NOTES AND STUDIES IN SPECTRAL METHODS

Computational Études

A Spectral Approach

Dr. Denys Dutykh

Mathematics Department
College of Computing and Mathematical Sciences
Khalifa University of Science and Technology
Abu Dhabi, UAE

2026

Contents

Preface

The purpose of this book is twofold: to explain to students why spectral methods work and why they are so remarkably efficient, and to teach them how to implement these methods using modern computational tools.

This text is conceived as a collection of **Computational Études**. In musical education, an étude is a composition designed to perfect a specific technique (be it rapid scales, complex arpeggios, or delicate phrasing) while remaining a pleasing piece of music in its own right. Similarly, each chapter here presents a focused mathematical concept paired with its computational realization: a study that is both instructive and complete.

Who Is This Book For?

We have written primarily for graduate students in applied mathematics, physics, and engineering who seek a hands-on understanding of spectral methods. The reader should be comfortable with calculus, linear algebra, and basic programming. Prior exposure to numerical methods is helpful but not essential; we build the necessary foundations as we proceed.

How Is the Book Organized?

Each chapter is designed to be largely self-contained. We begin with fundamental concepts (interpolation and differentiation) before advancing to time-stepping schemes and applications. The mathematical exposition is deliberately concise, favoring clarity over exhaustive rigor. Proofs are included when they illuminate; otherwise, we direct the reader to authoritative references.

Reproducible Science

This book is also an experiment in **reproducible science**. Every figure, every table, every numerical result you see in these pages was generated by code available in the accompanying repository. We provide implementations in both Python and MATLAB, allowing readers to choose their preferred environment. The Python code emphasizes accessibility and integration with the open-source scientific ecosystem; the MATLAB code leverages its historical significance in numerical computing and, where appropriate, the Advanpix Multiprecision Computing Toolbox for extended precision arithmetic.

How to Use This Book

We invite you to treat this book not as a static reference, but as a workshop. Clone the repository, run the scripts, modify the parameters, break the code, and fix it. That is the only way to truly master the art of spectral methods.

The complete source code and the Typst manuscript are available at:
https://github.com/dutykh/computational-etudes/

Dr. Denys Dutykh

Abu Dhabi, UAE
January 2026

Acknowledgements

I would like to express my sincere gratitude to the students who took this course, carefully reread the manuscript, and helped me present it in a clean and polished form. Their diligent attention to detail and thoughtful feedback have been invaluable.

In particular, I thank:

- **Farrell Adriano**
- **Aziz Sharofov**
- **Mark Essa Sukaiti**

Their contributions have significantly improved the quality of this book.

CHAPTER 1

Introduction

Differential equations serve as the fundamental language of the physical sciences, describing phenomena ranging from the propagation of sound waves to the flow of heat and the dynamics of fluids. Finding exact analytical solutions to these equations is a luxury rarely afforded in practical applications. Consequently, the scientist and the engineer must turn to numerical approximation.

Broadly speaking, numerical methods for differential equations fall into two categories: local methods and global methods. The former, including Finite Difference and Finite Element Methods, approximate the unknown solution using functions that are non-zero only on small sub-domains (elements or grid stencils). These methods are robust and flexible, handling complex geometries with grace. However, their accuracy is typically algebraic; refining the grid by a factor of two might improve the error by a factor of four or eight, but rarely more. From a computational perspective, local methods are *myopic*: to compute a derivative at a grid point, they look only at immediate neighbors.

Spectral methods represent the global approach. They approximate the solution as a linear combination of continuous, global basis functions, typically trigonometric polynomials (Fourier series) for periodic problems or Chebyshev polynomials for non-periodic ones. In stark contrast to local schemes, spectral methods are *holistic*: the derivative at any single point depends on the function values at *every other point* in the domain. Mathematically, this is equivalent to fitting a single high-degree polynomial through all data points. This global coupling is what allows information to propagate instantly across the grid, granting us the remarkable convergence that we call "spectral accuracy."

There is, however, a unifying viewpoint that bridges these apparently distinct philosophies. Pseudospectral methods can be understood as the natural limit of finite difference methods as the order of accuracy increases without bound. Imagine a sequence of finite difference stencils: first using two neighbors, then four, then eight, and so on. As the stencil width grows to encompass the entire grid, the local method transforms smoothly into a global one. This perspective, developed by Kreiss and Oliger and elaborated by Fornberg, offers both intuition and practical computational strategies. The theory and practice of spectral methods are comprehensively developed in the classical texts by [1], [2], and [3].

1.1 The Spectral Promise

The fundamental argument for spectral methods is one of efficiency. If the solution to a problem is smooth, the coefficients of its expansion in a proper global basis decay exponentially fast. This phenomenon is known as spectral accuracy.

In practical terms, this means that spectral methods can achieve a level of precision with a few dozen degrees of freedom that a finite difference scheme might require thousands of grid points to match. While a fourth-order finite difference method implies that the error $\varepsilon \sim O(N^{-4})$, a spectral method boasts $\varepsilon \sim O(c^{-N})$ for some constant $c > 1$. When the solution is analytic, the convergence is explosive; the error drops into the "spectral valley" until it hits the floor of machine precision.

Beyond raw accuracy, spectral methods possess another virtue that is often decisive in physical applications: they are virtually free of both dissipative and dispersive numerical errors. Finite difference schemes, by their very nature, introduce artificial dissipation that can overwhelm the true physical dissipation in problems such as high-Reynolds number fluid flows. They also suffer from dispersive errors that cause different frequency components to propagate

at slightly different speeds, turning sharp gradients into spurious wavetrains. Spectral methods avoid both pathologies. This fidelity to the underlying physics explains their dominance in demanding applications: turbulence modeling, global weather and climate prediction, nonlinear wave dynamics, and seismic analysis all rely heavily on spectral techniques.

This global dependence has a computational consequence: spectral differentiation matrices are *dense*, not sparse. Where a finite difference scheme produces banded matrices that are cheap to store and invert, spectral methods fill in every entry. However, the extraordinary accuracy means we need so few points (often just dozens where finite differences would require thousands) that we can afford this density. The cost per degree of freedom is higher, but the total cost for a given accuracy is dramatically lower.

However, this power is not without its price. Spectral methods are unforgiving regarding grid placement. We cannot simply choose points where we please; for non-periodic problems, the mathematics dictates that points must cluster at boundaries (the celebrated Chebyshev points) to prevent the interpolation from diverging. Attempting high-degree polynomial interpolation on an equispaced grid leads to the notorious Runge phenomenon, where oscillations grow without bound near the boundaries. This sensitivity to geometry is what restricts spectral methods primarily to simple domains, but within those domains, they reign supreme.

This book aims to demystify this "spectral magic." We will see that it is not magic at all, but a direct consequence of the smoothness of the underlying functions and the careful choice of basis and grid.

1.2 A Brief History

Spectral representations have been used for analytic studies of differential equations since the days of Fourier in the early nineteenth century. The idea of employing them for *numerical* computation, however, emerged much later. Lanczos, in the 1930s, pioneered the use of Chebyshev expansions for solving ordinary differential equations numerically. This approach remained somewhat academic until the early 1970s, when Kreiss and Oliger introduced the pseudospectral method for partial differential equations. Their innovation was to work with function values at grid points rather than expansion coefficients, dramatically simplifying the treatment of nonlinear terms. The practical viability of these methods was ensured by the fast Fourier transform algorithm, rediscovered by Cooley and Tukey in 1965, which reduced the cost of transforming between physical and spectral space from $O(N^2)$ to $O(N \log N)$ operations. This confluence of theoretical insight and algorithmic efficiency launched spectral methods into the mainstream of computational science.

1.3 Limitations and Trade-offs

Honesty compels us to acknowledge that spectral methods are not a panacea. Several factors can limit their applicability or efficiency:

- **Boundary conditions** can be awkward to impose, particularly for problems with complex constraints or time-dependent boundaries.

- **Irregular domains** resist the tensor-product structure that makes spectral methods efficient. While domain decomposition and mapping techniques exist, they sacrifice some of the method's elegance.
- **Strong shocks and discontinuities** violate the smoothness assumptions that underlie spectral accuracy. The Gibbs phenomenon produces persistent oscillations near discontinuities, requiring filtering or other remediation.
- **Variable resolution requirements** across a large domain are difficult to accommodate. Unlike adaptive mesh refinement in finite element methods, spectral grids are inherently uniform in their polynomial degree.

These limitations explain why finite element and finite difference methods continue to thrive in many applications. The art lies in recognizing when spectral methods are the right tool. When the geometry is simple, the solution is smooth, and high accuracy is paramount, no other approach comes close.

1.4 The Philosophy of "Études"

The title of this volume, Computational Études, reflects a specific pedagogical philosophy. In musical education, an étude is a composition designed to practice a particular technical skill (be it rapid scales or complex arpeggios) while remaining a pleasing piece of music in its own right.

In this text, our "technical skills" are not rapid scales or arpeggios, but rather handling stiffness in time-stepping, managing aliasing in nonlinear products, enforcing boundary conditions through tau methods or lifting functions, and filtering spurious oscillations. Just as a Chopin Étude transforms a technical exercise into art, a well-written spectral code transforms a mathematical formula into a robust simulation. The études collected here are designed to cultivate this virtuosity.

We approach spectral methods not through dry, abstract theorems, but through concrete, self-contained studies. Each chapter focuses on a specific mathematical concept (interpolation, differentiation, aliasing, or time-stepping) and explores it through a compact, runnable implementation.

We deliberately restrict our focus primarily to one-dimensional problems. This choice is strategic. The mathematical essence of spectral methods (the treatment of boundaries, the distribution of collocation points, and the structure of differentiation matrices) is fully present in one dimension. Extending these ideas to two or three dimensions usually involves tensor products, which add significant programming overhead without necessarily adding new conceptual depth. By staying in 1D, we keep our code short, readable, and focused on the physics and mathematics.

1.5 Collocation: Computing in Physical Space

While the theory of spectral methods relies on orthogonal expansions (Fourier series for periodic problems, Chebyshev series otherwise), the actual computation often proceeds differently. Rather than manipulating expansion coefficients directly (the *modal* or *Galerkin* approach), we

typically work with function values at carefully chosen grid points (the *nodal* or *collocation* approach, also called *pseudospectral*).

The collocation philosophy dominates this book for a practical reason: it handles nonlinear terms with ease. When the governing equation contains products like $u \cdot u_x$, the modal approach requires computing convolutions of coefficient sequences, a tedious operation. Collocation simply evaluates the product pointwise on the grid. This directness makes pseudospectral methods the tool of choice for most computational applications, and it is the approach we shall master through these études.

The reader should be aware that both viewpoints (modal and nodal) illuminate the same underlying mathematics. The Fast Fourier Transform provides the bridge, allowing us to move efficiently between coefficient space and physical space as needed.

1.6 A Modern Workflow

Finally, this book is an experiment in reproducible science. The days of presenting numerical results as static, unverifiable images are passing. The results you see in these pages were generated by the code available in the accompanying repository. We utilize a dual-language approach:

- **Python**: For accessibility and integration with the vast open-source scientific ecosystem.
- **Matlab**: For its historical significance in this field and its concise matrix syntax, often utilizing the Advanpix Multiprecision Computing Toolbox to explore phenomena that lie beyond standard double precision.

We invite you to treat this book not as a static reference, but as a workshop. Run the scripts, change the parameters, break the code, and fix it. That is the only way to truly learn the art of spectral methods.

CHAPTER 2

Classical Second Order PDEs and Separation of Variables

In this opening chapter we derive exact solutions for three classical linear partial differential equations: the *heat equation* (parabolic), the *wave equation* (hyperbolic), and the *Laplace equation* (elliptic). These solutions are found by the *method of separation of variables*, which expresses the solution as an infinite series of eigenfunctions.
Why begin a book on *numerical* methods with *analytical* solutions? Because separation of variables is the theoretical ancestor of spectral methods. When we later truncate these infinite series at some finite N and compute with only the first N modes, we are doing exactly what a spectral solver does, but with pen and paper first. This chapter thus serves as the conceptual bridge between classical analysis and modern computation. The methods presented here are classical and thoroughly developed in standard references such as [4].

We treat three model problems:

- heat equation with periodic boundary conditions in one spatial dimension,
- wave equation on a bounded interval,
- Laplace equation in a simple domain.

We begin with a complete analytic solution of the heat equation. The other two examples will follow the same pattern.

2.1 Heat Equation with Periodic Boundary Conditions

We consider the one dimensional heat equation on the interval $[0, 2\pi]$ with periodic boundary conditions. The unknown $u(x, t)$ represents, for example, the temperature at point x and time t.

The problem is

$$\frac{\partial u}{\partial t}(x,t) = \frac{\partial^2 u}{\partial x^2}(x,t), \quad x \in [0, 2\pi],\ t > 0,$$

with periodic boundary conditions

$$u(x + 2\pi, t) = u(x, t)$$

for all real x and all $t > 0$, and initial condition

$$u(x, 0) = f(x), \quad x \in [0, 2\pi].$$

We assume that f is smooth and 2π periodic:

$$f(x + 2\pi) = f(x).$$

Our goal is to obtain an explicit representation of $u(x, t)$ as an infinite series and to see how separation of variables leads naturally to a Fourier series in space. The technique we employ here follows the classical approach presented in [4].

2.1.1 Step 1: Separation Ansatz

We look for nontrivial solutions of the form

$$u(x, t) = X(x) \cdot T(t),$$

where X depends only on x and T depends only on t.

Substituting into the PDE gives

$$X(x) \cdot T'(t) = X''(x) \cdot T(t).$$

We assume X and T are not identically zero, so we can divide both sides by $X(x) \cdot T(t)$:

$$\frac{T'(t)}{T(t)} = \frac{X''(x)}{X(x)}.$$

The left side depends only on t, the right side only on x. Therefore both sides must be equal to the same constant, which we denote by $-\lambda$:

$$\frac{T'(t)}{T(t)} = \frac{X''(x)}{X(x)} = -\lambda.$$

We obtain two ordinary differential equations:

$$T'(t) + \lambda T(t) = 0,$$
$$X''(x) + \lambda X(x) = 0.$$

The periodic boundary conditions for u imply periodic conditions for X:

$$X(0) = X(2\pi), \quad X'(0) = X'(2\pi).$$

We have arrived at a spatial eigenvalue problem for X. The systematic treatment of such eigenvalue problems is a cornerstone of the theory of partial differential equations; see [4] for a comprehensive exposition.

2.1.2 Step 2: Spatial Eigenvalue Problem with Periodic Boundary Conditions

We now solve

$$X''(x) + \lambda X(x) = 0,$$

with

$$X(0) = X(2\pi), \quad X'(0) = X'(2\pi).$$

We consider three cases: $\lambda < 0$, $\lambda = 0$, and $\lambda > 0$.

Case 1: $\lambda < 0$

Write $\lambda = -\mu^2$ with $\mu > 0$. The equation becomes

$$X''(x) - \mu^2 X(x) = 0.$$

The general solution is

$$X(x) = Ae^{\mu x} + Be^{-\mu x}.$$

Imposing periodicity $X(0) = X(2\pi)$ gives

$$A + B = Ae^{2\mu\pi} + Be^{-2\mu\pi}.$$

Imposing $X'(0) = X'(2\pi)$ gives

$$\mu(A - B) = \mu(Ae^{2\mu\pi} - Be^{-2\mu\pi}).$$

Since $\mu > 0$, we can divide by μ and rewrite both conditions as a homogeneous linear system:

$$A(1 - e^{2\mu\pi}) + B(1 - e^{-2\mu\pi}) = 0,$$
$$A(1 - e^{2\mu\pi}) - B(1 - e^{-2\mu\pi}) = 0.$$

Adding these two equations gives

$$2A(1 - e^{2\mu\pi}) = 0.$$

Since $\mu > 0$, we have $e^{2\mu\pi} > 1$, so $1 - e^{2\mu\pi} \neq 0$. Therefore $A = 0$.

Subtracting the second equation from the first gives

$$2B(1 - e^{-2\mu\pi}) = 0.$$

Since $\mu > 0$, we have $e^{-2\mu\pi} < 1$, so $1 - e^{-2\mu\pi} \neq 0$. Therefore $B = 0$.

Hence the only solution is $A = B = 0$, the trivial solution. There are no nontrivial periodic eigenfunctions for $\lambda < 0$, and we discard this case.

Case 2: $\lambda = 0$

The equation reduces to

$$X''(x) = 0.$$

Its general solution is

$$X(x) = A + Bx.$$

Periodicity $X(0) = X(2\pi)$ gives

$$A = A + 2\pi B$$

so $B = 0$. Then $X(x) = A$ is constant.

The derivative is $X'(x) = 0$, so $X'(0) = X'(2\pi)$ is automatically satisfied.

Thus $\lambda = 0$ gives one eigenfunction

$$X_0(x) = 1$$

(up to a multiplicative constant).

Case 3: $\lambda > 0$

Write $\lambda = k^2$ with $k > 0$. The equation becomes

$$X''(x) + k^2 X(x) = 0.$$

The general solution is

$$X(x) = A\cos(kx) + B\sin(kx).$$

We now impose periodicity. First,

$$X(0) = A, \quad X(2\pi) = A\cos(2\pi k) + B\sin(2\pi k).$$

The condition $X(0) = X(2\pi)$ gives

$$A = A\cos(2\pi k) + B\sin(2\pi k).$$

Next,

$$X'(x) = -Ak\sin(kx) + Bk\cos(kx),$$

so

$$X'(0) = Bk, \quad X'(2\pi) = -Ak\sin(2\pi k) + Bk\cos(2\pi k).$$

The condition $X'(0) = X'(2\pi)$ gives

$$Bk = -Ak\sin(2\pi k) + Bk\cos(2\pi k).$$

We can divide by k the last identity (for $k \neq 0$) and write the system as

$$A(1 - \cos(2\pi k)) - B\sin(2\pi k) = 0,$$

$$A\sin(2\pi k) + B(1 - \cos(2\pi k)) = 0.$$

For a nontrivial pair (A, B) the determinant of the system must vanish:

$$(1 - \cos(2\pi k))^2 + (\sin(2\pi k))^2 = 0.$$

The left side is a sum of squares, so it is zero if and only if

$$1 - \cos(2\pi k) = 0, \quad \sin(2\pi k) = 0.$$

Hence

$$\cos(2\pi k) = 1, \quad \sin(2\pi k) = 0.$$

This happens exactly when k is an integer:

$$k = n, \quad n \in \mathbb{Z}.$$

The case $n = 0$ corresponds to $\lambda = 0$, which we have already treated. For $n \geqslant 1$ we obtain eigenvalues

$$\lambda_n = n^2, \quad n = 1, 2, 3, ...$$

For each $n \geqslant 1$ the corresponding eigenfunctions can be chosen as

$$X_n^{(c)}(x) = \cos(nx), \quad X_n^{(s)}(x) = \sin(nx).$$

These functions are 2π periodic, and their derivatives are also 2π periodic, so the boundary conditions are satisfied.

We have therefore found a complete set of spatial eigenfunctions for the heat equation with periodic boundary conditions:

- a constant mode $X_0(x) = 1$ (eigenvalue $\lambda_0 = 0$),
- cosine modes $X_n^{(c)}(x) = \cos(nx)$,
- sine modes $X_n^{(s)}(x) = \sin(nx)$,

with eigenvalues $\lambda_n = n^2$ for $n \geqslant 1$.

2.1.3 Step 3: Time Dependent Factors

For each eigenvalue λ the corresponding time factor satisfies

$$T'(t) + \lambda T(t) = 0.$$

The solution is

$$T(t) = Ce^{-\lambda t}.$$

For the constant mode $\lambda_0 = 0$ we obtain

$$T_0(t) = C_0$$

(constant in time).

For $n \geqslant 1$ we have

$$T_n(t) = C_n e^{-n^2 t}.$$

Combining space and time, we form products $u(x,t) = X(x) \cdot T(t)$ to obtain separated solutions. For each mode, the arbitrary constant from the time factor can be absorbed into a single new constant. Specifically, we write

$$u_0(x,t) = A_0,$$
$$u_n^{(c)}(x,t) = A_n \cos(nx) \cdot e^{-n^2 t},$$
$$u_n^{(s)}(x,t) = B_n \sin(nx) \cdot e^{-n^2 t},$$

where we have renamed the constants: $A_0 = C_0$, and for $n \geqslant 1$, the constant A_n absorbs the coefficient from the cosine mode while B_n absorbs the coefficient from the sine mode. Note that each separated solution carries its own independent arbitrary constant.

Since the heat equation is linear, any linear combination of these separated solutions is again a solution. Therefore the general solution that satisfies the periodic boundary conditions can be written as an infinite series

$$u(x,t) = A_0 + \sum_{n=1}^{\infty} (A_n \cos(nx) + B_n \sin(nx)) e^{-n^2 t}.$$

The coefficients A_0, A_n, B_n remain to be determined from the initial condition.

2.1.4 Step 4: Imposing the Initial Condition and Fourier Series

We impose the initial condition

$$u(x,0) = f(x).$$

Setting $t = 0$ in the general solution we obtain

$$u(x,0) = A_0 + \sum_{n=1}^{\infty} (A_n \cos(nx) + B_n \sin(nx)) = f(x).$$

This is exactly the Fourier series expansion of the 2π periodic function f. Under mild regularity assumptions, f has a Fourier series

$$f(x) = a_0 + \sum_{n=1}^{\infty} (a_n \cos(nx) + b_n \sin(nx))$$

with Fourier coefficients

$$a_0 = \frac{1}{2\pi} \int_0^{2\pi} f(x)\, \mathrm{d}x,$$

$$a_n = \frac{1}{\pi} \int_0^{2\pi} f(x) \cos(nx)\, \mathrm{d}x, \quad n \geqslant 1,$$

$$b_n = \frac{1}{\pi}\int_0^{2\pi} f(x)\sin(nx)\,\mathrm{d}x, \quad n \geqslant 1.$$

By uniqueness of the Fourier expansion, we must have

$$A_0 = a_0, \quad A_n = a_n, \quad B_n = b_n.$$

Thus the coefficients in the heat equation solution are exactly the Fourier coefficients of the initial data.

2.1.5 Step 5: Final Explicit Solution and Interpretation

Substituting these coefficients into the general solution we obtain the explicit formula

$$u(x,t) = a_0 + \sum_{n=1}^{\infty}(a_n\cos(nx) + b_n\sin(nx))e^{-n^2t}, \quad t > 0.$$

This infinite sum solves the heat equation with periodic boundary conditions and initial data f. Each Fourier mode decays exponentially in time at a rate proportional to its eigenvalue n^2. High frequency modes (large n) decay faster, which expresses the smoothing effect of the heat equation.

The constant term a_0 does not decay. It represents the average value of f on $[0, 2\pi]$, which is preserved by the heat flow.

From a spectral viewpoint, the functions

$$1, \cos(x), \sin(x), \cos(2x), \sin(2x), \ldots$$

form an eigenbasis of the spatial operator

$$Lu = \frac{\partial^2 u}{\partial x^2}$$

with periodic boundary conditions. The evolution of each eigenmode is independent and given simply by multiplication by e^{-n^2t} in time.

Later, in the numerical part of this book, we will approximate $u(x,t)$ by truncating the sum to finitely many modes:

$$u_{N(x,t)} = a_0 + \sum_{n=1}^{N}(a_n\cos(nx) + b_n\sin(nx))e^{-n^2t}.$$

This truncation is the essence of a Fourier spectral method. The analytic solution derived here is the infinite dimensional limit of that numerical procedure.

2.2 Numerical Illustration

To visualize the smoothing effect of heat diffusion, we compute the truncated Fourier series solution for a triangle wave initial condition:

$$f(x) = \pi - |x - \pi|, \quad x \in [0, 2\pi].$$

This function is continuous but has a corner (non-differentiable point) at $x = \pi$. Its Fourier series contains only cosine terms with coefficients decaying as $1/n^2$:

$$f(x) = \frac{\pi}{2} + \frac{4}{\pi}\sum_{k=1}^{\infty}\frac{\cos((2k-1)x)}{(2k-1)^2}.$$

The key portion of the implementation evaluates the truncated Fourier series at any point in space and time. In Python:

```python
def heat_solution(x, t, a0, a_n, b_n):
```

```
    u = np.full_like(x, a0, dtype=float)
    n_modes = len(a_n) - 1
    for n in range(1, n_modes + 1):
        decay = np.exp(-n**2 * t)
        u += (a_n[n] * np.cos(n * x) + b_n[n] * np.sin(n * x)) * decay
    return u
```

The equivalent MATLAB implementation:

```
u = a0 * ones(size(x));
for n = 1:N_MODES
    decay = exp(-n^2 * t);
    u = u + (a_n(n+1) * cos(n*x) + b_n(n+1) * sin(n*x)) * decay;
end
```

Figure 1 shows the evolution of $u_N(x,t)$ with $N = 50$ modes at several time values. At $t = 0$ the triangle wave is faithfully reproduced. As time increases, the higher frequency modes decay exponentially faster than the lower ones (the n-th mode decays as $e^{-n^2 t}$), and the solution rapidly smooths toward the constant equilibrium $u = \pi/2$.

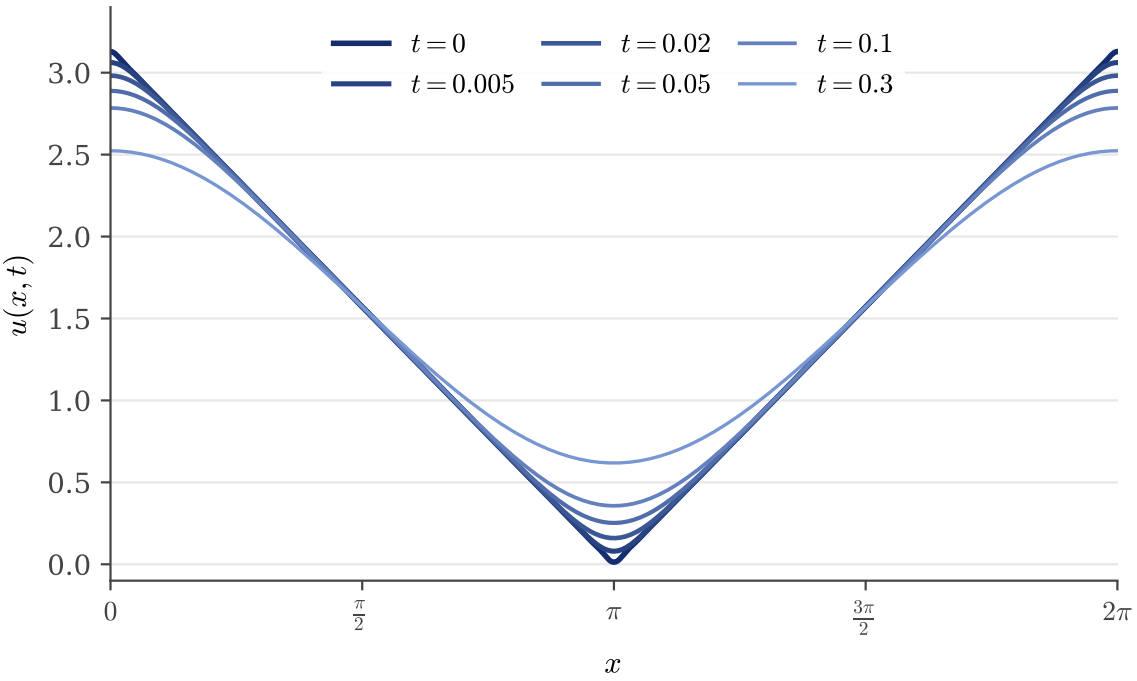

Figure 1: *Evolution of the heat equation solution with a triangle wave initial condition. The higher frequency modes decay rapidly, smoothing the initial corner at $x = \pi$.*

The code that generated this figure is available in both Python and MATLAB:

- `codes/python/ch02_classical_pdes/heat_equation_evolution.py`
- `codes/matlab/ch02_classical_pdes/heat_equation_evolution.m`

A complementary view of the solution is provided by the waterfall plot in Figure 2, which displays the entire space-time evolution as a three-dimensional surface. The smoothing effect of the heat equation is clearly visible: the initial sharp triangle wave rapidly flattens as time progresses, with the solution approaching the constant equilibrium state $u = \pi/2$.

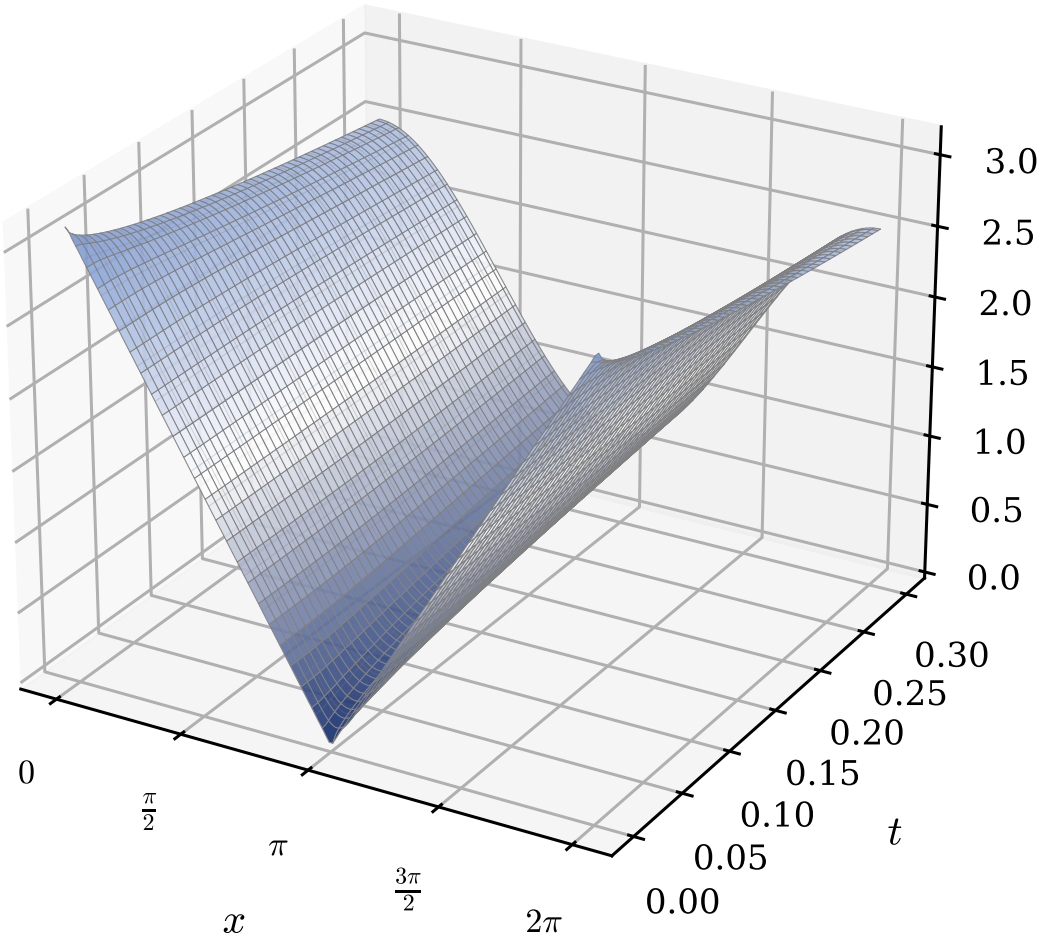

Figure 2: *Waterfall plot showing the complete space-time evolution of the heat equation solution. The initial triangle wave smooths rapidly as higher frequency modes decay exponentially.*

2.3 Wave Equation with Dirichlet Boundary Conditions

We now consider the one dimensional wave equation on a finite interval with homogeneous Dirichlet boundary conditions. This is the classical model for a vibrating string of length L with both ends fixed.

Let $u(x,t)$ denote the vertical displacement of the string at position $x \in [0,L]$ and time $t > 0$. The equation of motion is

$$\frac{\partial^2 u}{\partial t^2}(x,t) = c^2 \frac{\partial^2 u}{\partial x^2}(x,t), \quad 0 < x < L,\ t > 0,$$

where $c > 0$ is the wave speed.

The boundary conditions express that the endpoints of the string are clamped:

$$u(0,t) = 0, \quad u(L,t) = 0, \quad t > 0.$$

We prescribe the initial displacement and initial velocity:

$$u(x,0) = f(x), \quad \frac{\partial u}{\partial t}(x,0) = g(x), \quad 0 < x < L,$$

with suitable functions f and g that vanish at $x = 0$ and $x = L$.

As in the heat equation example, we use separation of variables and obtain a representation of the solution as an infinite series in spatial eigenfunctions. This time the temporal factors are oscillatory instead of decaying. The mathematical theory of the vibrating string is a classical topic covered extensively in [4].

2.3.1 Step 1: Separation Ansatz

We look for nontrivial solutions of the form

$$u(x,t) = X(x) \cdot T(t).$$

Substituting into the wave equation gives

$$X(x) \cdot T''(t) = c^2 X''(x) \cdot T(t).$$

Assuming X and T are not identically zero, we divide both sides by $c^2 X(x) \cdot T(t)$:

$$\frac{T''(t)}{c^2 T(t)} = \frac{X''(x)}{X(x)}.$$

The left side depends only on t, the right side only on x. Therefore both sides must be equal to a constant, which we denote by $-\lambda$:

$$\frac{T''(t)}{c^2 T(t)} = \frac{X''(x)}{X(x)} = -\lambda.$$

We obtain the pair of ordinary differential equations

$$T''(t) + c^2 \lambda T(t) = 0,$$
$$X''(x) + \lambda X(x) = 0,$$

with boundary conditions

$$X(0) = 0, \quad X(L) = 0.$$

As in the heat equation case, we have a spatial eigenvalue problem for X.

2.3.2 Step 2: Spatial Eigenvalue Problem with Dirichlet Boundary Conditions

We must solve

$$X''(x) + \lambda X(x) = 0, \quad 0 < x < L,$$

with

$$X(0) = 0, \quad X(L) = 0.$$

We again consider three cases: $\lambda < 0$, $\lambda = 0$, and $\lambda > 0$.

Case 1: $\lambda < 0$

Write $\lambda = -\mu^2$ with $\mu > 0$. The equation becomes

$$X''(x) - \mu^2 X(x) = 0.$$

The general solution is

$$X(x) = Ae^{\mu x} + Be^{-\mu x}.$$

The boundary condition at $x = 0$ gives

$$X(0) = A + B = 0 \quad \Rightarrow \quad B = -A.$$

Then

$$X(L) = Ae^{\mu L} - Ae^{-\mu L} = A(e^{\mu L} - e^{-\mu L}).$$

The condition $X(L) = 0$ implies

$$A(e^{\mu L} - e^{-\mu L}) = 0.$$

Since $e^{\mu L} \neq e^{-\mu L}$ for $\mu > 0$, we must have $A = 0$. Then $B = 0$ and the solution is trivial. Therefore there are no nontrivial eigenfunctions for $\lambda < 0$.

