SCIENTIFIC COMPUTATION

Implementing Spectral Methods for Partial Differential Equations

Scientific Computation

For other titles published in this series, go to
www.springer.com/series/718

David A. Kopriva

Implementing Spectral Methods for Partial Differential Equations

Algorithms for Scientists and Engineers

Springer

Prof. Dr. David A. Kopriva
Dept. of Mathematics
Florida State University
Tallahassee, FL 32306-4510
USA
e-mail: kopriva@math.fsu.edu

ISBN 978-90-481-2260-8 e-ISBN 978-90-481-2261-5

DOI 10.1007/978-90-481-2261-5

Library of Congress Control Number: 2009922124

Printed on acid-free paper

9 8 7 6 5 4 3 2 1

springer.com

To my Wife, my Mother, and in memory of my Dad.

Preface

This book is aimed to be both a textbook for graduate students and a starting point for applications scientists. It is designed to show how to implement spectral methods to approximate the solutions of partial differential equations. It presents a systematic development of the fundamental algorithms needed to write spectral methods codes to solve basic problems of mathematical physics, including steady potentials, transport, and wave propagation. As such, it is meant to supplement, not replace, more general monographs on spectral methods like the recently updated "Spectral Methods: Fundamentals in Single Domains" and "Spectral Methods: Evolution to Complex Geometries and Applications to Fluid Dynamics" by Canuto, Hussaini, Quarteroni and Zang, which provide detailed surveys of the variety of methods, their performance and theory.

I was motivated by comments that I have heard over the years that spectral methods are "too hard to implement." I hope to dispel this view—or at least to remove the "too". Although it is true that a spectral code is harder to hack together than a simple finite difference code (at least a low order finite difference method on a square domain), I show that only a few fundamental algorithms for interpolation, differentiation, FFT and quadrature—the subjects of basic numerical methods courses—form the building blocks of any spectral code, even for problems in complex geometries. I present the algorithms not only to solve problems in 1D, but 2D as well, to show the flexibility of spectral methods and to make as straightforward as possible the transition from simple, exploratory programs that illustrate the behavior of the methods to application programs.

I assume that the reader has a basic knowledge of numerical algorithms. The most important topics include interpolation, quadrature and the numerical integration of ordinary differential equations. An understanding of other methods for the solution of PDEs, such as finite difference or finite element methods is very helpful.

Although I assume some background in numerical methods, I have tried to make the presentation and the collection of algorithms self contained. The idea is to make it as straightforward as possible to implement the methods without the need to search for auxiliary routines. Of course, some of the routines (like FFTs, for example) should be later replaced by well-tested and optimally designed versions in the programmer's programming language of choice. Also, since I emphasize the implementation of spectral methods, I recommend having one of the many more general books, such as the two mentioned above, available to consult on the theory of the methods.

I have chosen to present the algorithms in a detailed pseudocode, rather than in a specific language such as C or Fortran or application such as Matlab, Maple or Mathematica. The idea is to present the algorithms unencumbered by a language syntax that the reader might not know, but that can nevertheless be quickly translated into the programmer's favorite computer language. I hope that this will make

the methods useful to the widest possible audience and for the widest range of applications. This is not a programming book, however, and the translation process of the pseudocode will depend on the particular computer language chosen. Fortran programmers will most likely replace loops with vector operations. C/C++ programmers will often have to adjust array indices. The object oriented concepts used for many of the algorithms will be natural to C++ programmers, but a number of tutorials on the web for Fortran programmers show how to implement object oriented ideas using modules, at least until F2003 compilers become available.

The book consists of two parts. The first is a quick introduction to spectral approximation to provide a reference point and to define the notation. It is followed by the development of the basic algorithms that form the building blocks of spectral codes. These algorithms include how to use the complex FFT to compute real transforms, how to compute Chebyshev and Legendre polynomials and then how to use them to approximate integrals and derivatives.

The second part presents algorithms to approximate the solutions of PDEs. It includes a short survey of spectral approximations that shows how to use the building blocks in one space dimension. Our main interest, however, is how to solve problems in complex geometries in two space dimensions, so we put the emphasis on collocation and nodal versions of Galerkin spectral methods. The development of the algorithms starts from basic approximation on the square, then moves to more complex geometries through the introduction of boundary-fitted mappings, and ends with spectral multidomain methods.

I am grateful for the efforts of my students and colleagues who did so much to help me prepare this work. I'd like to thank my students Wuming Zhu, James DeMarco, Matt Willyard and Cesar Acosta who gave suggestions on what they wanted to see in a book on spectral methods and implemented algorithms. Tom Zang deserves many thanks for taking the time to read through an entire draft of the manuscript and provide detailed comments. Last, but certainly not least, I want to thank Yousuff Hussaini, who started my research in spectral methods. He has mentored me and encouraged my career for two and a half decades. Many of the topics covered here came out of projects that we have worked on together.

Tallahassee, FL David A. Kopriva
October 2008

Contents

List of Algorithms

Part I
Approximating Functions, Derivatives and Integrals

Chapter 1
Spectral Approximation

1.1 Preamble: Series Solution of PDEs

Spectral methods owe their origins to series solutions of partial differential equations (PDEs). To motivate spectral methods, we think it is instructive to recall how PDEs are solved by analytical means.

To illustrate the Fourier series solution of a PDE, let's solve the diffusion(heat) equation in one space dimension for periodic boundary conditions. The temperature, φ, satisfies the time dependent partial differential equation

$$\begin{cases} \varphi_t = \varphi_{xx}, \quad 0 < x < 2\pi, \\ \varphi(x,0) = f(x), \quad 0 \le x \le 2\pi, \\ \varphi(0,t) = \varphi(2\pi,t). \end{cases} \tag{1.1}$$

To solve by separation of variables, we write the temperature in the form of a Fourier series

$$\varphi(x,t) = \sum_{k=-\infty}^{\infty} \hat{\varphi}_k(t) e^{ikx}. \tag{1.2}$$

To compute the solution unknowns, namely the Fourier coefficients $\hat{\varphi}_k$, we substitute the series into the differential equation and gather factors of the complex exponential functions

$$\sum_{k=-\infty}^{\infty} \left(\frac{d\hat{\varphi}_k(t)}{dt} + k^2 \hat{\varphi}_k \right) e^{ikx} = 0. \tag{1.3}$$

Since the complex exponentials e^{ikx} are linearly independent, the coefficient of each must individually vanish. From each coefficient we build a system of ordinary differential equations for the $\hat{\varphi}_k$

$$\frac{d\hat{\varphi}_k(t)}{dt} = -k^2 \hat{\varphi}_k, \quad -\infty < k < \infty. \tag{1.4}$$

The initial conditions are

$$\hat{\varphi}_k(0) = \hat{f}_k, \tag{1.5}$$

where $\hat{f}_k$ are the Fourier coefficients

$$\hat{f}_k = \frac{1}{2\pi} \int_0^{2\pi} f(x) e^{-ikx} dx. \tag{1.6}$$

D.A. Kopriva, *Implementing Spectral Methods for Partial Differential Equations,* Scientific Computation,

We then solve the system of equations for the coefficients individually for each value of k,

$$\hat{\varphi}_k = e^{-k^2 t} \hat{f}_k. \tag{1.7}$$

The analytic series solution of (1.1) is therefore

$$\varphi(x,t) = \sum_{k=-\infty}^{\infty} \hat{f}_k e^{ikx - k^2 t}. \tag{1.8}$$

The practical matter of the analytical solution (1.8) is that we must truncate the infinite series to evaluate it. Truncation creates an approximation

$$\sum_{k=-N/2}^{N/2} \hat{f}_k e^{ikx - k^2 t} \approx \varphi(x,t). \tag{1.9}$$

How good this approximation is, and how large N needs to be to make the approximation precise, depends on the rate at which the coefficients, $\hat{f}_k e^{-k^2 t}$, go to zero. From a purely theoretical point of view, the convergence rate is not so important. From a practical one, however, how large N has to be to get a sufficiently accurate approximation determines how long we have to wait while computing the solution.

When we solve PDEs by separation of variables in other coordinates, such as cylindrical or spherical polar coordinates, we would use other orthogonal expansion functions such as Bessel, Legendre or Chebyshev functions instead of the complex exponentials. We can approximate those infinite series solutions, too, by finite series.

The fundamental idea behind spectral methods is to approximate solutions of PDEs by finite series of orthogonal functions such as the complex exponentials (1.9), Chebyshev or Legendre polynomials. The devil, of course, is in the details. We will see that there are essentially three choices that we have to make to derive a spectral method: what expansion functions should be used, the form in which the approximation will be written, and finally the procedure by which the solution unknowns are determined. Over the course of the first part of this book, we will survey the mathematics we need to help make choices between different types of spectral approximations.

1.2 The Fourier Basis Functions and Fourier Series

The starting point for all spectral methods is to approximate the solution of a differential equation by a finite sum of orthogonal basis functions. For periodic problems, we use an expansion in the Fourier basis functions, which are the complex exponentials e^{inx}. We write that expansion as

$$S_N(x) = \sum_{n=-N/2}^{N/2} \hat{s}_n e^{inx}. \tag{1.10}$$

A spectral method defines the way we will find the $\hat{s}_n$.

We have just said that the complex exponentials are orthogonal on the interval $[0, 2\pi]$. Recall that a set of functions $\{\phi_n(x)\}_{n=0}^{N}$ is *orthogonal* on an interval $[a, b]$ with respect to a weight function, w, if

$$(\phi_n, \phi_m)_w \equiv \int_a^b \phi_n(x)\phi_m^*(x)w(x)dx = C_n\delta_{n,m}. \tag{1.11}$$

Here, $\delta_{n,m}$ is the Kronecker delta function, which is defined as

$$\delta_{n,m} = \begin{cases} 1, & n = m, \\ 0, & n \neq m \end{cases} \tag{1.12}$$

and ϕ_n^* is the complex conjugate of ϕ_n. The integral denoted by $(\phi_n, \phi_m)_w$ is the *weighted inner product* of the two functions ϕ_n and ϕ_m. The inner product of a function with itself gives the square of its *norm*,

$$\|\phi_n\|_w^2 = (\phi_n, \phi_n)_w = \int_a^b |\phi_n(x)|^2 w(x)dx = C_n. \tag{1.13}$$

The complex exponentials are periodic with period 2π and are orthogonal on that interval with respect to the weight function $w = 1$. The inner product of two complex exponentials is

$$\left(e^{inx}, e^{imx}\right) = \int_0^{2\pi} e^{i(n-m)x}dx = \begin{cases} 2\pi, & n = m, \\ 0 & n \neq m. \end{cases} \tag{1.14}$$

As we just did in (1.14), we will leave off the weight function from the inner product and the norm when the weight is equal to one.

Orthogonal function expansions are especially attractive because we can easily find the coefficients in a series by *orthogonal projection*. The orthogonal projection of the function S_N onto the basis function e^{imx} is the inner product

$$\left(S_N, e^{imx}\right) = \left(\sum_{n=-N/2}^{N/2} \hat{s}_n e^{inx}, e^{imx}\right) = \sum_{n=-N/2}^{N/2} \hat{s}_n\left(e^{inx}, e^{imx}\right) = 2\pi\hat{s}_m. \tag{1.15}$$

Thus, the coefficients in the expansion are simply

$$\hat{s}_m = \frac{1}{2\pi}\left(S_N, e^{imx}\right), \quad m = -N/2, \ldots, N/2. \tag{1.16}$$

In spectral methods, approximations are bettered by increasing the number of basis functions, $N + 1$, used in (1.10). For the series to converge as $N \to \infty$, it is necessary that the infinite set $\{\phi_n(x)\}_{n=0}^{\infty}$ forms a *basis*, in this case a basis for the space of all functions $f(x)$ that satisfy

$$\|f\|_w = \sqrt{\int_a^b |f|^2 wdx} < \infty. \tag{1.17}$$

This is the space of functions that are square integrable with respect to a weight w over the interval $[a, b]$ and is typically denoted by $L^2_w(a, b)$.

The complex exponentials do form a basis for $L^2(0, 2\pi)$, and we can represent any square integrable function, f, on the interval $[0, 2\pi]$ as an infinite series

$$f = \sum_{k=-\infty}^{\infty} \hat{f}_k e^{ikx}. \tag{1.18}$$

The coefficients, $\hat{f}_k$, of f are the *Fourier coefficients* and we find them from the orthogonal projection (f, e^{ikx}), which again produces

$$\hat{f}_k = \frac{1}{2\pi}\left(f, e^{ikx}\right). \tag{1.19}$$

Equation (1.19) is nothing but the well-known *Fourier Transform*, with (1.18) to represent its inverse. Furthermore, if f is sufficiently smooth, its derivative is the infinite series

$$f'(x) = \sum_{k=-\infty}^{\infty} ik\hat{f}_k e^{ikx} = \sum_{k=-\infty}^{\infty} \hat{f}_k^{(1)} e^{ikx}. \tag{1.20}$$

1.3 Series Truncation

Spectral methods typically use one of two methods to approximate a square integrable function as a finite expansion in orthogonal basis functions. The first, and seemingly most natural method, is to truncate the infinite series, as we did in (1.9). The second, which we will discuss later in Sect. 1.6, is to approximate the function by interpolation.

To derive the Fourier truncation operator, we split the sum (1.18) into two parts, one for wavenumbers, k, less than or equal to $N/2$ and the other for the remainder,

$$f(x) = \sum_{k=-N/2}^{N/2} \hat{f}_k e^{ikx} + \sum_{|k|=N/2+1}^{\infty} \hat{f}_k e^{ikx} \equiv \sum_{k=-N/2}^{N/2} \hat{f}_k e^{ikx} + \tau. \tag{1.21}$$

By the sum with the absolute value on $|k|$, we mean

$$\sum_{|k|=N/2+1}^{\infty} \equiv \sum_{k=-\infty}^{-(N/2+1)} + \sum_{k=N/2+1}^{\infty}. \tag{1.22}$$

From (1.21), we define the *Fourier truncation operator*, P_N, to have the action

$$P_N f(x) = \sum_{k=-N/2}^{N/2} \hat{f}_k e^{ikx}. \tag{1.23}$$

Since $P_N f$ is a finite sum of complex exponentials, we will say that $P_N f$ is a *Fourier polynomial of degree* $\leq N$. The remainder, τ, is the *truncation error*.

We now make a number of observations about Fourier truncation. First, the operator P_N is the *orthogonal projection* from the infinite dimensional space of square integrable functions onto the finite dimensional space of Fourier polynomials of degree $\leq N$. If we project a second time, nothing changes

$$P_N\left(P_N f(x)\right) = P_N f(x), \tag{1.24}$$

i.e. $P_N^2 = P_N$, and therefore

$$\left(\tau, e^{ikx}\right) = 0, \quad k = -N/2, \ldots, N/2. \tag{1.25}$$

Next, we approximate the derivative of a function by the derivative of the finite expansion, (1.23), which we write as $(P_N f)'$. Since $de^{ikx}/dx = ike^{ikx}$,

$$(P_N f(x))' = \sum_{k=-N/2}^{N/2} ik\hat{f}_k e^{ikx} = P_N\left(f'(x)\right) \quad (\textit{Fourier Truncation}). \tag{1.26}$$

So the order of truncation and differentiation is irrelevant; differentiation and the Fourier truncation operator commute.

The error that we introduce by the truncation approximation is the truncation error,

$$\tau = f(x) - P_N f(x) = \sum_{|k|=N/2+1}^{\infty} \hat{f}_k e^{ikx}. \tag{1.27}$$

Immediately, (1.27) tells us that the size of the error depends on how rapidly the coefficients, $\hat{f}_k$, decay to zero, which they must, ultimately, for the series to converge. We measure the size of this error by its norm, whose square is

$$\begin{aligned}
\|f(x) - P_N f(x)\|^2 &= \int_0^{2\pi} \left(\sum_{|k|=N+1}^{\infty} \hat{f}_k e^{ikx}\right)\left(\sum_{|l|=N+1}^{\infty} \hat{f}_l^* e^{-ilx}\right) dx \\
&= \sum_{|k|=N+1}^{\infty} \sum_{|l|}^{\infty} \hat{f}_k \hat{f}_l^* \int_0^{2\pi} e^{ikx} e^{-ilx} dx \\
&= \sum_{|k|=N+1}^{\infty} \sum_{|l|=N+1}^{\infty} \hat{f}_k \hat{f}_l^* \left(e^{ikx}, e^{ilx}\right) \\
&= \sum_{|k|=N+1}^{\infty} |\hat{f}_k|^2 \left\|e^{ikx}\right\|^2.
\end{aligned} \tag{1.28}$$

The fact that the basis functions are orthogonal and

$$\left\| e^{ikx} \right\|^2 = 2\pi \tag{1.29}$$

means that the error satisfies

$$\| f(x) - P_N f(x) \|^2 = 2\pi \sum_{|k|=N/2+1}^{\infty} |\hat{f}_k|^2. \tag{1.30}$$

(Note that we get the well-known *Parseval's equality*,

$$\| f \|^2 = 2\pi \sum_{k=-\infty}^{\infty} |\hat{f}_k|^2 \tag{1.31}$$

by the same argument.)

To get a feeling for how the smoothness of the function affects the error of the truncation approximation, let us study the approximations of three functions

$$\begin{aligned} f_1 &= x, \\ f_2 &= x\,(2\pi - x)\,, \\ f_3 &= \frac{3}{5 - 4\cos(x)}, \end{aligned} \tag{1.32}$$

that we define on the interval $[0, 2\pi]$ and extend periodically to the right and the left. We show these three functions plotted in Fig. 1.1. Remember, that the Fourier transform treats a function on $[0, 2\pi]$ as if it is replicated forever both to the right and to the left, that is, it computes the transform of the periodic extension of the function. The periodic extension of the first function, $f = x$ has a jump discontinuity. We compute the Fourier coefficients directly from the definition, and they decay as $1/k$. The periodic extension of the second function is continuous, but does not have continuous first derivatives. Its coefficients decay as $1/k^2$. The periodic extension of the final function, f_3 is continuous, and all of its derivatives are continuous, so it is a very smooth function. Its Fourier coefficients decay as $e^{-k\ln(2)}$. Figure 1.2 shows the logarithm of the Fourier coefficients as a function of the wavenumber, k. It shows the slow convergence of the coefficients for the first two functions and the rapid, exponential convergence of the last.

