{1}{2}\psi_{nn}(P)\Delta n^2 + O(\Delta n^3), \\ \Rightarrow -\psi_{nn}(P) &= \frac{2}{\Delta n^2}(\psi(P) - \psi(Q)) + \frac{2}{\Delta n}\psi_n(P) + O(\Delta n). \end{aligned}$$

Since $\psi_n(P) = U_{\text{wall}}$, a first-order accurate condition for the vorticity is

$$\omega(P) = \frac{2}{\Delta n^2}(\psi(P) - \psi(Q)) + \frac{2}{\Delta n} U_{\text{wall}} + O(\Delta n). \tag{7.2.12}$$

The sign in front of the U_{wall} term is the sign of $\vec{U}_{\text{wall}} \times \vec{n}$, which is positive as defined in Figure 7.2.1.

Higher-order accurate conditions can be derived, using more data for ψ from the interior, but instabilities can set in at high Reynolds numbers, $Re = U_{\text{wall}} L/\nu$, L = characteristic length.

7.2.3 Finite-Difference Discretization

Consider a regular $N \times N$ grid with uniform spacing, $\Delta x = \Delta y \equiv h$, and $\omega_{i,j}, \psi_{i,j}$ being the solution at point (i, j), with coordinates $(x = ih, y = jh)$. We use a forward-time centered-space (FTCS) discretization of the parabolic vorticity transport equation, Equation 7.2.7,

$$\begin{aligned} \omega_{i,j}^{n+1} \;=\; \omega_{i,j}^{n} - \Delta t \Bigg[& \frac{(\psi_{i,j+1}^{n} - \psi_{i,j-1}^{n})}{2h} \frac{(\omega_{i+1,j}^{n} - \omega_{i-1,j}^{n})}{2h} \\ & - \frac{(\psi_{i+1,j}^{n} - \psi_{i-1,j}^{n})}{2h} \frac{(\omega_{i,j+1}^{n} - \omega_{i,j-1}^{n})}{2h} \\ & - \nu \frac{(\omega_{i+1,j}^{n} + \omega_{i-1,j}^{n} + \omega_{i,j+1}^{n} + \omega_{i,j-1}^{n} - 4\omega_{i,j}^{n})}{h^2} \Bigg] \end{aligned} \tag{7.2.13}$$

and a centered space discretization of the stream function Poisson equation, Equation 7.2.8,

$$\frac{\psi_{i+1,j}^{n} + \psi_{i-1,j}^{n} + \psi_{i,j+1}^{n} + \psi_{i,j-1}^{n} - 4\psi_{i,j}^{n}}{h^2} = -\omega_{i,j}^{n}. \tag{7.2.14}$$

7.2.4 Solution Procedure

The unsteady solution iterations proceed as follows:

1. Initialize vorticity, ω^0.
2. Solve the stream function Poisson equation using ω^n.
3. Calculate vorticity on boundary using ψ on boundary and interior.
4. Update vorticity on interior nodes using FTCS iteration.
5. Increment n and go to step 2.

The time step restriction for FTCS is

$$\frac{\nu \Delta t}{\Delta x^2} \leq \frac{1}{4}, \qquad \frac{(|u| + |v|)^2 \Delta t}{\nu} \leq 4, \tag{7.2.15}$$

where u, v can be calculated from the stream function. These restrictions come from high and low frequency limits in a convection-diffusion stability analysis. This solution procedure

can be used to march a problem to steady-state. A convergence check on the changes in ω and ψ can be used as a termination criterion.

The equations could also be solved in an implicit manner. For steady-state, we have $2N^2$ unknowns $= \omega$ and ψ at each node on a $N \times N$ grid. For ψ we have a Dirichlet condition on each boundary node and the Poisson equation on each interior node. For ω we have a steady-state equation on each interior node, and ω in terms of ψ on each boundary node. An iterative procedure such a the Newton-Raphson method is necessary because the vorticity transport equation is nonlinear.

Example 7.1 (The lid-driven cavity problem)**.** Consider a square domain, with static walls on three of the boundaries and a moving wall on the top boundary, as shown in Figure 7.2.2. The square cavity is filled with a fluid of viscosity ν. No-slip boundary conditions are imposed

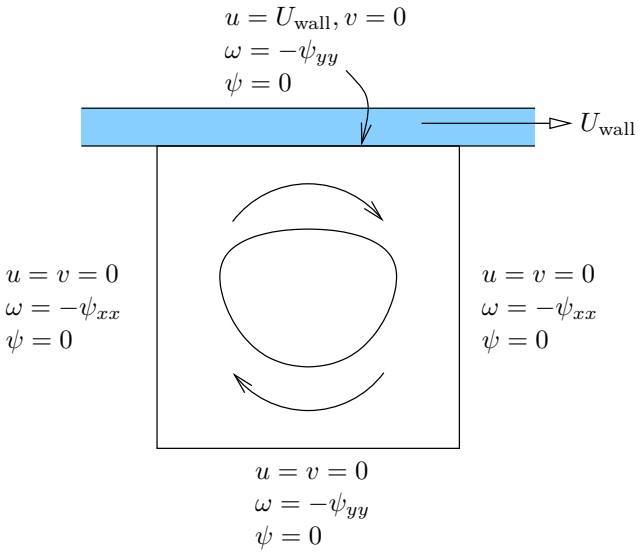


Figure 7.2.2: Setup of the lid-driven cavity problem.

on all of the walls, so that the velocity is zero on all boundaries except the top wall, where $u = U_{\text{wall}}$. Since no flow enters or leaves the domain, $\psi =$ constant $= 0$ (arbitrarily) on the boundary.

Listing 7.2.1 shows a Python code that implements the vorticity-stream function formulation of the incompressible Navier-Stokes equations for this problem. The code takes as input the size of the grid, the Reynolds number, and the target simulation time. The goal is steady state, and the longer the simulation time, the closer the solution will be to steady state. Note that the Poisson equation for the stream function is solved with a direct sparse solver. The time step is computed at every solver iteration, as the velocity changes during the iterations.

Listing 7.2.1: Finite difference code for solving the lid-driven cavity problem in a vorticity-stream function formulation.

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

def plotstate(X, Y, P, fname):
    dosave = not not fname
    f = plt.figure( figsize =(6,6))
    plt.contourf(X,Y,P, 10)
    plt.xlabel(r'$x$', fontsize=16); plt.ylabel(r'$y$', fontsize=16)
    plt.figure(f.number); plt.grid()
    plt.tick_params(axis='both', labelsize=12)
    f.tight_layout(); plt.show(block=(not dosave))
    if dosave: plt.savefig(fname)
    plt.close(f)

def Poisson(N, h):
    h2 = h*h;
    NN = (N-1)*(N-1) # total number of unknowns
    nnz = 5*NN - 4*(N-1) # number of nonzeros
    data = np.zeros(nnz, dtype=np.float);
    irow = np.zeros(nnz, dtype=np.int); icol = irow.copy()