Case 2: $\lambda = 0$

The equation reduces to

$$X''(x) = 0,$$

whose general solution is

$$X(x) = A + Bx.$$

The boundary conditions give

$$X(0) = A = 0,$$
$$X(L) = A + BL = BL = 0.$$

Hence $B = 0$ and X is again trivial. There is no nontrivial eigenfunction for $\lambda = 0$.

Case 3: $\lambda > 0$

Write $\lambda = k^2$ with $k > 0$. The equation becomes

$$X''(x) + k^2 X(x) = 0.$$

The general solution is

$$X(x) = A\cos(kx) + B\sin(kx).$$

The boundary condition at $x = 0$ gives

$$X(0) = A = 0.$$

So $X(x) = B\sin(kx)$. The boundary condition at $x = L$ gives

$$X(L) = B\sin(kL) = 0.$$

For a nontrivial solution we need $B \neq 0$, so we must have

$$\sin(kL) = 0.$$

Therefore

$$kL = n\pi, \quad n = 1, 2, 3, ...$$

The corresponding values of k are

$$k_n = \frac{n\pi}{L}, \quad n = 1, 2, 3, ...$$

We conclude that the eigenvalues and eigenfunctions are

$$\lambda_n = k_n^2 = \left(\frac{n\pi}{L}\right)^2,$$

$$X_n(x) = \sin\left(\frac{n\pi x}{L}\right), \quad n = 1, 2, 3, ...$$

Each X_n vanishes at $x = 0$ and $x = L$, as required by the Dirichlet boundary conditions. The family $\{X_n\}_{n\geqslant 1}$ is orthogonal in $L^2(0, L)$:

$$\int_0^L \sin\left(\frac{n\pi x}{L}\right)\sin\left(\frac{m\pi x}{L}\right)\mathrm{d}x = \begin{cases} 0 & \text{if } n \neq m \\ L/2 & \text{if } n = m. \end{cases}$$

These eigenfunctions will form the spatial basis in our series solution.

2.3.3 Step 3: Time Dependent Factors

For each eigenvalue λ_n the time factor T_n satisfies

$$T_n''(t) + c^2\lambda_n T_n(t) = 0.$$

Using $\lambda_n = (n\pi/L)^2$ we can write

$$T_n''(t) + \omega_n^2 T_n(t) = 0,$$

where

$$\omega_n = c\frac{n\pi}{L}, \quad n = 1, 2, 3, ...$$

The general solution of this second order linear ODE is

$$T_n(t) = A_n \cos(\omega_n t) + B_n \sin(\omega_n t),$$

where A_n and B_n are constants.

Combining the space and time factors, we obtain separated solutions

$$u_n(x,t) = (A_n \cos(\omega_n t) + B_n \sin(\omega_n t)) \sin\Big(\frac{n\pi x}{L}\Big), \quad n = 1, 2, 3, ...$$

Because the wave equation is linear, any linear combination of these separated solutions is again a solution. Therefore the general solution satisfying the Dirichlet boundary conditions can be written as an infinite series

$$u(x,t) = \sum_{n=1}^{\infty} (a_n \cos(\omega_n t) + b_n \sin(\omega_n t)) \sin\Big(\frac{n\pi x}{L}\Big),$$

for suitable coefficients a_n and b_n.

These coefficients will be determined from the initial conditions.

2.3.4 Step 4: Imposing the Initial Conditions and Sine Series

We now use the initial displacement and velocity.

At $t = 0$ we have

$$u(x,0) = \sum_{n=1}^{\infty} (a_n \cos(0) + b_n \sin(0)) \sin\Big(\frac{n\pi x}{L}\Big) = \sum_{n=1}^{\infty} a_n \sin\Big(\frac{n\pi x}{L}\Big).$$

The initial condition $u(x,0) = f(x)$ becomes

$$f(x) = \sum_{n=1}^{\infty} a_n \sin\Big(\frac{n\pi x}{L}\Big).$$

This is the Fourier sine series of f on the interval $(0, L)$.

Similarly, we differentiate u with respect to t:

$$\frac{\partial u}{\partial t}(x,t) = \sum_{n=1}^{\infty} (-a_n \omega_n \sin(\omega_n t) + b_n \omega_n \cos(\omega_n t)) \sin\Big(\frac{n\pi x}{L}\Big).$$

Evaluating at $t = 0$ gives

$$\frac{\partial u}{\partial t}(x,0) = \sum_{n=1}^{\infty} b_n \omega_n \sin\Big(\frac{n\pi x}{L}\Big).$$

The initial condition $\frac{\partial u}{\partial t}(x,0) = g(x)$ becomes

$$g(x) = \sum_{n=1}^{\infty} b_n \omega_n \sin\Big(\frac{n\pi x}{L}\Big).$$

So the sequence $\{a_n\}$ consists of the Fourier sine coefficients of f, and the sequence $\{b_n \omega_n\}$ consists of the Fourier sine coefficients of g.

Using the orthogonality relations, we obtain explicit formulas for the coefficients. For $n \geqslant 1$,

$$a_n = \frac{2}{L} \int_0^L f(x) \sin\Big(\frac{n\pi x}{L}\Big)\, \mathrm{d}x,$$

and

$$b_n \omega_n = \frac{2}{L} \int_0^L g(x) \sin\Big(\frac{n\pi x}{L}\Big)\, \mathrm{d}x.$$

Therefore

$$b_n = \frac{2}{L\omega_n} \int_0^L g(x) \sin\Big(\frac{n\pi x}{L}\Big)\, \mathrm{d}x = \frac{2}{L} \frac{1}{cn\pi/L} \int_0^L g(x) \sin\Big(\frac{n\pi x}{L}\Big)\, \mathrm{d}x.$$

In summary,

$$a_n = \frac{2}{L}\int_0^L f(x)\sin\Big(\frac{n\pi x}{L}\Big)\,\mathrm{d}x,$$

$$b_n = \frac{2}{L\omega_n}\int_0^L g(x)\sin\Big(\frac{n\pi x}{L}\Big)\,\mathrm{d}x, \quad \omega_n = c\frac{n\pi}{L}.$$

2.3.5 Step 5: Final Explicit Solution and Interpretation

Substituting these coefficients into the series, we obtain the explicit solution of the wave equation with Dirichlet boundary conditions:

$$u(x,t) = \sum_{n=1}^{\infty}[a_n\cos(\omega_n t) + b_n\sin(\omega_n t)]\sin\Big(\frac{n\pi x}{L}\Big),$$

where $\omega_n = cn\pi/L$ and the Fourier sine coefficients are

$$a_n = \frac{2}{L}\int_0^L f(y)\sin\Big(\frac{n\pi y}{L}\Big)\,\mathrm{d}y, \quad b_n = \frac{2}{n\pi c}\int_0^L g(y)\sin\Big(\frac{n\pi y}{L}\Big)\,\mathrm{d}y.$$

Each term in the sum is a normal mode of vibration: a standing wave with spatial shape $\sin(n\pi x/L)$ and temporal oscillation at frequency ω_n. The coefficients of $\cos(\omega_n t)$ and $\sin(\omega_n t)$ are determined by the initial displacement f and initial velocity g through their Fourier sine coefficients.

Comparing with the heat equation:

- For the heat equation, each mode decayed like $e^{-n^2 t}$ and the solution became smoother in time.
- For the wave equation, each mode oscillates periodically in time with constant amplitude, reflecting conservation of energy in the undamped string.

From a spectral viewpoint, the functions

$$\sin\Big(\frac{\pi x}{L}\Big), \sin\Big(\frac{2\pi x}{L}\Big), \sin\Big(\frac{3\pi x}{L}\Big), \ldots$$

form an eigenbasis of the spatial operator

$$Lu = \frac{\partial^2 u}{\partial x^2}$$

with Dirichlet boundary conditions. In this basis, the evolution is diagonal: each mode evolves independently according to a simple harmonic oscillator in time.

As in the heat equation example, a spectral method will approximate $u(x,t)$ by truncating the infinite sum. For some integer $N \geqslant 1$ we consider the finite approximation

$$u_N(x,t) = \sum_{n=1}^{N}(a_n\cos(\omega_n t) + b_n\sin(\omega_n t))\sin\Big(\frac{n\pi x}{L}\Big).$$

The analytic series above is the infinite dimensional limit of this spectral representation.

2.4 Numerical Illustration

To visualize the oscillatory behavior of the vibrating string, we compute the truncated Fourier sine series solution for a plucked string initial condition. The string is plucked at its center, forming a triangular initial displacement:

$$f(x) = \begin{cases} \frac{2h}{L}x & \text{for } 0 \leqslant x \leqslant L/2 \\ 2h(1 - x/L) & \text{for } L/2 \leqslant x \leqslant L \end{cases}$$

with zero initial velocity $g(x) = 0$. Here h denotes the height of the pluck at the center.

The Fourier sine coefficients of this triangular shape are

$$a_n = \frac{8h}{n^2\pi^2} \sin\left(\frac{n\pi}{2}\right),$$

which gives nonzero values only for odd n, with alternating signs.

The key portion of the implementation computes the solution at any point in space and time. In Python:

```python
def wave_solution(x, t, a_n, b_n, L, c):
    u = np.zeros_like(x, dtype=float)
    n_modes = len(a_n) - 1
    for n in range(1, n_modes + 1):
        omega_n = c * n * np.pi / L
        spatial = np.sin(n * np.pi * x / L)
        temporal = a_n[n] * np.cos(omega_n * t) + b_n[n] * np.sin(omega_n * t)
        u += temporal * spatial
    return u
```

The equivalent MATLAB implementation:

```matlab
u = zeros(size(x));
for n = 1:N_MODES
    omega_n = C * n * pi / L;
    spatial = sin(n * pi * x / L);
    temporal = a_n(n+1) * cos(omega_n * t) + b_n(n+1) * sin(omega_n * t);
    u = u + temporal * spatial;
end
```

Figure 3 shows the evolution of $u_N(x, t)$ with $N = 50$ modes at several time values within half a period $T = 2L/c$. The string oscillates back and forth, with the triangular shape inverting at $t = T/2$. Unlike the heat equation, the wave equation preserves energy and the solution does not decay; it continues oscillating indefinitely.

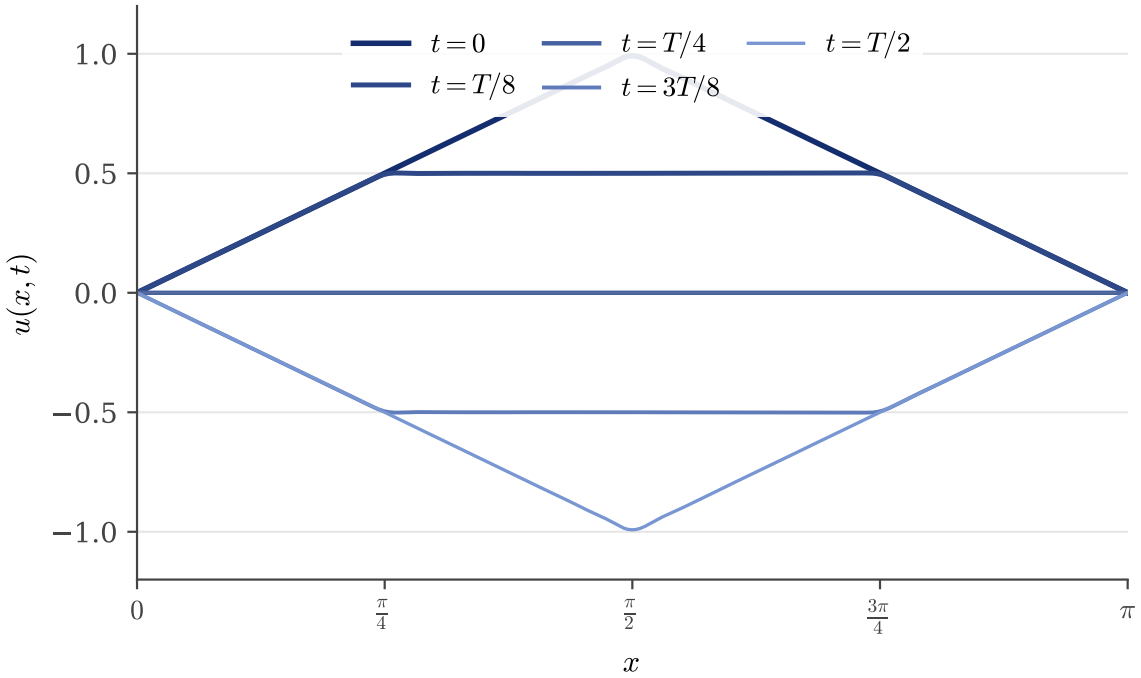

Figure 3: *Evolution of the wave equation solution with a plucked string initial condition. The string oscillates with period $T = 2\pi$, inverting at $t = T/2$.*

The code that generated this figure is available in both Python and MATLAB:

- `codes/python/ch02_classical_pdes/wave_equation_evolution.py`
- `codes/matlab/ch02_classical_pdes/wave_equation_evolution.m`

The waterfall plot in Figure 4 provides a complete view of the oscillatory dynamics over one full period. Unlike the heat equation, the wave equation conserves energy: the solution oscillates indefinitely without decay, and the periodic nature of the motion is clearly visible in the three-dimensional representation.

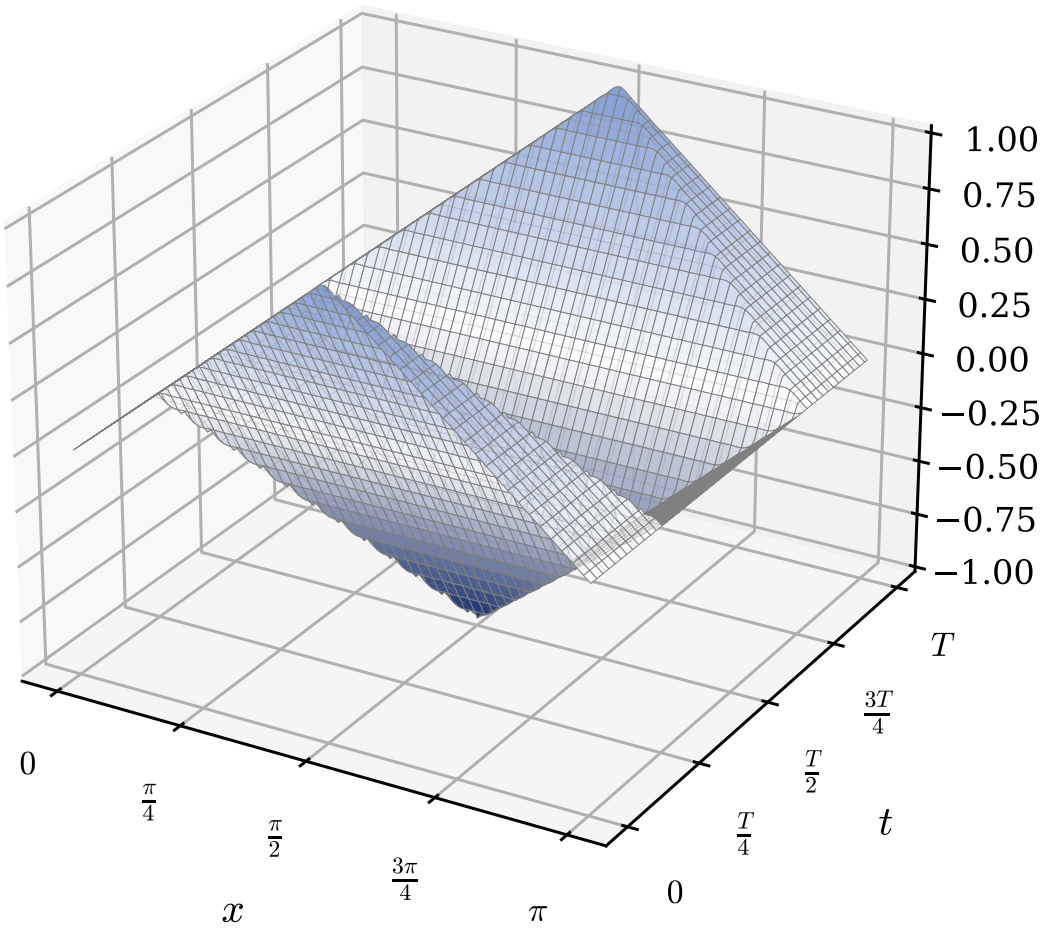

Figure 4: *Waterfall plot showing the complete space-time evolution of the wave equation solution over one period T. The plucked string oscillates between its initial shape and its mirror image.*

2.5 Laplace Equation in a Periodic Strip

For the elliptic case we consider the Laplace equation in a simple two dimensional domain that is periodic in one direction and bounded in the other. This setting connects naturally with the periodic heat equation example and again leads to a Fourier series representation in the periodic direction.

Let

$$D = \{(x, y) \in \mathbb{R}^2 : 0 < x < 2\pi,\ 0 < y < 1\}.$$

We seek a harmonic function $u(x, y)$ solving

$$u_{xx}(x, y) + u_{yy}(x, y) = 0, \quad (x, y) \in D,$$

with periodic boundary conditions in x

$$u(x + 2\pi, y) = u(x, y), \quad \text{for all real } \ x,\ 0 < y < 1,$$

and Dirichlet conditions in y

$$u(x,0) = f(x), \quad u(x,1) = 0, \quad 0 \leqslant x \leqslant 2\pi.$$

We assume that f is 2π periodic and smooth:

$$f(x + 2\pi) = f(x).$$

As in the parabolic and hyperbolic examples, we apply separation of variables and obtain a representation of u as an infinite Fourier series in x with y dependent coefficients. The theory of harmonic functions and Laplace's equation is presented in detail in [4].

2.5.1 Step 1: Separation Ansatz

We look for nontrivial separated solutions of the form

$$u(x,y) = X(x) \cdot Y(y).$$

Substituting into the Laplace equation gives

$$X''(x) \cdot Y(y) + X(x) \cdot Y''(y) = 0.$$

Assuming X and Y are not identically zero, we divide by $X(x) \cdot Y(y)$:

$$\frac{X''(x)}{X(x)} + \frac{Y''(y)}{Y(y)} = 0.$$

The first term depends only on x, the second only on y. Therefore both must be equal to constants whose sum is zero. We introduce a separation constant λ and write

$$\frac{X''(x)}{X(x)} = -\lambda, \quad \frac{Y''(y)}{Y(y)} = \lambda.$$

This leads to the two ordinary differential equations

$$X''(x) + \lambda X(x) = 0,$$
$$Y''(y) - \lambda Y(y) = 0.$$

The periodic boundary conditions for u in the x direction imply the periodic conditions

$$X(x + 2\pi) = X(x), \quad \text{for all real } \ x.$$

The Dirichlet conditions in y will be enforced later on the full series. For the separated functions Y we will impose appropriate conditions at $y = 1$, while the condition at $y = 0$ will be handled via the Fourier coefficients of f.

We have once more an eigenvalue problem in the periodic direction.

2.5.2 Step 2: Eigenfunctions in the Periodic Direction

The equation for X with periodic boundary conditions is exactly the same as in the heat equation example:

$$X''(x) + \lambda X(x) = 0, \quad X(x + 2\pi) = X(x).$$

We recall the result: there is a constant mode with eigenvalue $\lambda_0 = 0$,

$$X_0(x) = 1,$$

and for each integer $n \geqslant 1$ there are cosine and sine modes with eigenvalues

$$\lambda_n = n^2,$$
$$X_n^{(c)}(x) = \cos(nx), \quad X_n^{(s)}(x) = \sin(nx).$$

The family

$$1, \cos(x), \sin(x), \cos(2x), \sin(2x), ...$$

forms an orthogonal basis in $L^2(0, 2\pi)$ for 2π periodic functions.

For each such eigenvalue we now solve the corresponding equation for Y.

2.5.3 Step 3: Equations for the y Dependent Factors

For each eigenvalue λ we have

$$Y''(y) - \lambda Y(y) = 0.$$

We treat separately the constant mode $\lambda_0 = 0$ and the nonzero modes $\lambda_n = n^2$ for $n \geqslant 1$.

Constant mode $\lambda_0 = 0$

For $\lambda_0 = 0$ the equation reduces to

$$Y_0''(y) = 0.$$

The general solution is

$$Y_0(y) = A_0 + B_0 y.$$

We want separated solutions that satisfy the homogeneous boundary condition at $y = 1$:

$$u(x, 1) = 0 \quad \text{for all} \quad x.$$

For the constant mode this means

$$X_0(x) \cdot Y_0(1) = Y_0(1) = 0,$$

hence

$$Y_0(1) = A_0 + B_0 = 0.$$

We choose a convenient normalization so that $Y_0(0) = 1$. Then $A_0 = 1$ and the relation $A_0 + B_0 = 0$ gives $B_0 = -1$. Thus

$$Y_0(y) = 1 - y.$$

This separated mode

$$u_0(x, y) = X_0(x) \cdot Y_0(y) = 1 - y$$

is harmonic, periodic in x, and vanishes at $y = 1$, with value 1 at $y = 0$.

Higher modes $\lambda_n = n^2$ for $n \geqslant 1$

For $\lambda_n = n^2$ the equation is

$$Y_n''(y) - n^2 Y_n(y) = 0.$$

The general solution can be written in hyperbolic form

$$Y_n(y) = \alpha_n \cosh(ny) + \beta_n \sinh(ny).$$

We require that each separated mode vanish at $y = 1$:

$$Y_n(1) = 0.$$

To match later the Fourier coefficients of f at $y = 0$, it is convenient to normalize so that

$$Y_n(0) = 1.$$

Imposing $Y_n(0) = 1$ gives

$$\alpha_n = 1.$$

Then

$$Y_n(1) = \cosh(n) + \beta_n \sinh(n) = 0$$

so

$$\beta_n = -\frac{\cosh(n)}{\sinh(n)} = -\coth(n).$$

Thus

$$Y_n(y) = \cosh(ny) - \coth(n) \cdot \sinh(ny).$$

An alternative and simpler expression uses the hyperbolic sine function of the distance to the boundary $y = 1$. One checks that

$$Y_n(y) = \frac{\sinh(n(1-y))}{\sinh(n)}$$

satisfies

$$Y_n''(y) - n^2 Y_n(y) = 0,$$
$$Y_n(1) = 0,$$

$$Y_n(0) = 1.$$

Indeed, $Y_n(1) = \sinh(0)/\sinh(n) = 0$, $Y_n(0) = \sinh(n)/\sinh(n) = 1$, and

$$Y_n''(y) = n^2 \frac{\sinh(n(1-y))}{\sinh(n)} = n^2 Y_n(y).$$

We will use the form

$$Y_n(y) = \frac{\sinh(n(1-y))}{\sinh(n)}, \quad n \geqslant 1.$$

2.5.4 Step 4: Building the Series Solution

Each separated solution corresponding to the eigenfunctions in x and the functions Y_n in y has the form

$$u_0(x,y) = C_0 \cdot Y_0(y) = C_0(1-y),$$
$$u_n^{(c)}(x,y) = C_n \cdot \cos(nx) \cdot Y_n(y),$$
$$u_n^{(s)}(x,y) = D_n \cdot \sin(nx) \cdot Y_n(y), \quad n \geqslant 1,$$

for some constants C_0, C_n, D_n.

Since the Laplace equation is linear and the boundary condition at $y = 1$ is homogeneous, any linear combination of these separated solutions is again a solution that vanishes at $y = 1$. Therefore a general solution satisfying the periodic condition in x and the Dirichlet condition $u(x,1) = 0$ can be written as

$$u(x,y) = C_0(1-y) + \sum_{n=1}^{\infty} [C_n \cos(nx) + D_n \sin(nx)] \frac{\sinh(n(1-y))}{\sinh(n)}.$$

It remains to impose the boundary condition at $y = 0$,

$$u(x,0) = f(x).$$

At $y = 0$ we obtain

$$u(x,0) = C_0 + \sum_{n=1}^{\infty} [C_n \cos(nx) + D_n \sin(nx)],$$

because $Y_0(0) = 1$ and $Y_n(0) = 1$ for $n \geqslant 1$.

Hence the boundary condition $u(x,0) = f(x)$ becomes

$$f(x) = C_0 + \sum_{n=1}^{\infty} [C_n \cos(nx) + D_n \sin(nx)].$$

This is exactly the Fourier series expansion of f. For a 2π periodic f we have

$$f(x) = a_0 + \sum_{n=1}^{\infty} [a_n \cos(nx) + b_n \sin(nx)],$$

with coefficients

$$a_0 = \frac{1}{2\pi} \int_0^{2\pi} f(x)\,\mathrm{d}x,$$

$$a_n = \frac{1}{\pi} \int_0^{2\pi} f(x) \cos(nx)\,\mathrm{d}x, \quad n \geqslant 1,$$

$$b_n = \frac{1}{\pi} \int_0^{2\pi} f(x) \sin(nx)\,\mathrm{d}x, \quad n \geqslant 1.$$

By uniqueness of the Fourier expansion, we must have

$$C_0 = a_0, \quad C_n = a_n, \quad D_n = b_n.$$

2.5.5 Step 5: Final Explicit Solution and Interpretation

Substituting these coefficients into the series we obtain the explicit representation

$$u(x,y) = a_0(1-y) + \sum_{n=1}^{\infty}[a_n\cos(nx) + b_n\sin(nx)]\frac{\sinh(n(1-y))}{\sinh(n)}, \quad 0 < y < 1.$$

Here a_0, a_n, and b_n are the Fourier coefficients of the boundary data f as defined above.

This series converges (under mild assumptions on f) to the unique harmonic function that is periodic in x, equal to f on $y = 0$, and zero on $y = 1$.

From a spectral viewpoint:

- The functions 1, $\cos(nx)$, $\sin(nx)$ are eigenfunctions of the one dimensional Laplacian $X \mapsto X''$ with periodic boundary conditions in x, with eigenvalues $\lambda_0 = 0$ and $\lambda_n = n^2$.
- For each spatial frequency n in the periodic direction, the dependence in the transverse direction y is determined by the simple ordinary differential equation $Y'' - \lambda_n Y = 0$ with boundary condition $Y(1) = 0$ and normalization $Y(0) = 1$. This gives the hyperbolic profiles

$$Y_0(y) = 1 - y,$$
$$Y_n(y) = \frac{\sinh(n(1-y))}{\sinh(n)}, \quad n \geqslant 1.$$

- The boundary data f at $y = 0$ is expanded in the eigenbasis $\{1, \cos(nx), \sin(nx)\}$ and each Fourier mode is propagated into the interior of the strip with its own y dependent factor $Y_n(y)$.

Analytically, the solution is an infinite sum of separated solutions. In a spectral method we will truncate this sum to finitely many modes in x,

$$u_N(x,y) = a_0(1-y) + \sum_{n=1}^{N}[a_n\cos(nx) + b_n\sin(nx)]\frac{\sinh(n(1-y))}{\sinh(n)},$$

and approximate the harmonic function inside the strip by this finite Fourier representation.

2.6 Numerical Illustration

To visualize the structure of harmonic functions in the strip, we compute the truncated Fourier series solution for a boundary condition that contains two modes:

$$f(x) = \sin(x) + \frac{1}{2}\sin(3x).$$

For this particular boundary data, the Fourier coefficients are simply $b_1 = 1$ and $b_3 = 1/2$, with all other coefficients zero. The solution can be written explicitly as

$$u(x,y) = \sin(x)\frac{\sinh(1-y)}{\sinh(1)} + \frac{1}{2}\sin(3x)\frac{\sinh(3(1-y))}{\sinh(3)}.$$

The key portion of the implementation evaluates the truncated series on a two-dimensional grid. In Python:

```python
def laplace_solution(x, y, a0, a_n, b_n):
    u = a0 * (1 - y)
    n_modes = len(a_n) - 1
    for n in range(1, n_modes + 1):
```

```
        if abs(a_n[n]) < 1e-15 and abs(b_n[n]) < 1e-15:
            continue
        y_factor = np.sinh(n * (1 - y)) / np.sinh(n)
        u += (a_n[n] * np.cos(n * x) + b_n[n] * np.sin(n * x)) * y_factor
    return u
```

The equivalent MATLAB implementation:

```
U = a0 * (1 - Y);
for n = 1:N_MODES
    if abs(a_n(n+1)) < 1e-15 && abs(b_n(n+1)) < 1e-15
        continue;
    end
    y_factor = sinh(n * (1 - Y)) / sinh(n);
    U = U + (a_n(n+1) * cos(n*X) + b_n(n+1) * sin(n*X)) .* y_factor;
end
```

Figure 5 shows the solution $u(x, y)$ in the strip $[0, 2\pi] \times [0, 1]$. At the bottom boundary $y = 0$, the solution matches the prescribed boundary data $f(x)$. As y increases toward the top boundary, the solution decays to zero. Crucially, the higher frequency mode ($n = 3$) decays much faster than the lower frequency mode ($n = 1$), as the hyperbolic factor $\sinh(n(1-y))/\sinh(n)$ decreases more rapidly for larger n. This illustrates the smoothing effect of harmonic extension into the interior.

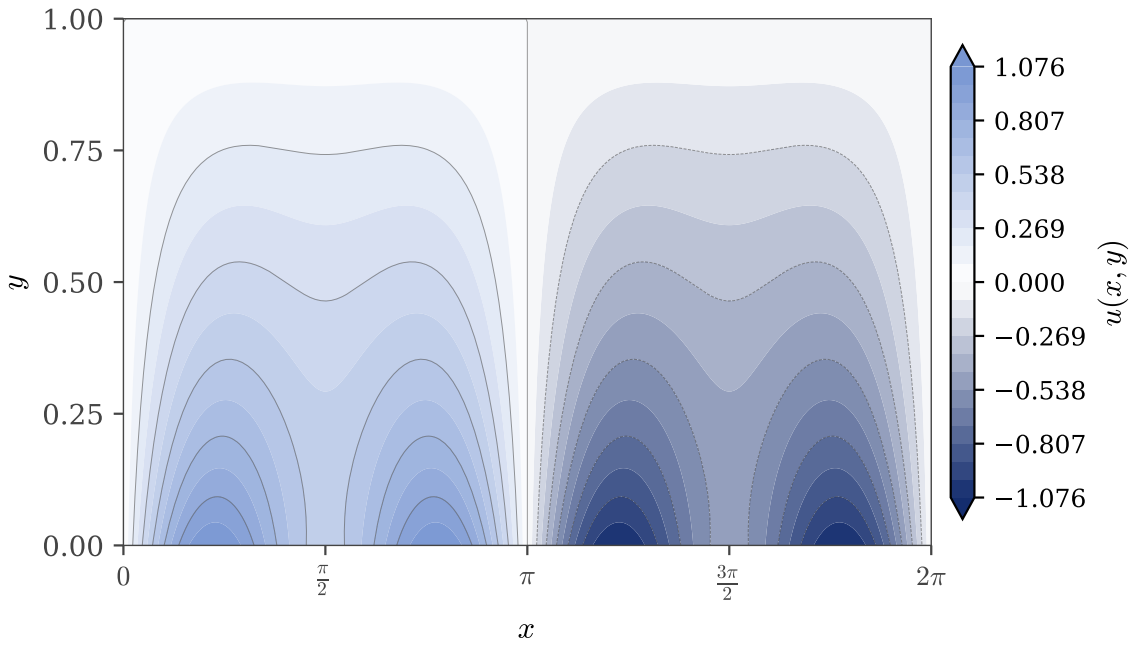

Figure 5: *Solution of the Laplace equation in the periodic strip with boundary data* $f(x) = \sin(x) + \frac{1}{2}\sin(3x)$ *at* $y = 0$ *and* $u = 0$ *at* $y = 1$. *Higher frequency modes decay faster toward the interior.*

The code that generated this figure is available in both Python and MATLAB:

- `codes/python/ch02_classical_pdes/laplace_equation_2d.py`
- `codes/matlab/ch02_classical_pdes/laplace_equation_2d.m`

2.7 Conclusions

The three examples presented in this chapter (the heat equation, the wave equation, and the Laplace equation) share a common mathematical structure that will guide us throughout the rest of this book. In each case, separation of variables reduces a partial differential equation to a family of ordinary differential equations: an eigenvalue problem in the spatial variable and a simpler equation governing the behavior in the remaining variable (time or the transverse coordinate). The eigenfunctions form an orthogonal basis, and the solution is expressed as an infinite series

$$u(x,t) = \sum_k \hat{u}_k(t)\varphi_k(x)$$

whose coefficients are determined by the initial or boundary data through Fourier projections. This is the DNA of spectral methods.

Let us distill the key ideas:

1. **Separation**. We decompose the solution into modes that evolve independently (or nearly so). Each mode satisfies a simpler equation than the original PDE.
2. **Basis**. The spatial part of each mode is an eigenfunction of a differential operator: Fourier exponentials for periodic problems, trigonometric functions for Dirichlet conditions, Chebyshev or Legendre polynomials for more general settings.
3. **Truncation**. In practice we cannot sum infinitely many terms due to the apparent finitude of our Universe. We retain only the first N modes, and the accuracy of this approximation depends critically on how fast the coefficients $\hat{u}_k$ decay.

When the solution is smooth, the coefficients decay *exponentially*, and a modest N suffices for high accuracy. This is the source of spectral methods' legendary efficiency. For a comprehensive treatment of these classical methods and their mathematical foundations, the reader is referred to [4].

The analytical solutions derived in this chapter are beautiful, but they are also fragile. They apply only to linear equations on simple domains with special boundary conditions. The moment we encounter a nonlinearity, a complicated geometry, or variable coefficients, we must turn to computation. The chapters that follow will develop the computational machinery needed to turn these analytical insights into practical algorithms:

- **FFT and its cousins**: how to move efficiently between physical space and coefficient space.
- **Differentiation matrices**: how to compute derivatives spectrally.
- **Quadrature rules**: how to compute inner products and projections.
- **Time stepping**: how to advance the ODE system for the coefficients.

Armed with these tools, we will be able to solve problems far beyond the reach of pen-and-paper analysis.

CHAPTER 3

Mise en Bouche

In French cuisine, a *mise en bouche* is a small appetizer offered by the chef to stimulate the palate before the main courses arrive. In this chapter we offer a similar intellectual appetizer: a compact, self-contained taste of spectral methods that illuminates the essential mechanics before we develop the full machinery of Fourier and Chebyshev approximation. Rather than jumping immediately to high-degree polynomials with $N = 100$, we perform hand calculations with just $N = 2$ or $N = 3$ unknowns. This low-dimensional setting makes every step transparent. We can verify each formula by direct computation and gain intuition that will guide us through the more sophisticated developments to come.

The techniques presented here follow the classical exposition in [3], adapted to our pedagogical goals. The Method of Weighted Residuals provides the unifying framework that connects the collocation (pseudospectral) approach we favor in this book with the Galerkin methods that dominate finite element analysis.

3.1 The Method of Weighted Residuals

3.1.1 Series Expansions and the Residual Function

The central idea of spectral methods is to approximate the unknown function $u(x)$ by a finite sum of basis functions:

$$u(x) \approx u_N(x) = \sum_{n=0}^{N} a_n \varphi_n(x),$$

where $\{\varphi_n(x)\}$ are known basis functions and $\{a_n\}$ are unknown coefficients to be determined.