We can compute the truncation error from the coefficients and (1.30). The first two functions have Fourier coefficients that decay as $1/k^p$, where $p = 1$ or 2. We bound the sum of the coefficients in (1.30) for functions that decay like this by

$$\sum_{|k|=N/2+1}^{\infty} \frac{1}{k^{2p}} < \int_{N/2+1}^{\infty} \frac{1}{z^{2p}} dz = \frac{1}{(2p-1)\,(N/2+1)^{2p-1}}. \tag{1.33}$$

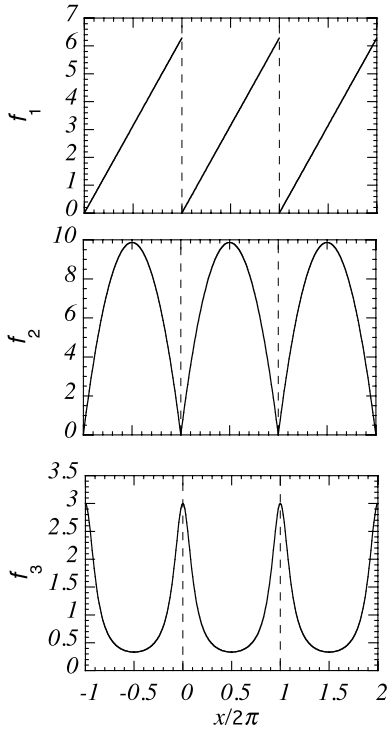

Fig. 1.1 The three functions of (1.32) whose periodic extensions have different degrees of smoothness

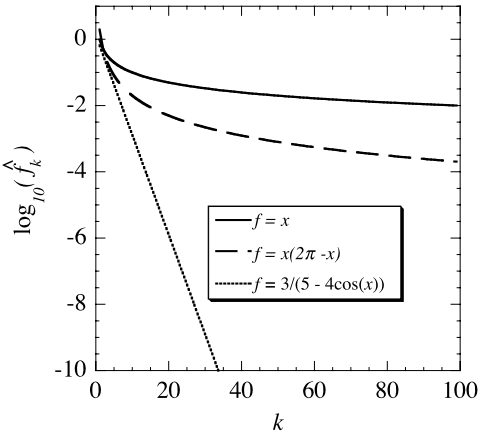

Fig. 1.2 Fourier coefficients of the three functions plotted in Fig. 1.1

Thus, if the coefficients decay with polynomial order p, we bound the truncation error by

$$\|f(x) - P_N f(x)\| < \frac{C}{(N/2+1)^{p-1/2}} \sim \left(\frac{N}{2}\right)^{1/2-p}, \tag{1.34}$$

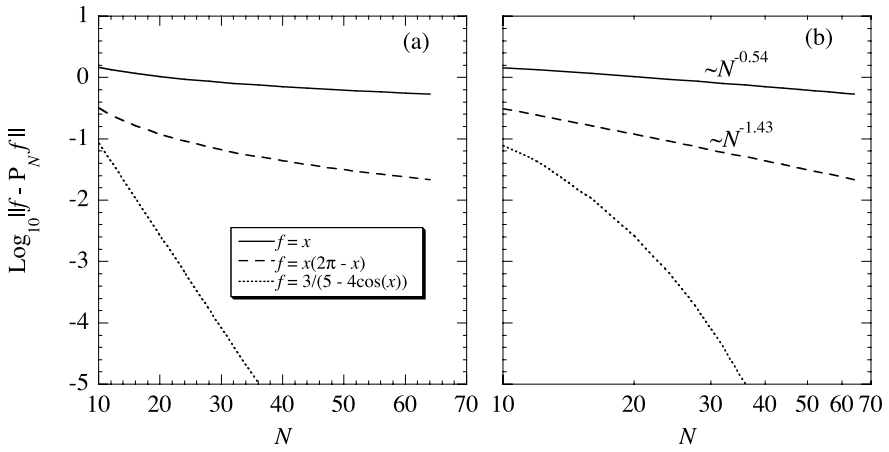

Fig. 1.3 Error in the truncation approximation for the three functions shown in Fig. 1.1. To show both exponential and polynomial order convergence, the errors are presented both as semi-log (**a**) and log-log (**b**) plots. The measured asymptotic behavior labelled in (**b**) is from a least squares fit to the lines

where C is some generic constant. If the coefficients decay exponentially fast, the truncation error also decays exponentially fast

$$\|f(x) - P_N f(x)\| \leq Ce^{-\alpha(N/2+1)} \sim Ce^{-\alpha N/2}. \tag{1.35}$$

We see these two types of behavior with the three example functions of (1.32). Figure 1.3 shows the logarithm of the truncation error, which we plot both as a function of N and $\text{Log}(N)$ to demonstrate polynomial and exponential convergence. Figure 1.3a shows that the truncation error of the smooth function, f_3 decays exponentially fast as the number of modes is increased, whereas Fig. 1.3b shows that the non-smooth functions converge with one-half less polynomial order than the order of the coefficients.

The convergence behavior described by equations (1.34) and (1.35) is known as *spectral accuracy*. In both cases the truncation error decays at a rate that depends on the rate of decay of the Fourier coefficients, which in turn decay at a rate that depends on the smoothness of the function. If the periodic extension of the function is not smooth, say if $p = 1$, then the approximation converges very slowly as $N^{-1/2}$, which is *polynomial order accuracy*. On the other hand, if the periodic extension of the function is infinitely smooth, i.e. $p \to \infty$, then spectral accuracy means that the error decays at a rate faster than any fixed power in $1/N$. In this case, we say that the approximation has *infinite order accuracy.* For very smooth (analytic) functions, those for which the coefficients decay exponentially fast, spectral accuracy means exponential convergence of the truncation error in the number of degrees of freedom.

Finally, we show that $P_N f$ is the best approximation to f on $[0, 2\pi]$ in the least squares sense. That is, of all Fourier polynomials of the form

$$S_N(x) = \sum_{k=-N/2}^{N/2} \hat{s}_k e^{ikx} \tag{1.36}$$

the one for which $\|f(x) - S_N\|$ is the smallest is $S_N = P_N f$. This is a consequence of the orthogonality of the basis functions, for

$$\begin{aligned} \|f(x) - S_N\|^2 &= \left\| \sum_{k=-N/2}^{N/2} (\hat{f}_k - \hat{s}_k) e^{ikx} - \sum_{|k|=N/2+1}^{\infty} \hat{f}_k e^{ikx} \right\|^2 \\ &= 2\pi \left\{ \sum_{k=-N/2}^{N/2} |\hat{f}_k - \hat{s}_k|^2 + \sum_{|k|=N/2+1}^{\infty} |\hat{f}_k|^2 \right\}. \end{aligned} \tag{1.37}$$

Since all terms in the sums are non-negative, the minimum occurs when $\hat{s}_k = \hat{f}_k$ for $|k| \leq N/2$.

1.4 Modal vs. Nodal Approximation

Equation (1.10) is an example of a *modal* approximation. It is called modal because the fundamental unknowns are the coefficients of the orthogonal expansion functions, or modes. An alternative is to represent the solution in terms of grid point values by way of an interpolant. The interpolation view is akin to that taken in finite difference approximations where the solution unknowns are the values at specific points in space. In finite difference approximations, derivatives are approximated at a grid point by the derivatives of a polynomial that interpolates the solution through the point and close neighbors. Just as we can represent a power series (a modal form of a polynomial) as a Lagrange or Newton form interpolant, we can also represent an orthogonal polynomial approximation as an interpolant through a set of grid point or *node* values. Such approximations are called *nodal*. Nodal approximations are intimately tied to the introduction of a quadrature rule to retain the orthogonality properties that we used to find the modal coefficients.

1.5 Discrete Orthogonality and Quadrature

As we have seen, the Fourier truncation operator requires us to evaluate integrals to compute the orthogonal projections. We know from experience that it is not necessarily an easy task to evaluate integrals analytically. For that reason, integrals are

often approximated by quadrature. The goal, now, is to find an appropriate quadrature that will retain the essential properties of spectral methods, namely orthogonality and spectral accuracy. With an appropriate quadrature, we can still compute the coefficients in an expansion by an orthogonal projection, much like we did in (1.16).

A quadrature rule is a formula to approximate the integral of a function such as

$$Q[f] = \sum_{j=0}^{N} f(x_j) w_j = \int_a^b f(x)dx + E. \tag{1.38}$$

Here, the set $\{x_j\}_{j=0}^N$ are known as the nodes, abscissas, or knots, the $\{w_j\}_{j=0}^N$ are the quadrature weights, and E is the error.

In this section we seek the quadrature rule under which as many orthogonal basis functions as possible remain orthogonal. In other words, we seek abscissas and weights so that for the complex exponentials

$$\sum_{j=0}^{N} e^{inx_j} e^{-imx_j} w_j = \int_0^{2\pi} e^{inx} e^{-imx} dx = \left(e^{inx}, e^{imx}\right) = 2\pi\delta_{n,m} \tag{1.39}$$

for the largest range of n and m. The result, as we will show in the following, turns out to be the well-known composite trapezoidal rule.

To start, let $k = n - m$ to change the problem to one of finding the nodes and weights so that

$$\sum_{j=0}^{N} e^{ikx_j} w_j = 2\pi\delta_{k,0} \tag{1.40}$$

over the largest range of k. The crucial result that we need is that

$$\sum_{j=0}^{N-1} e^{ik(2\pi j/N)} = N\delta_{k,\pm pN}, \quad p = 0, 1, \ldots. \tag{1.41}$$

Clearly (1.41) is true for $k = 0$ and for k being any multiple of N. Otherwise, for $k \neq \pm pN$, let's call $\xi = e^{2\pi ik/N}$ to convert the sum into a recognizable form

$$\sum_{j=0}^{N-1} e^{2\pi ijk/N} = \sum_{j=0}^{N-1} \xi^j = \frac{1-\xi^N}{1-\xi} = \frac{1-e^{2\pi ik}}{1-e^{2\pi ik/N}} = 0. \tag{1.42}$$

Equation (1.41) suggests that we should choose $x_j = 2\pi j/N$ as the quadrature nodes. Next, $e^{ix_0} = e^{ix_N}$, so we can combine the endpoints and remove the $j = N$ term from the sum. Finally, to match the normalization in (1.39), we choose $w_j = 2\pi/N$, so that by (1.41)

$$\frac{2\pi}{N} \sum_{j=0}^{N-1} e^{ikx_j} = 2\pi\delta_{k,\pm pN}. \tag{1.43}$$

This matches the integral when $p = 0$ so

$$\frac{2\pi}{N}\sum_{j=0}^{N-1} e^{ikx_j} = \int_0^{2\pi} e^{-ikx}dx = 2\pi\delta_{k,0}, \quad |k| = 0, 1, \ldots, N-1. \tag{1.44}$$

From this exercise let us define the Fourier quadrature rule

$$Q_F[f] = \frac{2\pi}{N}\sum_{j=0}^{N-1} f(x_j), \quad x_j = 2j\pi/N. \tag{1.45}$$

Equation (1.45) is nothing but the composite trapezoidal rule applied to the special case of a 2π-periodic integrand.

We use the Fourier quadrature to define a *discrete inner product*. When we replace $k = n - m$, (1.44) becomes

$$\frac{2\pi}{N}\sum_{j=0}^{N-1} e^{inx_j}e^{-imx_j} = \int_0^{2\pi} e^{inx}e^{-imx}dx, \quad |n-m| = 0, 1, \ldots, N-1, \tag{1.46}$$

so let us define

$$(u, v)_N = \frac{2\pi}{N}\sum_{j=0}^{N-1} u(x_j)v^*(x_j) \tag{1.47}$$

where $x_j = 2\pi j/N$. We have shown, therefore, the discrete orthogonality result

$$\left(e^{inx}, e^{imx}\right)_N = \left(e^{inx}, e^{imx}\right) = 2\pi\delta_{n,m}, \quad |n-m| = 0, 1, \ldots, N-1. \tag{1.48}$$

Not only that, the discrete norm of the basis functions matches the continuous norm,

$$\left\|e^{inx}\right\|_N^2 = \left(e^{inx}, e^{inx}\right)_N = 2\pi. \tag{1.49}$$

We now make two general comments. First, (1.44) shows that the composite trapezoidal rule exactly integrates the complex exponentials e^{ikx} for $k = 0, \pm 1, \pm 2, \ldots, \pm(N-1)$. Generally speaking, we would expect the composite trapezoidal rule to be exact only for piecewise linear integrands. However, we see that the quadrature is also exact when the integrand is a complex exponential of suitable degree.

Next, it is important to contrast the continuous and discrete orthogonality properties of the complex exponential basis functions. In the continuous case, the inner product of two complex exponentials e^{inx} and e^{imx} is nonzero only if $n = m$, i.e. $n - m = 0$. The discrete inner product, on the other hand, is nonzero if $n - m = \pm pN$, that is, if they differ by a multiple of N. The extra nonzero discrete inner products are a manifestation of aliasing errors, an important subject that we will discuss in the next section on Fourier polynomial interpolation.

1.6 Fourier Interpolation

The function

$$I_N f = \sum_{k=-N/2}^{N/2} \frac{\tilde{f}_k}{\bar{c}_k} e^{ikx} \tag{1.50}$$

where N is even and

$$\bar{c}_k = \begin{cases} 1, & k = -\frac{N}{2}+1, \ldots, \frac{N}{2}-1, \\ 2, & k = \pm\frac{N}{2} \end{cases} \tag{1.51}$$

is the *Fourier Interpolant* of a real function f if we compute the coefficients $\tilde{f}_k$ so that

$$I_N f\,(x_n) = f\,(x_n)\,, \quad n = 0, 1, \ldots, N-1, \tag{1.52}$$

for $x_n = 2\pi n/N$. (Since the complex exponentials are periodic with period 2π, the last point with $j = N$ is redundant. The interpolant is 2π-periodic because $I_N f(0) = I_N f(2\pi)$.) Note that we have written the expansion coefficients, which are called the *discrete coefficients*, as $\tilde{f}_k$ to distinguish them from the coefficients of the truncation approximation, $\hat{f}_k$, that we discussed in Sect. 1.3. Later in this section we will show how they are related.

In parallel to how we computed the Fourier transform coefficients, we compute the discrete Fourier coefficients using the discrete inner product

$$\tilde{f}_k = \frac{1}{N} \sum_{j=0}^{N-1} f_j e^{-ikx_j} = \frac{1}{2\pi} \big(f, e^{ikx}\big)_N, \quad k = -N/2, \ldots, N/2. \tag{1.53}$$

To show that $\tilde{f}_k$'s in (1.53) are indeed the coefficients of the interpolant, let us substitute them into (1.50) and swap the order of the summations

$$\begin{aligned} I_N f &= \sum_{k=-N/2}^{N/2} \left\{ \frac{1}{\bar{c}_k N} \sum_{j=0}^{N-1} f_j e^{-ikx_j} \right\} e^{ikx} \\ &= \sum_{j=0}^{N-1} f_j \left\{ \frac{1}{N} \sum_{k=-N/2}^{N/2} \frac{1}{\bar{c}_k} e^{ik(x-x_j)} \right\}. \end{aligned} \tag{1.54}$$

Let us next define the functions

$$h_j(x) = \frac{1}{N} \sum_{k=-N/2}^{N/2} \frac{1}{\bar{c}_k} e^{ik(x-x_j)}, \quad j = 0, 1, \ldots, N-1. \tag{1.55}$$

Then we can write the interpolant (1.50) in the equivalent *Lagrange form*

$$I_N f = \sum_{j=0}^{N-1} f_j h_j(x). \tag{1.56}$$

To show that $I_N f$ satisfies (1.52), we evaluate it at x_n

$$I_N f\,(x_n) = \sum_{j=0}^{N-1} f_j h_j\,(x_n), \quad n = 0, 1, \ldots, N-1. \tag{1.57}$$

So to finish showing that we have the right discrete Fourier coefficients to define the Fourier interpolant, we need only to show that

$$h_j\,(x_n) = \frac{1}{N} \sum_{k=-N/2}^{N/2} \frac{1}{\bar{c}_k} e^{2\pi i(n-j)k/N} = \delta_{j,n} \tag{1.58}$$

for $|n-j| < N$. First we note that

$$\begin{aligned} e^{2\pi i(n-j)(\frac{N}{2})/N} &= e^{i\pi(n-j)} = (-1)^{n-j}, \\ e^{2\pi i(n-j)(-\frac{N}{2})/N} &= e^{-i\pi(n-j)} = (-1)^{n-j}. \end{aligned} \tag{1.59}$$

Therefore, we can fold the $N/2$ mode in with the $-N/2$ mode

$$\frac{1}{N} \sum_{k=-N/2}^{N/2} \frac{1}{\bar{c}_k} e^{2\pi i(n-j)k/N} = \frac{1}{N} \sum_{k=-N/2}^{N/2-1} e^{2\pi i(n-j)k/N}. \tag{1.60}$$

To get the result we need, we will manipulate the second sum to put it into the form of (1.41). Let us replace $k \leftarrow k + N/2$

$$\begin{aligned} \frac{1}{N} \sum_{k=-N/2}^{N/2-1} e^{2\pi i(n-j)k/N} &= \frac{1}{N} \sum_{k=0}^{N-1} e^{2\pi i(n-j)(k+N/2)/N} \\ &= \frac{1}{N} \sum_{k=0}^{N-1} e^{2\pi i(n-j)k/N} e^{i\pi(n-j)} \\ &= \frac{(-1)^{n-j}}{N} \sum_{k=0}^{N-1} e^{2\pi i(n-j)k/N}. \end{aligned} \tag{1.61}$$

With a change of indices, we recognize the last sum to be (1.41). That is,

$$\sum_{k=0}^{N-1} e^{2\pi i(n-j)k/N} = N\delta_{n-j,\pm pN}, \quad p = 0, 1, \ldots. \tag{1.62}$$

Since $|n-j| < N$ and $(-1)^0 = 1$, we get (1.58).

The discrete coefficients have properties not shared with the true Fourier coefficients. First, the discrete Fourier coefficients are N-periodic, which means that

$$\begin{aligned}\tilde{f}_{k\pm N} &= \frac{1}{N}\sum_{j=0}^{N-1} f_j e^{-i(k\pm N)x_j} = \frac{1}{N}\sum_{j=0}^{N-1} f_j e^{-ikx_j} e^{\mp iN(2\pi j/N)} \\ &= \frac{1}{N}\sum_{j=0}^{N-1} f_j e^{-ikx_j} = \tilde{f}_k. \end{aligned} \tag{1.63}$$

Therefore, $\tilde{f}_{-N/2} = \tilde{f}_{N/2}$. Furthermore, when f is real, the coefficient of the highest mode is real. The $\mathrm{Im}(\tilde{f}_{-N/2}) = 0$, for

$$\tilde{f}_{-N/2} = \frac{1}{N}\sum_{j=0}^{N-1} f_j e^{-i\frac{N}{2}x_j} = \frac{1}{N}\sum_{j=0}^{N-1} f_j e^{-i\pi j} = \frac{1}{N}\sum_{j=0}^{N-1} f_j (-1)^j. \tag{1.64}$$

We derive two useful results about Fourier interpolants. The first follows immediately from (1.52), namely

$$\left(I_N f, e^{ikx}\right)_N = \left(f, e^{ikx}\right)_N. \tag{1.65}$$

The second is that the discrete and continuous inner products of two Fourier interpolants are equal. Suppose that U and V are two Fourier interpolants of degree less than or equal to N

$$U = \sum_{n=-N/2}^{N/2} \frac{\tilde{a}_n}{\bar{c}_n} e^{inx}, \qquad V = \sum_{m=-N/2}^{N/2} \frac{\tilde{b}_n}{\bar{c}_m} e^{imx}. \tag{1.66}$$

Now,

$$\begin{aligned} U(x_j) &= \sum_{n=-N/2}^{N/2-1} \frac{\tilde{a}_n}{\bar{c}_n} e^{inx_j} + \frac{\tilde{a}_{N/2}}{\bar{c}_{N/2}} e^{iN/2x_j} \\ &= \sum_{n=-N/2}^{N/2-1} \frac{\tilde{a}_n}{\bar{c}_n} e^{inx_j} + \frac{\tilde{a}_{-N/2}}{\bar{c}_{-N/2}} e^{-iN/2x_j} = \sum_{n=-N/2}^{N/2-1} \tilde{a}_n e^{inx_j}, \end{aligned} \tag{1.67}$$

and similarly for V. Therefore the discrete inner product of the two is

$$\begin{aligned}(U,V)_N &= \sum_{n=-N/2}^{N/2-1}\sum_{m=-N/2}^{N/2-1} a_n b_n \left(e^{inx}, e^{imx}\right)_N \\ &= \sum_{n=-N/2}^{N/2-1}\sum_{m=-N/2}^{N/2-1} a_n b_n \left(e^{inx}, e^{imx}\right) = (U,V). \end{aligned} \tag{1.68}$$

The equivalence of the discrete and exact inner products of two Fourier polynomials of degree $\leq N$ will be useful later to prove stability of Fourier spectral approximations.

Lastly, when we evaluate the Fourier interpolant at the nodes and use (1.67), we get the well known *Discrete Fourier Transform Pair*

$$\text{DFT} \quad \begin{cases} \tilde{f}_k = \frac{1}{N} \sum_{j=0}^{N-1} f(x_j) e^{-2\pi i j k/N}, & k = -N/2, \ldots, N/2 - 1, \\ f(x_j) = \sum_{k=-N/2}^{N/2-1} \tilde{f}_k e^{2\pi i j k/N}, & j = 0, 1, \ldots, N-1. \end{cases} \tag{1.69}$$

In Chap. 2, we will show how to compute the DFT rapidly using a Fast Fourier Transform.