    # fill in interior contributions; no boundary contributions (psi = 0 there)
    inz = 0
    for iy in range(1,N):
        for ix in range(1,N):
            k = (iy-1)*(N-1)+ix-1
            irow[inz] = icol[inz] = k; data[inz] = -4./h2; inz += 1
            if (ix > 1):
                irow[inz] = k; icol[inz] = k-1; data[inz] = 1./h2; inz += 1
            if (ix < N-1):
                irow[inz] = k; icol[inz] = k+1; data[inz] = 1./h2; inz += 1
            if (iy > 1):
                irow[inz] = k; icol[inz] = k-(N-1); data[inz] = 1./h2; inz += 1
            if (iy < N-1):
                irow[inz] = k; icol[inz] = k+(N-1); data[inz] = 1./h2; inz += 1

    print (inz-nnz)

    # build sparse matrix
    A = sparse.csr_matrix((data, (irow, icol)), shape=(NN,NN))
    return A


def cavity(Re, N, T):

    # parameters
    Uwall = 1.; L = 1.; nu = Uwall*L/Re

    # Mesh
```

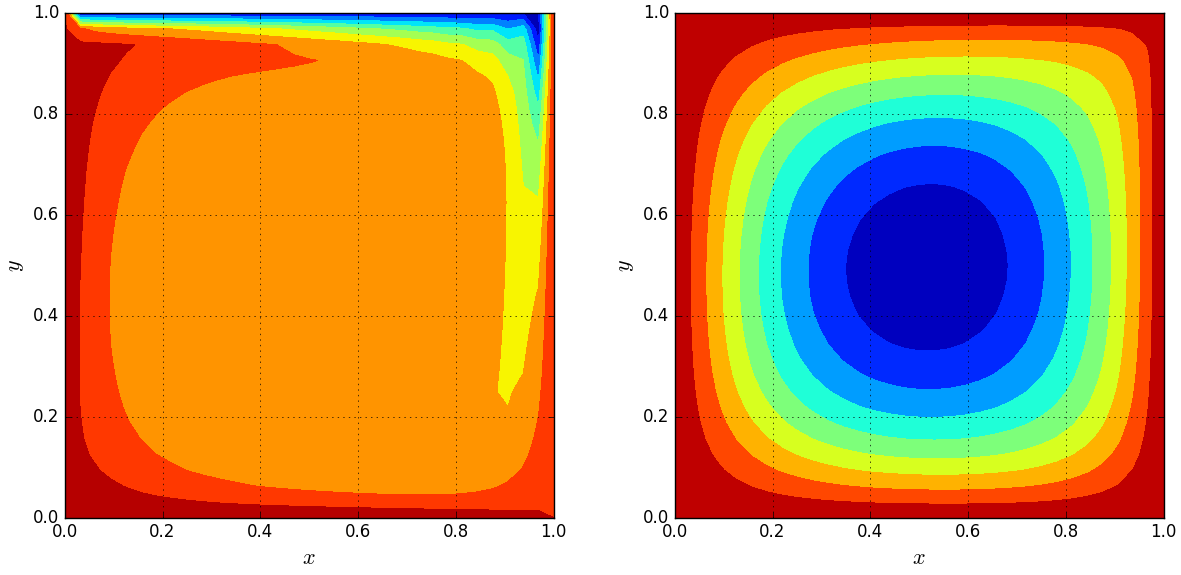
```
52      s = np.linspace(0.,1., N+1); h = s[1]-s[0]; h2 = h*h
53      [Y,X] = np.meshgrid(s,s)
54      O = np.zeros([N+1,N+1]); P = O.copy(); Q = O.copy()
55
56      # time stepping
57      t = 0.; dt0 = dt = min(0.25*h2/nu, 4.0*nu/Uwall**2); Nt = int(np.ceil(T/dt))
58
59      # Poisson solve matrix (5-point stencil)
60      A = Poisson(N, h)
61
62      # loop over time steps
63      for n in range(Nt):
64
65          # Poisson solve for stream function (P)
66          F = -np.reshape(O[1:N,1:N], ((N-1)*(N-1),1), order='F')
67          Pv = linalg.spsolve(A,F)     # solution at all points
68          P[1:N,1:N] = np.reshape(Pv, (N-1,N-1), order='F') # reshape into matrix
69
70          # vorticity update on boundary nodes
71          O[1:N,   0] = -2.*P[1:N,   1]/(h);                 # Bottom
72          O[1:N,   N] = -2.*P[1:N,   N]/(h2)-2./h*Uwall;  # Top
73          O[0  , 1:N] = -2.*P[1  , 1:N]/(h);                 # Left
74          O[N  , 1:N] = -2.*P[N  , 1:N]/(h2);                # Right
75
76          # vorticity update on interior nodes
77          for j in range(1,N):
78              for i in range(1,N):
79                  Q[i,j] = -0.25*((P[i,j+1]-P[i,j-1])*(O[i+1,j]-O[i-1,j]) \
80                                  - (P[i+1,j]-P[i-1,j])*(O[i,j+1]-O[i,j-1]))/h2 \
81                                  + nu*(O[i+1,j]+O[i-1,j]+O[i,j+1]+O[i,j-1]-4.*O[i,j])/h2
82
83          # print out the norm of the update to omega, stored in Q
84          print np.linalg.norm(Q)
85
86          # max velocity for time step calculation
87          U = abs(P[:,1:N+1]-P[:,0:N])/h
88          V = abs(P[1:N+1,:]-P[0:N,:])/h
89          vmax = max(np.max(np.max(U)) + np.max(np.max(V)), Uwall)
90          dt = min(0.25*h2/nu, 4.0*nu/vmax**2)
91
92          # Forward Euler time step
93          O[1:N,1:N] += dt*Q[1:N,1:N]
94          t = t+dt
95
96      return X, Y, O, P
97
98  def main():
99      X,Y,O,P = cavity(100., 32, 500.0)
100     plotstate(X,Y,P, '')
101
102 if __name__ == "__main__":
```

```
103     main()
```

Listing 7.2.2: Driver code for the functions in `cavitystream.py`.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from cavitystream import cavity,plotstate
4
5  def main():
6
7      X,Y,O,P = cavity(100., 32, 1000.0)
8      plotstate(X,Y,P, '../ figs /cavitystream_P.png')
9      plotstate(X,Y,O, '../ figs /cavitystream_O.png')
10
11 if __name__ == "__main__":
12     main()
```

Figure 7.2.3 shows contours of the vorticity and stream function for the case $Re = 100$ and $N = 32$. The contours of the stream function are streamlines, and these indicate that the flow rotates (clockwise) in response to the moving upper wall. The center of rotation is close to the center of the cavity; for higher Reynolds numbers, this location will move. The



(a) Vorticity, ω

(b) Stream function, ψ

Figure 7.2.3: Solution to the lid-driven cavity problem for $N = 32$ intervals in each direction and $Re = 100$.

vorticity plot exhibits more singular behavior. The vorticity is largest in the corners and

near boundaries. Inside most of the rest of the domain, the vorticity is constant and close to zero.

7.3 Primitive-Variable Approach

The vorticity stream-function approach is difficult to extend to 3D, and even in 2D, algorithms based on the primitive variables $\vec{v}, p$ are popular. These algorithms fall into two categories:

- A coupled approach (e.g. artificial compressibility)
- A pressure correction approach (pressure-based, uncoupled, segregated, sequential)

7.3.1 Artificial Compressibility (Chorin, 1967)

If only the steady-state solution is of interest, the $\nabla \cdot \vec{v} = 0$ condition can be relaxed in the transients by introducing a pressure time derivative term into the continuity equation:

$$p_t + \rho a^2 \nabla \cdot \vec{v} = 0, \tag{7.3.1}$$

where a is an artificial sound speed. At steady-state, $p_t = 0$, and the divergence-free condition will be enforced. Note that the elliptic Poisson equation has been eliminated in favor of the above parabolic equation. The resulting equations can be solved using standard time marching. Time-accurate calculations can also be carried out with this approach by iterating to pseudo-time convergence at each real (physical) time step.