When we substitute this approximation into a differential equation

$$\mathcal{L}u = f(x),$$

where $\mathcal{L}$ is a linear differential operator, the result is generally not zero. The *residual function* measures this discrepancy:

$$R(x; a_0, a_1, ..., a_N) = \mathcal{L}u_N - f.$$

For the exact solution, $R(x) \equiv 0$. The challenge is to choose the coefficients $\{a_n\}$ so that the residual is as small as possible. Different spectral methods correspond to different strategies for minimizing this residual.

3.1.2 Two Minimization Strategies

The two most important strategies are:

1. **Collocation (Pseudospectral) Method**: Force the residual to be exactly zero at a set of carefully chosen points $\{x_j\}$, called collocation points:
$$R(x_j; a_0, ..., a_N) = 0, \quad j = 1, 2, ..., N+1.$$
This gives $N + 1$ equations for $N + 1$ unknowns.
2. **Galerkin Method**: Require the residual to be orthogonal to each basis function in the sense of a weighted inner product:
$$\int_{-1}^{1} R(x)\varphi_k(x)w(x)\,\mathrm{d}x = 0, \quad k = 0, 1, ..., N,$$
where $w(x)$ is a weight function (often $w(x) = 1$ for polynomial bases).

Both methods convert the differential equation into a system of algebraic equations for the unknown coefficients. The collocation approach is simpler to implement and handles nonlinear terms easily, while the Galerkin approach often provides better global accuracy in weighted norms.

3.2 A First Collocation Example

We illustrate the collocation method with a complete worked example that can be verified by hand calculation.

3.2.1 Problem Statement

Consider the boundary value problem on $[-1, 1]$:

$$u''(x) - (4x^2 + 2)u(x) = 0, \quad -1 \leqslant x \leqslant 1,$$

with boundary conditions

$$u(-1) = 1, \quad u(1) = 1.$$

3.2.2 The Exact Solution

The exact solution is

$$u_{\text{exact}}(x) = e^{x^2-1}.$$

Let us verify this claim. The first derivative is

$$u'_{\text{exact}}(x) = 2xe^{x^2-1}.$$

The second derivative is

$$u''_{\text{exact}}(x) = (2 + 4x^2)e^{x^2-1} = (4x^2 + 2)u_{\text{exact}}(x).$$

Substituting into the ODE:

$$u''_{\text{exact}} - (4x^2 + 2)u_{\text{exact}} = (4x^2 + 2)u_{\text{exact}} - (4x^2 + 2)u_{\text{exact}} = 0.\checkmark$$

The boundary conditions are satisfied:

$$u_{\text{exact}}(\pm 1) = e^{1-1} = e^0 = 1.\checkmark$$

3.2.3 Trial Function

To satisfy the boundary conditions automatically, we write the approximation in a form that equals 1 at $x = \pm 1$ regardless of the coefficient values. A convenient choice is:

$$u_2(x) = 1 + (1 - x^2)(a_0 + a_1 x + a_2 x^2).$$

The factor $(1 - x^2)$ vanishes at the endpoints, so

$$u_2(\pm 1) = 1 + 0 \cdot (\cdots) = 1$$

for any values of a_0, a_1, a_2. We have three undetermined coefficients.

Expanding the trial function:

$$\begin{aligned} u_2(x) &= 1 + a_0 + a_1 x + a_2 x^2 - a_0 x^2 - a_1 x^3 - a_2 x^4 \\ &= (1 + a_0) + a_1 x + (a_2 - a_0)x^2 - a_1 x^3 - a_2 x^4. \end{aligned}$$

3.2.4 Computing the Residual

The residual is

$$R(x; a_0, a_1, a_2) = u_{2''}(x) - (4x^2 + 2)u_2(x).$$

Computing the second derivative of u_2:

$$u_2'(x) = a_1 + 2(a_2 - a_0)x - 3a_1x^2 - 4a_2x^3,$$
$$u_2''(x) = 2(a_2 - a_0) - 6a_1x - 12a_2x^2.$$

Substituting into the residual and simplifying (a calculation best verified with computer algebra), the residual is a polynomial of degree six in x with coefficients that depend linearly on a_0, a_1, a_2:

$$R(x) = (-2 - 4a_0 + 2a_2) - 8a_1x + (-4 - 2a_0 - 14a_2)x^2 - 2a_1x^3$$
$$+(4a_0 - 6a_2)x^4 + 4a_1x^5 + 4a_2x^6.$$

3.2.5 Collocation Conditions

We have three unknowns, so we choose three collocation points in the interior of the interval. A simple choice is

$$x_1 = -\frac{1}{2}, \quad x_2 = 0, \quad x_3 = \frac{1}{2}.$$

Setting the residual to zero at these points gives three linear equations:

At $x = -1/2$:

$$R\left(-\frac{1}{2}\right) = -\frac{17}{4}a_0 + \frac{33}{8}a_1 - \frac{25}{16}a_2 - 3 = 0.$$

At $x = 0$:

$$R(0) = -4a_0 + 2a_2 - 2 = 0.$$

At $x = 1/2$:

$$R\left(\frac{1}{2}\right) = -\frac{17}{4}a_0 - \frac{33}{8}a_1 - \frac{25}{16}a_2 - 3 = 0.$$

3.2.6 Solving the System

From the second equation:

$$-4a_0 + 2a_2 = 2 \quad \Rightarrow \quad a_2 = 1 + 2a_0.$$

Adding the first and third equations (the a_1 terms cancel):

$$-\frac{17}{2}a_0 - \frac{25}{8}a_2 = 6.$$

Substituting $a_2 = 1 + 2a_0$:

$$-\frac{17}{2}a_0 - \frac{25}{8}(1 + 2a_0) = 6$$
$$-\frac{17}{2}a_0 - \frac{25}{8} - \frac{25}{4}a_0 = 6$$
$$-\left(\frac{34}{4} + \frac{25}{4}\right)a_0 = 6 + \frac{25}{8}$$
$$-\frac{59}{4}a_0 = \frac{73}{8}$$
$$a_0 = -\frac{73}{118}.$$

Then

$$a_2 = 1 + 2 \cdot \left(-\frac{73}{118}\right) = 1 - \frac{146}{118} = -\frac{28}{118} = -\frac{14}{59}.$$

Subtracting the first equation from the third:

$$-\frac{33}{4}a_1 = 0 \quad \Rightarrow \quad a_1 = 0.$$

The vanishing of a_1 reflects the symmetry of the problem: both the differential equation and the boundary conditions are symmetric about $x = 0$, so the solution must be an even function. An odd coefficient like a_1 would break this symmetry.

3.2.7 The Approximate Solution

Substituting the coefficients back:

$$u_2(x) = 1 + (1 - x^2)\left(-\frac{73}{118} - \frac{14}{59}x^2\right).$$

After simplification, this becomes the even polynomial:

$$u_2(x) = \frac{14}{59}x^4 + \frac{45}{118}x^2 + \frac{45}{118}.$$

The boundary conditions are satisfied:

$$u_2(\pm 1) = \frac{14}{59} + \frac{45}{118} + \frac{45}{118} = \frac{28}{118} + \frac{90}{118} = \frac{118}{118} = 1.\checkmark$$

The implementation of this approximation is straightforward. In Python:

```python
def u_approx(x):
    """Evaluate the collocation approximation."""
    a0, a1, a2 = -73/118, 0, -14/59
    return 1 + (1 - x**2) * (a0 + a1*x + a2*x**2)
```

The equivalent MATLAB implementation:

```matlab
% Collocation coefficients
a0 = -73/118;  a1 = 0;  a2 = -14/59;

% Approximate solution (anonymous function)
u_approx = @(x) 1 + (1 - x.^2) .* (a0 + a1*x + a2*x.^2);
```

3.2.8 Error Analysis

The following table compares the exact and approximate solutions at several points:

x	$u_{\text{exact}}(x)$	$u_{\text{approx}}(x)$	Error
-1	1.00000	1.00000	0.00000
-0.5	0.47237	0.49153	-0.01916
0	0.36788	0.38136	-0.01348
0.5	0.47237	0.49153	-0.01916
1	1.00000	1.00000	0.00000

Table 1: *Comparison of exact and three-coefficient collocation approximation.*

The maximum error is approximately 2×10^{-2}, which is remarkably good for such a low-order approximation. Figure 6 shows the solutions graphically.

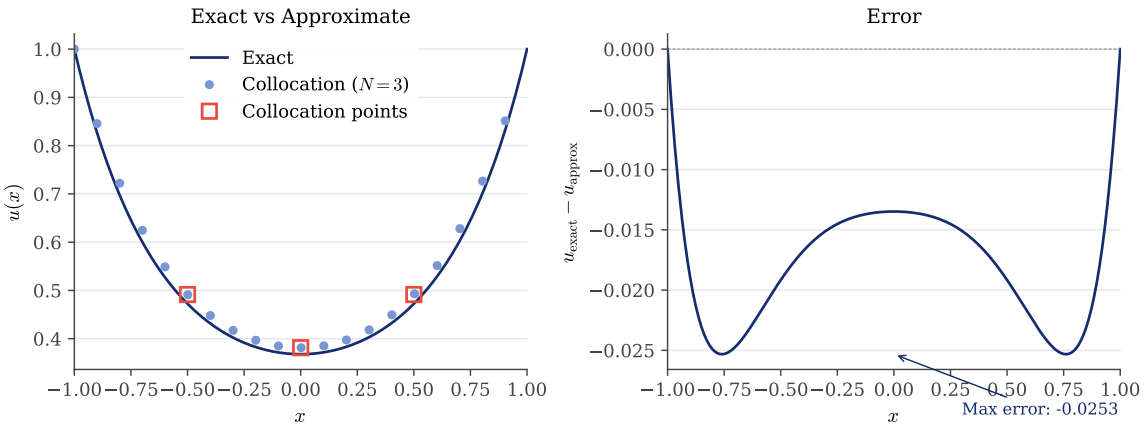

Figure 6: *Left: exact solution $u(x) = e^{x^2-1}$ compared with the three-coefficient collocation approximation. The collocation points $x = -1/2, 0, 1/2$ are marked with squares. Right: the error $u_{\text{exact}} - u_{\text{approx}}$.*

The code that generated this figure is available in both Python and MATLAB:

- `codes/python/ch03_mise_en_bouche/collocation_example1.py`
- `codes/matlab/ch03_mise_en_bouche/collocation_example1.m`

3.3 Collocation versus Galerkin

To compare the two main approaches to the Method of Weighted Residuals, we consider a second example where both methods can be applied with explicit hand calculations.

3.3.1 Problem Statement

Consider the reaction-diffusion equation on $[-1, 1]$:

$$\frac{\mathrm{d}^2 u}{\mathrm{d}x^2} - 4u = -1, \quad -1 \leqslant x \leqslant 1,$$

with homogeneous Dirichlet boundary conditions:

$$u(-1) = 0, \quad u(1) = 0.$$

3.3.2 The Exact Solution

The homogeneous equation $u'' - 4u = 0$ has the general solution $u_h = A\cosh(2x) + B\sinh(2x)$. A particular solution for the constant forcing -1 is $u_p = 1/4$. The general solution is therefore

$$u(x) = A\cosh(2x) + B\sinh(2x) + \frac{1}{4}.$$

The boundary condition $u(-1) = 0$ gives $A\cosh(2) - B\sinh(2) + 1/4 = 0$. The boundary condition $u(1) = 0$ gives $A\cosh(2) + B\sinh(2) + 1/4 = 0$.

Adding these equations: $2A\cosh(2) + 1/2 = 0$, so $A = -1/(4\cosh(2))$. Subtracting: $2B\sinh(2) = 0$, so $B = 0$.

The exact solution is:

$$u_{\text{exact}}(x) = \frac{1}{4}\left(1 - \frac{\cosh(2x)}{\cosh(2)}\right).$$

The maximum value occurs at $x = 0$:

$$u_{\text{exact}}(0) = \frac{1}{4}\left(1 - \frac{1}{\cosh(2)}\right) \approx 0.1834.$$

3.3.3 Trial Function and Basis

Since the boundary conditions are homogeneous, we choose basis functions that automatically vanish at $x = \pm 1$. For a symmetric problem like this one, we use even functions:

$$\varphi_0(x) = 1 - x^2, \quad \varphi_1(x) = (1 - x^2)x^2 = x^2 - x^4.$$

Both functions vanish at $x = \pm 1$ and are even in x. Our trial function is:

$$u_1(x) = a_0\varphi_0(x) + a_1\varphi_1(x) = a_0(1 - x^2) + a_1(x^2 - x^4).$$

3.3.4 The Residual

The operator is $\mathcal{L}u = u'' - 4u$. We compute:

$$\varphi_0'' = -2, \quad \varphi_1'' = 2 - 12x^2.$$

Applying the operator to each basis function:

$$\mathcal{L}\varphi_0 = -2 - 4(1 - x^2) = -6 + 4x^2,$$
$$\mathcal{L}\varphi_1 = (2 - 12x^2) - 4(x^2 - x^4) = 2 - 16x^2 + 4x^4.$$

The residual is:

$$R(x) = a_0\mathcal{L}\varphi_0 + a_1\mathcal{L}\varphi_1 - (-1)$$
$$= a_0(-6 + 4x^2) + a_1(2 - 16x^2 + 4x^4) + 1.$$

3.3.5 Collocation Method

With two unknowns, we need two collocation points. Due to symmetry, we can use points in $[0, 1)$. We choose $x_1 = 0$ and $x_2 = 0.5$.

At $x = 0$:

$$\mathcal{L}\varphi_0(0) = -6, \quad \mathcal{L}\varphi_1(0) = 2.$$
$$R(0) = -6a_0 + 2a_1 + 1 = 0 \quad \Rightarrow \quad 6a_0 - 2a_1 = 1.$$

At $x = 0.5$:

$$\mathcal{L}\varphi_0(0.5) = -6 + 4 \cdot 0.25 = -5,$$
$$\mathcal{L}\varphi_1(0.5) = 2 - 16 \cdot 0.25 + 4 \cdot 0.0625 = 2 - 4 + 0.25 = -1.75.$$
$$R(0.5) = -5a_0 - 1.75a_1 + 1 = 0 \quad \Rightarrow \quad 5a_0 + 1.75a_1 = 1.$$

Solving the system:

$$\begin{cases} 6a_0 - 2a_1 = 1 \\ 5a_0 + 1.75a_1 = 1 \end{cases}$$

Multiply the first equation by 5 and the second by 6:

$$30a_0 - 10a_1 = 5, \quad 30a_0 + 10.5a_1 = 6.$$

Subtracting: $20.5a_1 = 1$, so $a_1 \approx 0.04878$.

Substituting back: $6a_0 = 1 + 2 \cdot 0.04878 = 1.09756$, so $a_0 \approx 0.1829$.

The collocation solution is:

$$u_{\text{coll}}(x) \approx 0.1829(1 - x^2) + 0.0488(x^2 - x^4).$$

The following Python code assembles and solves the collocation system:

```python
def solve_collocation():
    """Solve the collocation system at x = 0 and x = 0.5."""
    # Operator L[φ] = φ'' - 4φ applied to basis functions
    L_phi0 = lambda x: -2 - 4*(1 - x**2)
    L_phi1 = lambda x: (2 - 12*x**2) - 4*(x**2 - x**4)

    # Build system matrix at collocation points
    A = np.array([[L_phi0(0.0), L_phi1(0.0)],
                  [L_phi0(0.5), L_phi1(0.5)]])
    b = np.array([-1.0, -1.0])  # RHS from f = -1
    return np.linalg.solve(A, b)
```

The equivalent MATLAB implementation:

```matlab
% Operator L = d²/dx² - 4 applied to basis functions
L_phi0 = @(x) -2 - 4*(1 - x.^2);
L_phi1 = @(x) (2 - 12*x.^2) - 4*(x.^2 - x.^4);

% Build and solve collocation system
A_coll = [L_phi0(0.0), L_phi1(0.0);
          L_phi0(0.5), L_phi1(0.5)];
coeffs = A_coll \ [-1; -1];
```

3.3.6 Galerkin Method

The Galerkin conditions require the residual to be orthogonal to each basis function:

$$\int_{-1}^{1} R(x)\varphi_k(x)\,\mathrm{d}x = 0, \quad k = 0, 1.$$

This gives a symmetric matrix system $\boldsymbol{A}\boldsymbol{a} = \boldsymbol{b}$ where:

$$A_{ij} = \int_{-1}^{1} \mathcal{L}\varphi_j(x) \cdot \varphi_i(x)\,\mathrm{d}x,$$

$$b_i = \int_{-1}^{1} (-1) \cdot \varphi_i(x)\,\mathrm{d}x = -\int_{-1}^{1} \varphi_i(x)\,\mathrm{d}x.$$

Computing the integrals (using standard formulas for powers of x):

For $\varphi_0 = 1 - x^2$:

$$\int_{-1}^{1} \varphi_0(x)\,\mathrm{d}x = \int_{-1}^{1} (1 - x^2)\,\mathrm{d}x = \left[x - \frac{x^3}{3}\right]_{-1}^{1} = 2 - \frac{2}{3} = \frac{4}{3}.$$

For $\varphi_1 = x^2 - x^4$:

$$\int_{-1}^{1} \varphi_1(x)\,\mathrm{d}x = \left[\frac{x^3}{3} - \frac{x^5}{5}\right]_{-1}^{1} = \frac{2}{3} - \frac{2}{5} = \frac{4}{15}.$$

The right-hand side is:

$$b_0 = -\frac{4}{3}, \quad b_1 = -\frac{4}{15}.$$

The matrix entries require more computation. Using computer algebra or careful integration:

$$A_{00} = -\frac{104}{15}, \quad A_{01} = A_{10} = -\frac{8}{7}, \quad A_{11} = -\frac{328}{315}.$$

Solving the system yields:

$$a_0 \approx 0.1832, \quad a_1 \approx 0.0550.$$

The Galerkin method requires numerical integration to assemble the system. In Python:

Python

```python
def solve_galerkin():
    """Solve using Galerkin: ⟨R, φₖ⟩ = 0 for k = 0, 1."""
    from scipy import integrate

    # Matrix entries: A_{ij} = ∫ L[φⱼ] φᵢ dx
    A00, _ = integrate.quad(lambda x: L_phi0(x) * phi0(x), -1, 1)
    A01, _ = integrate.quad(lambda x: L_phi1(x) * phi0(x), -1, 1)
    A10, _ = integrate.quad(lambda x: L_phi0(x) * phi1(x), -1, 1)
    A11, _ = integrate.quad(lambda x: L_phi1(x) * phi1(x), -1, 1)

    A = np.array([[A00, A01], [A10, A11]])
    b0, _ = integrate.quad(lambda x: -phi0(x), -1, 1)
    b1, _ = integrate.quad(lambda x: -phi1(x), -1, 1)
    return np.linalg.solve(A, [b0, b1])
```

The equivalent MATLAB implementation uses the built-in `integral` function:

Matlab

```matlab
% Matrix entries: A_{ij} = ∫ L[φⱼ] φᵢ dx
A00 = integral(@(x) L_phi0(x) .* phi0(x), -1, 1);
A01 = integral(@(x) L_phi1(x) .* phi0(x), -1, 1);
A10 = integral(@(x) L_phi0(x) .* phi1(x), -1, 1);
A11 = integral(@(x) L_phi1(x) .* phi1(x), -1, 1);

A_gal = [A00, A01; A10, A11];

% RHS: b_i = ∫ f φᵢ dx where f = -1
b0 = integral(@(x) -phi0(x), -1, 1);
b1 = integral(@(x) -phi1(x), -1, 1);
coeffs = A_gal \ [b0; b1];
```

3.3.7 Comparison

The following table compares the two methods at the central point $x = 0$:

Method	$u(0)$	**Absolute Error**
Exact	0.1835	0
Collocation ($N = 2$)	0.1829	0.0006
Galerkin ($N = 2$)	0.1832	0.0003

Table 2: *Comparison of spectral approximations at the central maximum.*

For this problem, the Galerkin method is more accurate both at the central point and in a global sense. This is consistent with the error plot in Figure 7, which shows the Galerkin error (green) remaining closer to zero across the entire interval. The Galerkin method minimizes the error in a root-mean-square sense, which typically leads to better overall accuracy for smooth problems.

Figure 7 shows both approximate solutions compared to the exact solution, along with the error profiles.

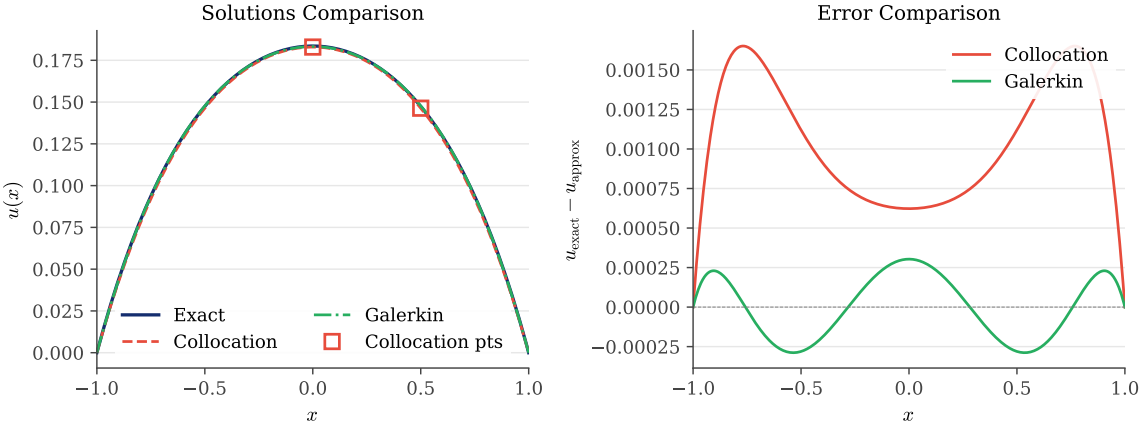

Figure 7: *Left: exact solution compared with collocation and Galerkin approximations. The collocation points $x = 0$ and $x = 0.5$ are marked. Right: error profiles for both methods.*

The code that generated this figure is available in both Python and MATLAB:

- `codes/python/ch03_mise_en_bouche/collocation_vs_galerkin.py`
- `codes/matlab/ch03_mise_en_bouche/collocation_vs_galerkin.m`

3.4 Conclusions and Questions

These simple examples illuminate several important features of spectral methods:

1. **Automatic satisfaction of boundary conditions**. By choosing trial functions that vanish at the boundaries (or equal the prescribed boundary values), we eliminate the boundary conditions from the algebraic system.
2. **Symmetry exploitation**. When the problem has symmetry, the solution inherits that symmetry. In our examples, the vanishing of odd coefficients ($a_1 = 0$) reflects the even symmetry of the exact solution.
3. **Simplicity of implementation**. Even with hand calculations, we can obtain remarkably accurate approximations with just a few coefficients.
4. **Trade-offs between methods**. Collocation is simpler to implement (no integrals to compute), while Galerkin typically provides better accuracy by minimizing a global error measure.

These examples raise important questions that we will address in subsequent chapters:

- **What is the optimal choice of basis functions?** Using simple powers of x works for small N, but becomes numerically unstable for large N. Chebyshev and Fourier bases are far superior.
- **What are the optimal collocation points?** Our ad hoc choices $x = -1/2, 0, 1/2$ worked well, but there exist optimal point distributions (Gauss and Gauss–Lobatto points) derived from orthogonal polynomial theory.
- **How fast does the error decrease as N increases?** For smooth solutions, spectral methods achieve exponential convergence (the error decreases like c^{-N} for some $c > 1$), which is dramatically faster than the algebraic convergence $O(N^{-p})$ of finite difference and finite element methods.

The following chapters will develop the theory and algorithms needed to answer these questions and to apply spectral methods to a wide range of problems.

3.5 A Broader Perspective

For demanding students who wish to understand how spectral methods fit into the wider landscape of numerical analysis, this section provides a high-level comparison with other approaches, particularly finite element methods. The discussion follows [3, Section 1.3].

3.5.1 Local versus Global Basis Functions

The fundamental distinction between spectral methods and finite element methods lies in the *support* of the basis functions. In finite element methods, the computational domain is divided into many small sub-intervals (or triangles, tetrahedra in higher dimensions), and the basis functions $\varphi_n(x)$ are *local*: they are polynomials of fixed, low degree (typically linear or quadratic) that are non-zero only over one or two adjacent elements.

In contrast, spectral methods use *global* basis functions. Each $\varphi_n(x)$ is a polynomial (or trigonometric polynomial) of potentially high degree that is non-zero (except at isolated points) over the entire computational domain. This global character is both the source of spectral methods' power and the reason for some of their limitations.

Figure 8 illustrates this distinction schematically. The finite element basis function (left) has compact support and contributes to the solution only locally. The spectral basis function (right) influences the solution everywhere.

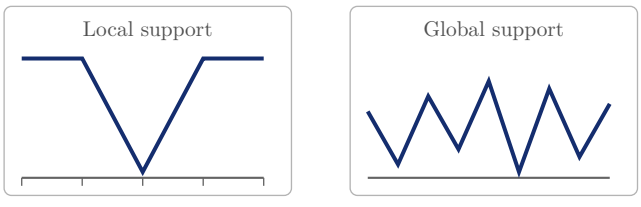

Figure 8: *Schematic comparison of local (finite element) and global (spectral) basis functions. The finite element "hat function" is non-zero only over two adjacent elements, while the spectral basis function oscillates across the entire domain.*

3.5.2 Refinement Strategies

When a numerical approximation is insufficiently accurate, there are three fundamentally different strategies to improve it, illustrated schematically in Figure 9:

1. ***h*-refinement**: Subdivide each element into smaller pieces, reducing the mesh spacing h uniformly throughout the domain. This increases the number of elements while keeping the polynomial degree fixed.
2. ***r*-refinement** (adaptive): Redistribute the mesh points, clustering them in regions where the solution has steep gradients or other features requiring high resolution. The total number of degrees of freedom remains roughly constant.
3. ***p*-refinement**: Keep the mesh fixed while increasing p, the polynomial degree within each element. For a single-element domain, this is precisely what spectral methods do.

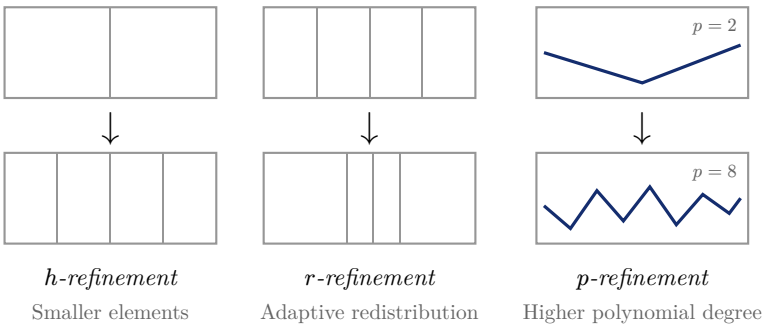

Figure 9: *Three strategies for improving accuracy in numerical methods. Spectral methods employ p-refinement: increasing the polynomial degree while using a single element (or few elements) spanning the entire domain.*

Spectral methods can be viewed as the extreme form of p-refinement: a single element spans the entire domain, and accuracy is improved solely by increasing the polynomial degree. This strategy is devastatingly effective when the solution is smooth, but struggles when the solution has discontinuities or sharp gradients.

3.5.3 Trade-offs: Sparse versus Full Matrices

The choice between local and global basis functions entails fundamental trade-offs:

Finite element advantages:

- *Sparse matrices*: Since each basis function is non-zero over only a few elements, the stiffness matrix has mostly zero entries. Sparse matrix solvers can exploit this structure, dramatically reducing computational cost for large systems.
- *Geometric flexibility*: The small elements (triangles, tetrahedra) can be fitted to irregularly shaped domains like automobile bodies or aircraft wings.

Finite element disadvantages:

- *Low accuracy per degree of freedom*: Each basis function is a polynomial of low degree, so many elements are needed for high accuracy.

Spectral method advantages:

- *High accuracy for smooth problems*: The high-degree global polynomials capture smooth solutions with far fewer degrees of freedom.

- *Efficiency with iterative solvers*: When fast iterative methods are used, spectral methods can be much more efficient than low-order methods for many problem classes.

Spectral method disadvantages:

- *Full matrices*: The global basis functions create dense matrices where most entries are non-zero.
- *Geometric limitations*: Spectral methods are most natural on simple domains (intervals, rectangles, disks) and require more sophisticated techniques for irregular geometries.

For problems with smooth solutions on regular domains (many important problems in fluid dynamics, quantum mechanics, and wave propagation fall into this category), the accuracy advantage of spectral methods often outweighs the matrix structure disadvantage.

3.5.4 Spectral Element Methods

A natural question arises: can we combine the geometric flexibility of finite elements with the high accuracy of spectral methods? The answer is yes, through *spectral element methods*.

In spectral element methods, the domain is subdivided into elements (as in finite elements), but within each element, the polynomial degree p is chosen to be moderately high, typically $p = 6$ to 8. This hybrid approach inherits several advantages:

- The element subdivision provides geometric flexibility and matrix sparsity.
- The high polynomial degree within each element provides spectral-like accuracy.
- The theoretical framework is essentially the same as for global spectral methods.

Spectral element codes are typically written so that p is a user-adjustable parameter, allowing practitioners to balance accuracy and cost for their specific application. We will not develop spectral element methods in detail in this book, but the reader should be aware that much of the theory developed for global spectral methods transfers directly to the spectral element context.

3.5.5 The Convergence of Methods at High Order

Perhaps the most profound insight from the comparison between finite element and spectral methods is this: *for sufficiently high polynomial degree, the two approaches become essentially equivalent.*

Low-order finite elements (linear, quadratic) can be derived, justified, and implemented without knowledge of Fourier or Chebyshev convergence theory. However, as the polynomial degree increases, ad hoc approaches become increasingly ill-conditioned and numerically unstable. The only practical way to implement well-behaved high-order finite elements (say, sixth order or higher) is to use the technology of spectral methods: Chebyshev or Legendre basis functions, Gaussian quadrature, and the convergence theory we will develop in subsequent chapters.

Thus, the question "Are finite elements or spectral methods better?" becomes somewhat artificial for high-order approximations. The real question is: *Does the problem at hand require high-order accuracy, or is second or fourth order sufficient?*

When the solution is smooth and high accuracy is needed, the spectral/high-order approach is clearly superior. When the solution has discontinuities, shocks, or boundary layers, or when the geometry is highly irregular, low-order methods with adaptive mesh refinement may be more practical. The wise practitioner chooses the tool appropriate to the problem.

CHAPTER 4

The Geometry of Nodes

Before we can differentiate functions numerically using spectral methods, we must first understand how to represent them. Polynomial interpolation is the process of constructing a polynomial that passes through a given set of data points. It is the foundation upon which pseudospectral methods are built. In this chapter, we explore a fascinating paradox: while polynomial interpolation seems entirely straightforward, the choice of interpolation nodes determines whether the method succeeds brilliantly or fails catastrophically. The story begins with a surprising discovery by the German mathematician Carl Runge in 1901. Attempting to approximate a simple, smooth function by interpolating polynomials, Runge found that increasing the polynomial degree made the approximation *worse*, not better. This counterintuitive phenomenon, now bearing his name, reveals deep connections between numerical analysis, complex analysis, and potential theory.

Our journey through this "étude in grid geometry" will explain the Runge phenomenon, develop the theoretical framework of potential theory that predicts where interpolation succeeds or fails, and introduce the Chebyshev points that form the cornerstone of practical spectral methods.

4.1 The Problem: Polynomial Interpolation

4.1.1 Lagrange Interpolation

Given $N+1$ distinct points $\{x_0, x_1, ..., x_N\}$ on an interval $[a, b]$ and corresponding function values $\{f_0, f_1, ..., f_N\}$ where $f_j = f(x_j)$, there exists a unique polynomial $p_N(x)$ of degree at most N such that

$$p_N(x_j) = f_j, \quad j = 0, 1, ..., N.$$

This interpolating polynomial can be written explicitly using the *Lagrange formula*:

$$p_N(x) = \sum_{k=0}^{N} f_k L_k(x),$$

where the *Lagrange basis polynomials* are

$$L_k(x) = \prod_{j=0, j\neq k}^{N} \frac{x - x_j}{x_k - x_j}.$$

Each basis polynomial $L_k(x)$ has the cardinal property: it equals 1 at x_k and 0 at all other nodes. This property ensures that substituting any node x_j into the Lagrange formula yields f_j.

4.1.2 Interpolation versus Best Approximation

It is important to distinguish interpolation from best approximation. The *Weierstrass Approximation Theorem*, proved by Karl Weierstrass in 1885 when he was 70 years old, guarantees that any continuous function on $[a, b]$ can be uniformly approximated to arbitrary accuracy by polynomials: if f is continuous on $[-1, 1]$ and $\varepsilon > 0$ is arbitrary, then there exists a polynomial p such that $\|f - p\|_\infty < \varepsilon$.

The theorem was independently discovered at about the same time by Carl Runge, the same mathematician whose name we attached to the phenomenon of divergent equispaced interpolation. This coincidence is not accidental: Runge's deep investigations into polynomial

approximation led him to both the positive existence result and the negative convergence phenomenon.

Weierstrass's original proof is remarkably elegant, connecting approximation theory to the heat equation. The idea is to extend $f(x)$ to a function $\tilde{f}$ with compact support on the real line, then convolve it with the Gaussian kernel

$$\varphi(x) = \frac{e^{-x^2/4t}}{\sqrt{4\pi t}}.$$

This convolution solves the diffusion equation $\partial u/\partial t = \partial^2 u/\partial x^2$ and converges uniformly to f as $t \to 0$. Since the convolution is an entire function (analytic throughout the complex plane), it has a uniformly convergent Taylor series that can be truncated to yield polynomial approximations of arbitrary accuracy.

The theorem stimulated an extraordinary amount of mathematics in the early twentieth century, with many alternative proofs discovered in rapid succession: Picard (1891), Lebesgue (1898, in his first paper at age 23), Fejér (1900, at age 20), Landau (1908), Jackson (1911), and Bernstein (1912), among others.

Bernstein Polynomials

The most constructive proof was given by Sergei Bernstein in 1912. For $f \in C([0,1])$, the *Bernstein polynomial* of degree n is defined by

$$B_n(x) = \sum_{k=0}^{n} f(k/n) \binom{n}{k} x^k (1-x)^{n-k}.$$

Bernstein proved that $B_n(x) \to f(x)$ uniformly as $n \to \infty$. The convergence can be understood through a beautiful probabilistic interpretation: $B_n(x)$ represents the expected value of f evaluated at the proportion of heads in n independent coin flips, where each coin has probability x of landing heads. As $n \to \infty$, the law of large numbers ensures this proportion concentrates near x, and the expected value converges to $f(x)$.

Bernstein polynomials provide explicit approximations that converge for *any* continuous function, though the convergence is typically slow: $O(1/\sqrt{n})$ for Lipschitz continuous functions. This is far inferior to the geometric convergence achieved by Chebyshev interpolation for analytic functions.