1.6.1 Direct Computation of the Fourier Interpolation

We see from equations (1.50) and (1.53) that it is a two step process to compute the interpolant from the Fourier modes. First, we compute and store the coefficients. From the coefficients, we construct the interpolant. Our first two algorithms: Algorithms 1 (DiscreteFourierCoefficients) and 2 (FourierInterpolantFromModes) compute these two steps directly. (For an explanation of the pseudocode, please see Appendix A.) Algorithm 1 computes the discrete Fourier coefficients by a direct sum. When we count the number of operations in the loops, we see that the direct sum takes order N^2 complex multiplications. In general, we would do this only for small values of N. For larger values we will use a Fast Fourier Transform, which we will present in Chap. 2. Algorithm 2 performs the second step; It evaluates the interpolant from the discrete Fourier coefficients.

We have also shown that we can write the interpolant $I_N f(x)$ in an equivalent Lagrange form, (1.56). The Lagrange form is useful for nodal approximations where

Algorithm 1: *DiscreteFourierCoefficients:* Direct Evaluation of the Discrete Fourier Coefficients

```
Procedure DiscreteFourierCoefficients
Input: {f_j}_{j=0}^{N-1}
for k = -N/2 to N/2 do
    s ← 0
    for j = 0 to N − 1 do
        s ← s + f_j ∗ e^{−2πijk/N}
    end
    f̃_k = s/N
end
return {f̃_k}_{k=−N/2}^{N/2}
End Procedure DiscreteFourierCoefficients
```

Algorithm 2: *FourierInterpolantFromModes:* Direct Evaluation of the Fourier Interpolant from Its Modes

Procedure FourierInterpolantFromModes
Input: x, $\left\{\tilde{f}_k\right\}_{k=-N/2}^{N/2}$
$s \leftarrow \left(\tilde{f}_{-N/2} * e^{-iNx/2} + \tilde{f}_{N/2} * e^{iNx/2}\right)/2$
for $k = -N/2+1$ **to** $N/2-1$ **do**
  $s \leftarrow s + \tilde{f}_k * e^{ikx}$
end
return $\mathrm{Re}(s)$
End Procedure FourierInterpolantFromModes

Algorithm 3: *FourierInterpolantFromNodes:* Direct Evaluation of the Fourier Interpolant from Its Nodes

Procedure FourierInterpolantFromNodes
Input: x,$\{x_j\}_{j=0}^{N-1}$, $\{f_j\}_{j=0}^{N-1}$
Uses Algorithms:
  Algorithm 139 (AlmostEqual)
for $j = 0$ **to** $N-1$ **do**
  if *AlmostEqual*(x, x_j) **then return** f_j
end
$s \leftarrow 0$
for $j = 0$ **to** $N-1$ **do**
  $t \leftarrow (x - x_j)/2$
  $s \leftarrow s + f_j * \sin(N * t) * \cot(t)/N$
end
$I_N f(x) \leftarrow s$
return $I_N f(x)$
End Procedure FourierInterpolantFromNodes

the fundamental unknowns are the values of the function at the nodes. It turns out that the function $h_j(x)$ has a closed form

$$h_j(x) = \frac{1}{N} \sin\left[\frac{N}{2}(x - x_j)\right] \cot\left[\frac{1}{2}(x - x_j)\right] \tag{1.70}$$

when $x_j = 2\pi j/N$.

Computation of the Lagrange form of the interpolation is a one step procedure. We have to be careful, however, because $h_j(x_j)$ is not defined numerically. Algorithm 3 (FourierInterpolantFromNodes) presents a procedure to calculate the Fourier interpolant in Lagrange form. To avoid the cot(0), it first checks to see if the evaluation point is a node. The test uses the function *AlmostEqual* of Algorithm 139, which we present in Appendix B. If the evaluation point is not a node, then the procedure evaluates the interpolant by (1.56).

1.6.2 Error of the Fourier Interpolation

As there is no free lunch, the easily computed Fourier interpolation is generally not as accurate as series truncation. In Sect. 1.3, we showed that the truncation operator P_N gives the best Fourier approximation in the least squares sense. It follows, then, that the interpolation error must be equal or larger.

The interpolation error is larger because the Fourier interpolation coefficients are not equal to the Fourier coefficients for $k = -N/2, \ldots, N/2$. To find the errors in the coefficients, we replace f in (1.53) by its Fourier series

$$\begin{aligned} \tilde{f}_k &= \frac{1}{N} \sum_{j=0}^{N-1} \left[\sum_{m=-\infty}^{\infty} \hat{f}_m e^{imx_j} \right] e^{-ikx_j} \\ &= \sum_{m=-\infty}^{\infty} \hat{f}_m \left[\frac{1}{N} \sum_{j=0}^{N-1} e^{i(m-k)x_j} \right]. \end{aligned} \tag{1.71}$$

Since $x_j = 2\pi j/N$, (1.43) says that the coefficient of $\hat{f}_m$ in the last sum is $\delta_{m-k,\pm pN}$. Thus, the only nonzero terms in the double sum are those for which $m = k \pm pN$, and the relation between the discrete and exact coefficients reduces to

$$\tilde{f}_k = \sum_{p=-\infty}^{\infty} \hat{f}_{k+pN} = \hat{f}_k + \sum_{\substack{p=-\infty \\ p\neq 0}}^{\infty} \hat{f}_{k+pN}. \tag{1.72}$$

The difference between the discrete and exact Fourier coefficients is called the *aliasing error*, and is a direct consequence of discrete sampling.

To see the effect of aliasing on a concrete example, let's interpolate the function $f(x) = e^{ikx}$ with $N = 8$ points for three values of k. Figure 1.4 shows the interpolations for $k = -8, -6$, and -4. The first column in the figure shows the magnitude of the exact Fourier coefficients, $\hat{f}_k$. The second column shows the discrete Fourier coefficients computed by (1.69). The interpolant can represent values of k only from -4 to 3, so it cannot represent wavenumbers $k = -8$ and -6. We see, therefore, that the DFT shifts the discrete coefficients by $N = 8$ for those values of k. The column on the far right of the figure shows the exact function, the interpolant, and the value of the interpolant evaluated at the points x_j, that is, the values computed by the inverse DFT in (1.69). We see that the interpolant and the exact solution differ significantly for $k = -8$ and -6, although they do match at the nodes as required. In fact, the interpolant for $k = -8$ looks like the constant, $k = 0$ function.

Once the wavenumber can be represented by the interpolant, there is no error. The bottom row of Fig. 1.4 shows the coefficients and interpolant for $k = -4$. That mode is included by the interpolant, so we see that the exact and discrete coefficients match, as do the function and its interpolant. The same will hold true for all values of k from -4 to 3. When k increases beyond 4, the function will be aliased again; for those, we will see the interpolation appear as the $k - 8$ mode.

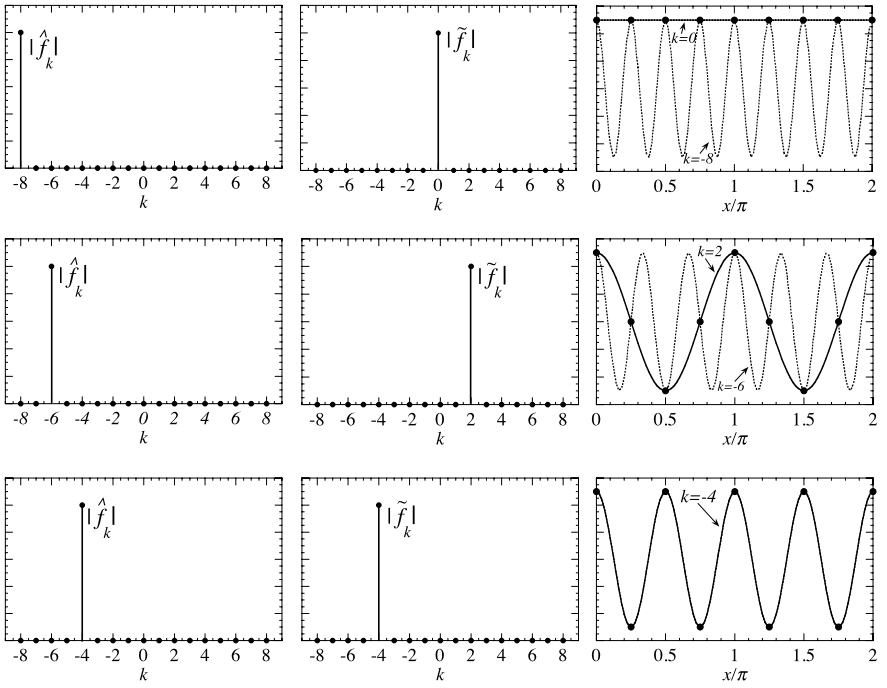

Fig. 1.4 Plot of the exact Fourier coefficients (*left column*), interpolation coefficients (*center column*) and interpolant (*right column*) for $f = \exp(ikx)$ and $N = 8$. In the *right column*, the *solid line* is the interpolant, the *dashed line* is the exact function, and the circles are the values of the interpolant at the grid points. The *top row* shows how the interpolant for $k = -8$ appears on the grid as the $k + N = -8 + 8 = 0$ mode. The *center row* shows the aliasing effects for $k = -6$. The *bottom row* shows that $k = -N/2$ is represented exactly by the interpolant and there is no aliasing error. It also corresponds to the highest wavenumber in magnitude/shortest wavelength for which this is true

In summary, when we view a sinusoid with wavenumber $k \pm N$ at the points $x_j = 2\pi j/N$, it looks exactly like a sinusoid with wavenumber k. It is the reason why the wagon wheels sometimes (appear to) go backwards in the old movie westerns even as the wagon goes forward.

We can compute the number of points per wavelength that we need to interpolate a sinusoid exactly. The length of the interval is 2π, so if the wavelength of a wave is λ, the number of wavelengths in the interval is $2\pi/\lambda$. Since the wavelength of a wave with wavenumber k is $\lambda = 2\pi/k$, the number of wavelengths on the interval $[0, 2\pi]$ is k. (Count them on Fig. 1.4.) Since N points are used to interpolate the interval, the number of points per wavelength is therefore N/k. From the example shown in Fig. 1.4, the wavenumber with $|k| = N/2$ is the largest that the grid can represent exactly. Thus, the minimum number of points per wavelength to represent a sinusoid exactly by Fourier interpolation is $N/(N/2) = 2$.

Although interpolation is less accurate than truncation, except when the original function is made up of modes with wavenumbers less than $N/2$, the difference is

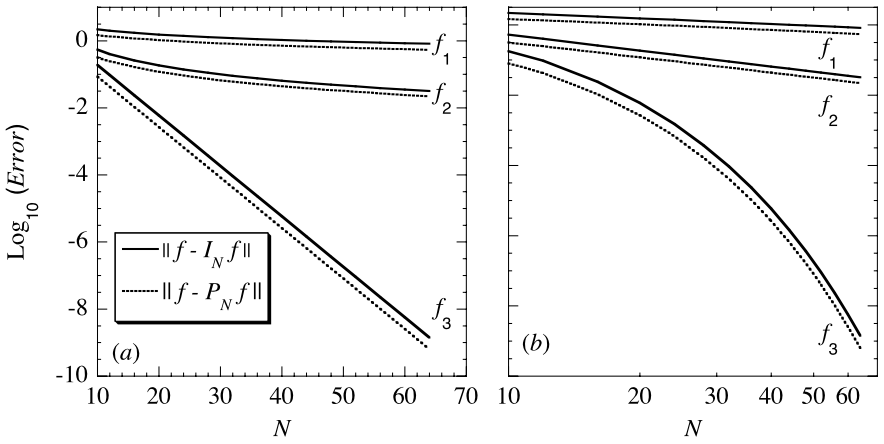

Fig. 1.5 Error in the interpolation and truncation approximation for the three functions shown in Fig. 1.1. To show both exponential and polynomial order convergence, the errors are presented both as semi-log (**a**) and log-log (**b**) plots

often not significant for smooth enough functions. The reason is that the aliasing error, like the truncation error, depends on the rate of decay of the exact Fourier coefficients for $|k| \geq N/2$. When the coefficients decay rapidly, which occurs when the function is very smooth, the aliasing error is dominated by the coefficient $\hat{f}_{N/2}$, same as the truncation error. In fact, it can be shown that the aliasing error is on the order of the truncation error, and that the interpolation error is at most twice the truncation error. This factor of two is insignificant for well-resolved approximations, that is, when the error is small in relation to the solution. Details can be found in Sect. 5.1 of [7].

To get a sense of the contribution of aliasing to the interpolation error, let us return to the three functions we showed in Fig. 1.1. We saw the truncation error for these three functions as a function of N in Fig. 1.3. In Fig. 1.5 we compare the convergence of the interpolation error to the truncation error in both semi-log (a) and log-log (b) formats. We see that the interpolation error is larger in each case, but decays at essentially the same rate as the truncation error. Furthermore, the computed size of the aliasing error is less than or approximately equal to the truncation error. We'll cover effects of aliasing errors on the solutions of PDEs later in Sect. 4.1.

1.7 The Derivative of the Fourier Interpolant

We approximate derivatives by analytically differentiating the Fourier interpolant and evaluating the result at the nodes

$$f'(x_j) \approx \mathscr{D} f_j = (I_N f)'(x_j) = \sum_{k=-N/2}^{N/2} \frac{ik\tilde{f}_k}{\tilde{c}_k} e^{2\pi ijk/N}. \tag{1.73}$$

We will show in the next chapter how to compute this sum, and hence the derivative approximation, with the Fast Fourier Transform.

Equivalently, we can differentiate the Lagrange form of the interpolant

$$\mathscr{D} f_n = (I_N f)'(x_n) = \sum_{j=0}^{N-1} f(x_j) h_j'(x_n) \tag{1.74}$$

where

$$h_j'(x_n) = \begin{cases} \frac{1}{2}(-1)^{n+j} \cot[\frac{(j-n)\pi}{N}], & j \neq n, \\ 0, & j = n. \end{cases} \tag{1.75}$$

This means that instead of using the DFT and the modal form of the interpolant, we can also evaluate the derivative at the nodes by a matrix-vector multiplication, where the matrix elements at the positions (n, j) are the values $h_j'(x_n)$.

The presence of aliasing error means that, unlike truncation and differentiation, interpolation and differentiation do not commute. If we call the difference between the truncation and interpolation polynomials $I_N f - P_N f = R_N f$, then derivative of the interpolant is

$$(I_N f)' = (P_N f)' + (R_N f)' = P_N(f') + (R_N f)', \tag{1.76}$$

whereas the interpolant of the derivative is

$$(I_N f') = (P_N f') + (R_N f'). \tag{1.77}$$

We've already shown that differentiation and truncation commute. To show that interpolation and differentiation do not commute we show that $(R_N f)' \neq R_N(f')$. This is easy, because

$$\begin{aligned} (R_N f)' = & \sum_{k=-N/2}^{N/2} \sum_{p \neq 0} \frac{ik}{\bar{c}_k} \hat{f}_{k+pN} e^{ikx} \\ & - \frac{1}{2} \left(\frac{iN}{2} \hat{f}_{N/2} e^{iNx/2} - \frac{iN}{2} \hat{f}_{-N/2} e^{-iNx/2} \right) \end{aligned} \tag{1.78}$$

whereas

$$\begin{aligned} R_N(f') = & \sum_{k=-N/2}^{N/2} \sum_{p \neq 0} \frac{i(k+pN)}{\bar{c}_k} \hat{f}_{k+pN} e^{ikx} \\ & - \frac{1}{2} \left(\frac{iN}{2} \hat{f}_{N/2} e^{iNx/2} - \frac{iN}{2} \hat{f}_{-N/2} e^{-iNx/2} \right) \end{aligned} \tag{1.79}$$

so the coefficients of $\hat{f}_{k+pN}$, differ between the two.

1.8 Polynomial Basis Functions

We use orthogonal polynomial series to approximate the solutions in nonperiodic problems. As we pointed out in the preamble, expansions in orthogonal polynomials such as Legendre or Bessel functions are useful to solve some types of boundary value problems analytically. Now we study polynomials that we can use to develop spectral approximations to PDEs.

The starting point for polynomial spectral methods is to construct an orthogonal basis for square integrable functions, specifically $L^2_w(a,b)$, in which to expand the functions that we want to approximate. One convenient way to generate these bases is to use the Sturm-Liouville theorem, which concerns the eigenfunctions, u, of the eigenvalue problem known as the Sturm-Liouville problem. Although it is not absolutely necessary to know the details of the Sturm-Liouville theorem to be able to use spectral methods, it is nevertheless helpful to know why certain choices of the expansion functions are more useful than others. For this reason, we describe the main results that indicate why Chebyshev and Legendre bases are useful to approximate solutions of differential equations.

The Sturm-Liouville problem is a second order boundary-value problem of the form

$$-\frac{d}{dx}\left(p(x)\frac{du}{dx}\right)+q(x)u=\lambda w(x)u, \quad a<x<b$$
$$+\text{ boundary conditions on } u. \tag{1.80}$$

If $p(a)=p(b)=0$ the problem is called *singular*, and regular otherwise. The Sturm-Liouville theorem tells us that the eigenvalues, λ, are real and that the eigenfunctions corresponding to distinct eigenvalues are orthogonal. With suitable boundary conditions for the Sturm-Liouville problem, the eigenvalues are countably infinite, i.e., they form a set $\{\lambda_n\}_{n=0}^{\infty}$. Most importantly, the theorem tells us that the set of eigenfunctions associated with these eigenvalues, $\{u_n\}_{n=0}^{\infty}$, form a basis for $L^2_w(a,b)$. Thus, we can use these eigenfunctions to represent functions that are square integrable with respect to the weight w.

To use a truncated series as an approximation to a function, it is important that the series coefficients decay rapidly. It turns out that they will converge at a rate dependent only on the smoothness of the function being expanded, without the need for special conditions (e.g. periodicity) on the function at the boundaries, if the Sturm-Liouville problem is singular. For details, we recommend consulting reference [7].

Finally, we are interested in the Sturm-Liouville problems for which the eigenfunctions are polynomials. If we scale the interval $[a,b]$ to the reference interval $[-1,1]$, there turns out to be only one Sturm-Liouville problem whose eigenfunctions satisfy these constraints. Those eigenfunctions satisfy

$$-\frac{d}{dx}\left((1-x)^{1+\alpha}(1+x)^{1+\beta}\frac{du}{dx}\right)=\lambda(1-x)^{\alpha}(1+x)^{\beta}u, \quad -1<x<1 \tag{1.81}$$

where $\alpha, \beta > -1$. The eigenfunctions are called the *Jacobi polynomials*, which are represented by $P_k^{(\alpha,\beta)}(x)$.

The Jacobi polynomials have computationally useful properties. The first is that they satisfy a three-term recursion relation, making them easy to evaluate.

A second useful property concerns their roots. For $k \geq 1$, $P_k^{(\alpha,\beta)}$ has k distinct real roots, all of which lie in the interval $(-1, 1)$. This property will be useful later when we develop high-precision Gauss quadratures to use for discrete inner products. Finally, the derivatives of the polynomials satisfy a three term recursion relation, which we will use later to compute the coefficients of the derivatives of functions.

The two important special cases of interest to spectral solutions of PDEs are the *Legendre Polynomials*, $L_k(x) = P_k^{(0,0)}(x)$ and the *Chebyshev Polynomials*, $T_k(x) = P_k^{(-1/2,-1/2)}(x)$. The Legendre polynomials are of interest because with $\alpha = \beta = 0$, the weight function $w(x) = 1$. The unit weight function makes integrals such as (1.11) and (1.13) easier to evaluate analytically. Chebyshev polynomials have well-known nice approximation properties and have the practical feature that we can evaluate Chebyshev series using the Fast Fourier Transform.