How do we choose a? For the pure hyperbolic problem (negligible viscosity) the wave speeds are u, $u \pm \sqrt{u^2 + a^2}$ in the x-direction and v, $v \pm \sqrt{v^2 + a^2}$ in the y-direction. Consider now two extreme cases:

- If $a \gg \max(u, v)$: the resulting equations will be stiff, requiring very small time steps for explicit methods:

$$\Delta t \lesssim h/a, \;\; h = \text{mesh size}$$

- If $a \ll \max(u, v)$: the pressure waves may not propagate far enough to reasonably balance viscous effects, and the iteration may not converge.

Besides these rough guidelines, the precise value of a is not critical. Reasonable values can be $0.1 - 10$ times the maximum speed expected in the flow. In the viscous limit, the parabolic time step restriction, $\Delta t \leq h^2/(4\nu)$ is also required.

7.3.2 Pressure-Correction Methods

MAC (Harlow and Welch, 1965)

The Marker-And-Cell (MAC) method is one of the earliest pressure-correction schemes. The name refers to the fact that the original method used zero-mass marker particles to aid in visualizing the flow. This is not a critical part of the algorithm, but the name stuck. An important contribution of Harlow and Welch was the use of staggered variables to avoid even-odd coupling of the pressure (Section 7.3.4). This is sometimes called "MAC storage." At a basic level, the algorithm consists of

1. Compute $\vec{v}^*$ at the next time step from the momentum equation, using current values of velocity and pressure, $\vec{v}^n$ and p^n.

2. Solve the pressure Poisson equation (PPE)

$$\nabla^2 p^{n+1} = -\rho \nabla \cdot (\vec{v}^* \cdot \nabla \vec{v}^*) \tag{7.3.2}$$

3. Iterate steps 1 and 2 until the velocity field $\vec{v}^* \to \vec{v}^{n+1}$ is divergence free.

Alternatively, any divergence in the velocity field introduced by the first step can be recognized, and the PPE can be designed to remove it:

$$\nabla^2 p^{n+1} = -\rho \nabla \cdot (\vec{v}^* \cdot \nabla \vec{v}^*) - \frac{\nabla \cdot \vec{v}^*}{\Delta t} \tag{7.3.3}$$

The modified PPE is derived in the same way as the original PPE, but $(\nabla \cdot \vec{v})_t$ is not assumed zero. A forward Euler time discretization is employed.

Projection Method (Chorin and Temam, 1968)

This method is also known as the method of fractional steps. This is another "splitting" procedure, in which each advance in time is performed in two steps:

1. An intermediate velocity field, $\vec{v}^{n+\frac{1}{2}}$, is computed from $\vec{v}^n$ using the momentum equation, but *neglecting the pressure gradient term.*

2. The velocity field is then corrected: $\vec{v}^{n+\frac{1}{2}} \to \vec{v}^{n+1}$ with the step

$$\rho \frac{\vec{v}^{n+1} - \vec{v}^{n+\frac{1}{2}}}{\Delta t} + \nabla p^{n+1} = 0, \tag{7.3.4}$$

where ∇p^{n+1} enforces $\nabla \cdot \vec{v}^{n+1} = 0$. Taking the divergence of the above equation leads to the Poisson equation:

$$\nabla^2 p^{n+1} = \rho \frac{\nabla \cdot \vec{v}^{n+\frac{1}{2}}}{\Delta t}. \tag{7.3.5}$$

A provisional pressure can also be included in the first step, in which case the Poisson solution from the second step becomes a pressure correction.

SIMPLE (Caretto et al. and Patankar and Spalding, 1972)

The Semi-Implicit Method for Pressure Linked Equations (SIMPLE) is a name given to a family of methods in which a cyclic series of guess-and-correct operations is used to solve for the new pressures and velocities at each time step. The distinction from MAC and projection methods is in the way in which pressure and velocity corrections are achieved. The steps are:

1. Guess a pressure field p^*.
2. Solve the momentum equation *implicitly* to obtain an estimate for the next-level velocity, $\vec{v}^*$

$$\frac{\vec{v}^* - \vec{v}^n}{\Delta t} + \vec{v}^* \cdot \nabla \vec{v}^* + \frac{\nabla p^*}{\rho} = \nu \nabla^2 \vec{v}^* \tag{7.3.6}$$

3. Solve the pressure correction equation

$$\nabla^2 p' = \frac{\rho}{\Delta t}(\nabla \cdot \vec{v}^*) \tag{7.3.7}$$

4. Update the pressure and velocity guesses:

$$p^* = p^* + p', \quad \vec{v}^* = \vec{v}^* - \frac{\Delta t}{\rho}\nabla p' \tag{7.3.8}$$

5. Return to step 2 if not converged.

The pressure correction equation is derived as follows. Define the corrected pressure and velocity as

$$p = p^* + p', \quad \vec{v} = \vec{v}^* + \vec{v}'. \tag{7.3.9}$$

Write down the momentum equation using the current and corrected quantities:

$$\frac{\vec{v}^* - \vec{v}^n}{\Delta t} + \vec{v}^* \cdot \nabla \vec{v}^* + \frac{\nabla p^*}{\rho} = \nu \nabla^2 \vec{v}^* \tag{7.3.10}$$

$$\frac{\vec{v} - \vec{v}^n}{\Delta t} + \vec{v} \cdot \nabla \vec{v} + \frac{\nabla p}{\rho} = \nu \nabla^2 \vec{v} \tag{7.3.11}$$

Take the difference of these two and assume (the solution process is iterative, so we can get away with this) $\vec{v} \cdot \nabla \vec{v} - \nu \nabla^2 \vec{v} = \vec{v}^* \cdot \nabla \vec{v}^* - \nu \nabla^2 \vec{v}^*$:

$$\frac{\vec{v} - \vec{v}^*}{\Delta t} + \frac{\nabla p'}{\rho} = 0. \tag{7.3.12}$$

Taking the divergence of this equation, with $\nabla \cdot \vec{v} = 0$, gives the pressure correction equation used above.

PISO (Issa, 1985)

The Pressure-Implicit with Splitting Operators (PISO) method is largely implicit, with two corrector steps:

1. Predictor step. Solve for $\vec{v}^*$ *implicitly* using p^n:

$$\frac{\vec{v}^* - \vec{v}^n}{\Delta t} = H(\vec{v}^*) - \frac{\nabla p^n}{\rho}, \tag{7.3.13}$$

$$H(\vec{v}^*) = -\vec{v}^* \cdot \nabla \vec{v}^* + \nu \nabla^2 \vec{v}^*. \tag{7.3.14}$$

   Note, the above is just the momentum equation discretized in time, with the convective and diffusive fluxes lumped into one operator, H.