The Limits of Interpolation

Despite the Weierstrass theorem's guarantee that approximating polynomials exist, a fundamental negative result constrains how we can find them. The *Faber–Bernstein theorem* (Faber 1914, Bernstein 1919) asserts that there is no fixed array of interpolation grids with $1, 2, 3, \ldots$ points that achieves convergence as $n \to \infty$ for *all* continuous functions.

This result might seem discouraging, but it has led to an unfortunate overemphasis on pathological functions in the numerical analysis literature. The practical reality is far more favorable: for functions with even modest smoothness, Chebyshev interpolation converges beautifully. The Weierstrass theorem applies to highly irregular functions like $\sin(1/x)\sin(1/\sin(1/x))$, which oscillates infinitely often near infinitely many points. Such functions are mathematical curiosities, not the smooth solutions to differential equations that arise in scientific computing.

Interpolation constructs a specific polynomial by enforcing exact agreement at the nodes. The hope is that as $N \to \infty$, the interpolating polynomials p_N converge uniformly to f. As we shall see, this hope is fulfilled for some node distributions but dramatically fails for others. The key insight of this chapter is that for smooth functions and well-chosen nodes, interpolation not only succeeds but achieves the optimal approximation rate up to a small constant factor.

4.1.3 Equispaced Nodes

The most natural choice of interpolation nodes is equally spaced points:

$$x_j = -1 + \frac{2j}{N}, \quad j = 0, 1, ..., N.$$

These nodes divide the interval $[-1, 1]$ into N equal subintervals. For low-degree polynomials and well-behaved functions, equispaced interpolation works reasonably well. However, a fundamental problem emerges as N increases.

4.2 The Runge Phenomenon

4.2.1 A Smooth but Troublesome Function

In 1901, Carl Runge studied the interpolation of a deceptively simple function:

$$f(x) = \frac{1}{1 + 25x^2}.$$

This *Runge function* is infinitely differentiable on the entire real line. Its graph is a smooth bell curve centered at the origin with maximum value $f(0) = 1$ and asymptotic decay to zero as $|x| \to \infty$.

Runge discovered that polynomial interpolation on equispaced nodes *diverges* for this function. Rather than improving with increasing N, the interpolating polynomials develop wild oscillations near the endpoints $x = \pm 1$.

4.2.2 Numerical Demonstration

Figure 10 illustrates this phenomenon. For $N = 6$, the interpolant reasonably approximates the function. But for $N = 10$ and $N = 14$, the approximation deteriorates dramatically at the boundaries, with oscillations that grow unboundedly as N increases.

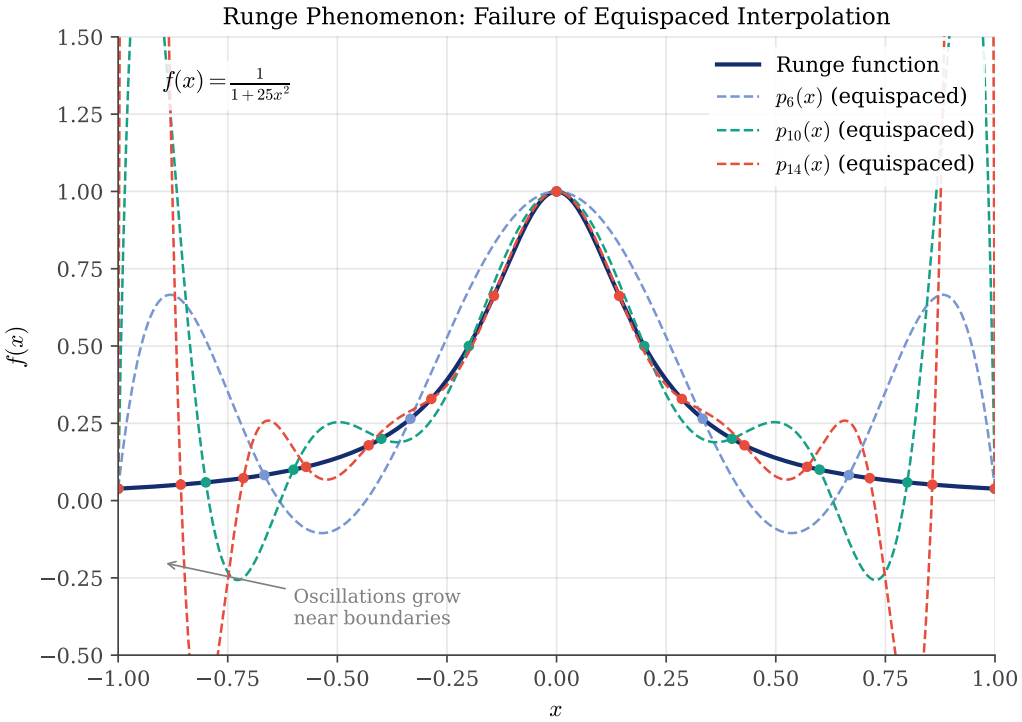

Figure 10: *The Runge phenomenon: polynomial interpolation of $f(x) = 1/(1 + 25x^2)$ on equispaced nodes. As the polynomial degree increases, oscillations near the boundaries $x = \pm 1$ grow unboundedly. The exact function (solid) and interpolants (dashed) are shown for $N = 6$, 10, and 14.*

The following Python code computes the Lagrange interpolant for a given set of nodes:

```python
def lagrange_interpolate(x_nodes, f_nodes, x_eval):
    """Evaluate the Lagrange interpolating polynomial."""
    n = len(x_nodes)
    p_eval = np.zeros_like(x_eval)

    for k in range(n):
        L_k = np.ones_like(x_eval)
        for j in range(n):
            if j != k:
                L_k *= (x_eval - x_nodes[j]) / (x_nodes[k] - x_nodes[j])
        p_eval += f_nodes[k] * L_k

    return p_eval
```

The equivalent MATLAB implementation:

```matlab
function p = lagrange_interp(x_nodes, f_nodes, x_eval)
    n = length(x_nodes);
    p = zeros(size(x_eval));

    for k = 1:n
        L_k = ones(size(x_eval));
        for j = 1:n
```

```
            if j ~= k
                L_k = L_k .* (x_eval - x_nodes(j)) / (x_nodes(k) - x_nodes(j));
            end
        end
        p = p + f_nodes(k) * L_k;
    end
end
```

The code generating Figure 10 is available in:

- `codes/python/ch04_geometry_of_nodes/runge_phenomenon.py`
- `codes/matlab/ch04_geometry_of_nodes/runge_phenomenon.m`

4.2.3 Why Does This Happen?

The Runge phenomenon seems paradoxical: how can adding more information (more interpolation points) make the approximation worse? The answer lies in the *Lebesgue constant*, which we will explore in Section 4.5. But first, let us develop a deeper understanding through potential theory.

4.3 Theoretical Explanation: Potential Theory

4.3.1 Singularities in the Complex Plane

The key to understanding the Runge phenomenon lies in extending our view from the real line to the complex plane. The Runge function has singularities (poles) where its denominator vanishes:

$$1 + 25z^2 = 0 \quad \Rightarrow \quad z = \pm\frac{i}{5} = \pm 0.2i.$$

These poles lie on the imaginary axis, at a distance of only 0.2 from the real interval $[-1, 1]$. This proximity is responsible for the divergence of equispaced interpolation.

4.3.2 The Potential Function

The convergence of polynomial interpolation is governed by a *potential function* associated with the node distribution. For a distribution with density $\mu(x)$ on $[-1, 1]$, the potential at a point z in the complex plane is:

$$\varphi(z) = -\int_{-1}^{1} \mu(x) \ln|z - x| \, \mathrm{d}x.$$

The *equipotential curves* $\varphi(z) = \text{const}$ form closed contours around the interval $[-1, 1]$. The largest equipotential curve that does not enclose any singularity of f determines the region of convergence.

4.3.3 Uniform versus Chebyshev Density

For equispaced nodes, the density is asymptotically uniform: $\mu(x) = 1/2$. The resulting equipotential curves are roughly elliptical, but they extend only a short distance from the interval before reaching the Runge function's poles at $\pm 0.2i$.

For Chebyshev nodes (which we introduce in the next section), the density is:

$$\mu(x) = \frac{1}{\pi\sqrt{1-x^2}}.$$

This density diverges at the endpoints, concentrating nodes near $x = \pm 1$. The corresponding potential simplifies to:

$$\varphi(z) = \ln|z + \sqrt{z^2 - 1}| - \ln 2 = \ln \rho - \ln 2,$$

where $\rho = |z + \sqrt{z^2 - 1}|$. The equipotential curves are *Bernstein ellipses* with foci at ± 1.

Figure 11 compares the equipotential curves for both distributions, showing why Chebyshev interpolation succeeds where equispaced interpolation fails.

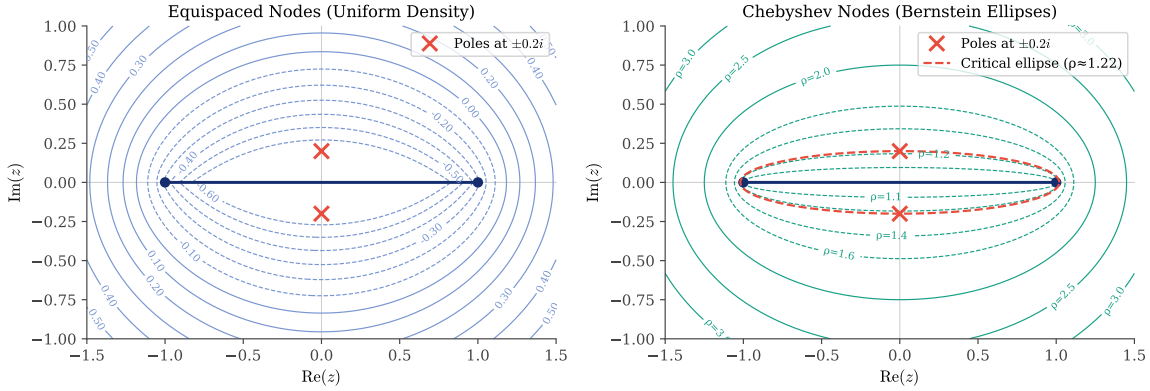

Figure 11: *Equipotential curves in the complex plane. Left: uniform node density (equispaced nodes). Right: Chebyshev density, where equipotentials are Bernstein ellipses. The poles of the Runge function at $z = \pm 0.2i$ are marked with crosses. For Chebyshev interpolation, the critical ellipse passing through the poles determines the convergence rate.*

The code generating Figure 11 is available in:

- `codes/python/ch04_geometry_of_nodes/equipotential_curves.py`
- `codes/matlab/ch04_geometry_of_nodes/equipotential_curves.m`

4.4 The Solution: Chebyshev Points

4.4.1 Definition and Geometric Construction

The *Chebyshev-Gauss-Lobatto points* (often simply called Chebyshev points) are defined by:

$$x_j = \cos\left(\frac{j\pi}{N}\right), \quad j = 0, 1, ..., N.$$

These points have a beautiful geometric interpretation: they are the projections onto the x-axis of $N+1$ equally spaced points on the upper semicircle of the unit circle. Figure 12 illustrates this construction.

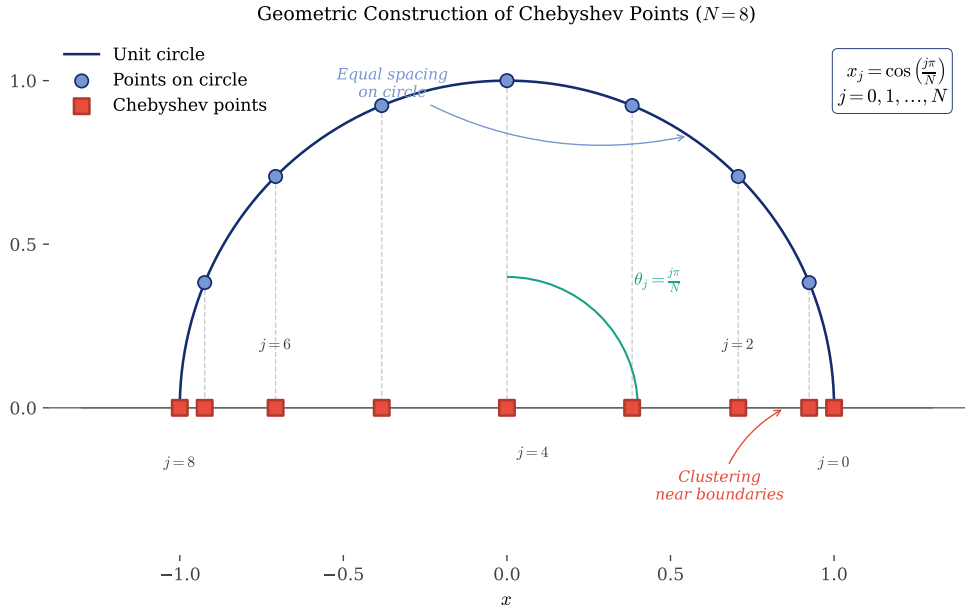

Figure 12: *Geometric construction of Chebyshev points. Points equally spaced on the upper semicircle (by angle $\theta_j = j\pi/N$) project vertically onto the Chebyshev nodes on the x-axis. Equal spacing on the circle maps to clustering near the endpoints $x = \pm 1$.*

4.4.2 Node Density and Clustering

The geometric construction reveals why Chebyshev points cluster near the endpoints. Equal angular spacing on the circle corresponds to denser horizontal spacing near $x = \pm 1$. The node density is approximately:

$$\rho(x) \approx \frac{N}{\pi\sqrt{1-x^2}},$$

which diverges as $x \to \pm 1$. This clustering counteracts the growth of interpolation error at the boundaries.

4.4.3 Chebyshev Interpolation of the Runge Function

Figure 13 demonstrates the dramatic improvement when using Chebyshev points instead of equispaced nodes for the Runge function. Where equispaced interpolation diverges, Chebyshev interpolation converges rapidly and uniformly.

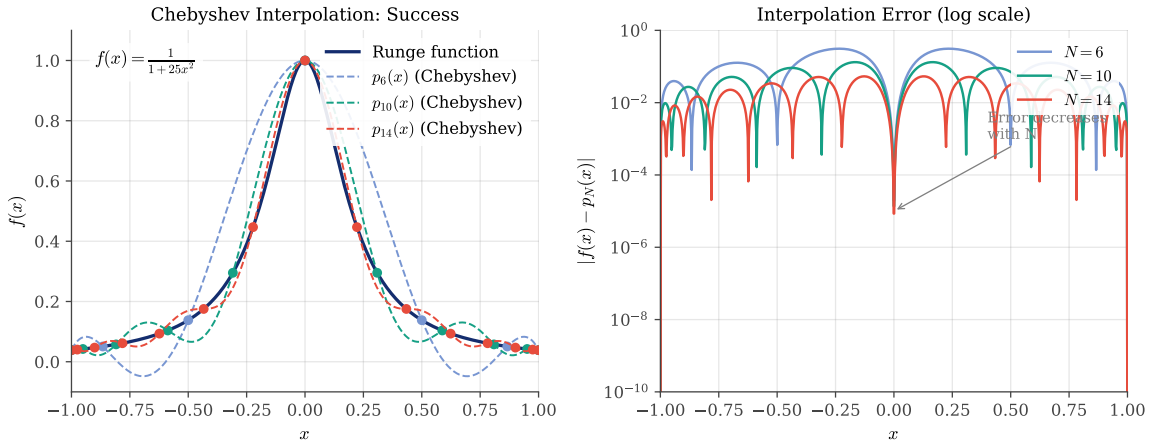

Figure 13: *Chebyshev interpolation of the Runge function. Left: the function and interpolants for $N = 6$, 10, and 14. Right: interpolation error on a logarithmic scale. Unlike equispaced interpolation, the error decreases rapidly with increasing N.*

The Chebyshev nodes in Python and MATLAB:

```python
def chebyshev_nodes(N):
    """Chebyshev-Gauss-Lobatto points on [-1, 1]."""
    j = np.arange(N + 1)
    return np.cos(j * np.pi / N)
```

```matlab
% Chebyshev-Gauss-Lobatto points
j = 0:N;
x_cheb = cos(j * pi / N);
```

Note that this formula produces nodes ordered from right to left: $x_0 = \cos(0) = +1$ down to $x_N = \cos(\pi) = -1$. This ordering is natural for the cosine function and is the standard convention in spectral methods.

The code generating Figure 13 is available in:

- `codes/python/ch04_geometry_of_nodes/chebyshev_success.py`
- `codes/matlab/ch04_geometry_of_nodes/chebyshev_success.m`

4.5 Lagrange Basis Functions and Lebesgue Constants

4.5.1 The Interpolation Operator

To understand why node choice matters so dramatically, we formalize the interpolation process. For any continuous function $u \in C([-1, 1])$, there exists a unique interpolating polynomial $\mathcal{P}_N(x) \in \mathbb{P}_N[\mathbb{R}]$ of degree N satisfying:

$$\mathcal{P}_N(x_j) = u_j \equiv u(x_j), \quad j = 0, 1, ..., N.$$

We denote this interpolating polynomial as $\mathcal{I}_N[u]$, emphasizing its role as a *linear operator* acting on continuous functions.

The *best approximating polynomial* $\mathcal{P}^* \in \mathbb{P}_N[\mathbb{R}]$ is defined by:

$$\|u - \mathcal{P}^*\|_\infty = \inf_{\mathcal{P} \in \mathbb{P}_N[\mathbb{R}]} \|u - \mathcal{P}\|_\infty.$$

While it is not obliged that $\mathcal{P}^* \equiv \mathcal{I}_N[u]$, since $\mathcal{P}^* \in \mathbb{P}_N[\mathbb{R}]$, we necessarily have $\mathcal{P}^* \equiv \mathcal{I}_N[u]$ when interpolating a polynomial of degree at most N.

4.5.2 The Lebesgue Function

Examining the Lagrange basis functions more closely, we define the *Lebesgue function* as the sum of absolute values of all basis polynomials:

$$\Lambda_N(x) = \sum_{k=0}^{N} |L_k(x)|.$$

The *Lebesgue constant* is its maximum over the interval:

$$\Lambda_N = \max_{x \in [-1,1]} \Lambda_N(x).$$

4.5.3 Error Bounds via Operator Norm

The Lebesgue constant bounds the interpolation error through a beautiful argument. For any continuous u with best approximation $\mathcal{P}^*$:

$$\begin{aligned}
\|u - \mathcal{I}_N[u]\|_\infty &= \|u - \mathcal{P}^* + \mathcal{P}^* - \mathcal{I}_N[u]\|_\infty \\
&\equiv \|u - \mathcal{P}^* + \mathcal{I}_N[\mathcal{P}^*] - \mathcal{I}_N[u]\|_\infty \\
&\leqslant \|u - \mathcal{P}^*\|_\infty + \|\mathcal{I}_N[\mathcal{P}^*] - \mathcal{I}_N[u]\|_\infty \\
&\leqslant \|u - \mathcal{P}^*\|_\infty + \|\mathcal{I}_N\| \cdot \|\mathcal{P}^* - u\|_\infty \\
&\leqslant (1 + \|\mathcal{I}_N\|) \cdot \|u - \mathcal{P}^*\|_\infty.
\end{aligned}$$

The norm of the interpolation operator $\mathcal{I}_N[\cdot]$ is defined as:

$$\|\mathcal{I}_N\| \coloneqq \sup_{\|u\|_\infty = 1} \|\mathcal{I}_N[u]\|_\infty.$$

This operator norm is precisely the Lebesgue constant Λ_N for the node set $\{x_j\}_{j=0}^N$.

Thus, if p_N^* is the best polynomial approximation of degree N to f, and p_N is the Lagrange interpolant, then:

$$\|f - p_N\|_\infty \leqslant (1 + \Lambda_N)\|f - p_N^*\|_\infty = (1 + \Lambda_N)E_N(f),$$

where $E_N(f)$ denotes the best approximation error.

This bound reveals the problem with equispaced nodes: even if $E_N(f)$ decreases with N (as guaranteed by the Weierstrass theorem), the factor $(1 + \Lambda_N)$ may grow so fast that the product increases.

4.5.4 Asymptotic Growth Rates

The growth rate of Λ_N depends critically on the node distribution:

- **Chebyshev nodes**: $\Lambda_N^{\text{Ch}} = \frac{2}{\pi} \ln N + O(1)$ (logarithmic growth)
- **Legendre nodes**: $\Lambda_N^{\text{Leg}} = O\left(\sqrt{N}\right)$ (slow algebraic growth)
- **Equispaced nodes**: $\Lambda_N^{\text{eq}} \approx \frac{2^{N+1}}{eN \ln N}$ (exponential growth!)

The exponential growth of the Lebesgue constant for equispaced nodes explains the Runge phenomenon: even though the best approximation error decreases geometrically for smooth functions, the exponentially growing factor $(1 + \Lambda_N)$ eventually dominates.

Remark. As shown by Vértesi [5], the Lebesgue constant for Chebyshev nodes is remarkably close to the smallest possible Lebesgue constant for *any* node distribution:

$$\Lambda_N^{\mathrm{Ch}} = \frac{2}{\pi}\left(\ln N + \gamma + \ln\frac{8}{\pi}\right) + o(1),$$
$$\Lambda_N^{\min} = \frac{2}{\pi}\left(\ln N + \gamma + \ln\frac{4}{\pi}\right) + o(1),$$

where $\gamma \approx 0.5772156649...$ is the Euler–Mascheroni constant. The difference between these two expressions is only $\frac{2}{\pi}\ln 2 \approx 0.44$, demonstrating that Chebyshev nodes are essentially optimal for polynomial interpolation.

4.5.5 Derivation of the Lebesgue Constant for Equi-Spaced Interpolation

We now derive the asymptotic formula for the Lebesgue constant of equi-spaced nodes. Let $L_k(x)$ denote the fundamental Lagrange polynomials and define the equidistant Lebesgue constant by

$$1 + \Lambda_N^{\mathrm{eq}} = \max_{x\in[-1,1]} \sum_{k=0}^{N} |L_k(x)|.$$

The interpretation is the usual one: if the interpolation error in the max norm satisfies $\|f - p_N\|_\infty \leqslant \varepsilon$, then the data values fed into the interpolation formula may differ from the exact polynomial values $p_N(x_j)$ by at most ε at each node x_j, and the factor $1 + \Lambda_N^{\mathrm{eq}}$ is the largest possible amplification of these perturbations by Lagrange interpolation.

The size of the interval does not affect the value of the Lebesgue constant, so for convenience we work with equi-spaced nodes on $[0, N]$ instead of $[-1, 1]$:

$$x_j = j, \quad j = 0, 1, ..., N.$$

For these nodes the k-th fundamental Lagrange polynomial can be written as

$$L_k(x) = \prod_{j=0, j\neq k}^{N} \frac{x-j}{k-j} = \frac{x(x-1)\cdots(x-k+1)(x-k-1)\cdots(x-N)}{k(k-1)\cdots 1\cdot(-1)\cdots(k-N)}.$$

Hence

$$1 + \Lambda_N^{\mathrm{eq}} = \max_{x\in[0,1]} \sum_{k=0}^{N} \left| \frac{x(x-1)\cdots(x-k+1)(x-k-1)\cdots(x-N)}{k(k-1)\cdots 1\cdot(-1)\cdots(k-N)} \right|.$$

We now multiply numerator and denominator of each term by $x - k$. This yields

$$x(x-1)\cdots(x-k+1)(x-k)(x-k-1)\cdots(x-N) = x(1-x)(2-x)\cdots(N-x),$$

so that

$$|L_k(x)| = \frac{x(1-x)(2-x)\cdots(N-x)}{|k-x|k!(N-k)!}.$$

Consequently,

$$1 + \Lambda_N^{\mathrm{eq}} = \max_{x\in[0,1]} \sum_{k=0}^{N} \frac{x(1-x)(2-x)\cdots(N-x)}{|k-x|k!(N-k)!}.$$

Using the Gamma function and repeated application of $\Gamma(z+1) = z\Gamma(z)$, we obtain

$$(N-x)(N-1-x)\cdots(1-x) = \frac{\Gamma(N+1-x)}{\Gamma(1-x)},$$

and therefore

$$x(1-x)(2-x)\cdots(N-x) = x\frac{\Gamma(N+1-x)}{\Gamma(1-x)}.$$

Thus the exact expression becomes

$$1+\Lambda_N^{\mathrm{eq}} = \max_{x\in[0,1]} x\frac{\Gamma(N+1-x)}{\Gamma(1-x)}\sum_{k=0}^{N}\frac{1}{|k-x|k!(N-k)!}.$$

Up to this point everything is exact. To estimate Λ_N^{eq} for large N, we introduce several standard approximations.

We first use the ratio asymptotics for the Gamma function:

$$\frac{\Gamma(N+1-x)}{\Gamma(N+1)} \sim N^{-x} \quad (N\to\infty),$$

which gives

$$\Gamma(N+1-x) \sim N!N^{-x}.$$

Next observe that, as N grows, the binomial weights $1/(k!(N-k)!)$ are sharply peaked near $k = N/2$. In this region $|k-x| \approx N/2$ (the optimal x will turn out to be small), so we approximate

$$\frac{1}{|k-x|} \approx \frac{2}{N}$$

and factor this out of the sum:

$$\sum_{k=0}^{N}\frac{1}{|k-x|k!(N-k)!} \approx \frac{2}{N}\sum_{k=0}^{N}\frac{1}{k!(N-k)!}.$$

Using the binomial identity

$$\sum_{k=0}^{N}\frac{N!}{k!(N-k)!} = 2^N,$$

we obtain

$$\sum_{k=0}^{N}\frac{1}{k!(N-k)!} = \frac{1}{N!}\sum_{k=0}^{N}\binom{N}{k} = \frac{2^N}{N!},$$

and hence

$$1+\Lambda_N^{\mathrm{eq}} \approx \max_{x\in[0,1]} x\frac{N!N^{-x}}{\Gamma(1-x)}\cdot\frac{2}{N}\cdot\frac{2^N}{N!}.$$

After cancelling $N!$ we arrive at

$$1+\Lambda_N^{\mathrm{eq}} \approx \max_{x\in[0,1]} \frac{2^{N+1}}{N}\frac{xN^{-x}}{\Gamma(1-x)}.$$

To understand the dominant dependence on N we analyse the factor xN^{-x} for large N. Consider

$$\varphi(x) = xN^{-x} \quad \text{for} \quad x>0.$$

Then

$$\ln\varphi(x) = \ln x - x\ln N, \quad \frac{\mathrm{d}}{\mathrm{d}x}\ln\varphi(x) = \frac{1}{x} - \ln N.$$

The maximum occurs where this derivative vanishes:

$$\frac{1}{x_*} - \ln N = 0 \quad \Rightarrow \quad x_* = \frac{1}{\ln N}.$$

Evaluating φ at x_* gives

$$\varphi(x_*) = x_*N^{-x_*} = \frac{1}{\ln N}\exp(-x_*\ln N) = \frac{1}{\ln N}e^{-1} = \frac{1}{e\ln N}.$$

For large N this maximiser lies in $(0,1)$, so it is admissible. Moreover $\Gamma(1-x)\to 1$ as $x\to 0$, hence

$$\Gamma(1-x_*) \approx 1,$$

and the prefactor 1 in $1+\Lambda_N^{\mathrm{eq}}$ is negligible compared with the rapidly growing right-hand side. Thus

$$\Lambda_N^{\text{eq}} \approx \frac{2^{N+1}}{N} \cdot \frac{1}{e \ln N} = \frac{2^{N+1}}{eN \ln N} = O\left(\frac{2^N}{N \ln N}\right).$$

A slightly sharper estimate follows from the expansion

$$\Gamma(1-x) = 1 + \gamma x + O(x^2),$$

where γ is the Euler–Mascheroni constant. Substituting $x_* = 1/\ln N$ yields

$$\Gamma(1-x_*) = 1 + \frac{\gamma}{\ln N} + O\left(\frac{1}{(\ln N)^2}\right),$$

so at leading order in $1/\ln N$ we obtain

$$\Lambda_N^{\text{eq}} \approx \frac{2^{N+1}}{eN(\gamma + \ln N)}.$$

This recovers the classical asymptotic growth of the Lebesgue constant for equi-spaced interpolation nodes: it increases essentially like $2^N/(N \ln N)$ as $N \to \infty$.

4.5.6 Lebesgue Points and Multi-Dimensional Considerations

The nodes that *minimize* the Lebesgue constant $\Lambda_N \to \min$ for a given polynomial space $\mathbb{P}_N[\mathbb{R}]$ are called *Lebesgue points*. Finding these optimal nodes is a challenging optimization problem that has been solved only for moderate values of N in one dimension.

In multiple dimensions, the situation becomes considerably more complex. The Lebesgue constant depends not only on the node distribution $\{\boldsymbol{x}_j\}_{j=0}^N$ but also on the *shape of the domain* $\mathcal{U}$. The estimation of the Lebesgue constant in multi-dimensional non-Cartesian domains is a problem that remains essentially open today. The most precious information would be to find the node distribution over a triangle that minimizes the Lebesgue constant—knowledge that would be crucial for the design of new spectral elements [6].

On *quadrilateral* domains, the best known point distributions are tensor products of Gauss–Lobatto[1] or Chebyshev nodes. These tensor product constructions inherit the favorable properties of their one-dimensional counterparts.

On *triangular* domains, the situation is more subtle. The *Fekete points*[2] currently represent the best known choice. Fekete points are defined as the nodes that maximize the determinant of the Vandermonde matrix—equivalently, they maximize the volume of the interpolation simplex in function space. While not provably optimal in the Lebesgue sense, they exhibit excellent numerical properties and remain the standard choice for triangular spectral elements.

4.5.7 Visualization of Basis Functions

Figure 14 compares the Lagrange basis polynomials for equispaced and Chebyshev nodes. For equispaced nodes, the basis functions near the endpoints develop large oscillations. For Chebyshev nodes, all basis functions remain well-behaved.

[1] Rehuel Lobatto (1797–1866) was a Dutch mathematician born into a Portuguese family.

[2] Michael Fekete (1886–1957) was a Hungarian mathematician who completed his PhD under the supervision of Lipót Fejér. Fekete also provided private tutoring to János Neumann, known today as John von Neumann.

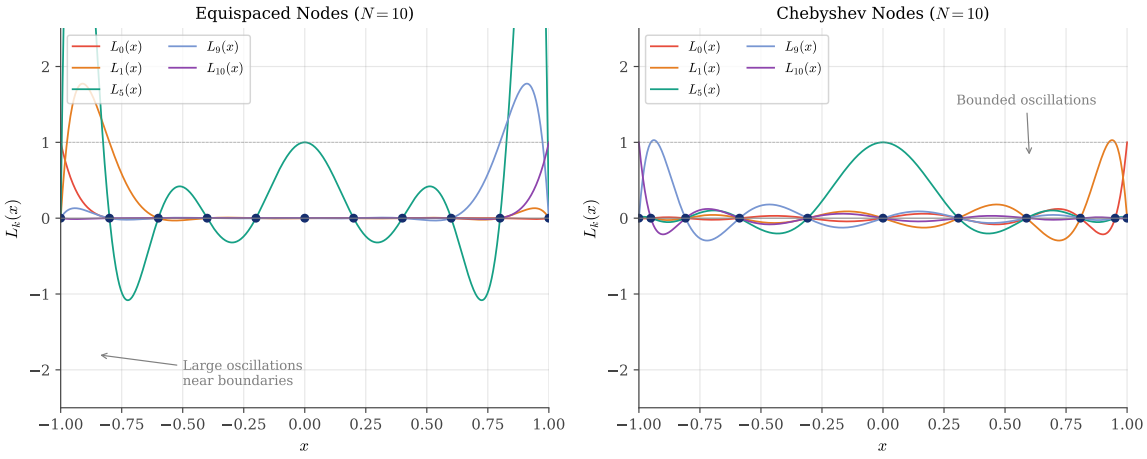

Figure 14: *Lagrange basis polynomials $L_{k(x)}$ for $N = 10$. Left: equispaced nodes show large oscillations near the boundaries. Right: Chebyshev nodes yield bounded basis functions throughout the interval.*

Figure 15 shows the Lebesgue functions and the growth of Lebesgue constants for the three node distributions.

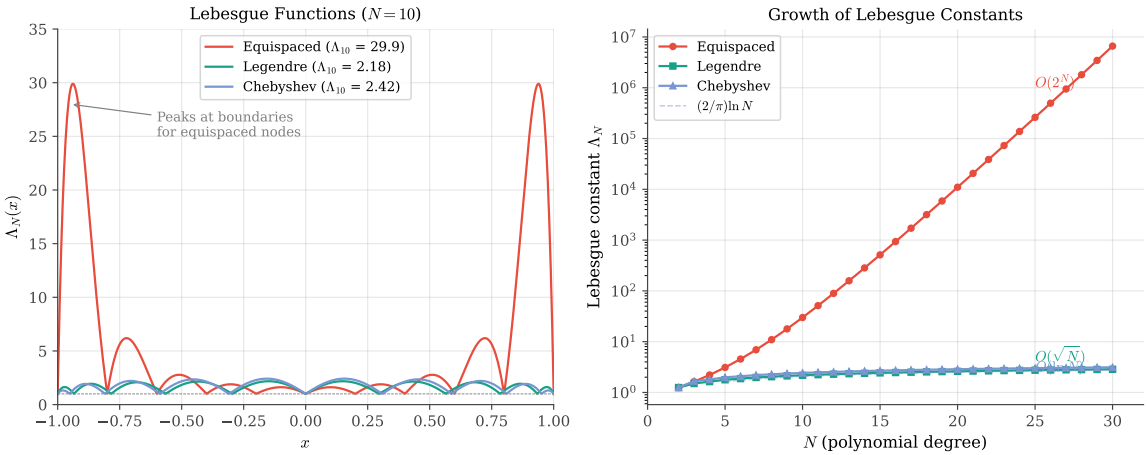

Figure 15: *Left: Lebesgue functions $\Lambda_N(x)$ for $N = 10$ with equispaced, Legendre, and Chebyshev nodes. The equispaced case has large peaks near the boundaries. Right: Growth of Lebesgue constants with N. The equispaced constant grows exponentially, while Chebyshev grows only logarithmically.*

The exponential growth of the equispaced Lebesgue constant dominates Figure 15, making it difficult to distinguish the Legendre and Chebyshev curves. Figure 16 provides a zoomed comparison of these two well-behaved distributions, clearly showing the $O\left(\sqrt{N}\right)$ growth of Legendre versus the superior $O(\ln N)$ growth of Chebyshev.