1.8.1 The Legendre Polynomials

The Legendre polynomials are the special case of the Jacobi polynomials when $\alpha = \beta = 0$. They satisfy the three term recursion

$$L_{k+1}(x) = \frac{2k+1}{k+1} x L_k(x) - \frac{k}{k+1} L_{k-1}(x) \tag{1.82}$$

with $L_0(x) = 1$ and $L_1(x) = x$. We show a plot of the Legendre polynomials of degree two through five in Fig. 1.6, which illustrates the fact that $L_k(\pm 1) = (\pm 1)^k$. The derivatives satisfy

$$(2k+1)\, L_k(x) = L'_{k+1}(x) - L'_{k-1}(x). \tag{1.83}$$

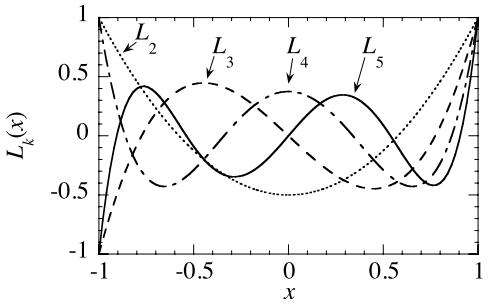

Fig. 1.6 The Legendre polynomials of degree two through five

The Legendre polynomials are normalized so that $L_k(1) = 1$. Their L^2 norms are

$$\|L_k\|^2 = \frac{2}{2k+1}. \tag{1.84}$$

1.8.2 The Chebyshev Polynomials

Chebyshev polynomials are the special case of Jacobi polynomials that we get when $\alpha = \beta = -1/2$. They are useful because of their approximation properties and because Chebyshev series can be computed efficiently by way of a Fast Fourier Transform. The relation to the Fourier transform is a consequence of the fact that the Chebyshev polynomials have an alternative representation, namely

$$T_k(x) = \cos\bigl(k \cos^{-1}(x)\bigr). \tag{1.85}$$

The representation of the Chebyshev polynomials by (1.85) gives us one way to evaluate them. Alternatively, there is still the three term recursion,

$$T_{k+1}(x) = 2xT_k(x) - T_{k-1}(x). \tag{1.86}$$

To start the sequence, (1.85) shows that $T_0 = 1$ and $T_1 = x$. We show a plot of the Chebyshev polynomials of degree two through five in Fig. 1.7.

The derivatives also satisfy a three term recursion,

$$2T_k(x) = \frac{T'_{k+1}}{k+1} - \frac{T'_{k-1}}{k-1}. \tag{1.87}$$

Finally, the L^2 norms are

$$\|T_k\|_w^2 = \int_{-1}^{1} \frac{T_k^2 dx}{\sqrt{1-x^2}} = c_k \frac{\pi}{2} \tag{1.88}$$

where

$$c_k = \begin{cases} 2, & k = 0, \\ 1, & k \geq 1. \end{cases} \tag{1.89}$$

It is possible to construct alternative polynomial bases that we could use to represent polynomials with particular properties. For example, for boundary value problems with Dirichlet boundary conditions for which the exact solution must vanish at the boundaries, it often makes sense to approximate the solution with a polynomial that also vanishes at the boundaries. This can be done either by choosing the coefficients so that the polynomial vanishes, or by choosing basis functions that vanish at the boundaries. That way any series in those basis functions also vanishes appropriately.

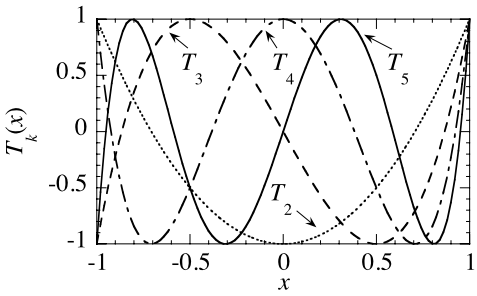

Fig. 1.7 The Chebyshev polynomials of degree two through five

1.9 Polynomial Series

Since the Jacobi polynomials form a basis for $L^2_w(-1, 1)$, as promised by the Sturm-Liouville theorem, we can represent any square integrable function, f, as an infinite series in them

$$f(x) = \sum_{k=0}^{\infty} \hat{f}_k P_k^{(\alpha,\beta)}(x). \tag{1.90}$$

Because the basis functions are orthogonal, the coefficients are

$$\hat{f}_k = \frac{(f, P_k^{(\alpha,\beta)}(x))_w}{\| P_k^{(\alpha,\beta)}(x) \|_w^2}. \tag{1.91}$$

The series that represents the derivative of a function expanded in orthogonal polynomials is more complicated than that of the Fourier series (1.20). The derivative of the Fourier series is simple because the basis functions are eigenfunctions of the derivative operator, i.e., $de^{ikx}/dx = ike^{ikx}$. The Legendre, Chebyshev and, indeed, all of the Jacobi polynomials are not. Instead, we have just seen that the derivatives of the Legendre and Chebyshev polynomials satisfy a three-term recursion relation that couples the polynomial modes.

To see how polynomial modes are coupled when we take derivatives, suppose f' is also square integrable. Then we can write its derivative as a Legendre series

$$f'(x) = \sum_{k=0}^{\infty} \hat{f}_k^{(1)} L_k(x). \tag{1.92}$$

To relate the coefficients of the derivative $\hat{f}_k^{(1)}$ to the coefficients of the function, $\hat{f}_k$, we use the recursion formula (1.83) that we rewrite as

$$L_k(x) = \frac{L'_{k+1}(x)}{2k+1} - \frac{L'_{k-1}(x)}{2k+1}. \tag{1.93}$$

When we substitute (1.93) into (1.92),

$$f'(x) = \sum_{k=0}^{\infty} \hat{f}_k^{(1)} L_k(x) = \sum_{k=0}^{\infty} \hat{f}_k^{(1)} \left[\frac{L'_{k+1}(x)}{2k+1} - \frac{L'_{k-1}(x)}{2k+1} \right] \tag{1.94}$$

or

$$f'(x) = \sum_{k=0}^{\infty} \hat{f}_k^{(1)} \frac{L'_{k+1}(x)}{2k+1} - \sum_{k=0}^{\infty} \hat{f}_k^{(1)} \frac{L'_{k-1}(x)}{2k+1}. \tag{1.95}$$

We now shift the indices each by one, note that $L'_0 = 0$, and set $L'_{-1} \equiv 0$. Finally, we gather terms to get

$$f'(x) = \sum_{k=1}^{\infty} \left[\frac{\hat{f}_{k-1}^{(1)}}{2k-1} - \frac{\hat{f}_{k+1}^{(1)}}{2k+3} \right] L'_k(x) = \sum_{k=1}^{\infty} \hat{f}_k L'_k(x). \tag{1.96}$$

Thus,

$$\hat{f}_k = \frac{\hat{f}_{k-1}^{(1)}}{2k-1} - \frac{\hat{f}_{k+1}^{(1)}}{2k+3}, \quad k \geq 1. \tag{1.97}$$

We use (1.97) to generate a recursion for the coefficients of the derivative:

$$\hat{f}_k^{(1)} = (2k+1) \left[\hat{f}_{k+1} + \frac{\hat{f}_{k+2}^{(1)}}{2k+5} \right]. \tag{1.98}$$

We use (1.98) to write (formally) the coefficient of the derivative as yet another infinite series. The recursion says that

$$\frac{\hat{f}_{k+2}^{(1)}}{2k+5} = \frac{2k+5}{2k+5} \left[\hat{f}_{k+3} + \frac{\hat{f}_{k+4}^{(1)}}{2k+9} \right] \tag{1.99}$$

so

$$\hat{f}_k^{(1)} = (2k+1) \left[\hat{f}_{k+1} + \hat{f}_{k+3} + \frac{\hat{f}_{k+4}^{(1)}}{2k+9} \right]. \tag{1.100}$$

Continuing the process leads to the infinite sum

$$\hat{f}_k^{(1)} = (2k+1) \left[\hat{f}_{k+1} + \hat{f}_{k+3} + \hat{f}_{k+5} + \cdots \right] = (2k+1) \sum_{\substack{p=k+1 \\ k+p \text{ odd}}}^{\infty} \hat{f}_p. \tag{1.101}$$

Equation (1.101) shows that the derivatives of the Legendre expansion couple the polynomial modes, whereas we have already seen that the Fourier modes do not couple.

Modes are coupled in the derivatives of the Chebyshev series in much the same way. For Chebyshev series, the recursion that relates the coefficients of the derivative to the coefficients of the original function is

$$c_k \hat{f}_k^{(1)} = \hat{f}_{k+2}^{(1)} + 2\,(k+1)\,\hat{f}_{k+1}, \quad k \geq 0. \tag{1.102}$$

Following an argument similar to that leading to (1.101) leads us to the series

$$\hat{f}_k^{(1)} = \frac{2}{c_k} \sum_{\substack{p=k+1 \\ k+p \text{ odd}}}^{\infty} p\,\hat{f}_p. \tag{1.103}$$

1.10 Polynomial Series Truncation

For non-periodic functions, we split an infinite series in orthogonal polynomials into

$$f(x) = \sum_{k=0}^{N} \hat{f}_k \phi_k(x) + \sum_{k=N+1}^{\infty} \hat{f}_k \phi_k(x) = \sum_{k=0}^{N} \hat{f}_k \phi_k(x) + \tau. \tag{1.104}$$

The action of the orthogonal projection operator, P_N, is defined to be

$$P_N f(x) = \sum_{k=0}^{N} \hat{f}_k \phi_k(x) \quad (\textit{Orthogonal Polynomial Truncation}). \tag{1.105}$$

Like the Fourier truncation operator, P_N for polynomial truncation is also the orthogonal projection operator, but this time with respect to the weighted inner product $(\cdot,\cdot)_w$. It is also the best approximation in the sense that the norm of the error, $\|f - P_N f\|_w$, is minimized. Furthermore, the norm of the truncation error, τ, is

$$\|\tau\|_{L_w^2}^2 = \sum_{k=N+1}^{\infty} |\hat{f}_k|^2 \, \|\phi_k\|_{L_w^2}^2, \tag{1.106}$$

so that the rate of convergence of the approximation $P_N f$ depends only on the rate of convergence of the coefficients $\hat{f}_k$.

What makes Jacobi polynomial expansion approximations useful as "spectral methods" is twofold. First, like the Fourier truncation, the Jacobi polynomial truncation converges spectrally fast as $N \to \infty$. The second is that it has this convergence behavior without specific restrictions (such as periodicity of the function and its derivatives) on the functions or their derivatives at the boundaries. The latter is due to the fact that the Sturm-Liouville problem for the Jacobi polynomials is singular with $p(\pm 1) = 0$. The former is due to the fact that the rate of decay of the expansion coefficients depends only on the smoothness of the function being approximated. The details for showing these facts are quite technical, and can be found, for example, in the book by Canuto et al. [7].

Although the approximation theory for polynomial truncation is rather technical, we get a good sense of the spectral convergence by looking at a couple of examples as we did with the Fourier approximations in Sect. 1.3. Let's first note that the first two example functions in (1.32) are not of much interest here since they are polynomials and are represented exactly as long as $N \geq 2$.

As examples that show the relationship between smoothness and convergence rate, let us consider Legendre polynomial truncation approximations to three functions defined on $[-1, 1]$

$$\begin{aligned} f_1(x) &= \begin{cases} -1, & -1 \leq x \leq 0, \\ +1, & 0 < x \leq 1, \end{cases} \\ f_2(x) &= |x|\,, \\ f_3(x) &= \sin(2\pi(x + 0.1)). \end{aligned} \tag{1.107}$$

The first has a jump discontinuity at the origin. The second has a slope discontinuity there. The last is infinitely smooth. We compute the coefficients of each by the orthogonal projection

$$\hat{f}_k = \frac{2k+1}{2} \int_{-1}^{1} f(x) L_k(x) dx. \tag{1.108}$$

The coefficients of the three functions are

$$\begin{aligned} \hat{f}_{1,k} &= (-1)^k \frac{(4k+3)(2k)!}{2^{2k+1}(k+1)!k!} = O\left(\frac{1}{k^{1/2}}\right), \quad k \text{ odd}, \\ \hat{f}_{2,k} &= (-1)^{k/2+1}(2k+1)\frac{k!}{2^{k+1}(k/2!)^2(k-1)(k/2-1)} = O\left(\frac{1}{k^{3/2}}\right), \quad k \text{ even}, \\ \hat{f}_{3,k} &= \frac{2k+1}{2} J_{k+1/2}(2\pi) \sin(0.2\pi + k\pi/2)\,. \end{aligned} \tag{1.109}$$

The coefficients of the first two functions converge as $k^{1/2-p}$ for $p = 1$ and $p = 2$. Now, since

$$\|f - P_N f\|^2 = \|\tau\|^2 = \sum_{k=N+1}^{\infty} \frac{2}{2k+1} |\hat{f}_k|^2, \tag{1.110}$$

we expect that the error will be bounded for large N as

$$\sqrt{\sum_{k=N+1}^{\infty} \frac{1}{k^{(p-1/2)^2+1}}} < \sqrt{\int_{N+1}^{\infty} \frac{1}{z^{(p-1/2)^2+1}} dz} \sim \frac{1}{N^{p-1/2}}. \tag{1.111}$$

Equation (1.111) says that the bound on the truncation error decays as $N^{-1/2}$ for f_1, which has the jump discontinuity. If only a slope discontinuity is present, as in f_2,

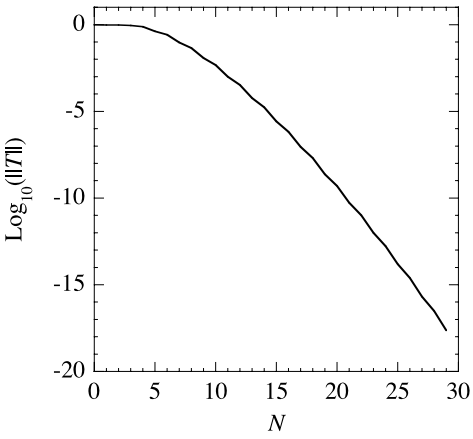

Fig. 1.8 Truncation error of $f_3 = \sin(2\pi(x + 0.1))$

then the convergence rate is higher, namely $N^{-3/2}$. Overall, we see that with discontinuities present, the truncation approximations converge with a low, polynomial order.

The truncation approximation of smooth functions, such as f_3, converges very rapidly, including exponentially fast. Figure 1.8 shows the truncation error for f_3 as a function of polynomial order. What we see is typical for the approximation of a sinusoidal function. Until N reaches a critical value, here about six, the convergence is slow. After the sine wave is adequately resolved, the approximation converges exponentially fast.

1.10.1 Derivatives of Truncated Series

We approximate the derivative of a function, f, by the derivative of its truncated series, $P_N f$

$$f' \approx (P_N f(x))' = \sum_{k=0}^{N} \bar{f}_k^{(1)} \phi_k. \tag{1.112}$$

The coefficients $\bar{f}_k^{(1)}$ are computed with recursions (1.98) and (1.102) for the Legendre and Chebyshev approximations, respectively. However, they are not the same as $\hat{f}_k^{(1)}$. Since the truncation $P_N f$ has no coefficients for $k > N$, and since the derivative of a polynomial of a degree N is a polynomial of degree $N - 1$, we start the recursion with the conditions

$$\bar{f}_N^{(1)} = \bar{f}_{N+1}^{(1)} = 0, \tag{1.113}$$

whereas the true starting values $\hat{f}_N^{(1)}$ and $\hat{f}_{N+1}^{(1)}$ are given by (1.101) and (1.103) for the Legendre and Chebyshev series, respectively. Algorithms 4 (LegendreDerivativeCoefficients) and 5 (ChebyshevDerivativeCoefficients) show how to compute

Algorithm 4: *LegendreDerivativeCoefficients:* Evaluate the Legendre Coefficients of the Derivative of a Polynomial

Procedure LegendreDerivativeCoefficients
Input: $\left\{\hat{f}_k\right\}_{k=0}^{N}$
$\bar{f}_N^{(1)} \leftarrow 0$
$\bar{f}_{N-1}^{(1)} \leftarrow (2N-1)\,\hat{f}_N$
for $k = N-2$ **to** 0 **step** -1 **do**
$\quad \bar{f}_k^{(1)} \leftarrow (2k+1)\left[\hat{f}_{k+1} + \dfrac{\bar{f}_{k+2}^{(1)}}{2k+5}\right]$
end
return $\left\{\bar{f}_k^{(1)}\right\}_{k=0}^{N}$
End Procedure LegendreDerivativeCoefficients

Algorithm 5: *ChebyshevDerivativeCoefficients:* Evaluate the Chebyshev Coefficients of the Derivative of a Polynomial

Procedure ChebyshevDerivativeCoefficients
Input: $\left\{\hat{f}_k\right\}_{k=0}^{N}$
$\bar{f}_N^{(1)} \leftarrow 0$
$\bar{f}_{N-1}^{(1)} \leftarrow (2N)\,\hat{f}_N$
for $k = N-2$ **to** 1 **step** -1 **do**
$\quad \bar{f}_k^{(1)} \leftarrow 2\,(k+1)\,\hat{f}_{k+1} + \bar{f}_{k+2}^{(1)}$
end
$\bar{f}_0^{(1)} \leftarrow \hat{f}_1 + \bar{f}_2^{(1)}/2$
return $\left\{\bar{f}_k^{(1)}\right\}_{k=0}^{N}$
End Procedure ChebyshevDerivativeCoefficients

the coefficients of the derivative of the truncated series for the Legendre and Chebyshev approximations.

Since the coefficients $\bar{f}_k^1$ and $\hat{f}_k^1$ differ, differentiation and truncation do not commute, i.e.

$$(P_N f(x))' \neq P_N f'(x) \quad (\textit{Polynomial Truncation}), \tag{1.114}$$

which means that the order in which we perform differentiation and truncation does matter.

1.11 Polynomial Quadrature

We will again use quadrature to generate discrete inner products with respect to which the polynomial basis functions remain orthogonal. For approximations with

orthogonal polynomials, the quadrature rules now should satisfy

$$\sum_{j=0}^{N} \phi_k(x_j)\phi_l(x_j)w_j = \int_{-1}^{1} \phi_k(x)\phi_l(x)w(x)dx \tag{1.115}$$

over the widest possible range of k and l. The quadrature rules with maximal precision, i.e, that are exact for the largest order polynomials are the Gauss rules. The Gauss rules are intimately tied to orthogonal polynomials and we will derive them by integrating an Hermite interpolant of a function.

To derive the Gauss quadrature rules, we start with the Hermite interpolant of a function, $f(x)$, which is the polynomial of degree $2N+1$

$$H(x) = \sum_{j=0}^{N} \bar{h}_j(x)\, f(x_j) + \sum_{j=0}^{N} \hat{h}_j(x)\, f'(x_j), \tag{1.116}$$

where

$$\begin{aligned} \bar{h}_j(x) &= \left[1 - 2(x - x_j)\ell_j'(x_j)\right]\ell_j^2(x), \\ \hat{h}_j(x) &= (x - x_j)\ell_j^2(x). \end{aligned} \tag{1.117}$$

The $\ell_j(x)$ are the usual Lagrange interpolating polynomials

$$\ell_j(x) = \prod_{\substack{i=0 \\ i\neq j}}^{N} \frac{x - x_i}{x_j - x_i}. \tag{1.118}$$

Since the Lagrange interpolating polynomials satisfy $\ell_j(x_i) = \delta_{i,j}$, the Hermite interpolant satisfies $H(x_j) = f(x_j)$ and $H'(x_j) = f'(x_j)$. Therefore, H interpolates both the function and its derivative. The interpolation is exact for all functions that are polynomials of degree $2N+1$ or less.