2. First corrector step. Solve for p^* and $\vec{v}^{**}$ such that

$$\frac{\vec{v}^{**} - \vec{v}^n}{\Delta t} = H(\vec{v}^*) - \frac{\nabla p^*}{\rho}, \tag{7.3.15}$$

   and $\nabla \cdot \vec{v}^{**} = 0$. Note, the velocity is treated explicitly. Taking the divergence of the above yields a pressure Poisson equation:

$$\nabla^2 p^* = \rho \nabla \cdot H(\vec{v}^*) + \frac{\rho}{\Delta t} \nabla \cdot \vec{v}^n. \tag{7.3.16}$$

   This equation is solved for p^*, from which $\vec{v}^{**}$ is calculated using the previous equation.

3. Second corrector step. This is essentially a re-application of the correction step. The explicitly-treated momentum equation is

$$\frac{\vec{v}^{***} - \vec{v}^n}{\Delta t} = H(\vec{v}^{**}) - \frac{\nabla p^{**}}{\rho}, \tag{7.3.17}$$

   and the associated pressure equation is

$$\nabla^2 p^{**} = \rho \nabla \cdot H(\vec{v}^{**}) + \frac{\rho}{\Delta t} \nabla \cdot \vec{v}^n. \tag{7.3.18}$$

More correction steps can be made, but two are usually sufficient.

7.3.3 Projection Method Implementation

Recall the steps in the projection method:

1. Compute an intermediate velocity field, $\vec{v}^{n+\frac{1}{2}}$, from $\vec{v}^n$ using the momentum equation, but *neglecting the pressure gradient term.*

2. Solve the pressure Poisson equation:

$$\nabla^2 p^{n+1} = \rho \frac{\nabla \cdot \vec{v}^{n+\frac{1}{2}}}{\Delta t}. \tag{7.3.19}$$

3. Correct the velocity field using the following equation:

$$\rho \frac{\vec{v}^{n+1} - \vec{v}^{n+\frac{1}{2}}}{\Delta t} + \nabla p^{n+1} = 0. \tag{7.3.20}$$

7.3.4 Staggered Storage

Storing each of the primitive variables u, v, p at the nodes of a regular grid ("collocated storage") makes straightforward discretizations susceptible to spurious oscillations. To address this problem, staggered storage is used to guarantee discrete well-posedness, as illustrated in Figure 7.3.1.

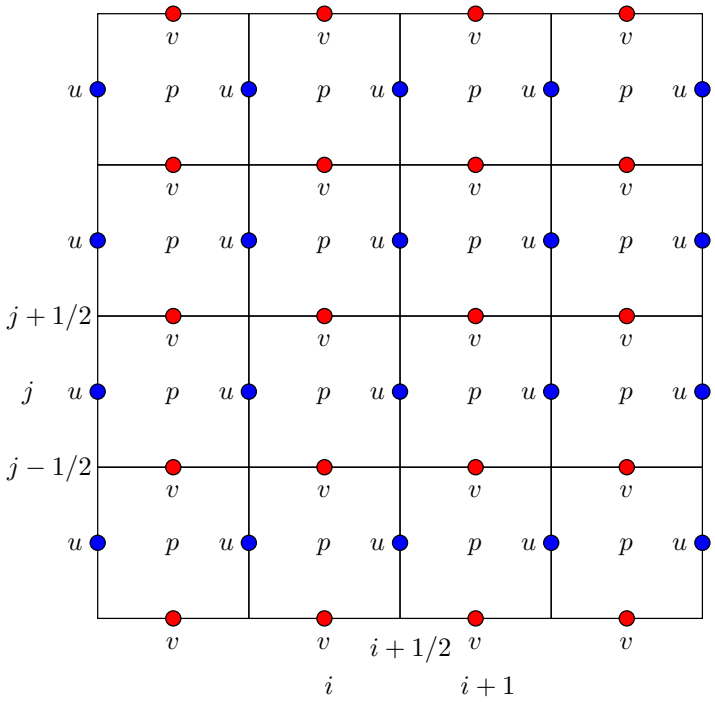


Figure 7.3.1: Staggered storage.

The *cells* are indexed with i, j, with half-integer indices for the grid lines, so that:

- The x-velocity $u_{i+\frac{1}{2},j}$ is stored at the midpoint of every vertical edge
- The y-velocity $v_{i,j+\frac{1}{2}}$ is stored at the midpoint of every horizontal edge
- The pressure $p_{i,j}$ is stored in the middle of each cell

This arrangement is possible on regular, axis-aligned grids, although it loses its advantage for curvilinear grids.

Consider a first-order elliptic system of equations for u, v:

$$\begin{aligned} u_x + v_y &= 0, \\ v_x - u_y &= 0. \end{aligned}$$

This system can easily be converted to one Laplace equation for a potential ($\phi : u = \phi_x, v = \phi_y$) or streamfunction ($\psi : u = \psi_y, v = -\psi_x$). However, the first-order system is representative of more complex equations, such as incompressible Navier-Stokes, which we would like

to analyze in primitive variables. On a *collocated grid* u, v are both stored at nodes of a regular grid. Central differencing yields

$$\begin{aligned} u_{j+1,k} - u_{j-1,k} + v_{j,k+1} - v_{j,k-1} &= 0, \\ v_{j+1,k} - v_{j-1,k} - u_{j,k+1} + u_{j,k-1} &= 0. \end{aligned}$$

To analyze Fourier modes admitted by these discrete equations, we write

$$u_{j,k} = \hat{u} \exp i(j\theta_x + k\theta_y), \qquad v_{j,k} = \hat{v} \exp i(j\theta_x + k\theta_y).$$

Substituting into the discrete equations gives

$$\sin^2 \theta_x + \sin^2 \theta_y = 0.$$

Low frequency modes satisfying this equation are consistent with Laplace's equation. However, there are three spurious solutions,

$$(\theta_x, \theta_y) = (0, \pi), \quad (\theta_x, \theta_y) = (\pi, 0), \quad (\theta_x, \theta_y) = (\pi, \pi),$$

that represent oscillatory/zebra and "checkerboard"/odd-even modes. For example, given a solution to the discrete equations above, adding 1 to all "red" points and subtracting 1 from all "black" points (checkerboard coloring) gives a solution that still satisfies the discrete equations. Numerically it is difficult to ensure that the solution procedure does not excite these modes.

In staggered storage, u is stored at the midpoint of every vertical edge while v is stored at the midpoint of every horizontal edge. The divergence equation $u_x + v_y = 0$ is enforced at every cell interior, while the curl equation $v_x - u_y = 0$ is enforced at every interior grid point. The matrix representing this discrete system can be shown to have non-zero determinant, yielding a discretely well-posed problem.