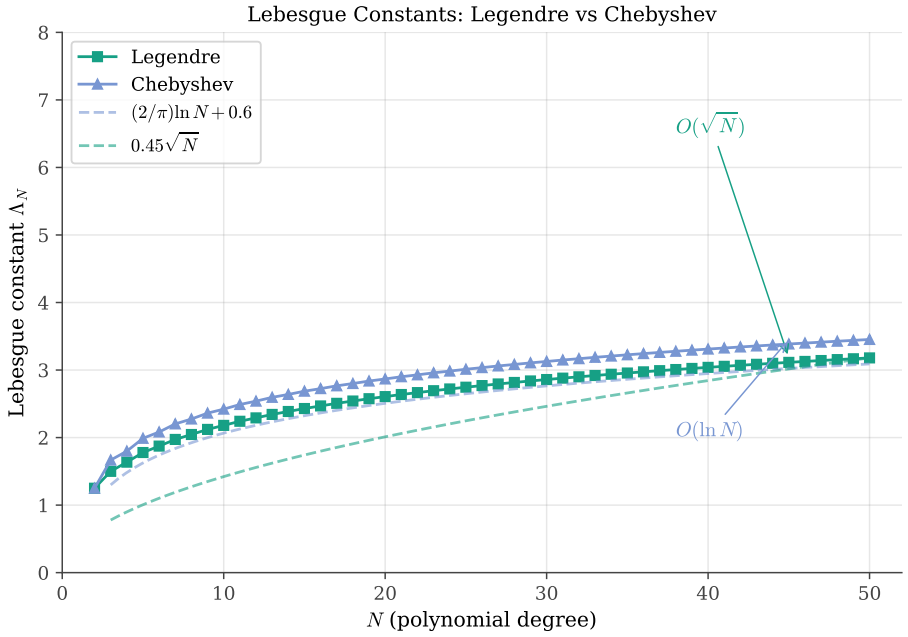

Figure 16: *Lebesgue constants for Legendre and Chebyshev nodes (equispaced omitted for clarity). Chebyshev nodes achieve the slowest possible growth rate $O(\ln N)$, while Legendre nodes grow as $O\left(\sqrt{N}\right)$. Both are far superior to equispaced nodes, but Chebyshev remains the optimal choice.*

The code generating these figures is available in:

- `codes/python/ch04_geometry_of_nodes/lagrange_basis.py`
- `codes/python/ch04_geometry_of_nodes/lebesgue_functions.py`
- `codes/python/ch04_geometry_of_nodes/lebesgue_constants_zoom.py`
- `codes/matlab/ch04_geometry_of_nodes/lagrange_basis.m`
- `codes/matlab/ch04_geometry_of_nodes/lebesgue_functions.m`

4.6 Barycentric Interpolation

4.6.1 Numerical Stability Issues

The Lagrange formula, while mathematically elegant, is numerically problematic. Computing each basis polynomial $L_k(x)$ requires N multiplications and divisions, and the resulting values can be very large or very small, leading to catastrophic cancellation when summed.

4.6.2 The Barycentric Formula

A more stable and efficient approach is the *barycentric interpolation formula*:

$$p_N(x) = \frac{\sum_{j=0}^{N} \frac{w_j}{x - x_j} f_j}{\sum_{j=0}^{N} \frac{w_j}{x - x_j}},$$

where the *barycentric weights* are:

$$w_j = \frac{1}{\prod_{k \neq j}(x_j - x_k)}.$$

This formula requires only $O(N)$ operations to evaluate $p_{N(x)}$ once the weights are precomputed.

4.6.3 Weights for Chebyshev Points

For Chebyshev-Gauss-Lobatto points, the weights have a particularly simple form:

$$w_j = (-1)^j \delta_j, \quad \text{where } \delta_j = \begin{cases} 1/2 \text{ if } j = 0 \text{ or } j = N \\ 1 \quad \text{otherwise.} \end{cases}$$

This simplicity, combined with the excellent approximation properties, makes Chebyshev points the standard choice for spectral methods.

4.7 Convergence Analysis

4.7.1 Geometric Convergence

For functions analytic in a region containing $[-1, 1]$, Chebyshev interpolation converges geometrically. If f is analytic inside the Bernstein ellipse $\mathcal{E}_\rho$ with parameter $\rho > 1$, then:

$$\|f - p_N\|_\infty \leqslant \frac{C}{\rho^N},$$

where C depends on f and ρ but not on N.

The parameter ρ is determined by the location of the nearest singularity: if the closest singularity to $[-1, 1]$ lies on the ellipse $\mathcal{E}_\rho$, then convergence is at rate $O(\rho^{-N})$.

4.7.2 The Runge Function Revisited

For the Runge function with poles at $z = \pm 0.2i$, we can compute:

$$\rho = |0.2i + \sqrt{(0.2i)^2 - 1}| = |0.2i + i\sqrt{1.04}| \approx 1.22.$$

Thus Chebyshev interpolation converges at rate $O(1.22^{-N})$; convergence is geometric but modest due to the poles being close to the real axis. In contrast, equispaced interpolation diverges because its effective $\rho < 1$.

The divergence of equispaced interpolation can be understood through the error bound $\|f - p_N\|_\infty \leqslant (1 + \Lambda_N) E_N(f)$. For the Runge function, the best approximation error $E_N(f)$ still decreases geometrically—the function is analytic, after all. However, recall from Section 4.5 that the equispaced Lebesgue constant grows as $\Lambda_N^{\text{eq}} \approx 2^{N+1}/(eN \ln N)$. This exponential growth eventually overwhelms the geometric decay of $E_N(f)$, causing the interpolation error $(1 + \Lambda_N)E_N(f)$ to grow without bound.

4.7.3 Numerical Verification

Figure 17 compares the convergence behavior for both node distributions. The Chebyshev error decreases geometrically, while the equispaced error grows without bound.

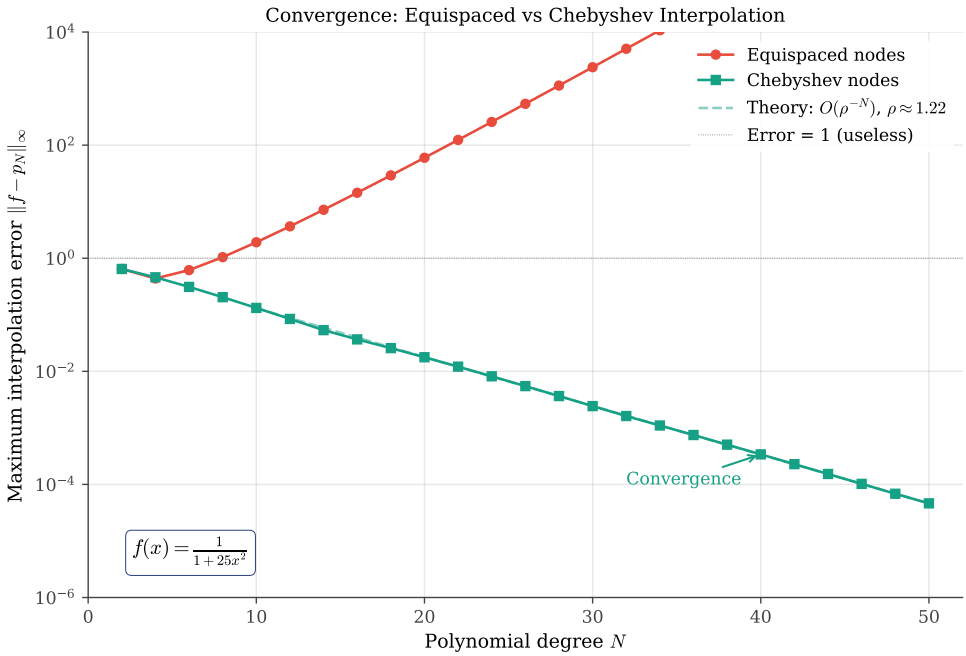

Figure 17: *Convergence comparison for the Runge function. The maximum interpolation error is plotted against polynomial degree N on a semilogarithmic scale. Chebyshev interpolation (green) converges at rate $O(\rho^{-N})$ with $\rho \approx 1.22$. Equispaced interpolation (red) diverges exponentially.*

Figure 18 shows the Chebyshev convergence alone, without the divergent equispaced curve that dominates the vertical scale. This reveals the beautiful geometric convergence: the error decreases by a factor of approximately $\rho \approx 1.22$ with each increase in polynomial degree. The theoretical rate matches the computed errors almost perfectly.

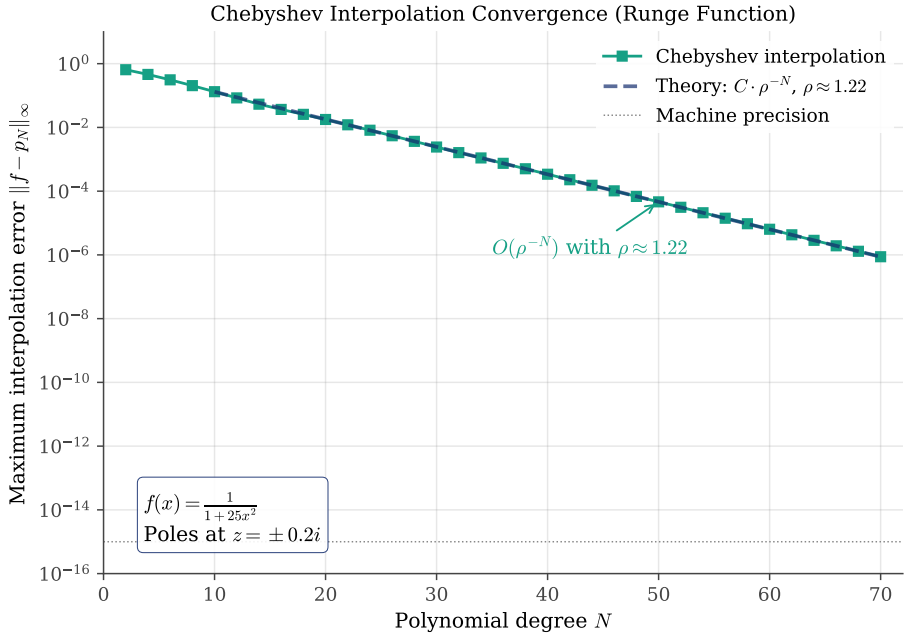

Figure 18: *Chebyshev interpolation convergence for the Runge function (equispaced omitted for clarity). The error decreases geometrically at rate $O(\rho^{-N})$ with $\rho \approx 1.22$, matching the theoretical prediction based on the pole locations at $z = \pm 0.2i$.*

The code generating these figures is available in:

- `codes/python/ch04_geometry_of_nodes/convergence_comparison.py`
- `codes/python/ch04_geometry_of_nodes/convergence_zoom.py`
- `codes/matlab/ch04_geometry_of_nodes/convergence_comparison.m`

4.8 Computational Experiment: Random Nodes

Having studied the optimal Chebyshev distribution and the problematic equispaced distribution, a natural question arises: what happens if we choose interpolation nodes *randomly*? This question leads us into the realm of *experimental mathematics*, where we use computation to discover and conjecture mathematical relationships.

4.8.1 Motivation

The dramatically different behaviors of equispaced and Chebyshev nodes might suggest that the key factor is *regularity* or *structure* in node placement. Perhaps any "reasonable" arrangement would work? To test this hypothesis, we investigate the simplest unstructured choice: nodes drawn uniformly at random from $[-1, 1]$.

This experiment serves two purposes. First, it tests whether the special clustering of Chebyshev points near the endpoints is truly essential, or whether it is merely one of many acceptable distributions. Second, it demonstrates the methodology of computational experimentation—using numerical evidence to formulate conjectures about asymptotic behavior.

4.8.2 Experimental Setup

For each polynomial degree N from 2 to 30, we generate $N+1$ random points uniformly distributed on $[-1, 1]$, sort them, and compute the Lebesgue constant. We repeat this process $M = 200$ times to obtain statistical estimates of the mean, standard deviation, and range of Λ_N for random nodes.

The Python implementation uses NumPy's random number generation:

```python
def random_nodes(N, rng=None):
    """Generate N+1 random nodes, sorted, on [-1, 1]."""
    if rng is None:
        rng = np.random.default_rng()
    return np.sort(rng.uniform(-1, 1, N + 1))

def monte_carlo_lebesgue(N, M=200):
    """Compute M samples of Lebesgue constant for random nodes."""
    samples = np.zeros(M)
    for m in range(M):
        x_rand = random_nodes(N)
        samples[m] = lebesgue_constant(x_rand)
    return samples
```

The equivalent MATLAB code:

```matlab
% Generate M samples for polynomial degree N
samples = zeros(M, 1);
for m = 1:M
    x_rand = sort(2 * rand(N+1, 1) - 1);  % Uniform on [-1, 1]
    samples(m) = max(lebesgue_function(x_rand, x_fine));
end
```

4.8.3 Results

Figure 19 shows the results of our Monte Carlo experiment. The left panel displays the growth of the Lebesgue constant with polynomial degree, including shaded regions showing the statistical variability. The right panel shows the distribution of Λ_N values for a fixed degree $N = 15$.

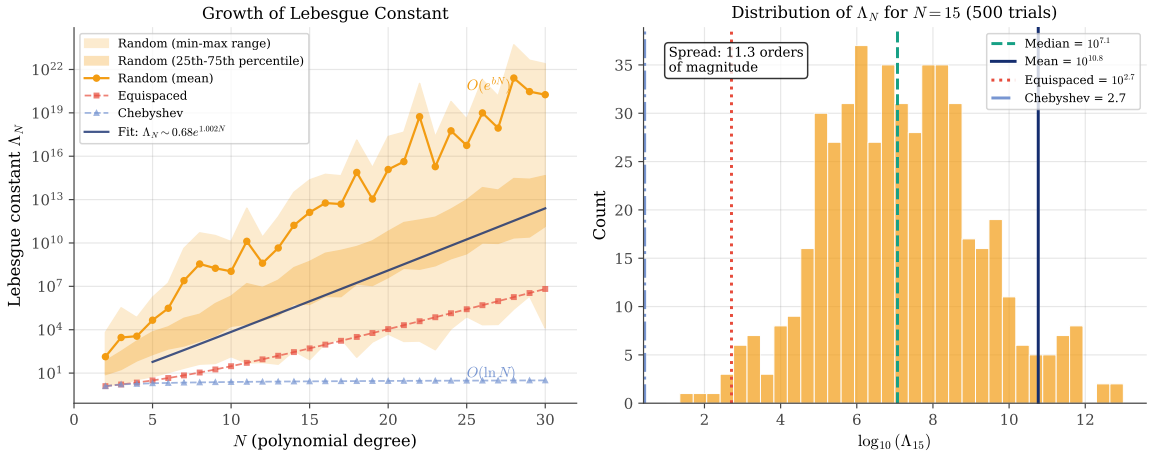

Figure 19: *Lebesgue constant for random nodes. Left: growth with polynomial degree N, showing mean (solid line) and min-max range (light shading). Right: distribution of $\log_{10}(\Lambda_{15})$ over 500 random trials on a logarithmic scale, revealing the enormous spread spanning many orders of magnitude. Vertical lines mark the mean, median, equispaced, and Chebyshev values.*

The key observations are striking:

1. **Explosive exponential growth**: Random nodes exhibit exponential growth of the Lebesgue constant, but surprisingly *faster* than equispaced nodes. The mean Λ_N grows roughly as e^{bN} with $b \approx 1.0$, compared to $b \approx 0.6$ for equispaced.
2. **Extreme variability**: There is enormous variability between different random samples. For $N = 15$, Λ_N can range from around 10^2 to $10^{\{10\}}$ or more, spanning many orders of magnitude. This variability increases dramatically with N.
3. **Worse than equispaced**: Counter to naive intuition, random nodes perform *worse* on average than the structured equispaced distribution. The reason is that random sampling occasionally produces clusters of nearby nodes, creating near-singular interpolation conditions. These worst-case realizations dominate the mean.

4.8.4 Empirical Asymptotic Formula

Fitting an exponential model to the mean Lebesgue constants yields an empirical formula:

$$\Lambda_N^{\text{rand}} \approx a \cdot e^{bN},$$

where the fitted constants are approximately $a \approx 0.7$ and $b \approx 1.0$, giving:

$$\Lambda_N^{\text{rand}} \approx 0.7 \cdot 2.7^N.$$

Remarkably, the growth rate for random nodes is *faster* than for equispaced nodes (which grow as $\approx 1.8^N$). This counterintuitive result arises because random sampling occasionally produces clusters of nodes that are extremely close together, causing the Lebesgue constant to explode. The mean is dominated by these worst-case realizations.

The fundamental problem is twofold: (1) the lack of *clustering near the endpoints*, which only carefully designed distributions like Chebyshev points possess, and (2) the risk of *random clustering* in the interior, which creates severely ill-conditioned interpolation problems.

4.8.5 Discussion

This computational experiment provides strong numerical evidence for a fundamental principle: **the clustering of Chebyshev points near the endpoints is not merely convenient but essential**. Random nodes, despite their apparent "fairness" in covering the interval, fail in two ways: they do not provide the boundary resolution needed to control Lagrange basis oscillations, and they risk interior clustering that creates catastrophically ill-conditioned systems.

The enormous variability in Λ_N for random nodes is perhaps the most striking finding. While equispaced nodes are suboptimal, they at least provide *predictable* (if exponentially growing) behavior. Random nodes introduce an additional layer of uncertainty—any given random realization might be acceptable or catastrophic.

The experiment illustrates the power of computational mathematics. By systematic numerical investigation, we have:

- Tested a natural hypothesis (random nodes might work)
- Discovered unexpected behavior (random is *worse* than equispaced)
- Quantified the asymptotic growth through data fitting
- Reinforced our understanding of why structured distributions matter

The code generating Figure 19 is available in:

- `codes/python/ch04_geometry_of_nodes/lebesgue_random_nodes.py`
- `codes/matlab/ch04_geometry_of_nodes/lebesgue_random_nodes.m`

4.9 Practical Guidelines and Outlook

4.9.1 When to Use Which Grid

Based on the theory developed in this chapter, we can state clear guidelines:

1. **Always prefer Chebyshev points** for polynomial interpolation on a finite interval. The $O(\ln N)$ growth of the Lebesgue constant ensures stability.
2. **Avoid equispaced nodes** for high-degree polynomial interpolation. The exponential growth of Λ_N makes this a losing proposition.
3. **Consider the singularity structure** of the function being approximated. The location of complex singularities determines the convergence rate.

4.9.2 Mapping to General Intervals

Chebyshev points are defined on $[-1, 1]$, but can be mapped to any interval $[a, b]$ via the linear transformation:

$$x_{\text{phys}} = \frac{a+b}{2} + \frac{b-a}{2} x_{\text{ref}},$$

where $x_{\text{ref}} \in [-1, 1]$ is the reference coordinate.

4.9.3 Preview of Differentiation Matrices

The Chebyshev points introduced in this chapter will play a central role in the differentiation matrices we develop in subsequent chapters. The clustering of nodes near the boundaries, far from being a peculiarity, is precisely what enables accurate spectral differentiation. The connection between node distribution, interpolation accuracy, and differentiation stability is one of the beautiful unifying themes of spectral methods.

In the next chapter, we will see how to convert function values at Chebyshev points into accurate approximations of derivatives, building on the geometric insights developed here.

CHAPTER 5

Differentiation Matrices

In the previous chapter, we mastered the art of polynomial interpolation—constructing polynomials that pass exactly through a set of data points. We discovered that the choice of nodes determines whether interpolation succeeds or fails, with Chebyshev points emerging as the optimal choice for non-periodic problems. Now we take the next logical step: having represented a function as an interpolating polynomial, how do we *differentiate* it? The answer leads us to one of the most elegant structures in numerical analysis: the differentiation matrix.

The remarkable insight of pseudospectral methods is that differentiation can be accomplished by a single matrix-vector multiplication. Given function values $\boldsymbol{u} = (u_0, u_1, ..., u_N)^\top$ at the grid points, we can approximate the derivative values $\boldsymbol{u}' = (u_0', u_1', ..., u_N')^\top$ as

$$\boldsymbol{u}' \approx D\boldsymbol{u}, \tag{1}$$

where D is the *differentiation matrix*. This matrix encapsulates the entire differentiation process: interpolate, differentiate, evaluate.

This chapter develops the theory and practice of constructing differentiation matrices. We begin with familiar finite difference approximations, viewing them through the lens of matrix algebra. We then take the crucial limiting step: what happens when the stencil extends to include *all* grid points? The answer reveals that spectral methods are not a separate species from finite differences, but rather their natural culmination—the limiting case as the stencil width grows without bound.

5.1 From Interpolation to Differentiation

5.1.1 The Matrix Perspective

Let us begin with a fundamental observation. Given $N+1$ distinct nodes $\{x_0, x_1, ..., x_N\}$ and function values $\{u_0, u_1, ..., u_N\}$, the unique interpolating polynomial of degree at most N can be written using the Lagrange formula:

$$p_N(x) = \sum_{j=0}^{N} u_j L_j(x), \tag{2}$$

where $L_j(x)$ are the Lagrange basis polynomials from Section 4.5.

To approximate the derivative of u at the nodes, we simply differentiate the interpolating polynomial:

$$p_N'(x) = \sum_{j=0}^{N} u_j L_j'(x). \tag{3}$$

Evaluating this at node x_i yields:

$$p_N'(x_i) = \sum_{j=0}^{N} u_j L_j'(x_i). \tag{4}$$

This is a linear combination of the function values! The coefficients $L_j'(x_i)$ depend only on the node locations, not on the function values. We can therefore write Equation 4 in matrix form:

$$\boldsymbol{u}' \approx D\boldsymbol{u}, \quad \text{where} \quad D_{ij} = L_j'(x_i). \tag{5}$$

The entry D_{ij} represents the weight given to the function value at x_j when approximating the derivative at x_i.

5.1.2 The Barycentric Formula for Differentiation Matrix Entries

For general (non-equispaced) nodes, the differentiation matrix entries can be computed using the barycentric weights w_j introduced in Section 4.5:

$$D_{ij} = \frac{w_j/w_i}{x_i - x_j} \quad \text{for} \quad i \neq j. \tag{6}$$

The diagonal entries are determined by the important *consistency condition*: the derivative of a constant function is zero. Since D applied to the constant vector $(1, 1, ..., 1)^\top$ must yield zero, we have

$$\sum_{j=0}^{N} D_{ij} = 0 \quad \Rightarrow \quad D_{ii} = -\sum_{j \neq i} D_{ij}. \tag{7}$$

This row-sum property provides a convenient way to compute the diagonal entries and serves as a useful sanity check.

5.2 Finite Difference Matrices

5.2.1 The Local Approach

Before tackling the full spectral differentiation matrix, let us review the familiar territory of finite differences. The classical approach approximates the derivative using only *nearby* function values—a local stencil.

The simplest example is the *second-order central difference*:

$$u'(x_i) \approx \frac{u_{i+1} - u_{i-1}}{2h}, \tag{8}$$

where h is the grid spacing. This formula is "second-order accurate" because the error is $O(h^2)$ for smooth functions, as can be verified by Taylor expansion.

For higher accuracy, we can include more neighbors. The *fourth-order central difference* uses five points:

$$u'(x_i) \approx \frac{-u_{i+2} + 8u_{i+1} - 8u_{i-1} + u_{i-2}}{12h}. \tag{9}$$

The *sixth-order central difference* extends to seven points:

$$u'(x_i) \approx \frac{u_{i+3} - 9u_{i+2} + 45u_{i+1} - 45u_{i-1} + 9u_{i-2} - u_{i-3}}{60h}. \tag{10}$$

Figure 20 illustrates these stencils schematically. Each formula uses only the function values at nodes within its stencil. The derivative at x_i depends only on nearby neighbors, not on distant points.

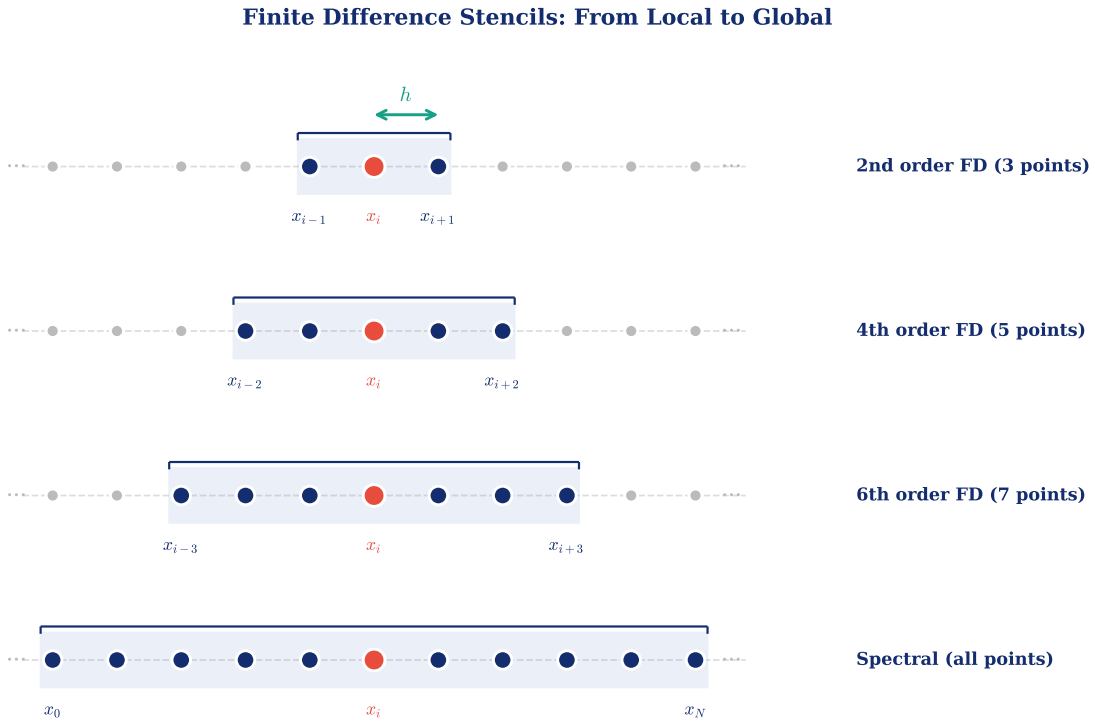

Figure 20: *Finite difference stencils in one dimension, progressing from local to global. The derivative at the central node x_i (red) is approximated using only the highlighted nodes (blue) within each stencil. From top to bottom: 3-point stencil (2nd order), 5-point stencil (4th order), 7-point stencil (6th order), and spectral method (all nodes). As the stencil widens, accuracy improves. The spectral method represents the limiting case where* every *node contributes to the derivative approximation, yielding exponential rather than algebraic convergence.*

5.2.2 Matrix View of Finite Differences

These formulas can be assembled into differentiation matrices. For the second-order scheme Equation 8, assuming periodic boundary conditions on N equispaced points with spacing $h = 2\pi/N$, the matrix is *tridiagonal* (plus corner entries for periodicity):

$$D^{(2)} = \frac{1}{2h}\begin{pmatrix} 0 & 1 & 0 & \cdots & 0 & -1 \\ -1 & 0 & 1 & \cdots & 0 & 0 \\ 0 & -1 & 0 & \cdots & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 1 & 0 & 0 & \cdots & -1 & 0 \end{pmatrix}. \tag{11}$$

The fourth-order scheme Equation 9 produces a *pentadiagonal* matrix with bandwidth 5, and the sixth-order scheme yields a *heptadiagonal* matrix with bandwidth 7.

Figure 21 illustrates this progression. As the order of accuracy increases, the stencil widens and the matrix bandwidth grows.

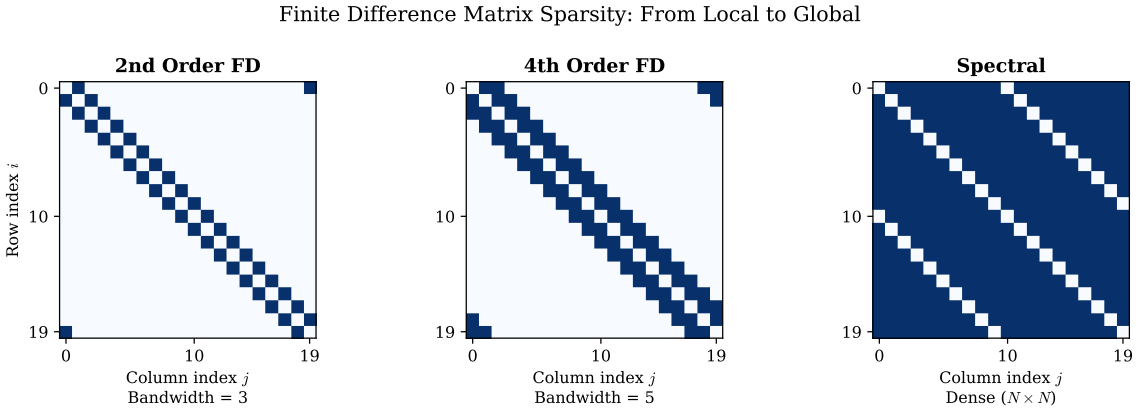

Figure 21: *Sparsity patterns of differentiation matrices for $N = 20$ grid points. Left: second-order finite differences (tridiagonal, bandwidth 3). Center: fourth-order finite differences (pentadiagonal, bandwidth 5). Right: spectral method (dense, bandwidth N). The progression illustrates the key insight: spectral methods are the limiting case of finite differences as the stencil width extends to the full domain.*

5.2.3 The Limit Question

This observation leads to a fundamental question: *if widening the stencil improves accuracy, what is the ultimate limit?*

The answer is a stencil that uses *all* N grid points. The resulting matrix is *dense*—every point influences the derivative at every other point. This is precisely the spectral differentiation matrix.

From this perspective, spectral methods are not a separate species from finite differences. They are the natural endpoint of the progression: as we increase the order of accuracy by including more neighbors, we eventually include all points. The sparse banded matrix becomes dense, and algebraic convergence transforms into spectral (exponential) convergence.

This unifying viewpoint, emphasized by Fornberg [1], provides both conceptual clarity and practical guidance. It suggests that finite differences and spectral methods lie on a continuum, with the choice of stencil width being a tunable parameter balancing accuracy against computational cost.

5.3 The Periodic Spectral Differentiation Matrix

5.3.1 Periodic Problems and Equispaced Nodes

A remarkable simplification occurs for *periodic* problems. On a domain like $[0, 2\pi)$ with periodic boundary conditions, the natural interpolation basis consists of trigonometric polynomials (sines and cosines) rather than algebraic polynomials.

For periodic problems, the optimal node distribution is *equispaced*:

$$x_j = \frac{2\pi j}{N}, \quad j = 0, 1, ..., N-1. \tag{12}$$

This is in stark contrast to the non-periodic case studied in Section 4.4, where equispaced nodes lead to the Runge phenomenon. The difference lies in the boundary: periodic functions have no

endpoints, so there is no need for the clustering that Chebyshev nodes provide. The periodicity itself regularizes the problem.

5.3.2 Derivation via the Fourier Basis

The spectral differentiation matrix for periodic problems can be derived by considering the trigonometric interpolant and differentiating it analytically. For N equispaced points (with N even), the interpolating trigonometric polynomial is

$$p(x) = \sum_{j=0}^{N-1} u_j \varphi_j(x), \tag{13}$$

where $\varphi_j(x)$ is the *periodic cardinal function* (or discrete Dirichlet kernel) satisfying $\varphi_j(x_k) = \delta_{jk}$.

The periodic cardinal function can be written as a sum of complex exponentials:

$$\varphi_j(x) = \frac{1}{N} \sum_{k=-N/2+1}^{N/2} e^{ik(x-x_j)}. \tag{14}$$

This sum can be evaluated in closed form. Writing $\theta = \big(x - x_j\big)/2$ and using the geometric series, we obtain:

$$\varphi_j(x) = \frac{\sin(N\theta)}{N\sin(\theta)} = \frac{\sin\big(N\big(x-x_j\big)/2\big)}{N\sin\big(\big(x-x_j\big)/2\big)}. \tag{15}$$

Figure 22 visualizes these periodic cardinal functions for $N = 16$ equispaced nodes. Each function peaks at value 1 at its corresponding node x_j and vanishes at all other nodes, satisfying the cardinal property $\varphi_j(x_k) = \delta_{jk}$. Unlike Lagrange basis polynomials on non-periodic domains, which can exhibit unbounded oscillations (recall Section 4.5), these periodic cardinal functions remain bounded. The damped oscillations away from the peak reflect the $\sin(x)/x$-like structure of the discrete Dirichlet kernel.

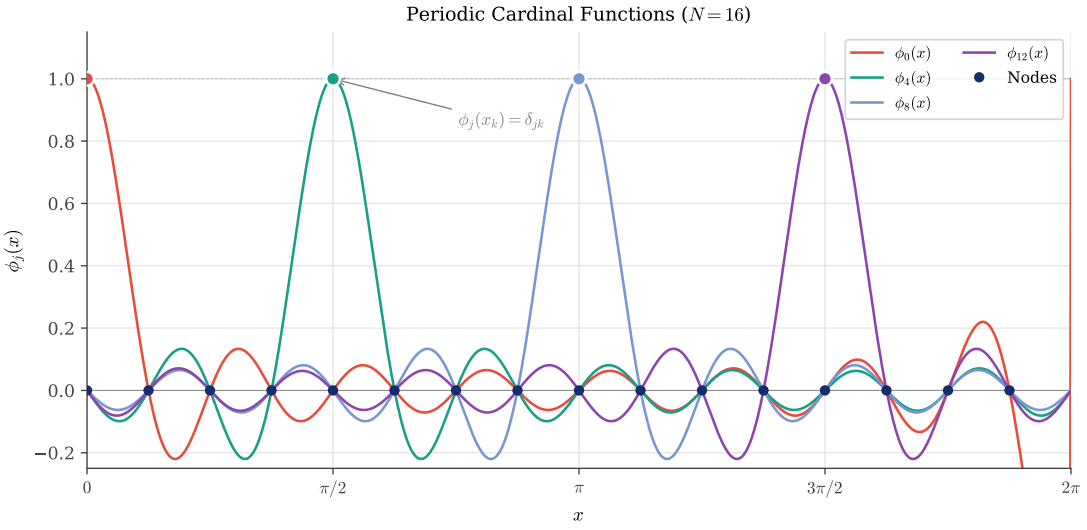

Figure 22: *Periodic cardinal functions $\varphi_j(x)$ for $N = 16$ equispaced nodes on $[0, 2\pi)$. Each function peaks at value 1 at its corresponding node x_j and vanishes at all other nodes, satisfying the cardinal property $\varphi_j(x_k) = \delta_{jk}$. The oscillations decay smoothly away from each peak, in contrast to the boundary-amplified oscillations seen in Lagrange basis polynomials for non-periodic interpolation.*

The following Python code computes the periodic cardinal function Equation 15:

```python
import numpy as np

def periodic_cardinal(x, x_j, N):
    """
    Compute the periodic cardinal function phi_j(x) centered at x_j.

    Parameters:
        x   : array - Points at which to evaluate
        x_j : float - Center point (node location)
        N   : int - Number of grid points

    Returns:
        phi : array - Values of phi_j at x
    """
    theta = (x - x_j) / 2.0
    phi = np.zeros_like(x, dtype=float)

    # Handle singularity at theta = 0 using L'Hopital's rule
    small = np.abs(np.sin(theta)) < 1e-14
    phi[~small] = np.sin(N * theta[~small]) / (N * np.sin(theta[~small]))
    phi[small] = 1.0  # Limit as theta -> 0

    return phi
```

The equivalent MATLAB implementation:

```matlab
function phi = periodic_cardinal(x, x_j, N)
    % Compute the periodic cardinal function phi_j(x) centered at x_j.
    % Input:
    %   x   - points at which to evaluate
    %   x_j - center point (node location)
    %   N   - number of grid points
    % Output:
    %   phi - values of phi_j at x

    theta = (x - x_j) / 2.0;
    phi = zeros(size(x));

    % Handle singularity at theta = 0
    small = abs(sin(theta)) < 1e-14;
    phi(~small) = sin(N * theta(~small)) ./ (N * sin(theta(~small)));
    phi(small) = 1.0;  % Limit as theta -> 0
end
```

The code implementing these functions and generating Figure 22 is available in:

- `codes/python/ch05_differentiation_matrices/periodic_cardinal_functions.py`
- `codes/matlab/ch05_differentiation_matrices/periodic_cardinal_functions.m`

To find the differentiation matrix, we need to compute $\varphi_j'(x_m)$ for all m. Let $\xi = (x - x_j)/2$ for brevity. Then Equation 15 becomes $\varphi_j = \sin(N\xi)/(N\sin\xi)$. Applying the quotient rule:

$$\frac{d\varphi_j}{d\xi} = \frac{N\cos(N\xi)\sin\xi - \sin(N\xi)\cos\xi}{N\sin^2\xi}. \tag{16}$$

Since $d\xi/dx = 1/2$, we obtain:

$$\varphi_j'(x) = \frac{1}{2}\cdot\frac{\cos(N\xi)}{\sin\xi} - \frac{1}{2}\cdot\frac{\sin(N\xi)\cos\xi}{N\sin^2\xi}. \tag{17}$$

We evaluate this at the grid points $x = x_m$.