To derive the quadrature rule, we integrate the polynomial approximation, $H(x)$, and use that integral to approximate the integral of f

$$\begin{aligned} \int_a^b f(x)w(x)dx \approx Q[f] = &\sum_{j=0}^{N} f(x_j) \int_a^b w(x)\bar{h}_j(x)dx \\ &+ \sum_{j=0}^{N} f'(x_j) \int_a^b w(x)\hat{h}_j(x)dx. \end{aligned} \tag{1.119}$$

For the quadrature rule to match (1.115), that is, to have no dependence on f', the integrals in the second sum of (1.119) must vanish. This is where orthogonality comes into play. First, note that we can write the Lagrange interpolating polynomi-

als, $\ell_j(x)$ in the alternate form

$$\ell_j(x) = \prod_{\substack{i=0 \\ i\neq j}}^{N} \frac{x - x_i}{x_j - x_i} = \frac{\psi_{N+1}(x)}{(x - x_j)\psi'_{N+1}(x_j)}, \tag{1.120}$$

where

$$\psi_{N+1}(x) = \prod_{i=0}^{N} (x - x_i). \tag{1.121}$$

Then

$$\hat{h}_j(x) = (x - x_j)\ell_j(x)\ell_j(x) = \frac{\psi_{N+1}(x)\ell_j(x)}{\psi'_{N+1}(x_j)}, \tag{1.122}$$

which means that we can rewrite

$$\int_a^b w(x)\hat{h}_j(x)dx = \frac{1}{\psi'_{N+1}(x_j)} \int_a^b w(x)\psi_{N+1}(x)\ell_j(x)dx. \tag{1.123}$$

Now we use orthogonality. The polynomial $\ell_j(x)$ is of degree $\leq N$, while $\psi_{N+1}(x)$ is a polynomial of degree $N+1$. We can ensure that the integral vanishes if $\psi_{N+1}(x)$ is orthogonal to all polynomials of degree N or less with respect to the weight $w(x)$. Orthogonality is immediate if we choose

$$\psi_{N+1}(x) = \prod_{i=0}^{N} (x - x_i) = \frac{\phi_{N+1}}{\gamma}, \tag{1.124}$$

where γ is the coefficient of x^{N+1} in the $N + 1^{st}$ basis function ϕ_{N+1}. Thus we choose the abscissas of the quadrature, the x_i's, to be the roots of ϕ_{N+1}.

Since our interest is in approximations with Jacobi polynomials, we now have the following *Jacobi Gauss Quadrature*:

$$Q_G[f] = \sum_{j=0}^{N} f(x_j)w_j, \tag{1.125}$$

where

$$\begin{aligned} x_j &= \text{zeros of } \phi_{N+1}, \\ w_j &= \int_{-1}^{1} w(x)\,\bar{h}_j(x)dx. \end{aligned} \tag{1.126}$$

The quadrature (1.125) is exact for all functions, f, that are polynomials of degree $2N + 1$ or less. Note that to compute the abscissas and weights we must find the roots of the $N + 1$st order polynomial and evaluate the integral for the quadrature weights.

The two polynomial basis functions of most interest are the Legendre and Chebyshev polynomials. The Legendre Gauss quadrature reduces to

$$\begin{aligned} x_j &= \text{zeros of } L_{N+1}(x), \\ w_j &= \frac{2}{(1-x_j^2)[L'_{N+1}(x_j)]^2}. \end{aligned} \tag{1.127}$$

The Chebyshev Gauss quadrature has a simpler form due to the trigonometric representation of the Chebyshev polynomials. Since $T_{N+1} = \cos((N+1)\cos^{-1}(x))$, we can easily compute the abscissas and weights with

$$\begin{aligned} x_j &= \cos\left(\frac{2j+1}{2N+2}\pi\right), \quad j = 0, 1, \ldots, N, \\ w_j &= \frac{\pi}{N+1}. \end{aligned} \tag{1.128}$$

As we mentioned in the previous section, one of the properties of the Jacobi polynomials is that their roots are interior to the interval. This fact is especially clear for the Chebyshev abscissas, (1.128). To solve boundary value problems, however, we usually want to include the boundary points in the approximations.

The Gauss quadrature rules that include the endpoints, $x = \pm 1$, are known as the *Gauss-Lobatto* rules. For problems with Legendre weighted integrals, the abscissas and weights for the Gauss-Lobatto rule are

$$\begin{aligned} x_j &= +1, -1, \text{ zeros of } L'_N(x), \\ w_j &= \frac{2}{N(N+1)} \frac{1}{[L_N(x_j)]^2}, \qquad j = 0, 1, \ldots, N. \end{aligned} \tag{1.129}$$

For Chebyshev weighted integrals, they are

$$\begin{aligned} x_j &= \cos\left(\frac{j\pi}{N}\right), \quad j = 0, 1, \ldots, N, \\ w_j &= \begin{cases} \frac{\pi}{2N}, & j = 0, N, \\ \frac{\pi}{N}, & j = 1, 2, \ldots, N-1. \end{cases} \end{aligned} \tag{1.130}$$

Requiring the end points to be abscissas reduces the number of degrees of freedom at which the abscissas can be located by two, so it should not be surprising that the Gauss-Lobatto rules have lower precision than the (optimal) Gauss rules. In fact, the Gauss-Lobatto rules are exact for polynomials of degree $2N-1$ or less.

With the quadrature rules, we can now define the discrete inner product for orthogonal polynomial approximations

$$(u, v)_N = \sum_{j=0}^{N} u(x_j) v(x_j) w_j. \tag{1.131}$$

If we replace u and v by the basis functions ϕ_n and ϕ_m, then

$$(\phi_n, \phi_m)_N = \int_{-1}^{1} \phi_n \phi_m w dx = \|\phi_n\|_N^2 \, \delta_{n,m}, \tag{1.132}$$

where $\|\phi_n\|_N^2 = (\phi_n, \phi_n)_N$, when

$$\begin{aligned} &n + m \leq 2N + 1 \quad \text{Gauss Rule,} \\ &n + m \leq 2N - 1 \quad \text{Gauss-Lobatto Rule.} \end{aligned} \tag{1.133}$$

We use the exactness of the quadratures to evaluate the discrete norms. For the Gauss points, the quadrature is exact for $n \leq N$, so

Gauss Points:

$$\|\phi_n\|_N^2 = \begin{cases} \frac{2}{2n+1} & \textit{Legendre Gauss,} \\ \frac{\pi}{2} c_n & \textit{Chebyshev Gauss.} \end{cases} \tag{1.134}$$

The coefficients c_n were defined previously. They are $c_0 = 2$ and $c_n = 1$ for $n > 0$.

The quadrature is exact for $n < N$ if we use the Gauss-Lobatto points, so we can use the analytical norm for those values of n, too. When $n = N$, we must compute the norm specially.

Lobatto Points:

$n < N$

$$\|\phi_n\|_N^2 = \begin{cases} \frac{2}{2n+1} & \textit{Legendre Gauss-Lobatto,} \\ \frac{\pi}{2} c_n & \textit{Chebyshev Gauss-Lobatto.} \end{cases} \tag{1.135}$$

$n = N$

$$\|\phi_N\|_N^2 = \begin{cases} \frac{2}{N} & \textit{Legendre Gauss-Lobatto,} \\ \pi & \textit{Chebyshev Gauss-Lobatto.} \end{cases} \tag{1.136}$$

1.12 Orthogonal Polynomial Interpolation

Now that we have derived the discrete inner products, we derive the polynomial interpolant

$$I_N f(x) = \sum_{k=0}^{N} \tilde{f}_k \phi_k(x) \tag{1.137}$$

that satisfies the interpolation conditions

$$I_N f(x_j) = f(x_j), \quad j = 0, 1, \ldots, N. \tag{1.138}$$

Note, again, that we have written the *discrete coefficients*, as $\tilde{f}_k$ to distinguish them from the coefficients of the truncation approximations $\hat{f}_k$ that we defined in Sect. 1.10.

To find the interpolant means formally to find the discrete coefficients. Equation (1.138) produces a linear system of equations to solve (uniquely) for the unknown coefficients. However, we will once again appeal to the orthogonality of the basis functions to find the coefficients. Since the interpolation condition (1.138) is defined at a discrete set of points, x_j, the discrete orthogonality discussed in the previous section is the important property now.

To simplify the computation of the interpolation expansion coefficients, we appeal to the fact that the orthogonal basis functions remain orthogonal under the discrete inner product created from the Gauss quadrature. For non-periodic functions, we choose the interpolation points x_j to be the abscissas of either a Gauss or Gauss-Lobatto quadrature, then take the discrete inner product

$$(I_N f, \phi_n)_N = \left(\sum_{k=0}^{N} \tilde{f}_k \phi_k, \phi_n \right)_N = \sum_{k=0}^{N} \tilde{f}_k \, (\phi_k, \phi_n)_N, \quad n = 0, 1, \ldots, N. \tag{1.139}$$

Thus, the coefficients of the orthogonal polynomial expansion are

$$\tilde{f}_n = \frac{(I_N f, \phi_n)_N}{\|\phi_n\|_N^2} = \frac{(f, \phi_n)_N}{\|\phi_n\|_N^2}, \quad n = 0, 1, \ldots, N \tag{1.140}$$

or

$$\tilde{f}_n = \frac{1}{\|\phi_n\|_N^2} \sum_{j=0}^{N} f(x_j) \phi_n(x_j) w_j. \tag{1.141}$$

In turn, we compute the nodal values of the interpolant, $I_N f(x_j)$, by a similar sum

$$f(x_j) = \sum_{k=0}^{N} \tilde{f}_k \phi_k(x_j). \tag{1.142}$$

Together, equations (1.141) and (1.142) form the *Discrete Polynomial Transform Pair.*

Polynomial interpolation is usually less accurate than series truncation, the difference again being due to aliasing. To show this, we follow the procedure that we used for the Fourier approximations, which begins by relating the exact and discrete coefficients $\hat{f}_k$ and $\tilde{f}_k$. If we replace f in (1.141) by its series representation, then

$$\tilde{f}_k = \frac{(\sum_{m=0}^{\infty} \hat{f}_m \phi_m, \phi_k)_N}{\|\phi_k\|_N^2} = \hat{f}_k + \frac{\sum_{m=N+1}^{\infty} \hat{f}_m (\phi_m, \phi_k)_N}{\|\phi_k\|_N^2}, \quad k = 0, 1, \ldots, N. \tag{1.143}$$

Since $m \geq N+1$ in the last sum, quadrature error that occurs when $k+m > 2N-1$ for Lobatto interpolation and $k+m > 2N+1$ for Gauss interpolation can cause

the discrete inner product of ϕ_k with ϕ_m to not vanish as it does for the true inner product.

We relate the interpolant and the truncated series by replacing the discrete coefficients in (1.137) by (1.143)

$$\begin{aligned} I_N f &= \sum_{k=0}^{N} \hat{f}_k \phi_k + \sum_{k=0}^{N} \left\{ \frac{1}{\|\phi_k\|_N^2} \sum_{m=N+1}^{\infty} \hat{f}_m \left(\phi_m, \phi_k\right)_N \right\} \phi_k \\ &= P_N f + R_N f. \end{aligned} \tag{1.144}$$

We can now show that the interpolation error cannot be less than the truncation error for polynomial approximations. The interpolation error is the difference $f - I_N f$, so

$$f - I_N f = (f - P_N f) - R_N f. \tag{1.145}$$

Now $R_N f$ has only terms of order N or less, while $(f - P_N f)$ has terms of order $N + 1$ or larger, so they are orthogonal to each other. Thus, the norm of the interpolation error satisfies

$$\|f - I_N f\|^2 = \|f - P_N f\|^2 + \|R_N f\|^2 . \tag{1.146}$$

As with Fourier interpolation, the aliasing error that arises in polynomial interpolation is not significant for smooth functions. For details, see [7].

Exercises

1.1 In Fig. 1.3 we showed how the norm of the Fourier truncation error decays with the number of modes. In this problem, plot $P_N f$ for the three functions as a function of x for $N = 4, 8, 16, 32$ and observe the behavior of the approximations. In particular, observe the well-known Gibbs phenomenon in the neighborhood of the discontinuities in the periodic extensions of the functions.

1.2 To reduce the size of the oscillations seen in the truncated Fourier series for non-smooth functions, we often *smooth/filter* it. The filtered approximation is of the form

$$\bar{P}_N f = \sum_{k=-N/2}^{N/2} \sigma_k \hat{f}_k e^{ikx},$$

where σ_k is the filter function. Typically, the filter function is a smooth low pass filter with $\sigma_0 = 1$ and $\sigma_{\pm N/2} = 0$. An example is the raised cosine filter,

$$\sigma_k = \frac{1 + \cos\left(2k\pi/N\right)}{2}, \quad k = -N/2, \ldots, N/2.$$

To see the effects of filtering, plot the Fourier truncation approximation of the function

$$f(x) = \begin{cases} 1, & \frac{\pi}{2} < x \le \frac{3\pi}{2}, \\ 0, & 0 < x \le \frac{\pi}{2} \text{ and } \frac{3\pi}{2} < x \le 2\pi \end{cases}$$

with and without the raised cosine filter. (The Fourier coefficients for this function are

$$\hat{f}_k = \begin{cases} \pi, & k = 0, \\ 0, & k \neq 0, \text{ even}, \\ \frac{(-1)^{(k-1)/2}}{k}, & k \neq 0, \text{ odd}.) \end{cases}$$

To learn more about filtering, see [7].

1.3 Equation (1.44) establishes that with enough nodes the composite trapezoidal rule integrates complex exponentials exactly. Otherwise, it does not. Compute and plot the logarithm of the error as a function of N for the following two integrals:

$$I_1 = \int_0^{2\pi} \cos(10x)\, dx,$$

$$I_2 = \int_0^{2\pi} \cos(x)\, e^{\sin(x)} dx.$$

Note the behavior of the error as a function of N and compare to the usual $O(\Delta x^2)$ behavior for the composite trapezoidal rule.

1.4 Repeat Problem 1.1 for $I_N f$.

1.5 Use the equivalence of (1.73) and (1.74) to show that the eigenvalues of the Fourier derivative matrix of $D_{ij} = h_j'(x_i)$ are $\lambda_k = ik$, $k = -N/2+1, \ldots, N/2-1$. Knowing the eigenvalues is important to approximate PDEs in time.

1.6 In Fig. 1.8 we showed how the norm of the Legendre truncation error decays with the number of modes. For this problem, plot the Legendre truncation approximation $P_N f$ for the three functions in (1.107) as a function of x for $N = 4, 8, 16, 32, \ldots$ and observe the behavior of the approximations. Again, observe the well-known Gibbs phenomenon in the neighborhood of the discontinuities in the functions.

1.7 In the likely event that the domain of interest is not $[0, 2\pi]$ for periodic problems or $[-1, 1]$ for non-periodic problems, it is necessary to map the integrals to the reference intervals. Usually one uses an *affine* mapping $\xi = a + bx$, for some a and b, to transform the intervals. Rewrite the quadratures (1.45) and (1.125) so that they can be used on an arbitrary, yet finite interval.

Chapter 2
Algorithms for Periodic Functions

In this chapter we show how to compute the Discrete Fourier Transform using a Fast Fourier Transform (FFT) algorithm, including not-so special case situations such as when the data to be transformed are real. In those situations, we speed up the transforms by about a factor of two by exploiting symmetries in the data and the coefficients. We end this chapter by showing how to approximate the derivatives of periodic functions, which are the fundamental approximations that we need to solve partial differential equations with periodic boundary conditions.

2.1 How to Compute the Discrete Fourier Transform

The sums in the Discrete Fourier Transform pair, (1.69), cost $O(N^2)$ operations to compute: N complex multiplications for each of the N values. For large N, they are too expensive to compute by the direct sum. However, we can compute the sums in a significantly more efficient $O(N \log_2(N))$ operations if we use the Fast Fourier Transform. The complex FFT is the core algorithm for the efficient implementation of Fourier spectral methods for large N.

It is beyond the scope of this book to give a thorough presentation of Fast Fourier Transform algorithms. FFTs are ubiquitous and are discussed in detail in numerous books (e.g. [5] and [6], among others), and the basic ideas are found in upper level numerical analysis books. Implementations exist in numerous programming languages tuned for virtually any computer. Rather than use an FFT presented in any book, we recommend libraries provided by one's computer vendor, a well-developed and tested FFT algorithm such as FFTW [12] (http://www.fftw.org/) or that can be found, for example, at netlib (www.netlib.org), or routines sold as part of one of the many commercial numerical libraries (e.g. NAG or IMSL). For completeness, however, we include Temperton's [23] self-sorting in-place complex FFT. Temperton's paper includes a mixed-radix FFT ($N = 2^m 3^q 5^r$), but for reasons of space and simplicity, we present only the radix 2 ($N = 2^m$) algorithm here.

In spectral method applications, the solutions of the differential equations are typically real. When the sequences are real, we can exploit symmetries in the coefficients to save approximately a factor of two in the work to compute the transforms. We present two algorithms that use these symmetries in this section. One simultaneously computes the coefficients of two real sequences with a single complex FFT. The other splits a single real sequence into two sequences of half the original length by putting the even indexed elements into the real part of a complex sequence and the odd indexed elements into the imaginary part.

D.A. Kopriva, *Implementing Spectral Methods for Partial Differential Equations,* Scientific Computation,

2.1.1 Fourier Transforms of Complex Sequences

We can recast the Discrete Fourier Transform (DFT) pair (1.69) as

$$\text{DFT} \quad \begin{cases} G_k = \frac{1}{N} \sum_{j=0}^{N-1} g_j e^{-2\pi ijk/N}, & k = 0, 1, \ldots, N-1, \\ g_j = \sum_{k=0}^{N-1} G_k e^{2\pi ijk/N}, & j = 0, 1, \ldots, N-1. \end{cases} \tag{2.1}$$

The first sum, which represents the decomposition of the sequence of physical space values into its Fourier components, is the *forward transform*. It takes an N-periodic sequence $\{g_j\}_{j=0}^{N-1}$ and returns an N-periodic sequence $\{G_j\}_{j=0}^{N-1}$ of *Discrete Fourier coefficients*. (An N-periodic sequence is one for which $g_{j\pm N} = g_j$.) The constant $i = \sqrt{-1}$. The second sum, which represents the synthesis of the Fourier modes back into the physical space, is the *backward transform*. Properties of the transforms can be found in numerous books, including [6].

Except for the sign of the exponents, the forward and backward transforms compute the same sum, namely

$$\text{DFTS:} \quad F_k = \sum_{j=0}^{N-1} f_j e^{\pm 2\pi ijk/N}, \quad k = 0, 1, \ldots, N-1. \tag{2.2}$$

We could compute the *Discrete Fourier Transform Sum*, DFTS of (2.2), directly by Algorithm 6 (DFT). The sign of the input variable s determines whether the procedure computes the forward or backward sum. To avoid having to remember which sign corresponds to which transform, let's define two constants $FORWARD = 1$ and $BACKWARD = -1$ to use as input to the procedure.

The range of wavenumbers, k, used in (2.1) is not the range we need to compute the Fourier interpolant (1.69). For that we need the coefficients in the order Algorithm 6 (DFT), and (2.2) each return a sequence with elements ordered as $k = 0, 1, \ldots, N-1$ instead.

Algorithm 6: *DFT*: Direct (and Slow) Evaluation of the Discrete Fourier Transform

```
Procedure DFT
Input: {f_j}_{j=0}^{N-1}, s
for k = 0 to N − 1 do
    F_k ← 0
    for j = 0 to N − 1 do
        F_k ← F_k + f_j ∗ e^{−2sπijk/N}
    end
end
return {F_k}_{k=0}^{N-1}
End Procedure DFT
```

For the DFT to be useful in spectral approximations, we must reinterpret the order of the sequences. Equation (1.63) shows that the discrete Fourier coefficient $\tilde{f}_k$ are N-periodic, that is, $\tilde{f}_k = \tilde{f}_{k\pm N}$. Thus, $\tilde{f}_{N/2} = \tilde{f}_{-N/2}$, and therefore the second half of the sequence returned by the transform with $s = -1$ corresponds to the negative values of the index. To use the results of the sum (2.2), i.e. of Algorithm 6 (DFT), we make the following correspondence:

$$\begin{bmatrix} F_0 \\ F_1 \\ \vdots \\ F_{N/2-1} \\ F_{N/2} \\ F_{N/2+1} \\ \vdots \\ F_N \end{bmatrix} \Longleftrightarrow \begin{bmatrix} N\tilde{f}_0 \\ N\tilde{f}_1 \\ \vdots \\ N\tilde{f}_{N/2-1} \\ N\tilde{f}_{-N/2} \\ N\tilde{f}_{-N/2+1} \\ \vdots \\ N\tilde{f}_{-1} \end{bmatrix}. \tag{2.3}$$

To reconstruct the set of values $\{f_j\}_{j=0}^{N-1}$ from the set of coefficients $\{\tilde{f}_k\}_{k=-N/2}^{N/2-1}$, we simply order them using the correspondence (2.3) before calling the transform with $s = +1$. We will show an example in Sect. 2.3 when we compute Fourier interpolation derivatives by FFTs.