7.3.5 Conservation Equations

The two-dimensional incompressible Navier-Stokes equations can be written in conservation form,

$$\begin{aligned} u_t + (F + p)_x + H^x_y &= 0, \\ v_t + H^y_x + (G + p)_y &= 0, \end{aligned} \tag{7.3.21}$$

where we have assumed $\rho = 1$ to simplify the expressions, and

$$F = u^2 - \nu u_x, \quad G = v^2 - \nu v_y, \quad H^x = uv - \nu u_y, \quad H^y = uv - \nu v_x. \tag{7.3.22}$$

The fluxes F and G are stored at the cell centers, while H^x, H^y are stored at the grid nodes, as shown in Figure 7.3.2. Assume a forward Euler time discretization and neglect the

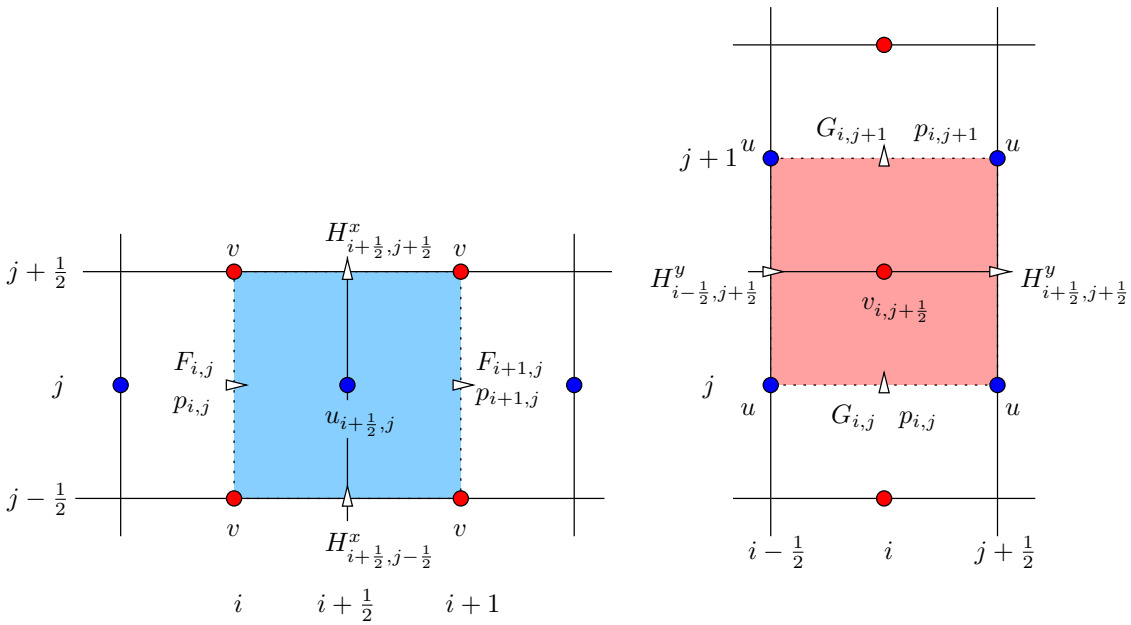


Figure 7.3.2: Storage of fluxes on a staggered grid.

pressure contribution for now (will correct in second step):

$$u^{n+\frac{1}{2}}_{i+\frac{1}{2},j} = u^n_{i+\frac{1}{2},j} - \Delta t \left(F_x|_{i+\frac{1}{2},j} + H^x_y|_{i+\frac{1}{2},j} \right) \tag{7.3.23}$$

$$v^{n+\frac{1}{2}}_{i,j+\frac{1}{2}} = v^n_{i,j+\frac{1}{2}} - \Delta t \left(H^y_x|_{i,j+\frac{1}{2}} + G_y|_{i,j+\frac{1}{2}} \right) \tag{7.3.24}$$

Updating $u_{i+\frac{1}{2},j}$ requires F_x, and H^x_y evaluated at $(i+\frac{1}{2}, j)$. Updating $v_{i,j+\frac{1}{2}}$ requires G_y, and H^y_x evaluated at $(i, j+\frac{1}{2})$. Use second-order accurate central differences:

$$\begin{aligned}
F_x|_{i+\frac{1}{2},j} &= \tfrac{1}{h}(F_{i+1,j} - F_{i,j}) &&\equiv \tfrac{1}{h}\delta_x|_{i+\frac{1}{2},j}\, F \\
H^x_y|_{i+\frac{1}{2},j} &= \tfrac{1}{h}(H^x_{i+\frac{1}{2},j+\frac{1}{2}} - H^x_{i+\frac{1}{2},j-\frac{1}{2}}) &&\equiv \tfrac{1}{h}\delta_y|_{i+\frac{1}{2},j}\, H^x \\
G_y|_{i,j+\frac{1}{2}} &= \tfrac{1}{h}(G_{i,j+1} - G_{i,j}) &&\equiv \tfrac{1}{h}\delta_y|_{i,j+\frac{1}{2}}\, G \\
H^y_x|_{i,j+\frac{1}{2}} &= \tfrac{1}{h}(H^y_{i+\frac{1}{2},j+\frac{1}{2}} - H^y_{i-\frac{1}{2},j+\frac{1}{2}}) &&\equiv \tfrac{1}{h}\delta_x|_{i,j+\frac{1}{2}}\, H^y
\end{aligned} \tag{7.3.25}$$

where $h = \Delta x = \Delta y$, and δ_x and δ_y are difference operators centered at the specified locations. The expression for updating $v_{i,j+\frac{1}{2}}$ is similar.

7.3.6 Residuals

When converging to steady state, we would like to monitor residuals. With the various equations appearing in the projection-based solution algorithm, residuals can be defined in various ways. One definition is based on the original conservation equations, in Equation 7.3.21. We

rewrite this system in vector form as

$$\frac{\partial \mathbf{v}}{\partial t} + \nabla \cdot \vec{\mathbf{F}} = \mathbf{0}, \tag{7.3.26}$$

where $\mathbf{v} = [u, v]^T$ and the total flux is

$$\vec{\mathbf{F}} = \begin{bmatrix} F + p \\ H^y \end{bmatrix} \hat{x} + \begin{bmatrix} H^x \\ G + p \end{bmatrix} \hat{y}. \tag{7.3.27}$$

Integrating Equation 7.3.26 over a control volume (area) A and applying the divergence form, we obtain

$$\frac{d\mathbf{v}_A}{dt} + \underbrace{\int_{\partial A} \vec{\mathbf{F}} \cdot \vec{n}\, dl}_{\mathbf{R}_A} = \mathbf{0}, \tag{7.3.28}$$

where $\mathbf{v}_A$ is the average velocity state inside A, and $\mathbf{R}_A$ is the flux residual for the control volume. When using staggered storage, two types of control volumes are most convenient for defining the flux residual: the two shaded regions in Figure 7.3.2. Call the left one $A_{i+\frac{1}{2},j}$, and the right one $A_{i,j+\frac{1}{2}}$. We then have two residuals,

$$\begin{aligned} R_{i+\frac{1}{2},j} &= \int_{\partial A_{i+\frac{1}{2},j}} \vec{\mathbf{F}} \cdot \vec{n}\, dl \\ &= h(F_{i+1,j} + p_{i+1,j} - F_{i,j} - p_{i,j}) + h(H^x_{i+\frac{1}{2},j+\frac{1}{2}} - H^x_{i+\frac{1}{2},j-\frac{1}{2}}) \qquad (7.3.29) \\ R_{i,j+\frac{1}{2}} &= \int_{\partial A_{i,j+\frac{1}{2}}} \vec{\mathbf{F}} \cdot \vec{n}\, dl \\ &= h(G_{i,j+1} + p_{i,j+1} - G_{i,j} - p_{i,j}) + h(H^y_{i+\frac{1}{2},j+\frac{1}{2}} - H^y_{i-\frac{1}{2},j+\frac{1}{2}}) \qquad (7.3.30) \end{aligned}$$