Case 1: Diagonal entries $(m = j)$.

When $x = x_j$, we have $\xi = 0$, so both numerator and denominator of Equation 17 vanish. Applying L'Hôpital's rule (twice) yields $\varphi_j'(x_j) = 0$. Geometrically, the cardinal function is symmetric about its peak at x_j, so its derivative must vanish there.

Case 2: Off-diagonal entries $(m \neq j)$.

When $x = x_m$ with $m \neq j$, we have $\xi = \pi(m-j)/N$. At these points:

- $\sin(N\xi) = \sin(\pi(m-j)) = 0$ (the second term in the numerator vanishes),
- $\cos(N\xi) = \cos(\pi(m-j)) = (-1)^{m-j}$.

Substituting into Equation 17:

$$\varphi_j'(x_m) = \frac{1}{2}\frac{(-1)^{m-j}}{\tan(\pi(m-j)/N)} = \frac{(-1)^{m-j}}{2}\cot\frac{(m-j)\pi}{N}. \tag{18}$$

Since $D_{mk} = \varphi_k'(x_m)$, the spectral differentiation matrix entries are:

$$D_{jk} = \begin{cases} \dfrac{1}{2}(-1)^{j-k}\cot\left(\dfrac{(j-k)\pi}{N}\right) & \text{if } j \neq k \\ 0 & \text{if } j = k. \end{cases} \tag{19}$$

5.3.3 Properties of the Spectral Matrix

The matrix defined by Equation 19 has several remarkable properties:

1. **Skew-symmetry**: $D^\top = -D$. This reflects the fact that differentiation is an antisymmetric operator in appropriate function spaces.
2. **Zero diagonal**: $D_{jj} = 0$. The derivative approximation at any point depends only on neighboring values, not on the value at the point itself.
3. **Toeplitz structure**: D_{jk} depends only on the difference $j - k$. This means the matrix has constant diagonals—a consequence of the translation-invariance of differentiation on periodic domains.
4. **Circulant**: Due to the periodic boundary conditions, the matrix is actually *circulant*: each row is a cyclic shift of the previous row. Circulant matrices can be diagonalized by the discrete Fourier transform, enabling $O(N\log N)$ matrix-vector products via the FFT.
5. **Dense**: Unlike finite difference matrices, every off-diagonal entry is nonzero. This is the price we pay for spectral accuracy.
6. **Exactness for trigonometric polynomials**: If $u(x)$ is a trigonometric polynomial of degree at most $N/2$, then $D\boldsymbol{u}$ gives the *exact* derivative values.

Figure 23 visualizes these properties for $N = 16$.

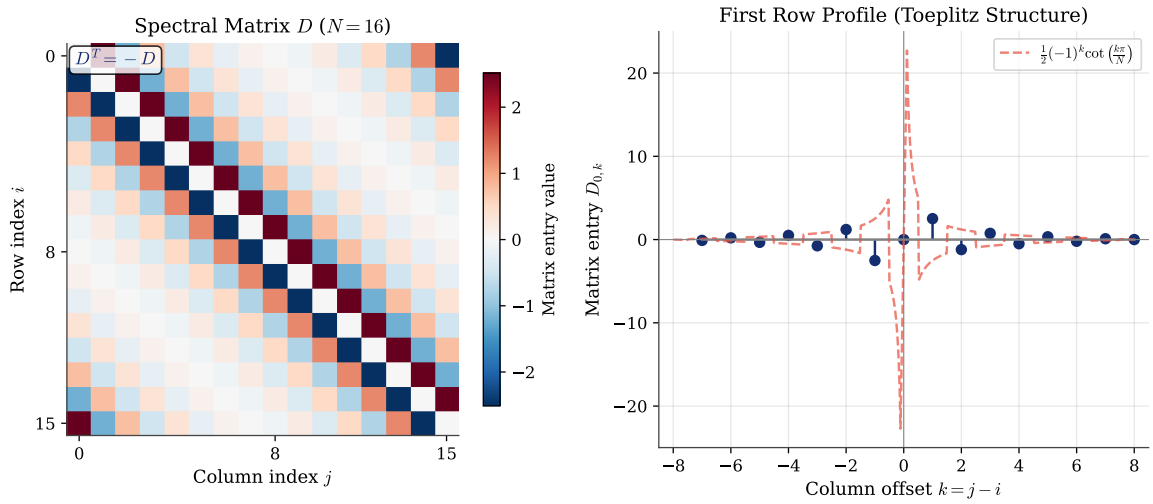

Figure 23: *Structure of the periodic spectral differentiation matrix for $N = 16$.* Left*: heatmap showing the matrix entries. The Toeplitz (constant diagonal) structure is evident, as is the skew-symmetry ($D^\top = -D$) with positive values (red) appearing where negative values (blue) appear in the transpose.* Right*: the first row entries $D_{0,k}$ (blue dots) plotted against the column offset $k = j - i$. The dashed red curve shows the continuous cotangent function $\frac{1}{2} \cdot (-1)^k \cot(k\pi/N)$ from Equation 19, confirming that the discrete matrix entries lie exactly on this curve. The antisymmetric pattern ($D_{0,k} = -D_{0,-k}$) reflects the skew-symmetry of D.*

5.3.4 Python Implementation

The following Python code constructs the periodic spectral differentiation matrix:

```python
import numpy as np

def spectral_diff_periodic(N):
    """
    Construct the periodic spectral differentiation matrix.

    Parameters:
        N : int - Number of grid points (should be even)

    Returns:
        D : ndarray (N, N) - Differentiation matrix
        x : ndarray (N,) - Grid points on [0, 2π)
    """
    h = 2 * np.pi / N
    x = h * np.arange(N)
    D = np.zeros((N, N))

    for i in range(N):
        for j in range(N):
            if i != j:
                D[i, j] = 0.5 * ((-1) ** (i - j)) / np.tan((i - j) * np.pi / N)
```

```

    return D, x
```

The equivalent MATLAB implementation:

```matlab
function [D, x] = spectral_diff_periodic(N)
    h = 2 * pi / N;
    x = h * (0:N-1)';
    D = zeros(N, N);

    for i = 1:N
        for j = 1:N
            if i ~= j
                D(i, j) = 0.5 * ((-1)^(i-j)) / tan((i-j) * pi / N);
            end
        end
    end
end
```

The code implementing these algorithms is available in:

- `codes/python/ch05_differentiation_matrices/spectral_matrix_periodic.py`
- `codes/matlab/ch05_differentiation_matrices/spectral_matrix_periodic.m`

5.3.5 A Practical Demonstration

Let us put our spectral differentiation matrix to work. Consider the smooth periodic function

$$u(x) = e^{\sin^2 x}, \tag{20}$$

which is *not* a trigonometric polynomial—its Fourier series has infinitely many terms. The derivatives are:

$$\begin{aligned} u'(x) &= \sin(2x) e^{\sin^2 x}, \\ u''(x) &= \left[\sin^2(2x) + 2\cos(2x)\right] e^{\sin^2 x}. \end{aligned} \tag{21}$$

With $N = 64$ grid points, the spectral method computes both derivatives to near machine precision! Figure 24 shows the results: the numerical values (markers) lie exactly on the exact curves (solid lines), with maximum errors around 10^{-14}.

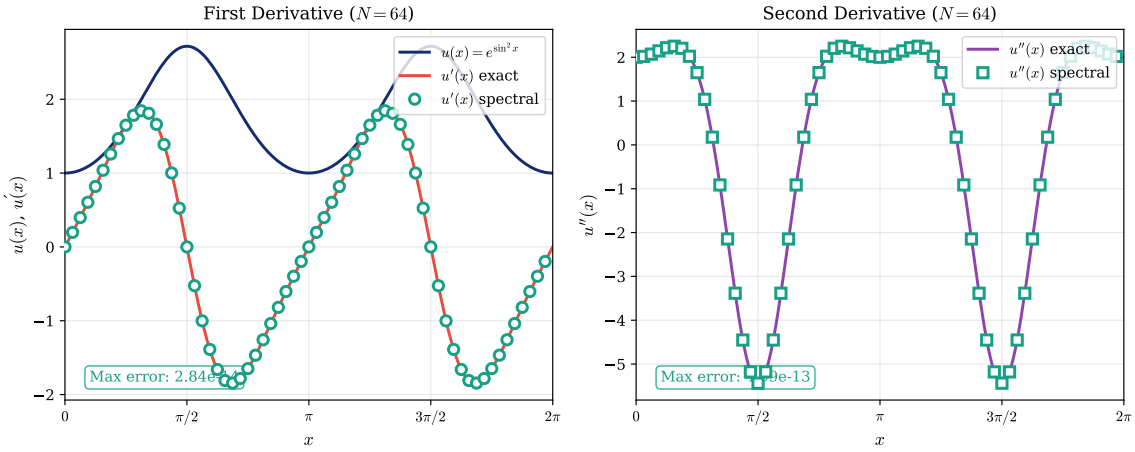

Figure 24: *Spectral differentiation of $u(x) = e^{\sin^2 x}$ using $N = 64$ grid points. Left: the function u (navy) and its first derivative u' exact (coral) vs. spectral (teal circles). Right: second derivative u'' exact (purple) vs. spectral (teal squares). The maximum errors, displayed in each panel, are near machine precision ($\approx 10^{-14}$). This remarkable accuracy with relatively few points is the hallmark of spectral methods for smooth periodic functions.*

Table 3 quantifies the convergence rate. The errors decrease dramatically—by roughly four orders of magnitude each time N doubles. This is the signature of *spectral convergence*: for analytic functions, the error decreases faster than any polynomial in $1/N$.

N	Error in u'	Error in u''
8	6.96×10^{-2}	2.00×10^{0}
16	4.33×10^{-4}	3.67×10^{-2}
32	1.12×10^{-9}	3.26×10^{-7}
64	2.84×10^{-14}	4.99×10^{-13}

Table 3: *Convergence of spectral differentiation for $u(x) = e^{\sin^2 x}$. The maximum errors $\|u' - D\boldsymbol{u}\|_\infty$ and $\|u'' - D^2\boldsymbol{u}\|_\infty$ decrease exponentially as N increases, reaching machine precision by $N = 64$.*

The code for this demonstration is available in:

- `codes/python/ch05_differentiation_matrices/spectral_derivatives_demo.py`
- `codes/matlab/ch05_differentiation_matrices/spectral_derivatives_demo.m`

5.4 Fornberg's Recursive Algorithm

5.4.1 Motivation: Robustness for Arbitrary Grids

The explicit formula Equation 19 is elegant but "brittle"—it applies only to equispaced periodic grids. What if we need differentiation weights for non-equispaced nodes? Or for non-periodic problems? Or for higher-order derivatives?

The direct approach would be to solve the Vandermonde system that arises from polynomial interpolation. However, Vandermonde matrices are notoriously ill-conditioned, especially for large N. We need a more robust algorithm.

In 1988, Bengt Fornberg published a remarkable recursive algorithm [7] that computes finite difference weights for *any* node distribution and *any* derivative order. The algorithm is:

- Numerically stable even for large stencils
- Efficient: $O(MN^2)$ for all derivatives up to order M on N nodes
- Elegant: the recursion has a beautiful mathematical structure

5.4.2 The Recursive Algorithm

The key insight is that we can build up the weights incrementally. Let $\delta_m^{(k,n)}$ denote the weight for node x_k when approximating the m-th derivative using nodes $x_0, x_1, ..., x_n$ (a stencil of $n+1$ points), evaluated at some point ξ.

The algorithm proceeds level by level:

- **Level 0** ($n = 0$): A single node. For interpolation ($m = 0$), the weight is simply 1.
- **Level 1** ($n = 1$): Two nodes. Weights are computed from level 0.
- **Level n**: Weights for $n+1$ nodes are computed from level $n-1$.

The recursion formulas are:

For $k < n$ (weights for nodes already in the previous stencil):

$$\delta_m^{(k,n)} = \frac{(\xi - x_n)\delta_m^{(k,n-1)} - m\delta_{m-1}^{(k,n-1)}}{x_n - x_k}. \tag{22}$$

For $k = n$ (weight for the newly added node):

$$\delta_m^{(n,n)} = \frac{c_n}{c_{n-1}}\left(m\delta_{m-1}^{(n-1,n-1)} - (\xi - x_{n-1})\delta_m^{(n-1,n-1)}\right), \tag{23}$$

where $c_n = \prod_{\ell=0}^{n-1}(x_n - x_\ell)$.

The base case is $\delta_0^{(0,0)} = 1$, and we define $\delta_m^{(k,n)} = 0$ when $m < 0$ or $m > n$.

5.4.3 The Stencil Pyramid

Figure 25 visualizes the recursive structure. Each level of the pyramid contains weights for one stencil size. Arrows show the data dependencies: weights at level n depend only on weights from level $n-1$.

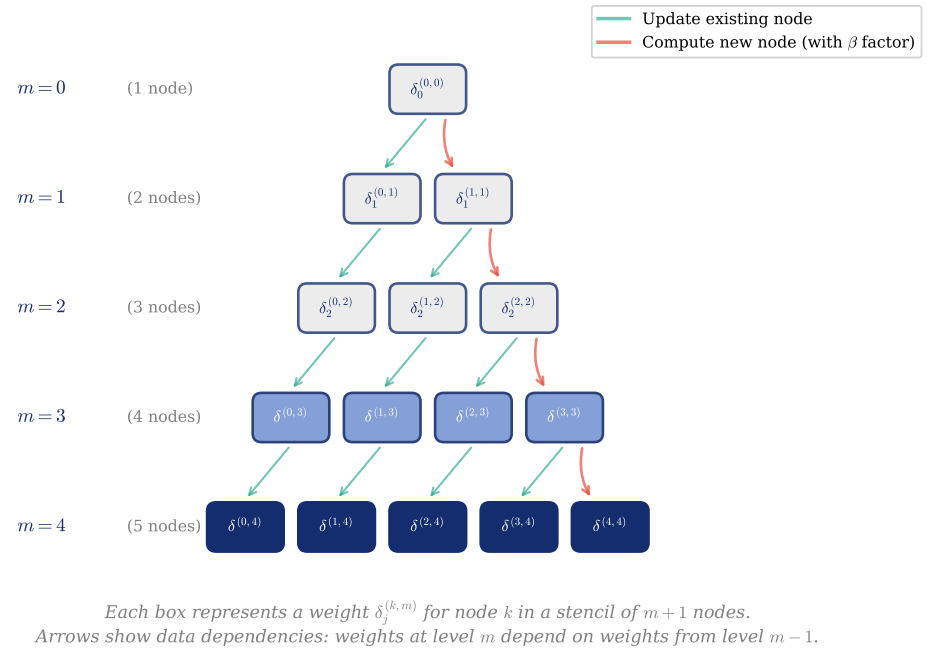

Figure 25: *Fornberg's recursive algorithm visualized as a "stencil pyramid." Each box represents a weight $\delta_m^{(k,n)}$ for node index k in a stencil of $n+1$ nodes. The algorithm builds from top (single node) to bottom (full stencil). Green arrows indicate updates to existing node weights via Equation 22; red arrows show computation of new node weights via Equation 23. The recursive structure ensures numerical stability.*

The pyramid structure explains why the algorithm is numerically stable. Each weight is computed as a simple combination of weights from the previous level, avoiding the accumulation of rounding errors that plagues direct methods.

5.4.4 Python Implementation

The following code implements Fornberg's algorithm, translated from the MATLAB version by Toby Driscoll [8]:

```python
import numpy as np

def fdweights(xi, x, m):
    """
    Compute finite difference weights using Fornberg's algorithm.

    Parameters:
        xi : float - Point where derivative is approximated
        x  : array - Node positions
        m  : int - Derivative order (0=interpolation, 1=first, etc.)

    Returns:
        w  : array - Weights for the m-th derivative approximation
```

```
    """
    x = np.asarray(x, dtype=float)
    n = len(x) - 1
    w = np.zeros(n + 1)
    x_shifted = x - xi  # Translate evaluation point to origin

    for k in range(n + 1):
        w[k] = _weight(x_shifted, m, n, k)

    return w

def _weight(x, m, j, k):
    """Recursive weight computation (evaluation point at 0)."""
    if m < 0 or m > j:
        return 0.0
    elif m == 0 and j == 0:
        return 1.0
    else:
        if k < j:
            c = (x[j] * _weight(x, m, j-1, k)
                 - m * _weight(x, m-1, j-1, k)) / (x[j] - x[k])
        else:
            beta = np.prod(x[j-1] - x[:j-1]) / np.prod(x[j] - x[:j])
            c = beta * (m * _weight(x, m-1, j-1, j-1)
                        - x[j-1] * _weight(x, m, j-1, j-1))
        return c
```

The equivalent MATLAB implementation, due to Toby Driscoll [8]:

Matlab

```
function w = fdweights(xi, x, m)
    % Compute finite difference weights using Fornberg's algorithm.
    % Input:
    %   xi  - evaluation point for the derivative
    %   x   - node positions (vector)
    %   m   - derivative order (0=interpolation, 1=first, etc.)
    % Output:
    %   w   - weights for the m-th derivative approximation

    p = length(x) - 1;
    w = zeros(size(x));
    x = x - xi;  % Translate evaluation point to origin

    for k = 0:p
        w(k+1) = weight(x, m, p, k);
    end
end

```

```
function c = weight(x, m, j, k)
    % Recursive weight computation (evaluation point at 0).
    if (m < 0) || (m > j)
        c = 0;
    elseif (m == 0) && (j == 0)
        c = 1;
    else
        if k < j
            c = (x(j+1) * weight(x, m, j-1, k) ...
                 - m * weight(x, m-1, j-1, k)) / (x(j+1) - x(k+1));
        else
            beta = prod(x(j) - x(1:j-1)) / prod(x(j+1) - x(1:j));
            c = beta * (m * weight(x, m-1, j-1, j-1) ...
                        - x(j) * weight(x, m, j-1, j-1));
        end
    end
end
```

This algorithm is the universal tool for computing differentiation matrix entries on any grid.

The code implementing this algorithm is available in:

- `codes/python/ch05_differentiation_matrices/fdweights.py`
- `codes/matlab/ch05_differentiation_matrices/fdweights.m`

5.5 Computational Étude: The Rational Trigonometric Test

We now conduct a computational experiment that reveals the dramatic difference between finite difference and spectral accuracy. The goal is not merely to verify theoretical convergence rates, but to develop intuition for *why* spectral methods achieve such remarkable precision.

5.5.1 Choosing a Test Function

The choice of test function is crucial. We need a function that is smooth and periodic (so that both finite difference and spectral methods apply), yet not so simple that it masks the differences between methods. A pure trigonometric function like $\sin(kx)$ would be inappropriate: since $\sin(kx)$ is an eigenfunction of both the continuous and discrete differentiation operators, spectral methods would give exact results trivially, revealing nothing about their convergence behavior.

Instead, we choose the *rational trigonometric function*:

$$u(x) = \frac{1}{2 + \sin(x)} \quad \text{on} \quad [0, 2\pi). \tag{24}$$

This function satisfies our requirements admirably. It is infinitely differentiable (analytic) on the entire real line, and clearly periodic with period 2π. Most importantly, it has an *infinite* Fourier series—unlike $\sin(x)$ or finite trigonometric polynomials, its spectral content extends to all frequencies, ensuring there are no "lucky cancellations" in our numerical differentiation.

The exact derivative, computed by the quotient rule, is:

$$u'(x) = -\frac{\cos(x)}{(2+\sin(x))^2}. \tag{25}$$

There is a deeper reason for choosing this particular function. Although $u(x)$ is smooth on the real line, it has singularities in the *complex plane*. The denominator $2+\sin(x)$ vanishes when $\sin(x) = -2$, which has no real solutions but does have complex solutions at $x = -\pi/2 \pm i \cdot \operatorname{arcsinh}(2)$. The distance from the real axis to these nearest singularities is $d = \operatorname{arcsinh}(2) \approx 1.44$. As we shall see, this distance controls the convergence rate of spectral methods—a beautiful connection to potential theory that we explored in Section 4.3.

5.5.2 The Experiment

Our experimental design is straightforward. For a sequence of grid sizes $N = 4, 6, 8, 10, ..., 64$, we construct four differentiation matrices: three finite difference matrices of orders 2, 4, and 6 (using Fornberg's algorithm from the previous section), and the periodic spectral differentiation matrix given by Equation 19.

For each matrix D, we sample the test function at the N equispaced grid points to form the vector $\boldsymbol{u} = (u(x_0), u(x_1), ..., u(x_{N-1}))^\top$, compute the numerical derivative $D\boldsymbol{u}$, and compare it to the exact derivative values $\boldsymbol{u}'_{\text{exact}} = (u'(x_0), u'(x_1), ..., u'(x_{N-1}))^\top$. The error is measured in the maximum norm:

$$\varepsilon_N = \|\boldsymbol{u}'_{\text{exact}} - D\boldsymbol{u}\|_\infty = \max_{0 \leqslant j \leqslant N-1} |u'(x_j) - (D\boldsymbol{u})_j|. \tag{26}$$

5.5.3 Results and Interpretation

Figure 26 displays the results on a semi-logarithmic plot. The visual contrast between finite difference and spectral methods is striking and reveals a fundamental distinction in their convergence behavior.

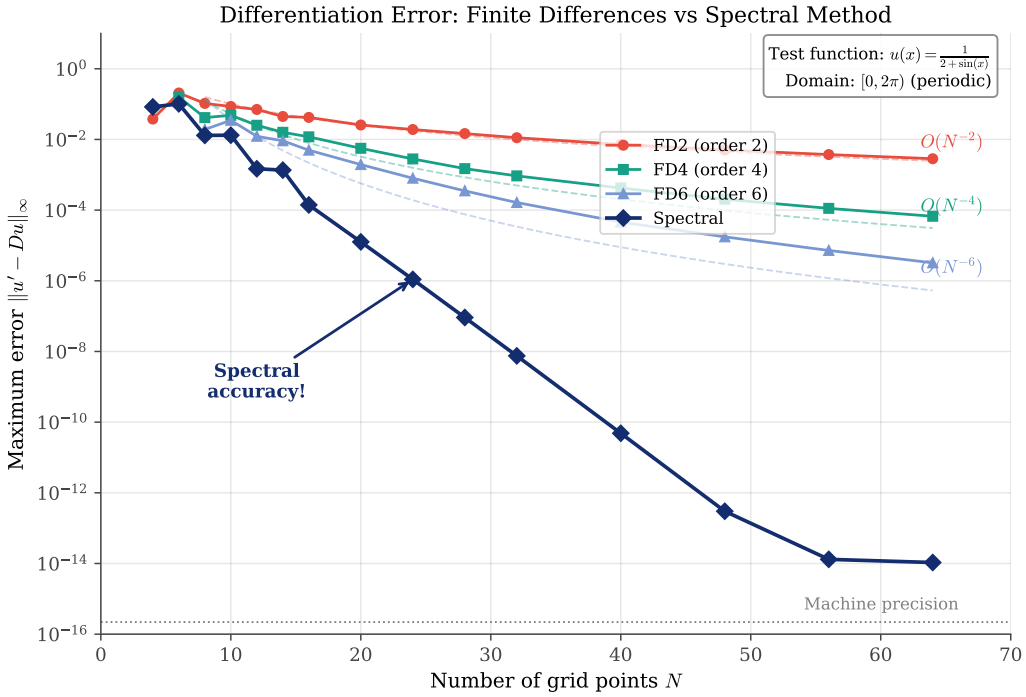

Figure 26: *Differentiation error comparison: finite differences versus spectral method. The test function is $u(x) = 1/(2 + \sin(x))$ on the periodic domain $[0, 2\pi)$. Finite difference methods (FD2, FD4, FD6) exhibit algebraic convergence with the expected rates $O(N^{-2})$, $O(N^{-4})$, $O(N^{-6})$. The spectral method achieves geometric (exponential) convergence, reaching near machine precision around $N = 50$. Dashed lines show theoretical convergence rates.*

The finite difference methods exhibit *algebraic* (polynomial) convergence. On a log-log plot, their error curves would appear as straight lines; on our semi-log plot, they curve downward with decreasing slope. The second-order method (FD2) shows error decreasing as $O(N^{-2})$, which is equivalent to $O(h^2)$ since $h = 2\pi/N$. The fourth-order method (FD4) improves to $O(N^{-4})$, and the sixth-order method (FD6) achieves $O(N^{-6})$. These rates match the theoretical predictions from Taylor series analysis: a p-th order finite difference scheme has truncation error $O(h^p)$.

The spectral method, in contrast, exhibits *geometric* (exponential) convergence. Its error curve appears as a straight line on the semi-log plot, indicating that $\log \varepsilon_N$ decreases linearly with N. Mathematically, $\varepsilon_N = O(c^{-N})$ for some constant $c > 1$. The method reaches near machine precision ($\approx 10^{-14}$) at around $N = 50$ with roughly 50 grid points, we have computed the derivative to nearly the full precision available in double-precision arithmetic!

To appreciate what this means in practice, consider trying to achieve 14-digit accuracy with finite differences. For the second-order method, we would need to solve $CN^{-2} \approx 10^{-14}$. Even with a modest constant $C \approx 1$, this requires $N \approx 10^7 \approx 10$ million grid points. For a problem in three dimensions, this would mean 10^{21} unknowns, which is utterly impractical. The spectral method achieves the same accuracy with only 50 points.

5.5.4 Discussion: Why Does Spectral Win?

The difference between algebraic and spectral convergence is fundamental:

Algebraic convergence ($O(N^{-p})$): Doubling N reduces the error by a factor of 2^p. This is good, but the improvement is polynomial.

Spectral convergence ($O(c^{-N})$): Doubling N *squares* the error (roughly). This exponential improvement is why spectral methods can achieve machine precision with modest grid sizes.

The source of spectral convergence is the *analyticity* of the function being approximated. For the test function Equation 24, the nearest singularities in the complex plane are at distance approximately $\operatorname{arcsinh}(2) \approx 1.44$ from the real axis. Potential theory (cf. Section 4.3) tells us that the convergence rate is controlled by this distance: the interpolation error decreases like ρ^{-N} where $\rho = e^d$ and d is the distance to the nearest singularity.

The code generating Figure 26 is available in:

- `codes/python/ch05_differentiation_matrices/convergence_comparison.py`
- `codes/matlab/ch05_differentiation_matrices/convergence_comparison.m`

5.6 Higher-Order Derivatives

5.6.1 Second Derivatives: Squaring vs. Direct Construction

For second derivatives, we have two options:

1. **Matrix squaring**: $D^{(2)} = D \cdot D$, where D is the first-derivative matrix
2. **Direct construction**: Build $D^{(2)}$ directly using second-derivative weights

Matrix squaring is simpler and often sufficient. The resulting matrix $(D \cdot D)_{ij}$ represents the combined effect of differentiating twice. For smooth functions, this gives accurate second derivatives.

For the periodic spectral case, the second-derivative matrix has a closed form:

$$D^{(2)}_{jk} = \begin{cases} -\dfrac{1}{2}\dfrac{(-1)^{j-k}}{\sin^2((j-k)\pi/N)} & \text{if } j \neq k \\ -\dfrac{\pi^2(N^2+2)}{3(2\pi)^2} & \text{if } j = k. \end{cases} \tag{27}$$

However, matrix squaring D^2 is often accurate enough and more convenient.

5.6.2 A Demonstration: Higher-Order Derivatives

Let us put higher-order spectral differentiation to the test. Consider the smooth periodic function

$$u(x) = e^{-\sin(2x)}, \tag{28}$$

which has higher frequency content than our earlier examples. Its derivatives become increasingly complex:

$$\begin{aligned} u'(x) &= -2\cos(2x)e^{-\sin(2x)}, \\ u''(x) &= 4\big[\sin(2x) + \cos^2(2x)\big]e^{-\sin(2x)}. \end{aligned} \tag{29}$$

Figure 27 shows spectral approximations of the first four derivatives using $N = 32$ grid points. The numerical values (markers) align precisely with the exact curves (solid lines), with errors displayed in each panel. Notice how the error grows with derivative order—a fundamental limitation of numerical differentiation.

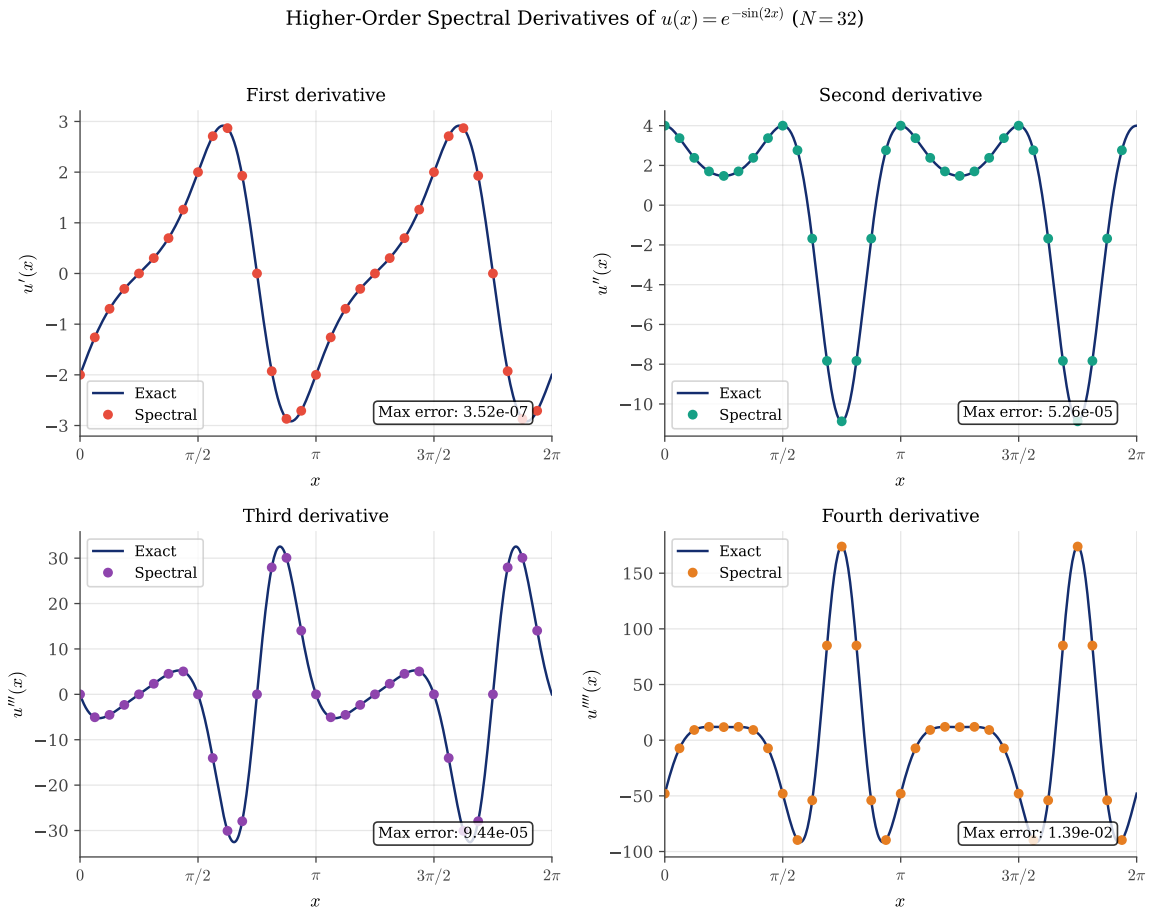

Figure 27: *Higher-order spectral derivatives of $u(x) = e^{-\sin(2x)}$ using $N = 32$ grid points. Each panel compares the exact derivative (navy curve) with the spectral approximation (colored markers). The maximum errors, displayed in each panel, grow with derivative order—from $\approx 10^{-7}$ for the first derivative to $\approx 10^{-2}$ for the fourth. This error amplification is intrinsic to numerical differentiation.*

Table 4 quantifies the convergence behavior. For each derivative order, the spectral method achieves exponential convergence as N increases. However, the errors at any fixed N grow significantly with derivative order, reflecting the ill-conditioning of higher-order differentiation.

N	Error u'	Error u''	Error u'''	Error u''''
8	3.50×10^{-1}	6.17×10^{0}	9.40×10^{0}	1.55×10^{2}
16	8.64×10^{-3}	3.92×10^{-1}	6.51×10^{-1}	2.82×10^{1}
32	3.52×10^{-7}	5.26×10^{-5}	9.44×10^{-5}	1.39×10^{-2}
64	2.42×10^{-14}	1.18×10^{-12}	1.45×10^{-11}	6.14×10^{-10}

Table 4: *Convergence of spectral differentiation for $u(x) = e^{-\sin(2x)}$ at different derivative orders. Each column shows the maximum error $\left\|u^{(m)} - D^m \boldsymbol{u}\right\|_\infty$ for the m-th derivative. While spectral convergence is achieved for all orders, higher derivatives require more grid points to reach a given accuracy.*

Figure 28 illustrates the matrix squaring approach for second derivatives. The left panel shows the structure of $D^2 = D \cdot D$, which inherits the Toeplitz property from D. The center panel confirms that the eigenvalues of D^2 are $-k^2$ for $k = -N/2+1, ..., N/2$, matching the eigenvalues of the continuous operator d^2/dx^2 acting on Fourier modes e^{ikx}. The right panel demonstrates the accuracy of the second derivative approximation.