It is highly unlikely that one would want use Algorithm 6 for anything but the simplest of exercises. That algorithm clearly requires $O(N^2)$ complex exponentiations and multiplications. Indeed, the early success of spectral methods was due the fact that one could use a Fast Fourier Transform (FFT) to compute these sums with only $O(N\,\mathrm{Log}_2(N))$ operations.

To help understand how to use an FFT routine, we present Temperton's self-sorting, in-place complex FFT. Like many FFT algorithms, Temperton's pre-computes and stores the complex exponential factors, $e^{-2s\pi ijk/N}$, since those evaluations are so expensive. To that end, we define the arrays $\{w_n^{\pm}\}_{n=0}^{N-1}$ whose elements are $e^{-2s\pi in/N}$ for $s = \pm 1$. Thus, we perform the forward transform when the w's are computed with $s = 1$, and the backward transform when they are computed with $s = -1$. The pre-computation of the trigonometric factors is done by Algorithm 7 (InitializeFFT).

Algorithm 7: *InitializeFFT:* Initialization Routine for FFT

Procedure InitializeFFT
Input: N, s
$w \leftarrow e^{-2s\pi i/N}$
for $j = 0$ **to** $N-1$ **do**
  $w_j \leftarrow w^j$
end
return $\{w_j\}_{j=0}^{N-1}$
End Procedure InitializeFFT

Algorithm 8: *Radix2FFT:* Temperton's Radix 2 Self Sorting Complex FFT

```
Procedure Radix2FFT
Input: {f_j}_{j=0}^{N-1}, {w_j}_{j=0}^{N-1}
Integers: noPtsAtLevel, a, b, c, d, p, N div 2, m, l, k
N div 2 ← N/2
m ← Log_2(N)
for l = 1 to (m + 1)/2 do
    noPtsAtLevel ← 2^{l-1}
    a ← 0
    b ← N div 2/noPtsAtLevel
    p ← 0
    for k = 0 to b − 1 do
        W ← w_p
        for i = k to N − 1 step N/noPtsAtLevel do
            z ← W ∗ (f_{a+i} − f_{b+i})
            f_{a+i} ← f_{a+i} + f_{b+i}
            f_{b+i} ← z
        end
        p ← p + noPtsAtLevel
    end
end
for l = (m + 3)/2 to m do
    noPtsAtLevel ← 2^{l-1}
    a ← 0
    b ← N div 2/noPtsAtLevel
    c ← noPtsAtLevel
    d ← b + noPtsAtLevel
    p ← 0
    for k = 0 to b − 1 do
        W ← w_p
        for j = k to noPtsAtLevel − 1 step N/noPtsAtLevel do
            for i = j to N − 1 step 2 ∗ noPtsAtLevel do
                z ← W ∗ (f_{a+i} − f_{b+i})
                f_{a+i} ← f_{a+i} + f_{b+i}
                f_{b+i} ← f_{c+i} + f_{d+i}
                f_{d+i} ← w ∗ (f_{c+i} − f_{d+i})
                f_{c+i} ← z
            end
        end
        p ← p + noPtsAtLevel
    end
end
return {f_k}_{j=0}^{N-1}
End Procedure Radix2FFT
```

We show the FFT algorithm itself in Algorithm 8 (Radix2FFT). The FFT takes a sequence of complex numbers $\{f_j\}_{j=0}^{N-1}$ and the pre-computed complex trigonometric factors $\{w_j\}_{j=0}^{N-1}$ and returns the sum (2.2) by overwriting the original complex sequence. Whether it computes the forward or backward transform depends on the

sequence of w's that it is supplied. For details on the algorithm, see [23]. We show examples that use the complex FFT in the sections that follow, where we study special cases of the transforms.

2.1.2 Fourier Transforms of Real Sequences

The complex FFT is about twice as expensive as is necessary when the data to be transformed are real. One of two methods is typically used to compute transforms of real sequences efficiently from the complex transform. The first method solves for the coefficients of two real sequences simultaneously. This is useful if two or more real FFTs have to be computed at once, as happens in multidimensional problems. The second is to use an even-odd decomposition where the transform is computed on a complex sequence of half the length of the original. In that algorithm, half of the values of the real sequence are placed in the real part of the complex sequence and the other half in the imaginary part.

2.1.2.1 Simultaneous Fourier Transformation of Two Real Sequences

We can use the complex FFT to compute the DFT of two real sequences simultaneously. Suppose that $\{x_j\}_{j=0}^{N-1}$ and $\{y_j\}_{j=0}^{N-1}$ are real sequences whose discrete transform coefficients we need. Then

$$\begin{aligned} X_k &= \frac{1}{N}\sum_{j=0}^{N-1} x_j e^{-2\pi ijk/N}, \\ & \qquad\qquad k = 0, 1, \ldots, N-1. \\ Y_k &= \frac{1}{N}\sum_{j=0}^{N-1} y_j e^{-2\pi ijk/N}, \end{aligned} \tag{2.4}$$

If we combine these into a single complex vector, $z = x + iy$, the discrete Fourier coefficients of z are

$$Z_k = \frac{1}{N}\sum_{j=0}^{N-1} z_j e^{-2\pi ijk/N} = \frac{1}{N}\sum_{j=0}^{N-1} x_j e^{-2\pi ijk/N} + \frac{i}{N}\sum_{j=0}^{N-1} y_j e^{-2\pi ijk/N} \tag{2.5}$$

or

$$Z_k = X_k + iY_k. \tag{2.6}$$

Since x_j and y_j are real,

$$Z^*_{-k} = \frac{1}{N}\sum_{j=0}^{N-1} x_j e^{-2\pi ijk/N} - \frac{i}{N}\sum_{j=0}^{N-1} y_j e^{-2\pi ijk/N} \tag{2.7}$$

so

$$Z^*_{-k} = X_k - i\,Y_k. \tag{2.8}$$

When we solve for X_k and Y_k from (2.6) and (2.8)

$$\begin{aligned} X_k &= \frac{1}{2}\left(Z_k + Z^*_{-k}\right), \\ Y_k &= \frac{-i}{2}\left(Z_k - Z^*_{-k}\right), \end{aligned} \qquad k = 0, 1, \ldots, N-1. \tag{2.9}$$

The discrete coefficients are N-periodic, so $Z_{-k} = Z_{N-k}$ and $Z_0 = Z_N$. Therefore,

$$\begin{aligned} X_k &= \frac{1}{2}\left(Z_k + Z^*_{N-k}\right), \\ Y_k &= \frac{-i}{2}\left(Z_k - Z^*_{N-k}\right), \end{aligned} \qquad k = 0, 1, \ldots, N-1. \tag{2.10}$$

We show how to compute the forward DFT of two real sequences in Algorithm 9 (FFFTOfTwoRealVectors). It takes as input the trigonometric factors pre-computed with $s = FORWARD$, the two real sequences, and the length of the sequences. It returns the scaled discrete Fourier coefficients of the two sequences. Since the complex FFT does not scale the forward transform by the $1/N$ factor, the procedure does the scaling when it extracts the coefficients. It is possible to modify the weights w^+ computed by Algorithm 7 (InitializeFFT) to include the factor of $1/N$ to save the cost of the divisions if multiple transforms are needed. We have added the scaling

Algorithm 9: *FFFTOfTwoRealVectors:* Simultaneous Computation of the DFT of Two Real Sequences. The Forward Transform

Procedure FFFTOfTwoRealVectors
Input: $\{x_j\}_{j=0}^{N-1}, \{y_j\}_{j=0}^{N-1}, \{w_j^+\}_{j=0}^{N-1}$
Uses Algorithms:
Algorithm 8 (Radix2FFT)

for $j = 0$ **to** $N-1$ **do**
  $Z_j \leftarrow x_j + i y_j$
end
$\{Z_k\}_{k=0}^{N-1} \leftarrow Radix2FFT\left(\{Z_j\}_{j=0}^{N-1}, \{w_j^+\}_{j=0}^{N-1}\right)$
$X_0 \leftarrow \mathrm{Re}(z_0)/N$
$Y_0 \leftarrow \mathrm{Im}(z_0)/N$
for $k = 1$ **to** $N-1$ **do**
  $X_k \leftarrow \frac{1}{2}\left(Z_k + Z^*_{N-k}\right)/N$
  $Y_k \leftarrow \frac{-i}{2}\left(Z_k - Z^*_{N-k}\right)/N$
end
return $\{X_k\}_{k=0}^{N-1}, \{Y_k\}_{k=0}^{N-1}$
End Procedure FFFTOfTwoRealVectors

Algorithm 10: *BFFTForTwoRealVectors:* Simultaneous Computation of the DFT of Two Real Sequences. The Backward Transform

Procedure BFFTForTwoRealVectors
Input: $\{X_j\}_{j=0}^{N-1}, \{Y_j\}_{j=0}^{N-1}, \{w_k^-\}_{k=0}^{N-1}$
Uses Algorithms:
Algorithm 8 (Radix2FFT)
for $j = 0$ **to** $N-1$ **do**
  $Z_j \leftarrow X_j + iY_j$
end
$\{Z_k\}_{k=0}^{N-1} \leftarrow Radix2FFT\left(\{Z_j\}_{j=0}^{N-1}, \{w_k^-\}_{k=0}^{N-1}\right)$
for $k = 0$ **to** $N-1$ **do**
  $x_k \leftarrow \mathrm{Re}(Z_k)$
  $y_k \leftarrow \mathrm{Im}(Z_k)$
end
return $\{x_k\}_{k=0}^{N-1}, \{y_k\}_{k=0}^{N-1}$
End Procedure BFFTForTwoRealVectors

here, since a modification of the trigonometric factors routine might not be possible if a library FFT is used.

To reverse the operation, we use (2.6) and the inverse FFT. The desired solutions are simply the real and imaginary parts of the complex sequence returned by the FFT. We present the procedure to compute the inverse transform in Algorithm 10 (BFFTForTwoRealVectors).

2.1.2.2 Fourier Transformation of a Real Sequence by Even-Odd Decomposition

We can also evaluate the DFT of a single sequence of real values efficiently with a complex FFT. The FFT will operate on a new sequence of half the original length that we create by putting half the original data into the real part and the other half into the complex part. Let the even and odd elements of a sequence $\{f_j\}_{j=0}^{N-1}$ be

$$\begin{aligned} e_j &= f_{2j}, \\ o_j &= f_{2j+1}, \end{aligned} \qquad j = 0, 1, \ldots, M-1, \tag{2.11}$$

where $M = N/2$. The forward Fourier transforms of these two sequences are

$$\begin{aligned} E_k &= \frac{1}{M} \sum_{j=0}^{M-1} e_j e^{-2\pi ijk/M}, \\ O_k &= \frac{1}{M} \sum_{j=0}^{M-1} o_j e^{-2\pi ijk/M}, \end{aligned} \qquad j = 0, 1, \ldots, M-1. \tag{2.12}$$

But

$$
\begin{aligned}
F_k &= \frac{1}{N}\sum_{j=0}^{N-1} f_j e^{-2\pi ijk/N} \\
&= \frac{1}{N}\sum_{j=0}^{2M-1} f_j e^{-2\pi ijk/2M} \\
&= \frac{1}{N}\sum_{j=0}^{M-1} f_{2j} e^{-2\pi ijk/M} + \frac{1}{N}\sum_{j=0}^{M-1} f_{2j+1} e^{-2\pi i(2j+1)k/2M} \\
&= \frac{1}{2M}\sum_{j=0}^{M-1} f_{2j} e^{-2\pi ijk/M} + \frac{e^{-2\pi ik/N}}{2M}\sum_{j=0}^{M-1} f_{2j+1} e^{-2\pi ijk/M} \\
&= \frac{1}{2}\left(E_k + e^{-2\pi ik/N} O_k\right), \quad k = 0, 1, \ldots, N/2-1.
\end{aligned} \tag{2.13}
$$

Thus, the complex DFT of the even-odd decomposition of the real sequence produces the first half of the discrete coefficients of the original sequence. We find the second half of the sequence of coefficients by recalling that if f_k is real and N-periodic, $F_{N-k} = F_k^*$.

We could compute coefficients E_k and O_k simultaneously using Algorithm 9 (FFFTOfTwoRealVectors), but it is probably best to combine the two algorithms. To combine them, we let $z_j = e_j + io_j$ and use (2.9) to define

$$
\begin{aligned}
F_k = \frac{1}{2N}\left\{\left(Z_k + Z_{N/2-k}^*\right) - ie^{-2\pi ijk/N}\left(Z_k - Z_{N/2-k}^*\right)\right\}, \\
k = 0, 1, \ldots, N/2-1.
\end{aligned} \tag{2.14}
$$

We find the second half of the F_k array from the symmetry relation, the first value from

$$
F_0 = \left(\mathrm{Re}(Z_0) + \mathrm{Im}(Z_0)\right)/N \tag{2.15}
$$

and the center from

$$
F_{N/2} = \left(\mathrm{Re}(Z_0) - \mathrm{Im}(Z_0)\right)/N. \tag{2.16}
$$

Algorithm 11 (FFFTEO) shows one way to implement the Even-Odd decomposition. It uses the fact that

$$
e^{\pm 2\pi ij/M} = e^{\pm 2\pi i(2j)/N} \tag{2.17}
$$

so that the factors w^+ required by the complex FFT of length M are simply the even indexed components of the factors computed for the full length transform, factors that are also required by (2.13). This saves us the re-computation of the trigonometric factors.

Algorithm 11: *FFFTEO:* The Forward DFT by Even-Odd Decomposition

Procedure FFFTEO
Input: $\{f_j\}_{j=0}^{N-1}, \{w_j^+\}_{j=0}^{N-1}$
Uses Algorithms:
Algorithm 8 (Radix2FFT)
for $j = 0$ **to** $N/2-1$ **do**
  $Z_j \leftarrow f_{2j} + i f_{2j+1}$
  $w_j \leftarrow w_{2j}^+$
end
$\{Z_k\}_{k=0}^{N/2-1} \leftarrow Radix2FFT\left(\{Z_j\}_{j=0}^{N/2-1}, \{w_j\}_{j=0}^{N/2-1}\right)$
$F_0 \leftarrow (\mathrm{Re}(Z_0) + \mathrm{Im}(Z_0))/N$
$F_{N/2} \leftarrow (\mathrm{Re}(Z_0) - \mathrm{Im}(Z_0))/N$
for $k = 1$ **to** $N/2-1$ **do**
  $F_k \leftarrow \frac{1}{2N}\left\{\left(Z_k + Z_{N/2-k}^*\right) - i w_k^+ \left(Z_k - Z_{N/2-k}^*\right)\right\}$
end
for $k = 1$ **to** $N/2-1$ **do**
  $F_{N-k} \leftarrow F_k^*$
end
return $\{F_k\}_{k=0}^{N-1}$
End Procedure FFFTEO

We can also evaluate the backward transform using the even-odd decomposition. Since $F_{k+N/2} = F_{N/2-k}^*$, (2.13) implies that

$$F_{N/2-k}^* = \frac{1}{2N}\left[\left(Z_k + Z_{M-k}^*\right) + i e^{-2\pi i k/N}\left(Z_k - Z_{N/2-k}^*\right)\right]. \tag{2.18}$$

If we add and subtract this and (2.13) we get

$$E_k = \left(F_k + F_{N/2-k}^*\right), \quad k = 0, 1, \ldots, N/2-1 \tag{2.19}$$

and

$$O_k = e^{2\pi i k/N}\left(F_k - F_{N/2-k}^*\right), \quad k = 0, 1, \ldots, N/2-1. \tag{2.20}$$

When we combine these into a single complex sequence

$$Z_k = E_k + i\, O_k, \quad k = 0, 1, \ldots, N/2-1, \tag{2.21}$$

the backward FFT produces

$$z_j = e_j + i o_j, \quad k = 0, 1, \ldots, N/2-1. \tag{2.22}$$

We extract the full sequence from the real and the imaginary parts,

$$\left.\begin{aligned} f_{2j} &= \mathrm{Re}(z_j) \\ f_{2j+1} &= \mathrm{Im}(z_j) \end{aligned}\right\}, \quad j = 0, 1, \ldots, N/2-1. \tag{2.23}$$

Algorithm 12: *BFFTEO:* The Backward DFT by Even-Odd Decomposition

Procedure BFFTEO
Input: $\{F_k\}_{k=0}^{N-1}$, $\{w_k^-\}_{k=0}^{N-1}$
Uses Algorithms:
Algorithm 8 (Radix2FFT)

$M \leftarrow N/2$
for $k = 0$ **to** $M-1$ **do**
  $E \leftarrow \left(F_k + F_{M-k}^*\right)$
  $O \leftarrow w_k^- \left(F_k - F_{M-k}^*\right)$
  $Z_k \leftarrow E_k + i\, O_k$
  $w_k \leftarrow w_{2k}^-$
end
$\left\{Z_j\right\}_{j=0}^{M-1} \leftarrow Radix2FFT(\{Z_k\}_{k=0}^{M-1}, \{w_k\}_{k=0}^{M-1})$
for $j \leftarrow 0$ **to** $M-1$ **do**
  $f_{2j} \leftarrow \mathrm{Re}(Z_j)$
  $f_{2j+1} \leftarrow \mathrm{Im}(Z_j)$
end
return $\left\{f_j\right\}_{j=0}^{N-1}$
End Procedure BFFTEO

We present the procedure for using the even-odd decomposition to compute the backward transform in Algorithm 12 (BFFTEO). As before, we use the periodicity of the trigonometric factors to avoid their re-computation.

2.1.3 The Fourier Transform in Two Space Variables

Multidimensional transforms take advantage of the tensor product basis, $\phi_{n,m}(x, y) = \phi_n(x)\phi_m(y) = e^{2\pi i n x} e^{2\pi i m y}$. In two space variables, the DFT of a two dimensional sequence $\{f_{j,k}\}_{j,k=0}^{N-1,M-1}$ is

$$F_{nm} = \frac{1}{NM} \sum_{k=0}^{M-1} \sum_{j=0}^{N-1} f_{j,k} e^{-2\pi i j n/N} e^{-2\pi i k m/M},$$

$$f_{j,k} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} F_{nm} e^{2\pi i j n/N} e^{2\pi i k m/M}. \tag{2.24}$$

In this section, since it is apropos to most spectral approximations of PDEs, we assume that f is real. Furthermore, we assume that N and M are even.

We can factor the forward transform into

$$F_{nm} = \frac{1}{NM} \sum_{k=0}^{M-1} \left\{ \sum_{j=0}^{N-1} f_{j,k} e^{-2\pi i j n/N} \right\} e^{-2\pi i k m/M}. \tag{2.25}$$

So let us define an intermediate array

$$\bar{F}_{n,k} = \frac{1}{N}\sum_{j=0}^{N-1} f_{j,k} e^{-2\pi i j n/N}, \quad \begin{array}{l} n = 0, 1, \ldots, N-1, \\ k = 0, 1, \ldots, M-1, \end{array} \tag{2.26}$$

so that

$$F_{nm} = \frac{1}{M}\sum_{k=0}^{M-1} \bar{F}_{n,k} e^{-2\pi i k m/M}, \quad \begin{array}{l} j = 0, 1, \ldots, N-1, \\ m = 0, 1, \ldots, M-1. \end{array} \tag{2.27}$$

Thus, the tensor product basis reduces the two dimensional transform to a sequence of one dimensional transforms.

As before, direct application of a complex FFT to compute the two dimensional transform of the real sequence would amount to doing twice as much work and use twice as much storage as is necessary. Fortunately, we can combine the ideas of the previous two sections to develop a more efficient algorithm than that.