Note that $R_{i+\frac{1}{2},j}$ is associated with each vertical edge, while $R_{i,j+\frac{1}{2}}$ is associated with each horizontal edge. On a square mesh with $N \times N$ cells, there are $(N-1)N$ interior vertical edges, and $N(N-1)$ interior horizontal edges, and to obtain a single L_1 residual norm, we can sum the absolute values of all the $R_{i+\frac{1}{2},j}$ and $R_{i,j+\frac{1}{2}}$, via

$$|R|_{L_1} = \sum_{i=1}^{N-1} \sum_{j=1}^{N} |R_{i+\frac{1}{2},j}| + \sum_{i=1}^{N} \sum_{j=1}^{N-1} |R_{i,j+\frac{1}{2}}| \tag{7.3.31}$$

7.3.7 Pressure Equation

The pressure correction step is used to make the velocity field at $n+1$ divergence free. The pressure equation at i, j discretized with central differencing is

$$\delta_x^2|_{i,j} p^{n+1} + \delta_y^2|_{i,j} p^{n+1} = \frac{h}{\Delta t} \left(\delta_x|_{i,j} u^{n+\frac{1}{2}} + \delta_y|_{i,j} v^{n+\frac{1}{2}} \right), \tag{7.3.32}$$

where

$$\delta_x^2|_{i,j} p^{n+1} = p_{i-1,j}^{n+1} - 2p_{i,j}^{n+1} + p_{i+1,j}^{n+1}, \tag{7.3.33}$$

$$\delta_y^2|_{i,j} p^{n+1} = p_{i,j-1}^{n+1} - 2p_{i,j}^{n+1} + p_{i,j+1}^{n+1}. \tag{7.3.34}$$

Note that $u^{n+\frac{1}{2}}$ and $v^{n+\frac{1}{2}}$ are required precisely at the locations at which they are stored.

7.3.8 Boundary Conditions

At a wall, no slip requires the tangential fluid velocity to match the wall velocity. Zero flow-through requires the normal fluid velocity to be zero. Specifically, in our case of staggered storage, this means $H^x = uv - \nu u_y = 0$ on vertical walls and $H^y = uv - \nu v_x = 0$ on horizontal walls. However, we still need a pressure boundary condition and a means to evaluate H^x on horizontal boundaries and H^y on vertical boundaries. One way to do this is the *ghost-point* method, in which the fluid is assumed to extend through the wall, but in such a way that the required boundary conditions are satisfied.

On a horizontal wall, at $y = 0$, $u_x = 0$, so by continuity $v_y = -u_x = 0$. This means v is an even function of y, and its derivative $v_y = -u_x$ (and hence u) is an odd function of y, as illustrated in Figure 7.3.3. Thus, near the wall $\boxed{v = \frac{1}{2} v_{yy} y^2 + O(y^3), \quad u = U_{\text{wall}} + u_y y + O(y^2)}$

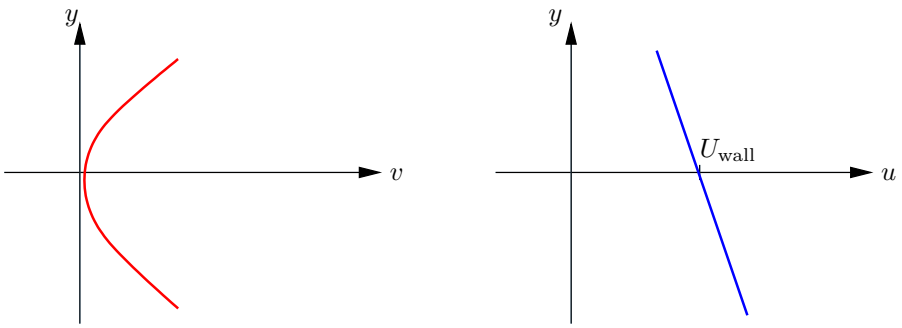


Figure 7.3.3: Behavior of the horizontal and vertical velocities on a horizontal wall boundary.

where v_{yy} and u_y are evaluated at $y = 0$.

Consider a horizontal bottom wall, padded with an additional layer of cells below the wall ($j = 0$), as shown in Figure 7.3.4. $u_{i+\frac{1}{2},0}$ is required for computing u_y in the viscous component of H^x at $y = 0$. Since, from above, u is an odd function of y,

$$u_{i+\frac{1}{2},1} - U_{\text{wall}} = U_{\text{wall}} - u_{i+\frac{1}{2},0} + O(h^2), \tag{7.3.35}$$

$$u_{i+\frac{1}{2},0} = 2U_{\text{wall}} - u_{i+\frac{1}{2},1} + O(h^2). \tag{7.3.36}$$

From the pressure correction equation, the pressure boundary condition is of the Neumman type. We discretize it with a centered difference:

$$\frac{p_{i,1}^{n+1} - p_{i,0}^{n+1}}{h} = \frac{v_{i,\frac{1}{2}}^{n+\frac{1}{2}} - v_{i,\frac{1}{2}}^{n+1}}{\Delta t}. \tag{7.3.37}$$

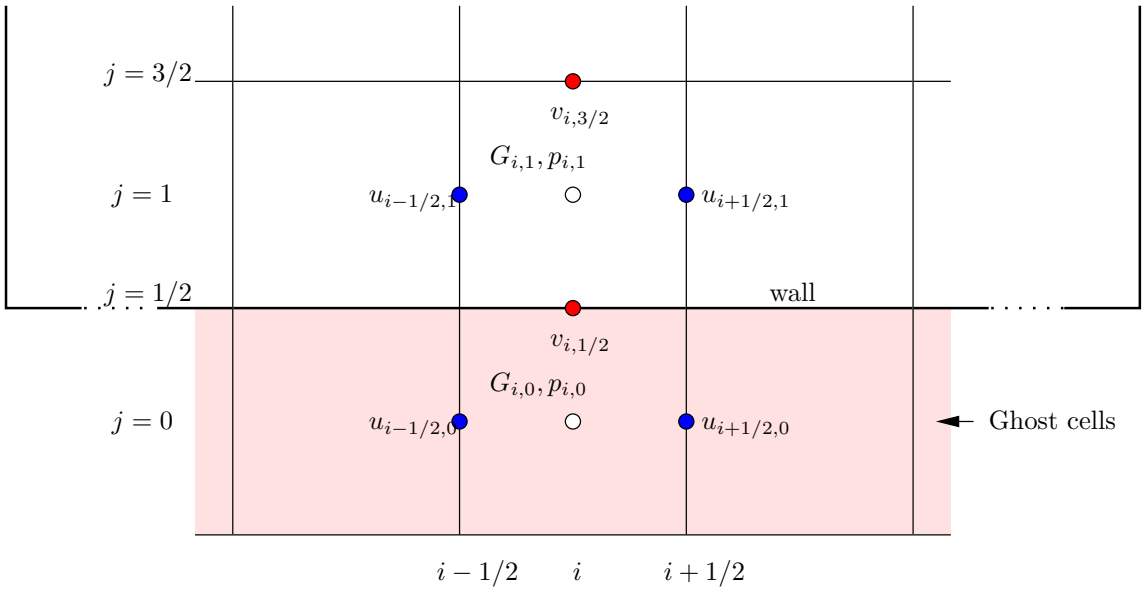


Figure 7.3.4: Quantities used in the enforcement of boundary conditions on a horizontal wall.

For zero flow normal to the wall, this reduces to

$$p_{i,0}^{n+1} = p_{i,1}^{n+1}. \tag{7.3.38}$$

This condition specifies the pressure just outside the domain boundary; it can be used in an iterative solution technique like successive over relaxation or multigrid. Conditions for the other walls can be derived in a similar manner.