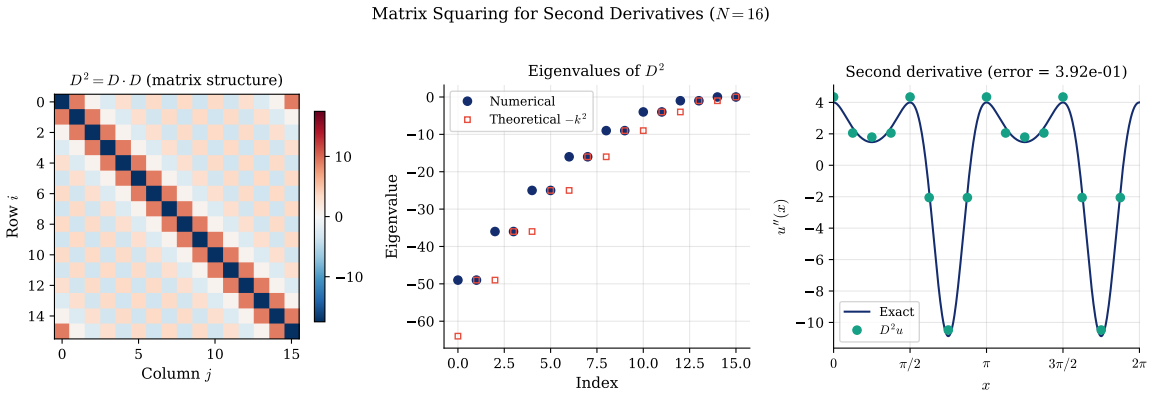

Figure 28: *Matrix squaring for second derivatives with $N = 16$. Left: structure of $D^2 = D \cdot D$, showing the Toeplitz pattern inherited from D. Center: eigenvalues of D^2 (circles) compared to the theoretical values $-k^2$ (squares), confirming spectral accuracy. Right: second derivative of $u(x) = e^{-\sin(2x)}$ computed via $D^2\boldsymbol{u}$, showing excellent agreement with the exact solution.*

The following Python code computes higher-order derivatives via matrix powers:

Python

```python
import numpy as np

def higher_order_derivative(D, u, order):
    """
    Compute the m-th derivative using matrix powers.

    Parameters:
        D     : ndarray (N, N) - First-derivative matrix
        u     : ndarray (N,) - Function values at grid points
        order : int - Derivative order (1, 2, 3, ...)

    Returns:
        u_m   : ndarray (N,) - m-th derivative values
    """
    D_m = np.linalg.matrix_power(D, order)
    return D_m @ u
```

The equivalent MATLAB implementation:

Matlab

```matlab
function u_m = higher_order_derivative(D, u, order)
    % Compute the m-th derivative using matrix powers.
    % Input:
    %   D     - first-derivative matrix (N x N)
    %   u     - function values at grid points (N x 1)
    %   order - derivative order (1, 2, 3, ...)
```

```
    % Output:
    %   u_m   - m-th derivative values

    D_m = D^order;
    u_m = D_m * u;
end
```

The code generating Figure 27 and Figure 28 is available in:

- `codes/python/ch05_differentiation_matrices/higher_order_derivatives.py`
- `codes/matlab/ch05_differentiation_matrices/higher_order_derivatives.m`

5.6.3 Fornberg's Algorithm for Higher Derivatives

A key advantage of Fornberg's algorithm is that it handles any derivative order m with no additional complexity. The same recursive structure computes interpolation weights ($m = 0$), first-derivative weights ($m = 1$), second-derivative weights ($m = 2$), and so on.

This generality is valuable when solving PDEs that involve mixed derivatives or high-order terms.

5.7 Looking Ahead: The Non-Periodic Case

The periodic spectral matrix Equation 19 relies crucially on the periodicity of the problem. For *non-periodic* problems on finite intervals, such as boundary value problems with Dirichlet or Neumann conditions, we need a different approach.

The solution, which we develop in the next chapter, is to use *Chebyshev differentiation matrices*. These combine the Chebyshev nodes from Section 4.4 with differentiation matrix techniques similar to those presented here. The resulting matrices are dense but not Toeplitz, reflecting the non-uniform node distribution.

The key formulas are similar in spirit to Equation 19 but more complex due to the clustering of Chebyshev nodes near the boundaries. The Chebyshev differentiation matrix will become our primary tool for spectral solutions of differential equations on bounded domains.

5.8 Summary

This chapter has established differentiation matrices as the computational heart of pseudospectral methods:

1. **Matrix representation**: Differentiation of interpolating polynomials can be expressed as matrix-vector multiplication: $\boldsymbol{u}' \approx D\boldsymbol{u}$.
2. **Finite differences as sparse approximations**: Classical FD methods correspond to sparse (banded) differentiation matrices. Higher-order FD methods use wider stencils and denser matrices.

3. **Spectral methods as the limit**: When the stencil extends to all grid points, we obtain the dense spectral differentiation matrix. This limiting viewpoint unifies finite difference and spectral approaches.
4. **Periodic spectral matrix**: For periodic problems on equispaced grids, the matrix entries are given by the cotangent formula Equation 19.
5. **Fornberg's algorithm**: A stable, efficient recursive algorithm computes differentiation weights for arbitrary node distributions and derivative orders.
6. **Spectral accuracy**: For smooth functions, spectral methods achieve exponentially fast convergence, vastly outperforming any fixed-order finite difference scheme.

The differentiation matrix is our passport to spectral solutions of differential equations. In the chapters ahead, we will use these matrices to solve boundary value problems, initial value problems, and eventually the partial differential equations that motivated our journey.

CHAPTER 6

Chebyshev Differentiation Matrices

Having developed the theory of differentiation matrices for periodic problems in the previous chapter, we now face a new challenge: what happens when the domain is *bounded*? Many problems in science and engineering—heat conduction, wave propagation, fluid dynamics—are posed on finite intervals with boundary conditions at the endpoints. For such problems, the elegant trigonometric framework of Fourier methods must give way to something new.

The key insight of this chapter is that polynomial interpolation, when done correctly, provides the foundation for spectral methods on bounded domains. The "correct" approach, as we discovered in Section 4, requires carefully chosen interpolation nodes. Equispaced points lead to the Runge phenomenon; Chebyshev points do not. This chapter builds on that foundation to construct the *Chebyshev differentiation matrix*—the non-periodic analog of the Fourier differentiation matrix.

6.1 The Non-Periodic Challenge

6.1.1 From Periodic to Bounded Domains

In the previous chapter, we exploited the periodicity of the domain $[0, 2\pi)$ to construct differentiation matrices using equispaced nodes and trigonometric interpolation. The resulting matrices had beautiful structure: circulant, with entries determined by a simple cotangent formula.

For problems on a bounded interval like $[-1, 1]$, we face several new difficulties:

1. **No periodicity**: The function values at the endpoints are independent, not related by periodicity.
2. **The Runge phenomenon**: Equispaced nodes lead to wild oscillations near the boundaries, as we demonstrated dramatically in Section 4.2.
3. **Boundary conditions**: Physical problems typically impose conditions at $x = \pm 1$, which must be incorporated into the differentiation process.

The solution to these challenges comes from our study of interpolation theory: the Chebyshev-Gauss-Lobatto points

$$x_j = \cos(j\pi/N), \quad j = 0, 1, ..., N, \tag{30}$$

cluster near the boundaries, counteracting the Runge phenomenon and enabling spectral accuracy.

6.1.2 Grid Comparison: Equispaced vs. Chebyshev

Figure 29 illustrates the fundamental difference between equispaced and Chebyshev grids. The equispaced grid distributes points uniformly across $[-1, 1]$, while the Chebyshev grid clusters points near the boundaries according to the projection from a circle.

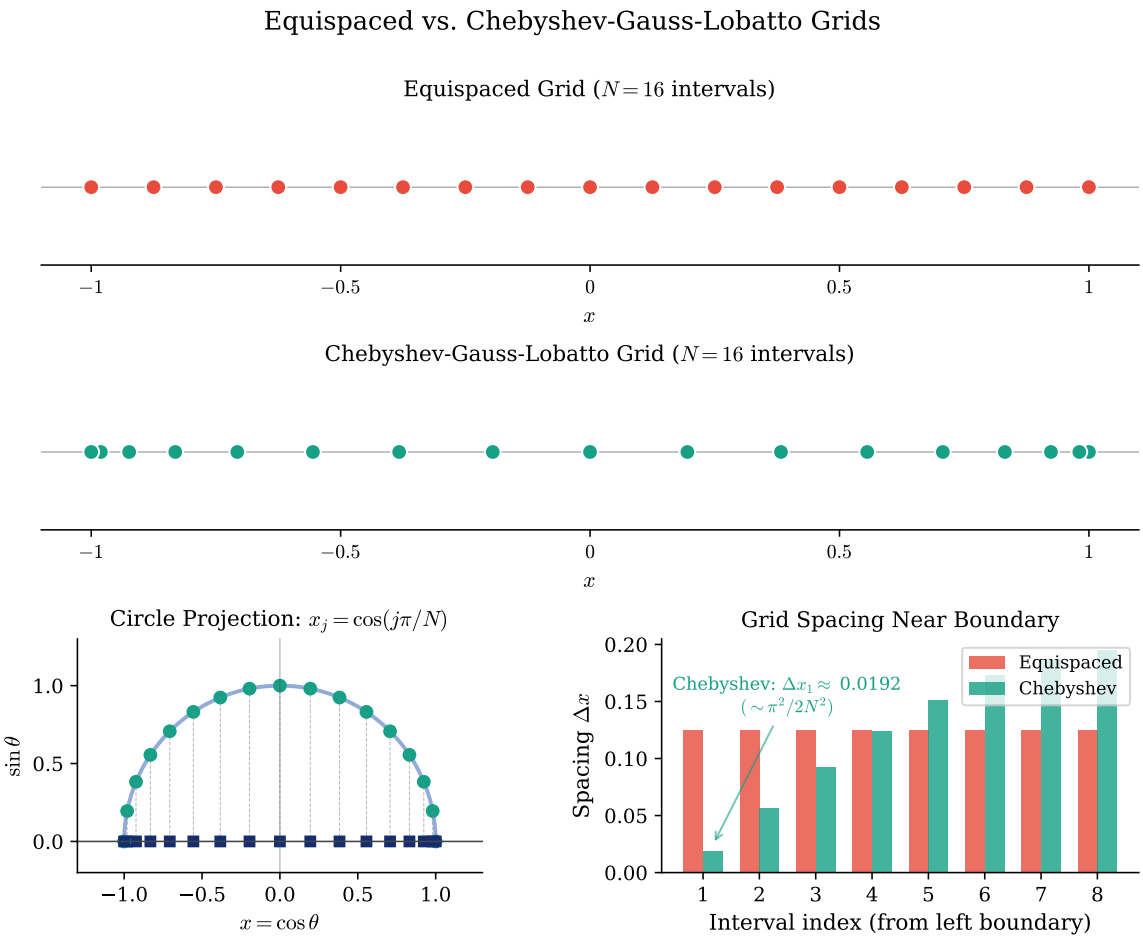

Figure 29: *Comparison of equispaced and Chebyshev-Gauss-Lobatto grids for $N = 16$ intervals. Top: equispaced points are distributed uniformly. Middle: Chebyshev points cluster near the boundaries. Bottom left: the circle projection interpretation—Chebyshev points are the projections of equally-spaced points on a semicircle. Bottom right: comparison of grid spacing near the left boundary, showing the $O(N^{-2})$ clustering of Chebyshev points.*

The boundary clustering is not a minor detail—it is essential for stability. Near the boundaries, where polynomial interpolation tends to oscillate wildly, the Chebyshev grid provides many closely-spaced points to control the behavior. Near the center, where interpolation is naturally well-behaved, fewer points suffice.

The spacing of Chebyshev points near the boundary is $O(N^{-2})$, compared to $O(N^{-1})$ for equispaced points. This denser boundary clustering has important implications for time-stepping in differential equations, as we shall see in later chapters.

6.2 The Chebyshev Differentiation Matrix

6.2.1 Differentiation via Interpolation

The construction of the Chebyshev differentiation matrix follows the same principle as in the periodic case: interpolate, then differentiate.

Given function values $\boldsymbol{v} = (v_0, v_1, ..., v_N)^\top$ at the Chebyshev points Equation 30, we first construct the unique interpolating polynomial $p(x)$ of degree at most N satisfying $p(x_j) = v_j$. The derivative approximation at the grid points is then

$$\boldsymbol{w} = D_N \boldsymbol{v}, \quad \text{where} \quad w_i = p'(x_i). \tag{31}$$

The matrix D_N is the *Chebyshev differentiation matrix*. Its entries can be computed from the Lagrange basis polynomials:

$$(D_N)_{ij} = L_j'(x_i), \tag{32}$$

where L_j is the Lagrange interpolating polynomial centered at x_j.

6.2.2 Explicit Formulas

The entries of the Chebyshev differentiation matrix can be written explicitly. Define the weights

$$c_j = \begin{cases} 2 \text{ if } j = 0 \text{ or } j = N \\ 1 \text{ otherwise.} \end{cases} \tag{33}$$

Then the matrix entries are:

Off-diagonal entries ($i \neq j$):

$$(D_N)_{ij} = \frac{c_i}{c_j} \frac{(-1)^{i+j}}{x_i - x_j}. \tag{34}$$

Diagonal entries (interior nodes, $j = 1, ..., N-1$):

$$(D_N)_{jj} = -\frac{x_j}{2\left(1 - x_j^2\right)}. \tag{35}$$

Corner entries:

$$(D_N)_{00} = \frac{2N^2 + 1}{6}, \quad (D_N)_{NN} = -\frac{2N^2 + 1}{6}. \tag{36}$$

6.2.3 The Negative Sum Trick

While the formulas Equation 35 and Equation 36 give exact expressions for the diagonal entries, direct evaluation can be numerically unstable. A more robust approach uses the *negative sum trick*: since the derivative of a constant function must be zero, each row of D_N must sum to zero:

$$(D_N)_{jj} = -\sum_{k \neq j} (D_N)_{jk}. \tag{37}$$

This ensures that $D_N \mathbf{1} = \mathbf{0}$ to machine precision, where $\mathbf{1}$ is the vector of all ones.

The following code implements the Chebyshev differentiation matrix:

```python
def cheb_matrix(N):
    """Chebyshev differentiation matrix."""
    if N == 0:
        return np.array([[0.0]]), np.array([1.0])

    # Chebyshev-Gauss-Lobatto points
    x = np.cos(np.pi * np.arange(N + 1) / N)

    # Weights: c_0 = c_N = 2, others = 1
    c = np.ones(N + 1)
    c[0], c[N] = 2.0, 2.0

```

```
    # Off-diagonal entries
    X = np.tile(x, (N + 1, 1))
    dX = X - X.T
    C = np.outer(c, 1.0 / c)
    sign = np.outer((-1.0)**np.arange(N + 1), (-1.0)**np.arange(N + 1))

    with np.errstate(divide='ignore'):
        D = C * sign / (-dX)

    # Negative sum trick for diagonal
    np.fill_diagonal(D, 0.0)
    D[np.diag_indices(N + 1)] = -np.sum(D, axis=1)

    return D, x
```

Matlab

```
function [D, x] = cheb_matrix(N)
% Chebyshev differentiation matrix.
    if N == 0
        D = 0; x = 1; return
    end

    % Chebyshev-Gauss-Lobatto points
    x = cos(pi * (0:N)' / N);

    % Weights: c_0 = c_N = 2, others = 1
    c = ones(N+1, 1);
    c(1) = 2; c(N+1) = 2;

    % Off-diagonal entries
    X = repmat(x, 1, N+1);
    dX = X - X';
    C = c * (1 ./ c');
    sign = (-1).^((0:N)' + (0:N));

    D = C .* sign ./ (-dX);

    % Negative sum trick for diagonal
    D(1:N+2:end) = 0;
    D(1:N+2:end) = -sum(D, 2);
end
```

6.3 Small-N Examples

6.3.1 Hand Calculations

To develop intuition, let us compute the smallest Chebyshev matrices by hand.

For $N = 1$, we have two nodes: $x_0 = 1$ and $x_1 = -1$. The only polynomial passing through two points is a line, and its derivative is constant:

$$D_1 = \begin{pmatrix} \frac{1}{2} & -\frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} \end{pmatrix}. \tag{38}$$

For $N = 2$, we have three nodes: $x_0 = 1$, $x_1 = 0$, and $x_2 = -1$. The middle row of D_2 is particularly illuminating:

$$D_2 = \begin{pmatrix} \frac{3}{2} & -2 & \frac{1}{2} \\ \frac{1}{2} & 0 & -\frac{1}{2} \\ -\frac{1}{2} & 2 & -\frac{3}{2} \end{pmatrix}. \tag{39}$$

The middle row $\left(\frac{1}{2}, 0, -\frac{1}{2}\right)$ is exactly the *centered finite difference* formula! This reveals a beautiful connection: at interior points where the grid happens to be locally symmetric, the spectral method reduces to the familiar finite difference formula.

6.4 Matrix Structure and Properties

6.4.1 The Dense Matrix

Figure 30 visualizes the structure of the Chebyshev differentiation matrix for $N = 16$.

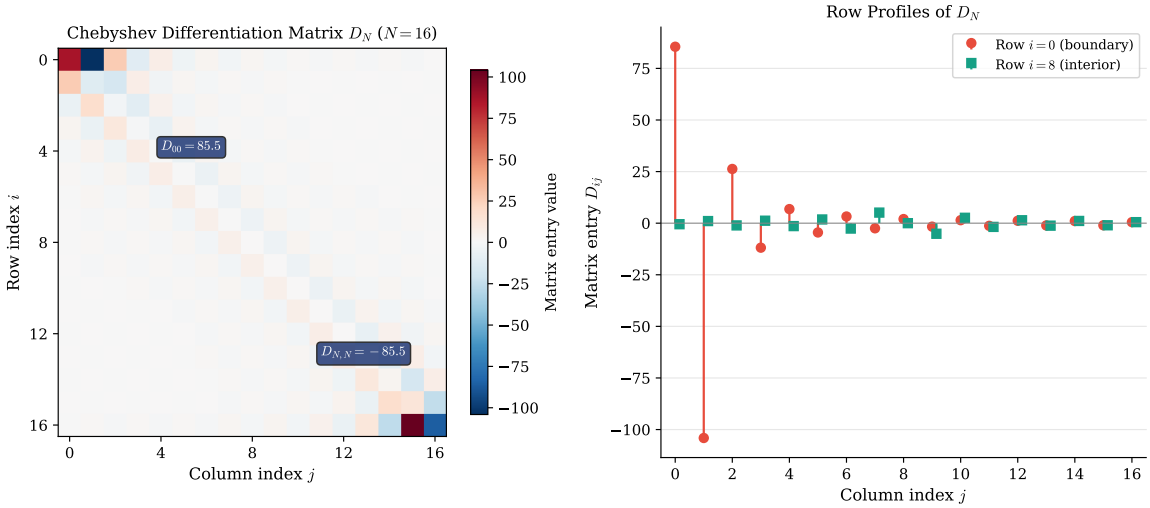

Figure 30: *Structure of the Chebyshev differentiation matrix for $N = 16$. Left: heatmap showing the matrix entries, with red indicating positive values and blue indicating negative. The large corner entries $(D_N)_{00}$ and $(D_N)_{NN}$ are visible. Right: row profiles showing boundary row (red) and interior row (green). The boundary row has large entries reflecting the $O(N^2)$ corner values.*

Unlike the sparse banded matrices of finite difference methods, the Chebyshev differentiation matrix is *dense*: every entry is generally nonzero. This is the price we pay for spectral accuracy —information from every grid point contributes to the derivative at every other point.

6.4.2 Cardinal Functions

The columns of D_N have a natural interpretation: column j contains the derivatives of the jth Lagrange cardinal function evaluated at all the grid points. Figure 31 illustrates this connection.

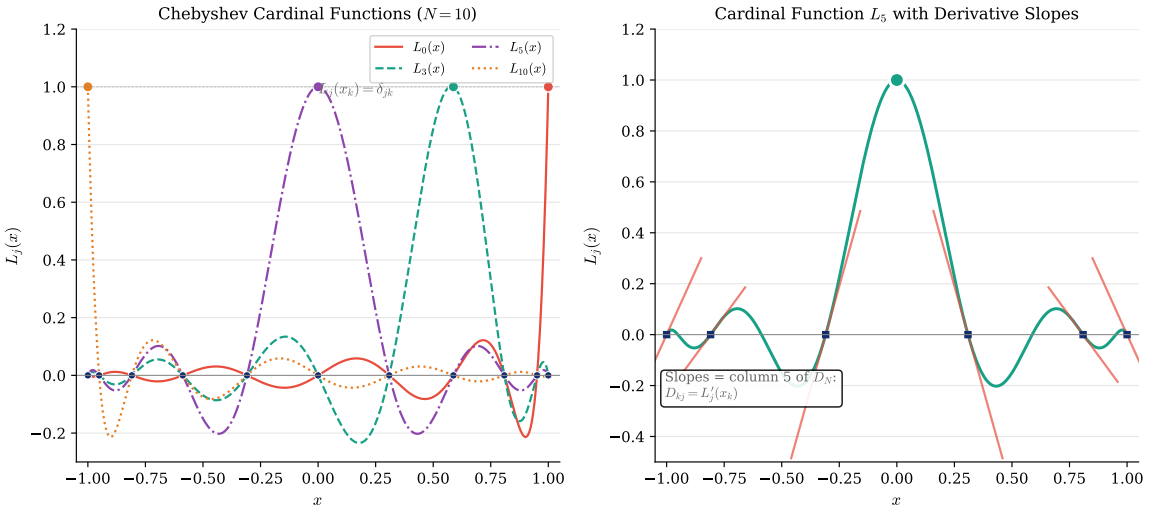

Figure 31: *Chebyshev cardinal functions (Lagrange basis polynomials). Left: several cardinal functions for $N = 10$, each peaking at value 1 at its corresponding node and vanishing at all others. Right: a single cardinal function with tangent lines at the grid points—the slopes of these tangent lines are precisely the entries in the corresponding column of the differentiation matrix.*

6.5 Demonstration: The Witch of Agnesi

6.5.1 A Smooth Test Function

To demonstrate spectral differentiation in action, we use the *Witch of Agnesi*:

$$u(x) = \frac{1}{1 + 4x^2}, \tag{40}$$

with exact derivative

$$u'(x) = \frac{-8x}{(1 + 4x^2)^2}. \tag{41}$$

This function is smooth and analytic on $[-1, 1]$, with poles at $x = \pm i/2$ in the complex plane. The distance from $[-1, 1]$ to the nearest singularity determines the rate of exponential convergence.

Figure 32 shows the function and its spectral derivative approximation for $N = 10$ and $N = 20$ grid points.

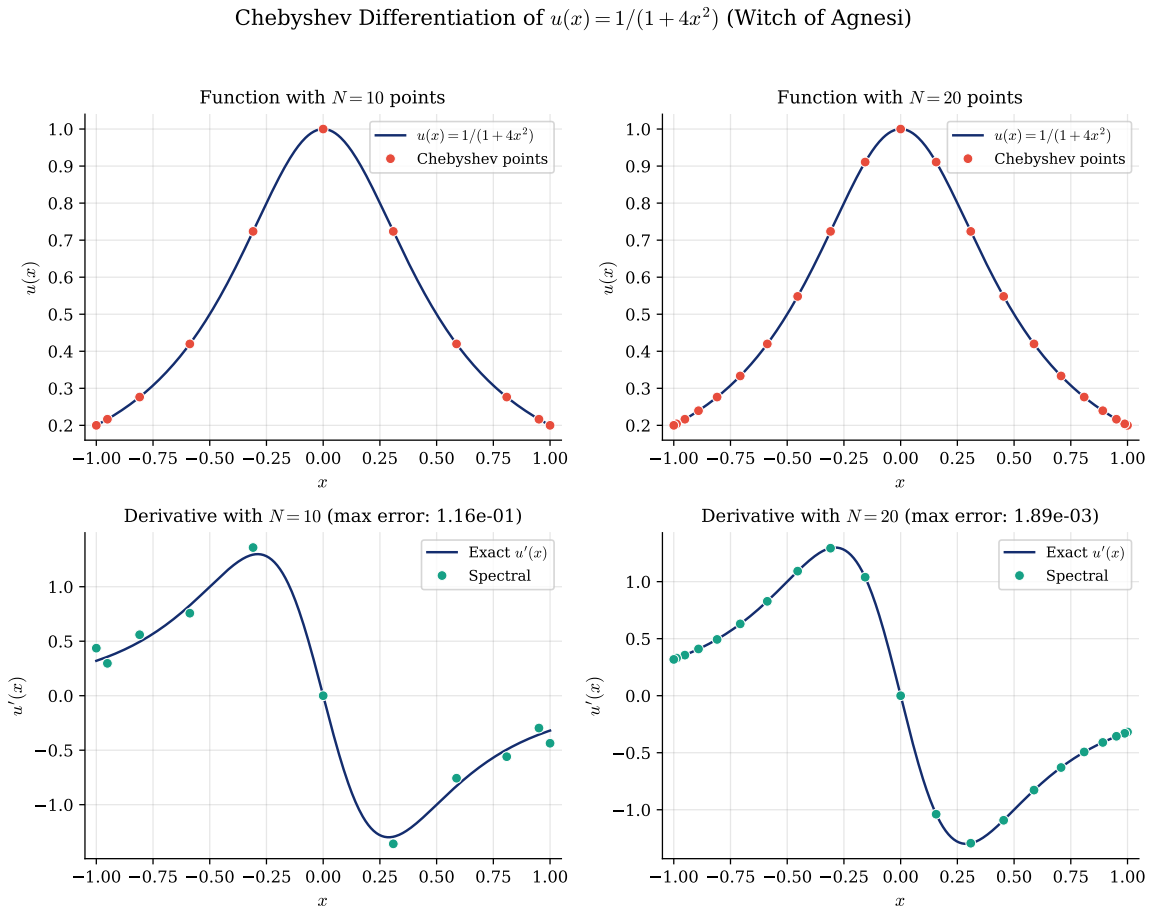

Figure 32: *Chebyshev spectral differentiation of the Witch of Agnesi $u(x) = 1/(1 + 4x^2)$. Top row: function values at Chebyshev points. Bottom row: comparison of exact and spectral derivatives, with maximum errors indicated. The error decreases exponentially with N.*

The exponential convergence is evident from Table 5, which shows the maximum differentiation error for various values of N.

N	Max error	N	Max error
4	8.5×10^{-1}	20	1.9×10^{-3}
6	4.8×10^{-1}	24	3.3×10^{-4}
8	2.4×10^{-1}	28	5.6×10^{-5}
10	1.2×10^{-1}	32	9.4×10^{-6}
12	5.3×10^{-2}	36	1.5×10^{-6}
14	2.4×10^{-2}	40	2.5×10^{-7}
16	1.0×10^{-2}	50	5.8×10^{-9}

Table 5: *Maximum differentiation error for the Witch of Agnesi $u(x) = 1/(1 + 4x^2)$. The error decreases exponentially with N.*

The following code demonstrates spectral differentiation:

Python
```python
def differentiate_witch(N):
    """Differentiate u = 1/(1+4x²) using Chebyshev spectral method."""
    D, x = cheb_matrix(N)

    # Function and exact derivative
    u = 1.0 / (1.0 + 4.0 * x**2)
    du_exact = -8.0 * x / (1.0 + 4.0 * x**2)**2

    # Spectral derivative
    du_spectral = D @ u

    # Maximum error
    max_error = np.max(np.abs(du_spectral - du_exact))
    return x, du_spectral, du_exact, max_error
```

Matlab
```matlab
function [x, du_spectral, du_exact, max_error] =
differentiate_witch(N)
% Differentiate u = 1/(1+4x²) using Chebyshev spectral method.
    [D, x] = cheb_matrix(N);

    % Function and exact derivative
    u = 1 ./ (1 + 4*x.^2);
    du_exact = -8*x ./ (1 + 4*x.^2).^2;

    % Spectral derivative
    du_spectral = D * u;

    % Maximum error
    max_error = max(abs(du_spectral - du_exact));
end
```

6.6 Spectral Convergence

6.6.1 Four Functions of Increasing Smoothness

The rate of spectral convergence depends critically on the smoothness of the function being differentiated. To illustrate this, we examine four test functions with different regularity:

1. $|x|^{5/2}$: Second derivative continuous, third derivative in bounded variation. Expected convergence: $O(N^{-2.5})$.
2. $e^{-1/(1-x^2)}$: The "bump function," infinitely differentiable (C^∞) but not analytic at $x = \pm 1$. Expected convergence: faster than any algebraic rate, but not exponential.
3. $\tanh(5x)$: Analytic on $[-1, 1]$ with poles at $x = \pm i\pi/10$. Expected convergence: exponential, $O(\rho^{-N})$ with $\rho = 1 + \pi/10 \approx 1.31$.
4. x^8: Polynomial of degree 8. Expected convergence: exact for $N \geqslant 8$.

Figure 33 displays the maximum differentiation error versus N for these four functions.

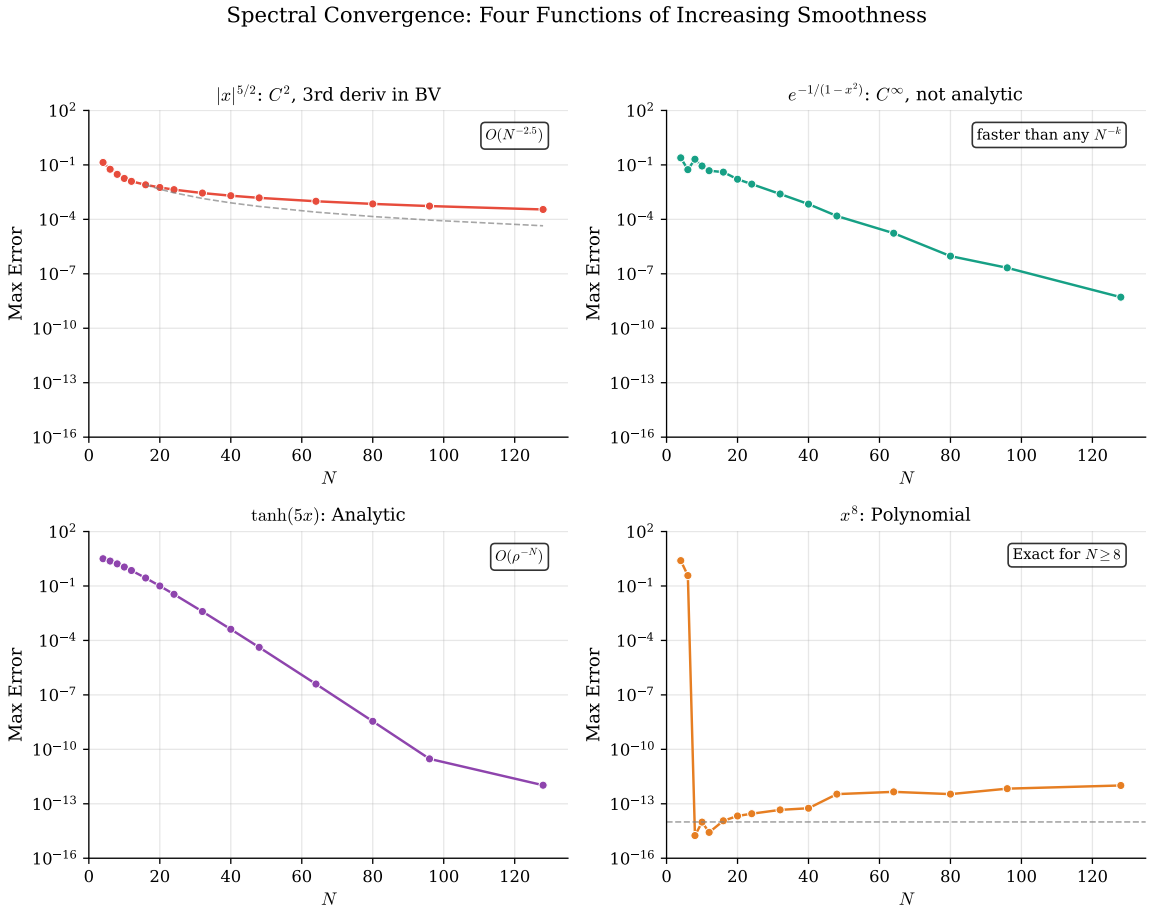

Figure 33: *Spectral convergence for four functions of increasing smoothness. Top left:* $|x|^{5/2}$ *shows algebraic convergence* $O(N^{-2.5})$ *due to limited smoothness. Top right: the bump function* $e^{-1/(1-x^2)}$ *achieves superalgebraic (faster than any power) but not exponential convergence. Bottom left:* $\tanh(5x)$ *demonstrates exponential convergence until machine precision. Bottom right: the polynomial* x^8 *is differentiated exactly for* $N \geqslant 8$.

The message is clear: spectral methods achieve their promised exponential convergence only for analytic functions. For less smooth functions, convergence is still rapid but algebraic, with the rate determined by the degree of smoothness.

6.7 Summary

This chapter has developed the Chebyshev differentiation matrix for spectral methods on bounded domains. Key points include:

- **Chebyshev points** Equation 30 provide the optimal node distribution, clustering near boundaries to avoid the Runge phenomenon.
- **The negative sum trick** Equation 37 ensures numerical stability by computing diagonal entries from row sums.

- **Matrix structure**: The Chebyshev differentiation matrix is dense, with $O(N^2)$ corner entries.
- **Spectral convergence** depends on smoothness: exponential for analytic functions, algebraic for functions with limited regularity.

In the next chapter, we will use these differentiation matrices to solve boundary value problems —the natural next step in applying spectral methods to differential equations.

CHAPTER 7

Boundary Value Problems

With the Chebyshev differentiation matrix in hand, we are ready to tackle one of the most important classes of problems in applied mathematics: boundary value problems (BVPs). Unlike initial value problems, where we march forward in time from given initial conditions, BVPs impose constraints at multiple locations—typically at the boundaries of the domain. This spatial coupling makes BVPs inherently global, and spectral methods are ideally suited to exploit this structure.

This chapter demonstrates how to transform differential equations into linear algebra problems. The differentiation matrix D_N from Section 6 becomes the workhorse, and its square D_N^2 handles second-order equations. Imposing boundary conditions requires a simple "matrix surgery"—removing rows and columns corresponding to boundary points. The result is a systematic approach that handles linear, variable-coefficient, and even nonlinear problems with remarkable ease.

7.1 Second Derivatives and Matrix Squaring

7.1.1 The Need for D^2

Most BVPs in physics involve second-order derivatives: the heat equation, the wave equation, the Poisson equation, and many others all feature u_{xx}. To apply spectral collocation, we need a second derivative matrix.

Two approaches present themselves:

1. **Direct formulas**: Derive explicit expressions for $\left(D^2\right)_{ij}$ analogous to the first derivative formulas in Section 6. These exist but are complex.
2. **Matrix squaring**: Simply compute $D^2 = D \times D$. This is $O(N^3)$ but entirely adequate for spectral N-values (typically $N < 200$).