Since we assume the initial sequence is real, only half the number of FFTs is required in the first coordinate direction. To that end, let us define the complex sequence

$$z_{j,l} = f_{j,2l} + i f_{j,2l+1}, \quad \begin{array}{l} j = 0, 1, \ldots, N-1, \\ l = 0, 1, \ldots, M/2-1, \end{array} \tag{2.28}$$

whose Fourier transform in the first index is

$$\bar{Z}_{n,l} = \frac{1}{N}\sum_{j=0}^{N-1} \left(f_{j,2l} + i f_{j,2l+1}\right) e^{-2\pi i j n/N}, \quad \begin{array}{l} n = 0, 1, \ldots, N-1, \\ l = 0, 1, \ldots, M/2-1. \end{array} \tag{2.29}$$

Note that the intermediate array $\bar{Z}_{n\,l}$ is half the size of the $\bar{F}$ array. Now, we know the intermediate transform of the original data, namely,

$$\begin{aligned} \bar{F}_{n,2l} &= \frac{1}{2}\left(\bar{Z}_{n,l} + \bar{Z}^*_{n,M-l}\right), \\ \bar{F}_{n,2l+1} &= \frac{-i}{2}\left(\bar{Z}_{n,l} - \bar{Z}^*_{n,M-l}\right), \end{aligned} \quad \begin{array}{l} n = 0, 1, \ldots, N-1, \\ l = 0, 1, \ldots, M/2-1, \end{array} \tag{2.30}$$

but there is no need to compute it at this stage. Instead, we go back to (2.27) and split it into even and odd components as in the previous subsection

$$F_{nm} = \frac{1}{M}\sum_{l=0}^{M/2-1} \left\{ \bar{F}_{n,2l} e^{-2\pi i (2l) m/M} + e^{-2\pi i m/M} \bar{F}_{n,2l+1} e^{-2\pi i (2l) m/M} \right\}. \tag{2.31}$$

Then the transforms to be computed on the second index are the half-length transforms

$$F_{nm} = \frac{1}{M}\sum_{l=0}^{M/2-1} X_{n,l} e^{-2\pi i l m/(M/2)}, \quad \begin{array}{l} n = 0, 1, \ldots, N-1, \\ m = 0, 1, \ldots, M/2-1, \end{array} \tag{2.32}$$

Algorithm 13: *Forward2DFFT:* A Two-Dimensional Forward FFT of a Real Array with an Even Number of Points in Each Direction

Procedure Forward2DFFT
Input: $\left\{f_{j,k}\right\}_{j,k=0}^{N-1,M-1}, \left\{w_j^{+,1}\right\}_{j=0}^{N-1}, \left\{w_j^{+,2}\right\}_{j=0}^{M-1}$
Uses Algorithms:
Algorithm 9 (FFFTOfTwoRealVectors)
Algorithm 8 (Radix2FFT)

for $k = 0$ **to** $M-2$ **step** 2 **do**
  $\left\{\left\{\bar{F}_{n,k}\right\}_{n=0}^{N-1}, \left\{\bar{F}_{n,k+1}\right\}_{n=0}^{N-1}\right\} \leftarrow$
  $FFFTOfTwoRealVectors\left(\left\{f_{j,k}\right\}_{j=0}^{N-1}, \left\{f_{j,k+1}\right\}_{j=0}^{N-1}, \left\{w_j^{+,1}\right\}_{j=0}^{N-1}\right)$
end
for $n = 0$ **to** $N-1$ **do**
  $\left\{F_{n,m}\right\}_{m=0}^{M-1} \leftarrow Radix2FFT\left(\left\{\bar{F}_{n,k}\right\}_{k=0}^{M-1}, \left\{w_j^{+,2}\right\}_{j=0}^{M-1}\right)$
end
for $m = 0$ **to** $M-1$ **do**
  for $n = 0$ **to** $N-1$ **do**
    $F_{n,m} \leftarrow F_{n,m}/M$
  end
end
return $\left\{F_{n,m}\right\}_{n,m=0}^{N-1,M-1}$
End Procedure Forward2DFFT

where the values to be transformed are

$$X_{n,l} = \frac{1}{2}\left(\bar{Z}_{n,l} + \bar{Z}^*_{n,N-l}\right) - \frac{ie^{-2\pi im/M}}{2}\left(\bar{Z}_{n,l} - \bar{Z}^*_{n,N-l}\right). \tag{2.33}$$

We compute the second half of the array from the symmetry $F_{N-n\,M-m} = F^*_{n,m}$. Overall we see that approximately one half the work of the direct use of the complex FFT is done. Under the assumption that the initial data are real and that N and M are even, we use Algorithms 9 (FFFTOfTwoRealVectors) and 10 (BFFTForTwoRealVectors) to compute the two dimensional transforms efficiently. We show the procedures in Algorithm 13 (Forward2DFFT) for the forward transform and Algorithm 14 (Backward2DFFT) for the backward transform.

2.2 The Real Fourier Transform

Though we do not directly need it in typical Fourier spectral computations, we derive the real transform for the purposes of the next chapter where we study Chebyshev transforms. We can compute the real transform from the complex transform,

Algorithm 14: *Backward2DFFT:* Two-Dimensional Backward FFT of a Real Array with an Even Number of Points in Each Direction

Procedure Backward2DFFT
Input: $\{F_{n,m}\}_{n,m=0}^{N-1,M-1}, \{w_j^{-,1}\}_{j=0}^{N-1}, \{w_j^{-,2}\}_{j=0}^{M-1}$
Uses Algorithms:
Algorithm 8 (Radix2FFT)
Algorithm 10 (BFFTForTwoRealVectors)

for $n = 0$ **to** $N-1$ **do**
  $\{F_{n,k}\}_{k=0}^{M-1} \leftarrow Radix2FFT\left(\{\bar{F}_{n,m}\}_{m=0}^{M-1}, \{w_j^{-,2}\}_{j=0}^{M-1}\right)$
end
for $k = 0$ **to** $M-2$ **step** 2 **do**
  $\left\{\{f_{j,k}\}_{j=0}^{N-1}, \{f_{j,k+1}\}_{j=0}^{N-1}\right\} \leftarrow$
  $BFFTForTwoRealVectors\left(\{F_{n,k}\}_{n=0}^{N-1}, \{F_{n,k+1}\}_{n=0}^{N-1}, \{w_j^{-,1}\}_{j=0}^{N-1}\right)$
end
return $\{f_{j,k}\}_{j,k=0}^{N-1,M-1}$
End Procedure Backward2DFFT

too. If N is even, the real transform takes the form

$$x_j = \frac{a_0}{2} + \sum_{k=1}^{N/2-1} \left\{ a_k \cos\left(\frac{2\pi jk}{N}\right) + b_k \sin\left(\frac{2\pi jk}{N}\right) \right\} + \frac{(-1)^j a_{N/2}}{2}. \tag{2.34}$$

In terms of the complex transform,

$$x_j = \sum_{k=0}^{N-1} X_k e^{2\pi ijk/N} = \sum_{k=0}^{N/2-1} X_k e^{2\pi ijk/N} + \sum_{k=1}^{N/2} X_{N-k} e^{2\pi ij(N-k)/N}. \tag{2.35}$$

Since $X_k = X_{N-k}^*$,

$$x_j = X_0 + 2 \sum_{k=0}^{N/2-1} \mathrm{Re}\left\{ X_k e^{2\pi ijk/N} \right\} + (-1)^j X_{N/2}. \tag{2.36}$$

Using the fact that

$$\mathrm{Re}\left\{ X_k e^{2\pi ijk/N} \right\} = \mathrm{Re}(X_k) \cos\left(\frac{2\pi jk}{N}\right) - \mathrm{Im}(X_k) \sin\left(\frac{2\pi jk}{N}\right), \tag{2.37}$$

$$x_j = X_0 + 2 \sum_{k=0}^{N/2-1} \left\{ \mathrm{Re}(X_k) \cos\left(\frac{2\pi jk}{N}\right) - \mathrm{Im}(X_k) \sin\left(\frac{2\pi jk}{N}\right) \right\} + (-1)^j X_{N/2}. \tag{2.38}$$

Algorithm 15: *ForwardRealFFT:* The Forward Real Transform

Procedure ForwardRealFFT
Input: $\{x_j\}_{j=0}^{N-1}, \{w_j^+\}_{j=0}^{N-1}$
Uses Algorithms:
Algorithm 11 (FFFTEO)
$\{X_k\}_{k=0}^{N-1} \leftarrow FFFTEO\left(\{x_j\}_{j=0}^{N-1}, \{w_j^+\}_{j=0}^{N-1}\right)$
for $k = 0$ **to** $N/2$ **do**
  $a_k \leftarrow 2\,\mathrm{Re}(X_k)$
  $b_k \leftarrow -2\,\mathrm{Im}(X_k)$
end
$b_0 \leftarrow 0$
$b_{N/2} \leftarrow 0$
return $\{a_k\}_{k=0}^{N/2}, \{b_k\}_{k=0}^{N/2}$
End Procedure ForwardRealFFT

Algorithm 16: *BackwardRealFFT*: The Backward Real Transform

Procedure BackwardRealFFT
Input: $\{a_k\}_{k=0}^{N/2}, \{b_k\}_{k=0}^{N/2}$
Uses Algorithms:
Algorithm 12 (BFFTEO)
Algorithm 7 (InitializeFFT)
$\left\{w_j^+\right\}_{j=0}^{N/2-1} \leftarrow InitializeFFT(N/2, FORWARD)$
$X_0 \leftarrow a_0/2$
$X_{N/2} \leftarrow a_{N/2}/2$
for $k = 1$ **to** $N/2 - 1$ **do**
  $X_k \leftarrow (a_k + ib_k)/2$
end
for $k = 1$ **to** $N/2 - 1$ **do**
  $X_{N-k} \leftarrow X_k^*$
end
$\left\{x_j\right\}_{j=0}^{N-1} \leftarrow BFFTEO(\{X_k\}_{k=0}^{N-1}, \{w_k^+\}_{k=0}^{N-1})$
return $\left\{x_j\right\}_{j=0}^{N-1}$
End Procedure BackwardRealFFT

When we match terms with (2.34) we get the relation between the coefficients of the real and complex transforms

$$\begin{aligned} a_k &= 2\,\mathrm{Re}(X_k), \\ b_k &= -2\,\mathrm{Im}(X_k), \end{aligned} \qquad k = 0, 1, \ldots, N/2 - 1. \tag{2.39}$$

Thus, we can compute the forward and backward real transforms from the complex transform, and we show the procedures in Algorithms 15 (ForwardRealFFT) and 16 (BackwardRealFFT). Note that half of the X_k's returned by the forward FFT are

not used in Algorithm 15. We can get additional savings if we explicitly incorporate Algorithm 11 (FFFTEO) minus the final loop that produces that second half of the coefficients (Problem 2.2).

2.3 How to Evaluate the Fourier Interpolation Derivative by FFT

For large N, it is efficient to use the FFT to compute the derivative of Fourier interpolants written in the form (1.73). Exactly what is that value of N is very implementation dependent. It depends on the FFT code, the architecture of the machine and the matrix multiplication code. Crossover points, where the FFT becomes more efficient than the matrix multiplication method that we discuss in the next section, have been reported to vary from eight to 128. Before deciding on which method to incorporate into production codes, it is probably best to program both and test for the particular computer architecture to be used.

To use the FFT to compute the derivative of the interpolant at the nodes, we must put the derivative in the form of the DFT. Let's rewrite the derivative of the interpolant by separating the $N/2$ mode

$$(I_N f)'(x_j) = \sum_{k=-N/2}^{N/2} \frac{ik\tilde{f}_k}{\bar{c}_k} e^{ikx_j} = \sum_{k=-N/2}^{N/2-1} \frac{ik\tilde{f}_k}{\bar{c}_k} e^{ikx_j} + \frac{i(\frac{N}{2})\tilde{f}_{N/2}}{\bar{c}_{N/2}} e^{i(\frac{N}{2})x_j}. \quad (2.40)$$

Since $\tilde{f}_{N/2} = \tilde{f}_{-N/2}$ and $e^{i(N/2)x_j} = e^{-i(N/2)x_j}$, we can rewrite the derivative as

$$\begin{aligned}(I_N f)'(x_j) &= \sum_{k=-N/2}^{N/2-1} \frac{ik\tilde{f}_k}{\bar{c}_k} e^{ikx_j} - \frac{i(\frac{-N}{2})\tilde{f}_{-N/2}}{\bar{c}_{-N/2}} e^{i(-\frac{N}{2})x_j} \\ &= \sum_{k=-N/2+1}^{N/2-1} ik\tilde{f}_k e^{ikx_j}. \end{aligned} \quad (2.41)$$

Therefore, we can use the DFT to evaluate the derivative of the Fourier interpolant at the nodes by setting $\tilde{f}_{-N/2} = 0$.

We show how to compute the interpolation derivative using the FFT in Algorithm 17 (FourierDerivativeByFFT). The procedure first computes the discrete Fourier coefficients by Algorithm 11 (FFFTEO), with the assumption that the exponential factors have been pre-computed and stored. It then uses the correspondence we made in (2.3) to compute the discrete Fourier coefficients of the derivative. Finally, it computes the derivative values at the nodes by the backward FFT, Algorithm 12 (BFFTEO). Note that the procedure sets the $-N/2$ coefficient to zero to represent (2.41).

Algorithm 17 easily generalizes to higher order derivatives. To modify Algorithm 17 to compute the approximation of the mth derivative, we merely need to

Algorithm 17: *FourierDerivativeByFFT:* Fast Evaluation of the Fourier Polynomial Derivative

Procedure FourierDerivativeByFFT
Input: $\{f_j\}_{j=0}^{N-1}, \{w_j^+\}_{j=0}^{N-1}, \{w_j^-\}_{j=0}^{N-1}$
Uses Algorithms:
Algorithm 11 (FFFTEO)
Algorithm 12 (BFFTEO)
$\{F_j\}_{j=0}^{N-1} \leftarrow FFFTEO(\{F_j\}_{j=0}^{N-1}, \{w_j^+\}_{j=0}^{N-1})$
for $k = 0$ **to** $N/2 - 1$ **do**
  $F_k \leftarrow ik * F_k$
end
$F_{-N/2} \leftarrow 0$
for $k = N/2 + 1$ **to** $N - 1$ **do**
  $F_k \leftarrow i(k - N) * F_k$
end
$\{Df_j\}_{j=0}^{N-1} \leftarrow BFFTEO(\{F_j\}_{j=0}^{N-1}, \{w_j^-\}_{j=0}^{N-1})$
return $\{(Df)_j\}_{j=0}^{N-1}$
End Procedure FourierDerivativeByFFT

add m to the input list and replace ik by $(ik)^m$ in the first loop and $i(k - N)$ by $(i(k - N))^m$ in the second. However, we do not set $\tilde{f}_{-N/2} = 0$ for even derivatives. We leave the difference between even and odd derivatives to Problem 2.4. The difference also implies that computing the first derivative twice is not the same as computing the second derivative (Problem 2.5).

2.4 How to Compute Derivatives by Matrix Multiplication

For small enough N, we can compute the derivative of the Fourier interpolant at the interpolation points efficiently by matrix vector multiplication. Differentiation of the interpolant at the nodes is

$$(I_N f)'_n = \sum_{j=0}^{N-1} D_{nj} f_j. \tag{2.42}$$

The matrix D, whose elements are D_{nj}, is called the *Fourier derivative matrix*. Formally, the elements are the values of $h'_j(x_n)$ presented in (1.75). In practice, we pre-compute and store this matrix.

The first issue in the use of matrix multiplication to compute the Fourier derivative approximation is how to construct the derivative matrix itself. The construction of spectral derivative matrices has been the subject of much discussion since it was noticed that derivatives computed with the Chebyshev differentiation matrix (which we discuss in Sect. 3.5) were very sensitive to rounding errors. To reduce the effects of rounding errors, several modifications to the matrices have been proposed,

Algorithm 18: *FourierDerivativeMatrix:* Computation of the Fourier Derivative Matrix Using the Negative Sum Trick

Procedure FourierDerivativeMatrix
Input: N
for $i = 0$ **to** $N - 1$ **do**
  $D_{i,i} \leftarrow 0$
  for $j = 0$ **to** $N - 1$ **do**
    if $j \neq i$ **then**
      $D_{i,j} \leftarrow \frac{1}{2}(-1)^{i+j} \cot\left[\frac{(i-j)\pi}{N}\right]$
      $D_{i,i} \leftarrow D_{i,i} - D_{i,j}$
    end
  end
end
return $\{D_{i,j}\}_{i,j=0}^{N-1}$
End Procedure FourierDerivativeMatrix

including preconditioning, use of the symmetry properties, and the use of what has come to be called the "Negative Sum Trick" to compute the diagonal entries. Comparisons of the numerous approaches have favored the use of the Negative Sum Trick to evaluate the derivative matrix.

The Negative Sum Trick comes from the observation that the derivative of a constant must be zero. This means that $\sum_{j=0}^{N-1} D_{nj} = 0$ for $n = 0, 1, \ldots, N-1$. To enforce that condition we compute the diagonal elements to satisfy it explicitly, i.e. we evaluate the diagonal entries of the matrix to satisfy

$$D_{nn} = -\sum_{\substack{j=0 \\ j \neq n}}^{N-1} D_{nj}. \tag{2.43}$$

The diagonal elements computed with (2.43) will not be exactly equal to zero, but overall the effects of rounding errors are minimized.

Algorithm 18 (FourierDerivativeMatrix) shows how to pre-compute the Fourier derivative matrix using the Negative Sum Trick. For the best roundoff error properties, however, the diagonal entries should be computed separately, after the rest of the matrix has been computed, since the order in which the sum (2.43) is computed is important. The off-diagonal terms should be sorted and summed from smallest in magnitude to largest. For the sake of simplicity, we present Algorithm 18 without the ordered sum.

The next issue is to decide how to implement the matrix multiplication. Since this is also an issue for the polynomial approximations discussed in the next chapter, and is not specifically related to spectral methods *per se*, we leave the detailed discussion of matrix multiplication for later. If efficiency is not of the utmost importance, of course, we would use the standard implementation shown in Algorithm 19 (MxVDerivative), with $s = 0$ and $e = N - 1$. That algorithm takes the derivative

Algorithm 19: *MxVDerivative:* A Matrix-Vector Multiplication Procedure

Procedure MxVDerivative
Input: $\{D_{i,j}\}_{i,j=s}^{e}$, $\{f_j\}_{j=s}^{e}$
for $i = s$ **to** e **do**
  $t = 0$
  for $j = s$ **to** e **do**
    $t \leftarrow t + D_{i,j} * f_j$
  end
  $(I_N f)'_i \leftarrow t$
end
return $\{(I_N f)'_i\}_{i=s}^{e}$
End Procedure MxVDerivative

matrix, pre-computed by Algorithm 18 (FourierDerivativeMatrix) and the sequence of values at the nodes, and returns derivative of the interpolant, evaluated at the nodes.

Exercises

2.1 Compare the speed of the simple, direct DFT, Algorithm 6 to the FFT, Algorithm 8 for various values of N. Above what value of N is the FFT faster? Are there any advantages that the direct DFT has over the FFT? Make the same comparisons to a library FFT.

2.2 Half of the X_k's returned by the forward FFT are not used in Algorithm 15 (ForwardRealFFT). Modify the procedure to get additional savings by explicitly incorporating Algorithm 11 (FFFTEO) minus the final loop that produces that second half of the coefficients.

2.3 Compute and plot as a function of x the Fourier interpolation derivative of the functions

$$f(x) = \sin(x/2),$$
$$f(x) = e^{\sin(x)}$$

on the interval $[0, 2\pi]$ for several values of N. Observe the behavior or the derivative approximations and explain their differences. Also, plot the maximum error at the nodes as a function of N and compare.

2.4 Show that

$$(I_N f)^{(m)}(x_j) = \begin{cases} \sum_{k=-N/2+1}^{N/2-1} (ik)^m \tilde{f}_k e^{ikx_j}, & m \text{ odd}, \\ \sum_{k=-N/2}^{N/2-1} (ik)^m \tilde{f}_k e^{ikx_j}, & m \text{ even}. \end{cases}$$

Use the result to modify Algorithm 17 (FourierDerivativeByFFT) to compute mth order derivatives.

2.5 Use the result of Problem 2.4 to show that taking the first derivative twice is not the same as taking the second derivative. By how much do they differ? How significant is that difference for smooth functions?

2.6 For what values of N is it faster to compute the Fourier interpolation derivative of a function by FFT than by matrix multiplication?

Chapter 3
Algorithms for Non-Periodic Functions

For non-periodic boundary value problems, we use expansions in orthogonal polynomials instead of the complex exponentials to approximate solutions, i.e.,

$$P(x) = \sum_{n=0}^{N} \hat{p}_n \phi_n(x). \tag{3.1}$$

The basis functions, ϕ_n, are typically chosen to be Jacobi polynomials, of which the popular Legendre, $L_n(x)$, or Chebyshev polynomials, $T_n(x)$, are special cases. Polynomial bases are attractive because they are smooth and easy to compute. With proper choice of the polynomials, approximations converge spectrally fast without restriction on the boundary conditions to be imposed, unlike Fourier approximations, which require that the function and all its derivatives to be periodic.