7.3.9 Flux Evaluation

Recall the flux definitions

$$F = u^2 - \nu u_x, \quad G = v^2 - \nu v_y, \quad H^x = uv - \nu u_y, \quad H^y = uv - \nu v_x. \tag{7.3.39}$$

Simple central differencing/averaging for all the terms,

$$F_{i,j} = \frac{1}{2}(u_{i+\frac{1}{2},j}^2 + u_{i-\frac{1}{2},j}^2) - \frac{\nu}{h}\delta_x u \tag{7.3.40}$$

$$G_{i,j} = \frac{1}{2}(v_{i,j+\frac{1}{2}}^2 + v_{i,j-\frac{1}{2}}^2) - \frac{\nu}{h}\delta_y v \tag{7.3.41}$$

$$H^x_{i+\frac{1}{2},j+\frac{1}{2}} = \frac{u_{i+\frac{1}{2},j} + u_{i+\frac{1}{2},j+1}}{2}\frac{v_{i,j+\frac{1}{2}} + v_{i+1,j+\frac{1}{2}}}{2} - \frac{\nu}{h}\delta_y|_{i+\frac{1}{2},j+\frac{1}{2}} u \tag{7.3.42}$$

$$H^y_{i+\frac{1}{2},j+\frac{1}{2}} = \frac{u_{i+\frac{1}{2},j} + u_{i+\frac{1}{2},j+1}}{2}\frac{v_{i,j+\frac{1}{2}} + v_{i+1,j+\frac{1}{2}}}{2} - \frac{\nu}{h}\delta_x|_{i+\frac{1}{2},j+\frac{1}{2}} v \tag{7.3.43}$$

will result in, just like for the FTCS method, unphysical oscillations for any reasonable Reynolds number (cell Reynolds number > 2).

Averaging is appropriate for the viscous terms, but not for the convection terms. Simple upwinding of the entire quantities u^2 and v^2 results in the *donor cell* method, which is overly dissipative. In fact, for cell Reynolds number > 1, the numerical dissipation will drown out the physical dissipation.

A closer look at the inviscid fluxes reveals

$$\begin{array}{lcll} F & : & uu & = \text{ transport of } u \text{ in the } x\text{-direction} \\ G & : & vv & = \text{ transport of } v \text{ in the } y\text{-direction} \\ H^x & : & vu & = \text{ transport of } u \text{ in the } y\text{-direction} \\ H^y & : & uv & = \text{ transport of } v \text{ in the } x\text{-direction} \end{array}$$

All of these are in the form: flux $= q\phi$, where q is the transport velocity (u when in the x-direction and v when in the y direction) and ϕ is the transported quantity. Thus, a reasonable approach is:

- Evaluate q by averaging the transport velocity
- Upwind ϕ according to the sign of q

For example, to evaluate the convective component u^2 for $F_{i,j}$, take $q_{i,j} = (u_{i-\frac{1}{2},j} + u_{i+\frac{1}{2},j})/2$:

- if $q_{i,j} > 0$,

$$\phi_{i,j} = \frac{3u_{i+\frac{1}{2},j} + 6u_{i-\frac{1}{2},j} - u_{i-\frac{3}{2},j}}{8} \tag{7.3.44}$$

- if $q_{i,j} \leq 0$,

$$\phi_{i,j} = \frac{3u_{i-\frac{1}{2},j} + 6u_{i+\frac{1}{2},j} - u_{i+\frac{3}{2},j}}{8} \tag{7.3.45}$$

The above high-order upwinding constitutes a scheme called Quadratic Upstream Interpolation for Convection Kinematics (*QUICK*).

7.3.10 Limiting

The QUICK method is a great improvement over simple upwinding, but still suffers from oscillations at high Reynolds numbers, especially for data that is nearly discontinuous. This is expected from Godunov's theorem.

As in TVD methods for compressible flow, we address this problem via limiting. Suppose we have three consecutive data values on a regular grid: $\phi_{j-1}, \phi_j, \phi_{j+1}$, and we are interested in evaluating $\phi_{j+\frac{1}{2}}$ (i.e. upwinding with $q_{j+\frac{1}{2}} > 0$), as illustrated in Figure 7.3.5.

Define a smoothness monitor,

$$\hat{\phi}_j = \frac{\phi_j - \phi_{j-1}}{\phi_{j+1} - \phi_{j-1}}. \tag{7.3.46}$$

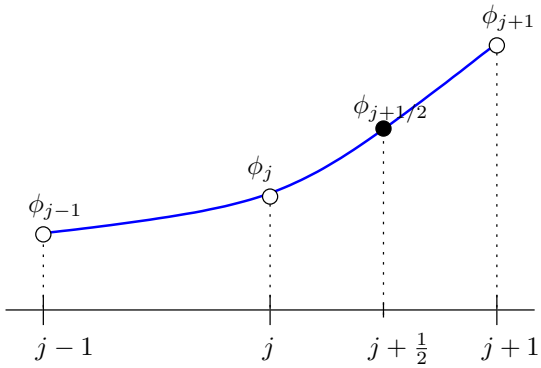


Figure 7.3.5: Evaluation of $\phi_{j+\frac{1}{2}}$ from consecutive data.

$\hat{\phi}_j = 1/2$ denotes smooth data, while $\hat{\phi}_j$ outside of $[0,1]$ denotes non-monotone data. We need a limiter that takes $\hat{\phi}_j$ as input and produces the output

$$\hat{\phi}_{j+\frac{1}{2}} = \frac{\phi_{j+\frac{1}{2}} - \phi_{j-1}}{\phi_{j+1} - \phi_{j-1}}. \tag{7.3.47}$$

One option is the SMART (Simple Monotone Advection for Realistic Transport) limiter (Gaskell and Lau, 1988), which takes the form

$$\hat{\phi}_{j+\frac{1}{2}} = \begin{cases} \hat{\phi}_j & \hat{\phi}_j < 0 \quad \text{or} \quad \hat{\phi}_j > 1 \\ 3\hat{\phi}_j & 0 < \hat{\phi}_j < \frac{1}{6} \\ \frac{3}{8}(2\hat{\phi}_j + 1) & \frac{1}{6} < \hat{\phi}_j < \frac{5}{6} \\ 1 & \frac{5}{6} < \hat{\phi}_j < 1 \end{cases} \tag{7.3.48}$$

The theory of limiters for incompressible flow is very similar to that of compressible flow. The underlying idea of preserving monotonicity is the same. The main difference is a different language: whereas for compressible flow we used the slope ratio, r, as a smoothness monitor, for incompressible flow the smoothness monitor is defined as above. The information it contains is the same, but it is plotted differently. Specifically, plotting $\hat{\phi}_{j+\frac{1}{2}}$ versus $\hat{\phi}_j$ results in what's called a normalized variable diagram instead of the Sweby diagram.

7.3.11 Post-processing

A simple technique for visualizing the streamlines on a regular staggered grid consists of plotting contours of the stream function ψ. Recall that

$$\frac{\partial \psi}{\partial y} = u, \qquad \frac{\partial \psi}{\partial x} = -v. \tag{7.3.49}$$

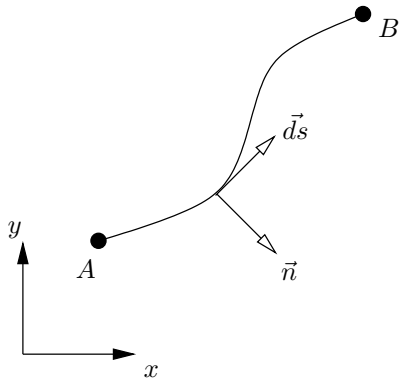


Figure 7.3.6: Net flow between points A and B causes a change in the stream function, ψ, between these two points.