We adopt the second approach for its simplicity. The product D_N^2 gives us the second derivative matrix: if $\boldsymbol{v}$ contains function values at the Chebyshev points, then $D_N^2\boldsymbol{v}$ approximates the second derivative values.

7.2 Imposing Boundary Conditions

7.2.1 Dirichlet Conditions: Matrix Stripping

The most common boundary conditions specify the function values at the endpoints:

$$u(-1) = \alpha, \quad u(1) = \beta. \tag{42}$$

These are *Dirichlet conditions*.

With Chebyshev points ordered as $x_0 = 1$, x_1, ..., x_{N-1}, $x_N = -1$, the boundary conditions fix $v_0 = \beta$ and $v_N = \alpha$. The differential equation need only be enforced at the *interior* points $x_1, ..., x_{N-1}$.

The implementation is straightforward:

- **Remove** the first and last rows of the equation (we don't need the ODE at boundary points).
- **Remove** the first and last columns (the boundary values are known, not unknowns).

The result is an $(N-1) \times (N-1)$ interior system:

$$\tilde{D}^2 = D_N^2[1:N-1, 1:N-1]. \tag{43}$$

For homogeneous conditions ($\alpha = \beta = 0$), the boundary terms vanish and we simply solve $\tilde{D}^2 \boldsymbol{u}_{\text{int}} = \boldsymbol{f}_{\text{int}}$.

7.3 Linear BVP: The Poisson Equation

7.3.1 A Model Problem

Consider the one-dimensional Poisson equation:

$$u_{xx} = \sin(\pi x) + 2\cos(2\pi x), \quad x \in (-1, 1), \quad u(\pm 1) = 0. \tag{44}$$

This has the exact solution

$$u(x) = -\frac{\sin(\pi x)}{\pi^2} + \frac{1 - \cos(2\pi x)}{2\pi^2}. \tag{45}$$

7.3.2 Spectral Solution

The solution procedure is direct:

1. Construct D_N and compute D_N^2.
2. Extract the interior submatrix $\tilde{D}^2 = D_N^2[1:N-1, 1:N-1]$.
3. Evaluate the right-hand side at interior points.
4. Solve the linear system $\tilde{D}^2 \boldsymbol{u}_{\text{int}} = \boldsymbol{f}_{\text{int}}$.
5. Embed the result in the full vector with boundary values.

Figure 34 shows the solution and demonstrates spectral convergence.

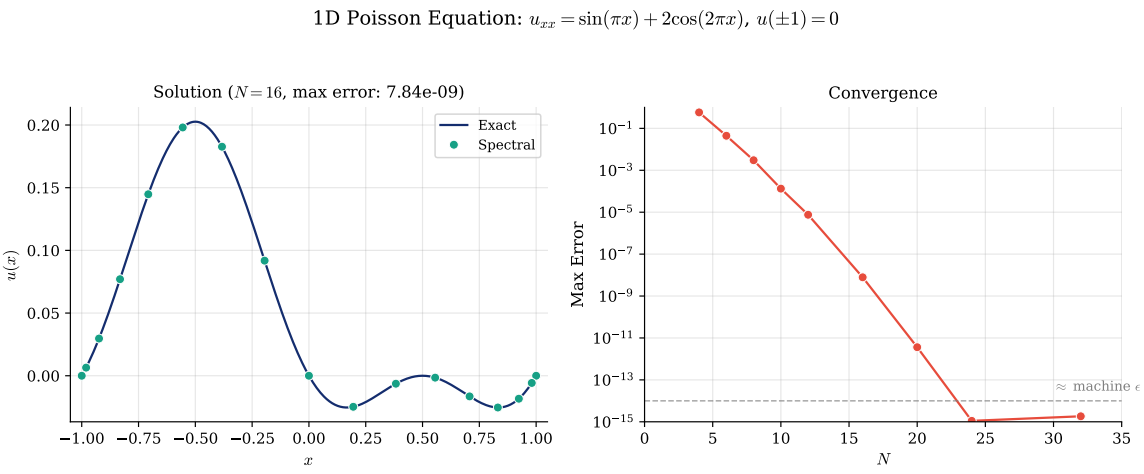

Figure 34: *Solution of the 1D Poisson equation Equation 44. Left: numerical solution (circles) compared with exact solution (line) for $N = 16$. Right: exponential convergence of the maximum error, reaching machine precision by $N = 24$.*

The implementation is remarkably concise:

Python
```python
def solve_poisson(N, f):
    """Solve u_xx = f(x) with u(±1) = 0."""
    D, x = cheb_matrix(N)
    D2 = D @ D

    # Extract interior system
    D2_int = D2[1:N, 1:N]
    f_int = f(x[1:N])

    # Solve
    u_int = np.linalg.solve(D2_int, f_int)

    # Assemble full solution
    u = np.zeros(N + 1)
    u[1:N] = u_int
    return x, u
```

Matlab
```matlab
function [x, u] = solve_poisson(N, f)
% Solve u_xx = f(x) with u(±1) = 0.
    [D, x] = cheb_matrix(N);
    D2 = D * D;

    % Extract interior system
    D2_int = D2(2:N, 2:N);
    f_int = f(x(2:N));

    % Solve
    u_int = D2_int \ f_int;

    % Assemble full solution
    u = zeros(N+1, 1);
    u(2:N) = u_int;
end
```

7.4 Variable Coefficient Problems

7.4.1 The Airy-Type Equation

Variable coefficients pose no additional difficulty for spectral methods. Consider:

$$u_{xx} - (1 + x^2)u = 1, \quad x \in (-1, 1), \quad u(\pm 1) = 0. \tag{46}$$

The variable coefficient $(1 + x^2)$ becomes a diagonal matrix. The discretized operator is:

$$L = D_N^2 - \mathrm{diag}(1 + x^2). \tag{47}$$

After extracting the interior system, we solve $\tilde{L}\boldsymbol{u}_{\text{int}} = \mathbf{1}_{\text{int}}$.

Figure 35 compares this solution with the constant-coefficient case.

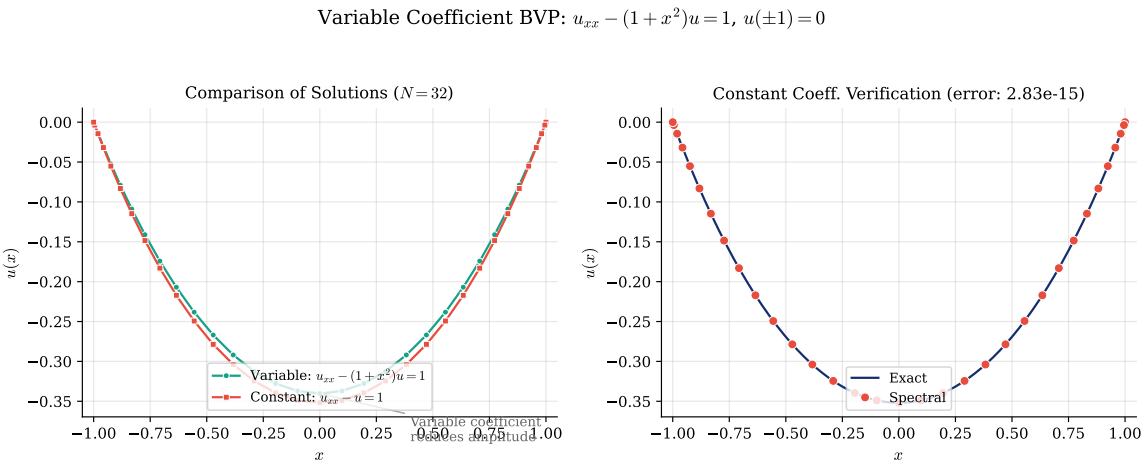

Figure 35: *Variable coefficient BVP Equation 46. Left: comparison of solutions for variable coefficient $(1 + x^2)$ and constant coefficient (1). The variable coefficient reduces the solution amplitude, especially near the boundaries where $1 + x^2$ is largest. Right: verification of the constant-coefficient case against its exact solution.*

The implementation requires only a minor modification to the Poisson solver:

Python
```python
def solve_variable_coeff(N, coeff_func):
    """Solve u_xx - c(x)*u = 1 with u(±1) = 0."""
    D, x = cheb_matrix(N)
    D2 = D @ D

    # Build operator L = D² - diag(c(x))
    c = coeff_func(x)
    L = D2 - np.diag(c)

    # Extract interior system
    L_int = L[1:N, 1:N]
    rhs = np.ones(N - 1)

    # Solve and assemble
    u_int = np.linalg.solve(L_int, rhs)
    u = np.zeros(N + 1)
    u[1:N] = u_int
    return x, u
```

Matlab
```matlab
function [x, u] = solve_variable_coeff(N, coeff_func)
% Solve u_xx - c(x)*u = 1 with u(±1) = 0.
    [D, x] = cheb_matrix(N);
    D2 = D * D;

    % Build operator L = D² - diag(c(x))
    c = coeff_func(x);
    L = D2 - diag(c);
```

```
9
10      % Extract interior system
11      L_int = L(2:N, 2:N);
12      rhs = ones(N-1, 1);
13
14      % Solve and assemble
15      u_int = L_int \ rhs;
16      u = zeros(N+1, 1);
17      u(2:N) = u_int;
18  end
```

7.5 Nonlinear BVP: The Bratu Equation

7.5.1 A Classic Nonlinear Problem

The Bratu equation models combustion and thermal explosion:

$$u_{xx} + \lambda e^u = 0, \quad x \in (-1, 1), \quad u(\pm 1) = 0. \tag{48}$$

This equation exhibits a *turning point* phenomenon: solutions exist only for $\lambda \leqslant \lambda_c$, where $\lambda_c \approx 0.878$ for the domain $[-1, 1]$. Above this critical value, no solution exists. For $\lambda = 0.5$ (well below the critical value), a unique solution exists.

The nonlinearity e^u prevents direct linear algebra. Instead, we use *Newton iteration*: linearize, solve, update, repeat.

7.5.2 Newton Iteration

Newton's method for the discrete system $F(\boldsymbol{u}) = D^2\boldsymbol{u} + \lambda e^{\boldsymbol{u}} = \boldsymbol{0}$ proceeds as follows:

1. Compute the residual: $\boldsymbol{F} = D^2\boldsymbol{u} + \lambda e^{\boldsymbol{u}}$.
2. Compute the Jacobian: $J = D^2 + \lambda \operatorname{diag}(e^{\boldsymbol{u}})$.
3. Solve for the Newton step: $J\delta\boldsymbol{u} = -\boldsymbol{F}$.
4. Update: $\boldsymbol{u} \leftarrow \boldsymbol{u} + \delta\boldsymbol{u}$.

Figure 36 shows the solution and convergence behavior.

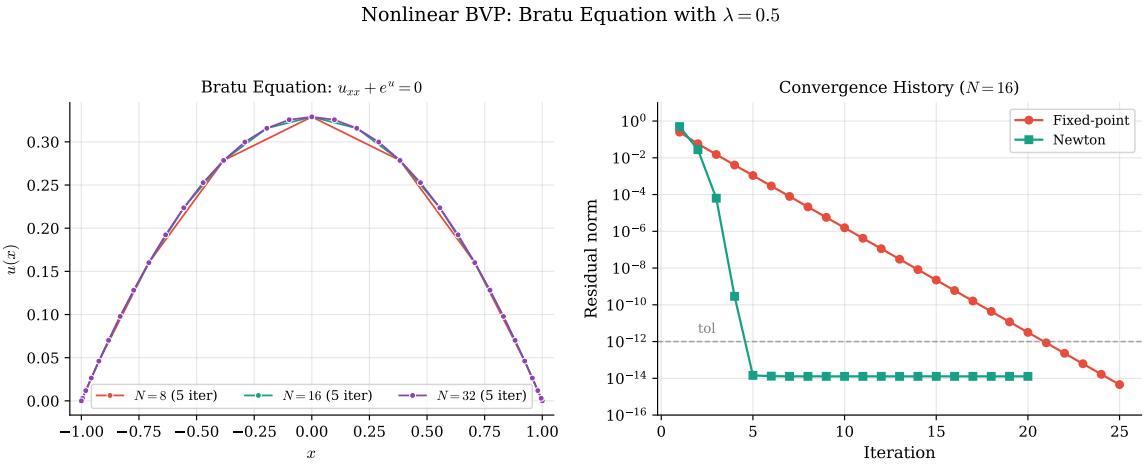

Figure 36: *Nonlinear BVP: the Bratu equation Equation 48 with $\lambda = 0.5$. Left: solutions for different values of N. Right: convergence history comparing Newton iteration (quadratic convergence) with fixed-point iteration (linear convergence). Newton typically converges in 5–8 iterations.*

The Newton iteration is straightforward to implement:

Python

```python
def solve_bratu_newton(N, lam=0.5, tol=1e-10, max_iter=50):
    """Solve u_xx + λ*exp(u) = 0 with u(±1) = 0 using Newton iteration."""
    D, x = cheb_matrix(N)
    D2 = D @ D
    D2_int = D2[1:N, 1:N]

    u = np.zeros(N - 1)  # Initial guess

    for k in range(max_iter):
        exp_u = np.exp(u)
        # Residual: F = D²u + λ*exp(u)
        F = D2_int @ u + lam * exp_u
        # Jacobian: J = D² + λ*diag(exp(u))
        J = D2_int + lam * np.diag(exp_u)
        # Newton step
        delta_u = np.linalg.solve(J, -F)
        u = u + delta_u

        if np.max(np.abs(delta_u)) < tol:
            break

    # Assemble full solution
    u_full = np.zeros(N + 1)
    u_full[1:N] = u
    return x, u_full, k + 1
```

Matlab

```matlab
function [x, u_full, iterations] = solve_bratu_newton(N, lam, tol, max_iter)
% Solve u_xx + λ*exp(u) = 0 with u(±1) = 0 using Newton iteration.
```

```
    if nargin < 4, max_iter = 50; end
    if nargin < 3, tol = 1e-10; end
    if nargin < 2, lam = 0.5; end

    [D, x] = cheb_matrix(N);
    D2 = D * D;
    D2_int = D2(2:N, 2:N);

    u = zeros(N-1, 1);  % Initial guess

    for k = 1:max_iter
        exp_u = exp(u);
        % Residual: F = D²u + λ*exp(u)
        F = D2_int * u + lam * exp_u;
        % Jacobian: J = D² + λ*diag(exp(u))
        J = D2_int + lam * diag(exp_u);
        % Newton step
        delta_u = J \ (-F);
        u = u + delta_u;

        if max(abs(delta_u)) < tol
            break
        end
    end

    % Assemble full solution
    u_full = zeros(N+1, 1);
    u_full(2:N) = u;
    iterations = k;
end
```

7.6 Eigenvalue Problems

7.6.1 Resolution Limits

The eigenvalue problem for the Laplacian,

$$u_{xx} = \lambda u, \quad x \in (-1, 1), \quad u(\pm 1) = 0, \tag{49}$$

has exact eigenvalues $\lambda_n = -(n\pi/2)^2$ and eigenfunctions $u_n(x) = \sin(n\pi(x+1)/2)$.

Spectral methods compute these eigenvalues with spectral accuracy—*for the low modes*. High-frequency modes require many points per wavelength (ppw) for accurate representation. The rule of thumb is:

$$\text{Need} \geqslant \pi \quad \text{points per wavelength for spectral accuracy.} \tag{50}$$

Figure 37 demonstrates this resolution limit.

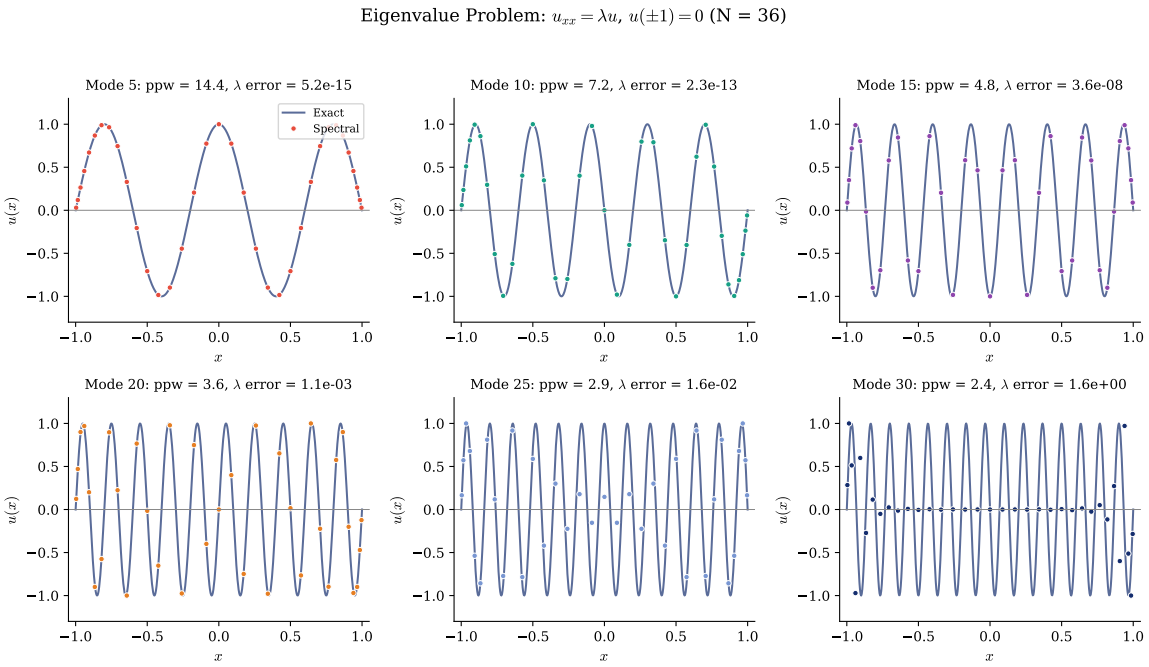

Figure 37: *Eigenvalue problem Equation 49 for $N = 36$. Six eigenmodes are shown with their points per wavelength (ppw) and eigenvalue errors. Low modes (high ppw) are resolved to near machine precision. As ppw drops below $\pi \approx 3.14$, accuracy degrades rapidly. This is not a bug but a fundamental resolution limit.*

The eigenvalue computation uses standard linear algebra:

Python
```python
def compute_laplacian_eigenvalues(N):
    """Compute eigenvalues of u_xx = λu with u(±1) = 0."""
    D, x = cheb_matrix(N)
    D2 = D @ D
    D2_int = D2[1:N, 1:N]

    # Compute eigenvalues and eigenvectors
    eigenvalues, eigenvectors = np.linalg.eig(D2_int)

    # Sort by magnitude (most negative first)
    idx = np.argsort(eigenvalues)
    eigenvalues = eigenvalues[idx]
    eigenvectors = eigenvectors[:, idx]

    # Exact eigenvalues: λ_n = -(nπ/2)²
    n = np.arange(1, N)
    exact_eigenvalues = -(n * np.pi / 2)**2

    return eigenvalues, eigenvectors, exact_eigenvalues
```

Matlab
```matlab
function [eigenvalues, eigenvectors, exact_eigenvalues] =
compute_laplacian_eigenvalues(N)
% Compute eigenvalues of u_xx = λu with u(±1) = 0.
    [D, x] = cheb_matrix(N);
```

```
    D2 = D * D;
    D2_int = D2(2:N, 2:N);

    % Compute eigenvalues and eigenvectors
    [eigenvectors, Lambda] = eig(D2_int);
    eigenvalues = diag(Lambda);

    % Sort by magnitude (most negative first)
    [eigenvalues, idx] = sort(eigenvalues);
    eigenvectors = eigenvectors(:, idx);

    % Exact eigenvalues: λ_n = -(nπ/2)²
    n = (1:N-1)';
    exact_eigenvalues = -(n * pi / 2).^2;
end
```

7.7 Two-Dimensional Problems

7.7.1 Tensor Products

For problems on a rectangle $[-1, 1]^2$, we use *tensor product* grids: Chebyshev points in both x and y directions. The total number of grid points is $(N + 1)^2$.

The 2D Laplacian operator is built using *Kronecker products*:

$$L = I \otimes D^2 + D^2 \otimes I, \tag{51}$$

where I is the identity matrix and $\otimes$ denotes the Kronecker product.

Figure 38 shows the tensor product grid structure.

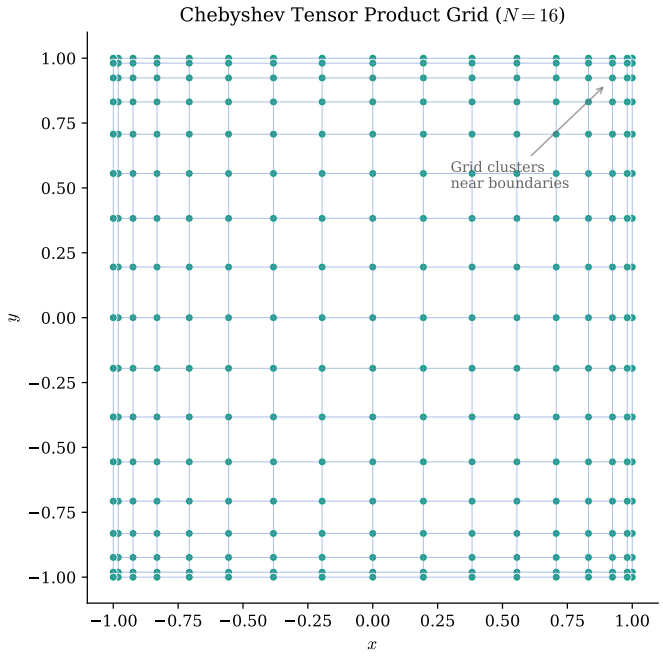

Figure 38: *Chebyshev tensor product grid for $N = 16$. Points cluster near all four boundaries, providing the boundary resolution needed for spectral accuracy in two dimensions.*

7.7.2 2D Poisson Problem

Consider the 2D Poisson equation:

$$u_{xx} + u_{yy} = -2\pi^2 \sin(\pi x)\sin(\pi y), \quad u = 0 \text{ on boundary.} \tag{52}$$

The exact solution is $u(x, y) = \sin(\pi x)\sin(\pi y)$.

Figure 39 shows the solution.

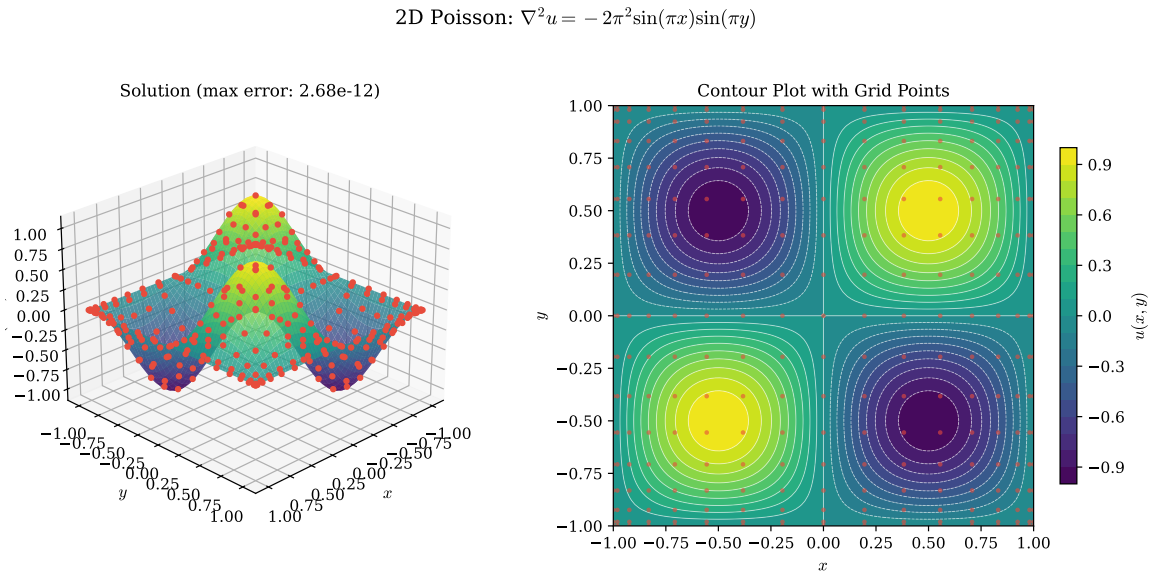

Figure 39: *Solution of the 2D Poisson equation Equation 52. Left: 3D surface plot of the solution. Right: contour plot with Chebyshev grid points overlaid. For $N = 16$, the maximum error is approximately 10^{-12}.*

The Kronecker product formulation leads to concise code:

Python
```python
def solve_poisson_2d(N, f):
    """Solve ∇²u = f on [-1,1]² with u = 0 on boundary."""
    D, x = cheb_matrix(N)
    D2 = D @ D
    D2_int = D2[1:N, 1:N]
    x_int = x[1:N]

    # Build 2D Laplacian: L = I ⊗ D² + D² ⊗ I
    I = np.eye(N - 1)
    L = np.kron(I, D2_int) + np.kron(D2_int, I)

    # Right-hand side on interior grid
    X, Y = np.meshgrid(x_int, x_int)
    F = f(X, Y).flatten(order='F')

    # Solve and reshape
    u_vec = np.linalg.solve(L, F)
    U_int = u_vec.reshape((N-1, N-1), order='F')

    # Embed in full grid with zero boundary
    U = np.zeros((N+1, N+1))
    U[1:N, 1:N] = U_int
    return np.meshgrid(x, x), U
```

Matlab
```matlab
function [grids, U] = solve_poisson_2d(N, f)
% Solve ∇²u = f on [-1,1]² with u = 0 on boundary.
    [D, x] = cheb_matrix(N);
    D2 = D * D;
    D2_int = D2(2:N, 2:N);
    x_int = x(2:N);

    % Build 2D Laplacian: L = I ⊗ D² + D² ⊗ I
    I = eye(N - 1);
    L = kron(I, D2_int) + kron(D2_int, I);

    % Right-hand side on interior grid
    [X, Y] = meshgrid(x_int, x_int);
    F = f(X, Y);

    % Solve and reshape
    u_vec = L \ F(:);
    U_int = reshape(u_vec, N-1, N-1);

    % Embed in full grid with zero boundary
    U = zeros(N+1, N+1);
    U(2:N, 2:N) = U_int;
```

```
    [grids{1}, grids{2}] = meshgrid(x, x);
end
```

The sparsity pattern of the 2D Laplacian operator, shown in Figure 40, reveals the Kronecker product structure.

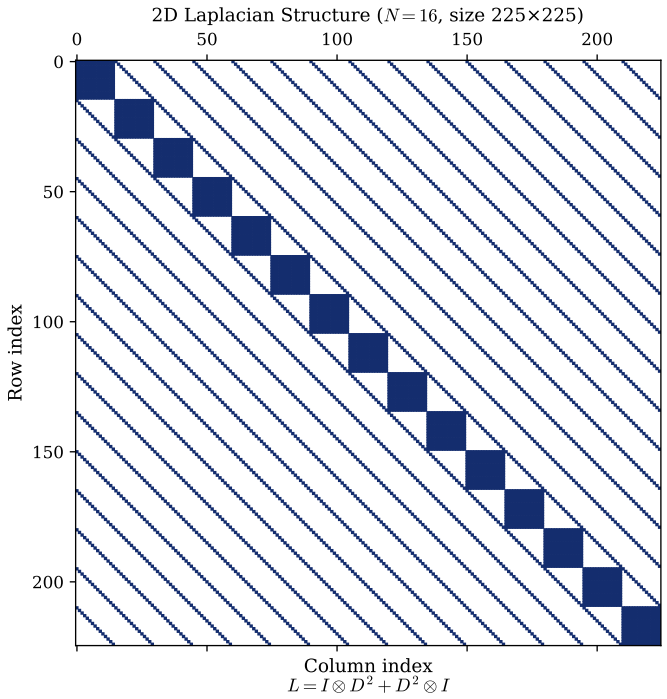

Figure 40: *Sparsity pattern of the 2D Laplacian matrix for $N = 16$. The $(N-1)^2 \times (N-1)^2$ matrix shows the characteristic block structure arising from the Kronecker product formulation Equation 51.*

7.8 The Helmholtz Equation

7.8.1 Near-Resonance Behavior

The Helmholtz equation models wave phenomena:

$$u_{xx} + u_{yy} + k^2 u = f(x, y), \quad u = 0 \text{ on boundary.} \tag{53}$$

When k^2 approaches an eigenvalue of the Laplacian, the system becomes *nearly resonant* and the solution amplitude grows dramatically.

For a localized Gaussian forcing $f(x, y) = e^{-20[(x-0.3)^2+(y+0.4)^2]}$ and $k = 7$, we are near resonance with the $(2, 4)$ mode (theoretical $k \approx 7.02$).

Figure 41 shows the characteristic modal structure of the near-resonant solution.

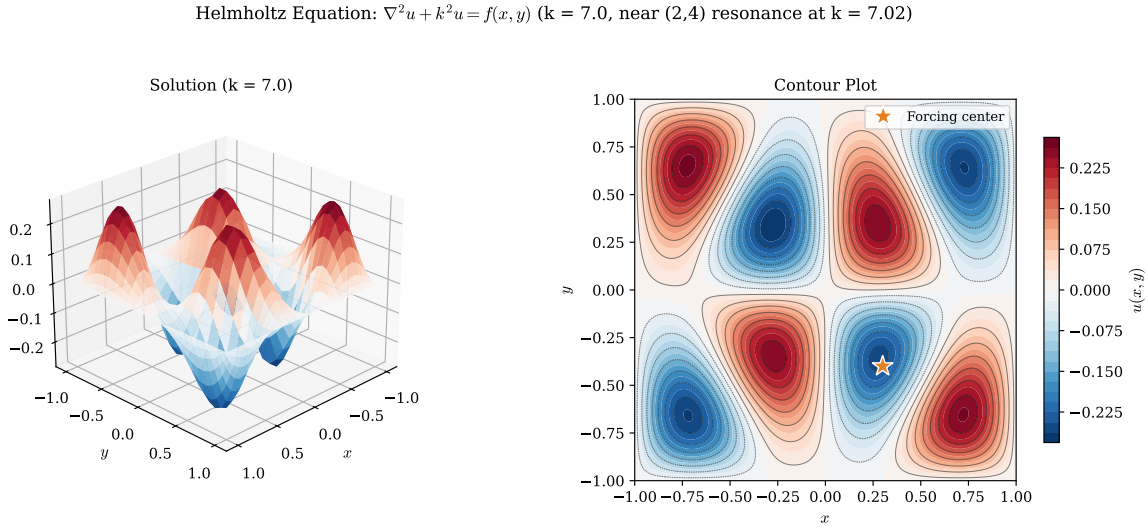

Figure 41: *Helmholtz equation Equation 53 with $k = 7$, near resonance with the $(2, 4)$ eigenmode ($k_{2,4} \approx 7.02$). Left: 3D surface showing the wave-like structure. Right: contour plot with forcing location marked. The solution exhibits the characteristic pattern of the $(2, 4)$ eigenmode.*

The Helmholtz solver modifies the 2D Laplacian by adding the $k^2 I$ term:

Python

```python
def solve_helmholtz(N, k, f):
    """Solve ∇²u + k²u = f on [-1,1]² with u = 0 on boundary."""
    D, x = cheb_matrix(N)
    D2 = D @ D
    D2_int = D2[1:N, 1:N]
    x_int = x[1:N]

    # Build Helmholtz operator: L = I ⊗ D² + D² ⊗ I + k²I
    I = np.eye(N - 1)
    n_int = (N - 1)**2
    L = np.kron(I, D2_int) + np.kron(D2_int, I) + k**2 * np.eye(n_int)

    # Right-hand side
    X, Y = np.meshgrid(x_int, x_int)
    F = f(X, Y).flatten(order='F')

    # Solve
    u_vec = np.linalg.solve(L, F)
    U_int = u_vec.reshape((N-1, N-1), order='F')

    U = np.zeros((N+1, N+1))
    U[1:N, 1:N] = U_int
    return np.meshgrid(x, x), U
```

Matlab

```matlab
function [grids, U] = solve_helmholtz(N, k, f)
% Solve ∇²u + k²u = f on [-1,1]² with u = 0 on boundary.
    [D, x] = cheb_matrix(N);
```

```
    D2 = D * D;
    D2_int = D2(2:N, 2:N);
    x_int = x(2:N);

    % Build Helmholtz operator: L = I ⊗ D² + D² ⊗ I + k²I
    I = eye(N - 1);
    n_int = (N - 1)^2;
    L = kron(I, D2_int) + kron(D2_int, I) + k^2 * eye(n_int);

    % Right-hand side
    [X, Y] = meshgrid(x_int, x_int);
    F = f(X, Y);

    % Solve
    u_vec = L \ F(:);
    U_int = reshape(u_vec, N-1, N-1);

    U = zeros(N+1, N+1);
    U(2:N, 2:N) = U_int;
    [grids{1}, grids{2}] = meshgrid(x, x);
end
```

7.9 Summary

This chapter has demonstrated the power and simplicity of spectral collocation for boundary value problems:

- **Matrix stripping** handles Dirichlet boundary conditions by removing boundary rows and columns.
- **Variable coefficients** become diagonal matrices—no additional complexity.
- **Nonlinear problems** yield to Newton iteration, with the Jacobian easily computed from the differentiation matrix.
- **Eigenvalue problems** reveal the resolution limits of spectral methods: $\geqslant \pi$ points per wavelength are needed.
- **Tensor products** extend the method to higher dimensions via Kronecker products.

The key insight throughout is that spectral methods transform differential equations into *dense linear algebra*. The matrices are small (because spectral accuracy requires few points), and the resulting systems can be solved directly. This directness—no iteration, no mesh refinement—is the hallmark of spectral methods for smooth problems.

Bibliography

[1] B. Fornberg, *A practical guide to pseudospectral methods*. Cambridge University Press, 1996, p. 242.
[2] L. N. Trefethen, *Spectral methods in MatLab*. Society for Industrial, Applied Mathematics, Philadelphia, PA, USA, 2000, p. 184. [Online]. Available: http://web.comlab.ox.ac.uk/oucl/work/nick.trefethen/spectral.html
[3] J. P. Boyd, *Chebyshev and Fourier Spectral Methods*, 2nd ed. Dover Publications, New York, 2000, p. 688.
[4] A. N. Tikhonov and A. A. Samarskii, *Equations of Mathematical Physics*. Dover Publications, Inc., 1963, p. 785.
[5] P. Vértesi, "Optimal Lebesgue constant for Lagrange interpolation," *SIAM J. Numer. Anal.*, vol. 27, pp. 1322–1331, 1990.
[6] J. S. Hesthaven and T. Warburton, *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications*, vol. 54. in Texts in Applied Mathematics, vol. 54. Springer, 2007.
[7] B. Fornberg, "Generation of finite difference formulas on arbitrarily spaced grids," *Mathematics of Computation*, vol. 51, no. 184, pp. 699–706, 1988, doi: 10.1090/S0025-5718-1988-0935077-0.
[8] T. A. Driscoll, "Finite difference weights MATLAB function." 2007.