Integrating the stream function on an arbitrary path between points A and B in (x, y), see Figure 7.3.6, gives:

$$\begin{aligned}
\psi(B) - \psi(A) &= \int_A^B \nabla\psi \cdot \vec{ds} \\
&= \int_A^B \frac{\partial\psi}{\partial x} ds_x + \frac{\partial\psi}{\partial y} ds_y \\
&= \int_A^B (-\frac{\partial\psi}{\partial x} n_y + \frac{\partial\psi}{\partial y} n_x) ds \\
&= \int_A^B (v n_y + u n_x) ds \\
&= \int_A^B \vec{v} \cdot \vec{n} ds
\end{aligned} \tag{7.3.50}$$

where $\vec{n} = [n_x, n_y]$ is the normal to the path, and $\vec{ds} = [ds_x, ds_y] = [-n_y, n_x]ds$ is the tangent to the path. The above equation says that the volume flow rate through any path joining points A and B will be the same, and equal to $\psi(B) - \psi(A)$.

We can use this property of the stream function to evaluate it discretely. First, pick one corner, A, of the grid and set $\psi = 0$ there. Next, to calculate ψ at any other point B on the grid, connect B to A via any sequence of horizontal and vertical edges. Since we have u at the midpoints of vertical edges, and v at the midpoints of horizontal edges, we can calculate a discrete flow rate through any edge. Summing these up (making sure we stay consistent in our normal definition) will give us $\psi(B)$. The order of edges (path taken) does not matter as long as the velocity field is divergence free, in the discrete sense. Streamlines can then be visualized by plotting the contours of ψ.

7.3.12 Summary

To review, the steps involved in implementing a projection-based primitive-variable solution algorithm on a staggered mesh are:

1. Initialize velocity and pressure fields. Set $n = 0$.

2. Calculate u^n and v^n on boundary "ghost" points using the appropriate boundary conditions.

3. Calculate fluxes F, G, H^x, H^y from the current velocity field.

4. Compute a fractional-step update to the velocity field, not using the pressure:

$$u^{n+\frac{1}{2}}_{i+\frac{1}{2},j} = u^n_{i+\frac{1}{2},j} - \Delta t \left(F_x|_{i+\frac{1}{2},j} + H^x_y|_{i+\frac{1}{2},j} \right) \tag{7.3.51}$$

$$v^{n+\frac{1}{2}}_{i,j+\frac{1}{2}} = v^n_{i,j+\frac{1}{2}} - \Delta t \left(H^y_x|_{i,j+\frac{1}{2}} + G_y|_{i,j+\frac{1}{2}} \right) \tag{7.3.52}$$

5. Solve the pressure Poisson equation,

$$\nabla^2 p^{n+1} = \frac{\nabla \cdot \vec{v}^{n+\frac{1}{2}}}{\Delta t}, \tag{7.3.53}$$

   which in discretized form for cell i, j is Equation 7.3.32.

6. Correct the velocity field using Equation 7.3.4,

$$\frac{\vec{v}^{n+1} - \vec{v}^{n+\frac{1}{2}}}{\Delta t} + \nabla p^{n+1} = 0, \tag{7.3.54}$$

   which in discretized form reads

$$u^{n+1}_{i+\frac{1}{2},j} = u^{n+\frac{1}{2}}_{i+\frac{1}{2},j} - \frac{\Delta t}{h}(p_{i+1,j} - p_{i,j}) \tag{7.3.55}$$

$$v^{n+1}_{i,j+\frac{1}{2}} = v^{n+\frac{1}{2}}_{i,j+\frac{1}{2}} - \frac{\Delta t}{h}(p_{i,j+1} - p_{i,j}) \tag{7.3.56}$$

7. If converging to steady-state, calculate the L_1 residual norm via Equation 7.3.31, and terminate the iterations if the residual norm is below a prescribed tolerance.

8. Set $n \to n + 1$ and return to step 2.

A convergence criterion is necessary on the overall algorithm when converging to steady state.

Chapter 8

Appendix

8.1 More Codes

Listing 8.1.1: Computes the temperature in a heated 2D plate with Dirichlet boundary conditions. Uses an efficient construction of the sparse matrix.

```
import numpy as np
from scipy import sparse
from scipy.sparse import linalg

def source(x, y, Lx, Ly, kappa):
    return np.sin(np.pi*x/Lx)*(-(np.pi/Lx)**2*y/Ly*(1-y/Ly) - 2/Ly**2)*(-kappa)

def heat2d(Lx, Ly, Nx, Ny, kappa):
    dx = float(Lx)/Nx; dy = float(Ly)/Ny;  # spacing
    x = np.linspace(0, Lx, Nx+1)  # x nodes
    y = np.linspace(0, Ly, Ny+1)  # y nodes
    Y, X = np.meshgrid(y, x)      # matrices of all nodes
    N = (Nx+1)*(Ny+1) # total number of unknowns
    nnz = (4+2*(Nx-1)+2*(Ny-1))*1 + (Nx-1)*(Ny-1)*5 # number of nonzeros
    data = np.zeros(nnz, dtype=np.float);
    irow = np.zeros(nnz, dtype=np.int); icol = np.zeros(nnz, dtype=np.int)
    q = np.zeros(N)                 # empty rhs vector

    # fill in interior contributions
    inz = 0
    for iy in range(1,Ny):
        for ix in range(1,Nx):
            i = iy*(Nx+1)+ix; x = ix*dx; y = iy*dy
            iL = i-1; iR = i+1; iD = i-(Nx+1); iU = i+(Nx+1)
            q[i] = source(x,y,Lx,Ly,kappa)/kappa
            I = range(inz+0,inz+5)
            irow[I] = i
            icol[I] = [i, iL, iR, iD, iU]
            data[I] = [2./dx**2 + 2./dy**2, -1./dx**2, -1./dx**2, -1./dy**2, -1./dy**2]
            inz += 5

```

```
32      # enforce Dirichlet boundary conditions
33      for ix in range(Nx+1):
34          irow[inz] = icol[inz] = ix; data[inz] = 1.; inz+=1
35          irow[inz] = icol[inz] = Ny*(Nx+1)+ix; data[inz] = 1.; inz+=1
36      for iy in range(1,Ny):
37          irow[inz] = icol[inz] = iy*(Nx+1); data[inz] = 1.; inz+=1
38          irow[inz] = icol[inz] = iy*(Nx+1)+Nx; data[inz] = 1.; inz+=1
39
40      # build sparse matrix
41      A = sparse.csr_matrix((data, (irow, icol )), shape=(N,N))
42
43      # solve system
44      Tv = linalg.spsolve(A,q)     # solution at all points
45      T = np.reshape(Tv, (Nx+1,Ny+1), order='F') # reshape into matrix
46
47      return X, Y, T
48
49  def main():
50      X, Y, T = heat2d(2.0, 1.0, 4, 3, 0.5)
51
52  if __name__ == "__main__":
53      main()
```