In this chapter, we develop the basic algorithms that we need to compute Chebyshev and Legendre spectral approximations to PDEs. We start with how to compute the Legendre and Chebyshev polynomials themselves. We use those algorithms to compute the Gauss quadrature nodes and weights (at least for the Legendre polynomials) that we need to compute interpolants and discrete inner products. The chapter ends with algorithms that we use to approximate the derivatives that appear in PDEs.

3.1 How to Compute the Legendre and Chebyshev Polynomials

The three-term recursions (1.82) and (1.86) make Legendre and Chebyshev polynomials mathematically easy and efficient to compute. As a historical note, the basic algorithm was presented by Galler in 1960 as the Transactions on Mathematical Software (TOMS) Algorithm 13. We will compute the Legendre Polynomial of order k at the point x with Algorithm 20 (LegendrePolynomial). For efficiency, we might want to return special cases for more than just the L_0 and L_1 values. The cutoff point will depend on the implementation.

We can compute the Chebyshev polynomials either by recursion (1.86) or by using the trigonometric form (1.85) directly. The cosine and its inverse are very expensive to compute, however, so for small enough k, recursion may be significantly faster than the trigonometric form. Algorithm 21 (ChebyshevPolynomial) computes the Chebyshev polynomial of degree k either by recursion or by the trigonometric form of the polynomial. The crossover value of K_s, where it becomes more efficient to use the trigonometric form, will depend on the implementation. (The Chebyshev recursion algorithm was also presented in 1960 as TOMS 10.)

Although Algorithms 20 (LegendrePolynomial) and 21 (ChebyshevPolynomial) are quite simple, practical issues of efficiency and numerical stability arise. First, the

D.A. Kopriva, *Implementing Spectral Methods for Partial Differential Equations,* Scientific Computation,

Algorithm 20: *LegendrePolynomial:* Evaluate the Legendre Polynomial of Degree k using Three Term Recursion

Procedure LegendrePolynomial
Input: k, x
if $k = 0$ **then return** 1
if $k = 1$ **then return** x
$L_{k-2} \leftarrow 1$
$L_{k-1} \leftarrow x$
for $j = 2$ **to** k **do**
  $L_k \leftarrow \frac{2j-1}{j} x L_{k-1}(x) - \frac{j-1}{j} L_{k-2}$
  $L_{k-2} \leftarrow L_{k-1}$
  $L_{k-1} \leftarrow L_k$
end
return L_k
End Procedure LegendrePolynomial

Algorithm 21: *ChebyshevPolynomial:* The Chebyshev Polynomial of Degree k using Three Term Recursion and Trigonometric Forms

Procedure ChebyshevPolynomial
Input: k, x
if $k = 0$ **then return** 1
if $k = 1$ **then return** x
if $k \leq K_s$ **then**
  $T_{k-2} \leftarrow 1$
  $T_{k-1} \leftarrow x$
  for $j = 2$ **to** k **do**
    $T_k \leftarrow 2xT_{k-1} - T_{k-2}$
    $T_{k-2} \leftarrow T_{k-1}$
    $T_{k-1} \leftarrow T_k$
  end
else
  $T_k \leftarrow \cos(k \cos^{-1}(x))$
end
return T_k
End Procedure ChebyshevPolynomial

choice of an efficient algorithm will be strongly dependent on computer architecture and code. To compute Chebyshev polynomials, for instance, Algorithm 21 uses a switch from the recursion to the trigonometric form at some value of $k = K_s$ after which the trigonometric form is more efficient. Figure 3.1 compares the time it takes to compute $T_k(0.667)$ as a function of k using recursion and using the trigonometric form. From this simple exercise, we see that the crossover point, K_s, changes from about six to 70—over an order of magnitude—simply by turning on the optimizer. We might just as easily expect differences of such magnitudes if we change from one compiler to another, or from one computer architecture to another. The moral is that even the simplest algorithms should be tested for efficiency.

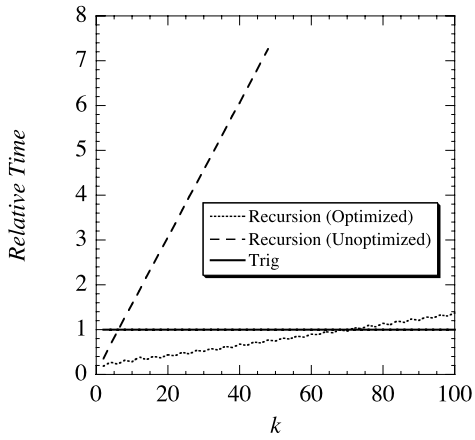

Fig. 3.1 Computing time relative to the trigonometric form (1.85) to compute $T_k(x)$. Considered are the recursion (1.86) with optimizer on and off

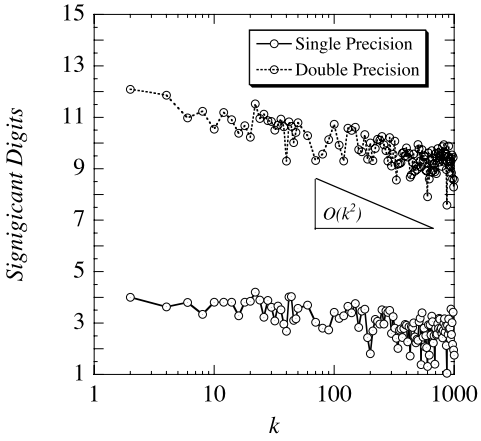

Fig. 3.2 Number of significant digits for the recursive computation of $T_k(x)$ as a function of polynomial order, k

We must also consider stability when we compute functions by recursion. Let's look at the recursion for the Chebyshev polynomials first. For $|x| > 1/2$ the coefficient of the leading order polynomial is larger than one. This makes the algorithm unstable for large k since (rounding) errors in the lower order polynomials are amplified. Near roots of the polynomials, which we will need to find later in Sect. 3.2 to compute the Gauss quadrature points and weights, the cancellation between the two terms can be catastrophic.

To get a sense of the significance of the errors when we compute Chebyshev polynomials of large degree, let's compare the number of significant digits in T_k over the interval $[-1, 1]$ for the recursion algorithm in both single and double precision arithmetic compared to the trigonometric formula. The behavior of Algorithm 20 (LegendrePolynomial) should be similar since as k gets large the coefficients of the Legendre recursion approach the those of the Chebyshev recursion formula.

We show a graph of the minimum number of significant digits on $[-1, 1]$ as a function of the Chebyshev polynomial order in Fig. 3.2 for k between two and 1000. We see that even at low orders there is a significant loss of precision for both single

and double precision evaluations. In single precision, recursion is essentially useless for $N \geq 100$. By $N = 1000$ the double precision computation approaches single precision accuracy. Thus, for both accuracy and efficiency, the Chebyshev polynomials should be computed using Algorithm 21 (ChebyshevPolynomial) with K_s no larger than about 70. The Legendre polynomials do not have a simple alternative expression, so beware. For a discussion of how to compute Legendre polynomials for large k, see [26].

3.2 How to Compute the Gauss Quadrature Nodes and Weights

We next develop algorithms to compute the Legendre Gauss and Gauss-Lobatto points and weights. As we saw in Sect. 1.11, the Chebyshev Gauss and Gauss-Lobatto quadrature points and weights are easy to compute.

We could use one of several methods to compute the Legendre Gauss quadrature nodes and weights. A common one that is suitable for small N solves an eigenvalue problem where the nodes are the eigenvalues and the weights are computed from the eigenvectors. Methods that are good for very large N (> 1000) are presented in papers by Yakimiw [26] and Swarztrauber [22].

We will develop algorithms like those of Yakimiw and Swartztrauber that solve for the nodes as roots of the appropriate Legendre polynomial, but without the special (and more complicated) forms needed to minimize the effects of rounding errors. Even so, our algorithms will compute the nodes to 15 significant digits for orders up to at least $N = 200$ when IEEE double precision arithmetic is used, as verified by 32 digit floating point computations with the same algorithms. The weights are less precise. (Yakimiw has a detailed explanation why.) We find the Gauss weights to within 12 significant digits in double precision arithmetic, and the Gauss-Lobatto weights to within 13 digits, again for $N \leq 200$.

3.2.1 Legendre Gauss Quadrature

The most straightforward way to find the Gauss nodes and weights uses the result derived in Sect. 1.11 that the quadrature points of the $N + 1$ point Legendre Gauss quadrature are the zeros of L_{N+1}. We can use a simple rootfinding procedure to find the roots to sufficient precision. We then find the weights from the derivatives of the Legendre polynomials at the nodes via (1.127).

To find the quadrature points, we use Newton's method to solve for the roots of L_{N+1}. Let x_j^k be the kth iterate for the jth root. Newton's method updates the iterate by

$$x_j^{k+1} = x_j^k - \frac{L_{N+1}(x_j^k)}{L'_{N+1}(x_j^k)}. \tag{3.2}$$

We compute the Legendre polynomial of degree $N + 1$ in (3.2) from the recursion relation (1.82) by Algorithm 20 (LegendrePolynomial). The derivatives of

Algorithm 22: *LegendrePolynomialAndDerivative:* The Legendre Polynomial of Degree k and Its Derivative using the Three Term Recursion

Procedure LegendrePolynomialAndDerivative
Input: N, x
if $N = 0$ **then**
  $L_N \leftarrow 1$
  $L'_N \leftarrow 0$
else if $N = 1$ **then**
  $L_N \leftarrow x$
  $L'_N \leftarrow 1$
else
  $L_{N-2} \leftarrow 1$
  $L_{N-1} \leftarrow x$
  $L'_{N-2} \leftarrow 0$
  $L'_{N-1} \leftarrow 1$
  for $k = 2$ **to** N **do**
    $L_N \leftarrow \frac{2k-1}{k} x L_{N-1}(x) - \frac{k-1}{k} L_{N-2}$
    $L'_N \leftarrow L'_{N-2} + (2k-1) L_{N-1}$
    $L_{N-2} \leftarrow L_{N-1}$
    $L_{N-1} \leftarrow L_N$
    $L'_{N-2} \leftarrow L'_{N-1}$
    $L'_{N-1} \leftarrow L'_N$
  end
end
return $L_N, L'_N(x)$
End Procedure LegendrePolynomialAndDerivative

the Legendre polynomials satisfy the recursion (1.93). For efficiency, we should compute the Legendre polynomial and its derivative simultaneously. We present a procedure to compute L_{N+1} and L'_{N+1} at the same time in Algorithm 22 (LegendrePolynomialAndDerivative).

Newton's method requires an initial guess for the roots. A simple approximation that works for small N is to use the Chebyshev Gauss points (with a sign change to make the points vary from left to right)

$$x_j^0 = -\cos\left(\frac{2j+1}{2N+2}\pi\right), \quad j = 0, 1, \ldots, N. \tag{3.3}$$

Better starting values use asymptotic representations for the roots [26].

Algorithm 23 (LegendreGaussNodesAndWeights) computes the nodes and weights for the Legendre Gauss quadrature. For efficiency, we use the fact that the roots of the Legendre polynomials are symmetric about the origin so we only need to compute half of them. If N is even, the origin is one of the nodes. We use a special case to set that. We use (3.3) for the starting values of the Newton iteration. Even with this rough starting value, the number of iterations, n_{it}, is small. For $N \leq 200$, $n_{it} \leq 4$ to get double precision accuracy. We choose the stopping criterion for the iteration so that the relative error in the root is within a small factor of the

Algorithm 23: *LegendreGaussNodesAndWeights:*

Procedure LegendreGaussNodesAndWeights
Input: N
Uses Algorithms:
Algorithm 22 (LegendrePolynomialAndDerivative)

if $N = 0$ **then**
  $x_0 \leftarrow 0$
  $w_0 \leftarrow 2$
else if $N = 1$ **then**
  $x_0 \leftarrow -\sqrt{1/3}$
  $w_0 \leftarrow 1$
  $x_1 \leftarrow -x_0$
  $w_1 \leftarrow w_0$
else
  for $j = 0$ **to** $\lfloor (N+1)/2 \rfloor - 1$ **do**
    $x_j \leftarrow -\cos\left(\frac{2j+1}{2N+2}\pi\right)$
    for $k = 0$ **to** n_{it} **do**
      $\left\{L_{N+1}, L'_{N+1}\right\} \leftarrow LegendrePolynomialAndDerivative(N+1, x_j)$
      $\Delta \leftarrow -L_{N+1}/L'_{N+1}$
      $x_j \leftarrow x_j + \Delta$
      if $|\Delta| \leq TOL * |x_j|$ **then** Exit
    end
    $\left\{L_{N+1}, L'_{N+1}\right\} \leftarrow LegendrePolynomialAndDerivative(N+1, x_j)$
    $x_{N-j} \leftarrow -x_j$
    $w_j \leftarrow \frac{2}{(1-x_j^2)[L'_{N+1}]^2}$
    $w_{N-j} \leftarrow w_j$
  end
end
if $N \bmod 2 = 0$ **then**
  $\left\{L_{N+1}, L'_{N+1}\right\} \leftarrow LegendrePolynomialAndDerivative(N+1, 0.0)$
  $x_{N/2} \leftarrow 0$
  $w_{N/2} \leftarrow \frac{2}{[L'_{N+1}]^2}$
end
return $\left\{x_j\right\}_{j=0}^{N}, \left\{w_j\right\}_{j=0}^{N}$
End Procedure LegendreGaussNodesAndWeights

machine epsilon, typically $TOL = 4\epsilon$. (Appendix B.) We use special cases for $N = 0$ and $N = 1$ where it is simply easier to compute the nodes and weights analytically.

3.2.2 Legendre Gauss-Lobatto Quadrature

The Gauss-Lobatto points include the endpoints of the interval, $x = \pm 1$. The interior points (see Sect. 1.11) are the zeros of L'_N. Equivalently, the derivative recursion

formula (1.83) tells us that the interior nodes are the roots of the polynomial

$$q(x) = L_{N+1} - L_{N-1}. \tag{3.4}$$

The weights are found by

$$w_j = \frac{2}{N(N+1)[L_N(x_j)]^2}. \tag{3.5}$$

To find the roots, we again use Newton's method,

$$x_j^{k+1} = x_j^k - \frac{q(x_j^k)}{q'(x_j^k)}. \tag{3.6}$$

For each iteration at an interior point, we need q, its derivative, and L_N. These we compute simultaneously using the recursion relations for the Legendre polynomials and their derivatives, much as we did in Algorithm 22 (LegendrePolynomialAndDerivative). We present an auxiliary procedure to compute these values in Algorithm 24 (qAndLEvaluation).

Algorithm 24: *qAndLEvaluation:* Combined Algorithm to Compute $L_N(x)$, $q(x) = L_{N+1} - L_{N-1}$, and $q'(x)$

Procedure qAndLEvaluation
Input: N, x
$k = 2$
$L_{N-2} \leftarrow 1$
$L_{N-1} \leftarrow x$
$L'_{N-2} \leftarrow 0$
$L'_{N-1} \leftarrow 1$
for $k = 2$ **to** N **do**
  $L_N \leftarrow \frac{2k-1}{k} x L_{N-1}(x) - \frac{k-1}{k} L_{N-2}$
  $L'_N \leftarrow L'_{N-2} + (2k-1) L_{N-1}$
  $L_{N-2} \leftarrow L_{N-1}$
  $L_{N-1} \leftarrow L_N$
  $L'_{N-2} \leftarrow L'_{N-1}$
  $L'_{N-1} \leftarrow L'_N$
end
$k \leftarrow N+1$
$L_{N+1} \leftarrow \frac{2k-1}{k} x L_N(x) - \frac{k-1}{k} L_{N-1}$
$L'_{N+1} \leftarrow L'_{N-1} + (2k-1) L_N$
$q \leftarrow L_{N+1} - L_{N-1}$
$q' \leftarrow L'_{N+1} - L'_{N-1}$
return $q(x), q'(x), L_N(x)$
End Procedure qAndLEvaluation

As with the Gauss quadrature computation, we need an initial value for each root to start the Newton iteration. For this, we use an asymptotic relation due to Parter

$$x_j^0 = -\cos\left\{\frac{(j+1/4)\pi}{N} - \frac{3}{8N\pi}\frac{1}{j+1/4}\right\}, \quad j = 1, 2, \ldots, N-1. \tag{3.7}$$

We show the final procedure to find the Gauss-Lobatto nodes and weights in Algorithm 25 (LegendreGaussLobattoNodesAndWeights).

Algorithm 25: *LegendreGaussLobattoNodesAndWeights:*

Procedure LegendreGaussLobattoNodesAndWeights
Input: N
Uses Algorithms:
Algorithm 24 (qAndLEvaluation)

if $N = 1$ **then**
  $x_0 \leftarrow -1$
  $w_0 \leftarrow 1$
  $x_1 \leftarrow 1$
  $w_1 \leftarrow w_0$
end
else
  $x_0 \leftarrow -1$
  $w_0 \leftarrow 2/(N(N+1))$
  $x_N \leftarrow 1$
  $w_N \leftarrow w_0$
  for $j = 1$ **to** $\lfloor (N+1)/2 \rfloor - 1$ **do**
    $x_j \leftarrow -\cos\left\{\frac{(j+1/4)\pi}{N} - \frac{3}{8N\pi}\frac{1}{j+1/4}\right\}$
    for $k = 0$ **to** n_{it} **do**
      $\{q, q', L_N\} \leftarrow qAndLEvaluation(N, x_j)$
      $\Delta \leftarrow -q/q'$
      $x_j \leftarrow x_j + \Delta$
      if $|\Delta| \leq TOL * |x_j|$ **then** Exit
    end
    $\{q, q', L_N\} \leftarrow qAndLEvaluation(N, x_j)$
    $x_{N-j} \leftarrow -x_j$
    $w_j \leftarrow \frac{2}{N(N+1)\,[L_N]^2}$
    $w_{N-j} \leftarrow w_j$
  end
end
if $N \bmod 2 = 0$ **then**
  $\{q, q', L_N\} \leftarrow qAndLEvaluation(N, 0.0)$
  $x_{N/2} \leftarrow 0$
  $w_{N/2} \leftarrow \frac{2}{N(N+1)\,[L_N]^2}$
end
return $\{x_j\}_{j=0}^N, \{w_j\}_{j=0}^N$
End Procedure LegendreGaussLobattoNodesAndWeights

Table 3.1 First four Legendre nodes and weights for $N = 6$

j	Gauss		Gauss-Lobatto	
	x_j	w_j	x_j	w_j
0	−0.949107912342759	0.129484966168870	−1.00000000000000	4.761904761904762E−002
1	−0.741531185599395	0.279705391489277	−0.830223896278567	0.276826047361566
2	−0.405845151377397	0.381830050505119	−0.468848793470714	0.431745381209863
3	0.000000000000000	0.417959183673469	0.000000000000000	0.487619047619048

Algorithm 26: *ChebyshevGaussNodesAndWeights:*

Procedure ChebyshevGaussNodesAndWeights
Input: N
for $j = 0$ **to** N **do**
  $x_j \leftarrow -\cos\left(\frac{2j+1}{2N+2}\pi\right)$
  $w_j \leftarrow \frac{\pi}{N+1}$
end
return $\{x_j\}_{j=0}^{N}, \{w_j\}_{j=0}^{N}$
End Procedure ChebyshevGaussNodesAndWeights

3.2.2.1 Benchmark Solution: Legendre Nodes and Weights

For benchmark purposes, we include Table 3.1 to show the double precision computed values of the nodes and weights for both the Legendre Gauss and Legendre Gauss-Lobatto quadrature when $N = 6$. For further comparisons, see the tables in [1].

3.2.3 Chebyshev Gauss Quadratures

For completeness, and since we will use them later to compute Chebyshev spectral approximations, we include two algorithms, Algorithms 26 (ChebyshevGaussNodesAndWeights) and 27 (ChebyshevGaussLobattoNodesAndWeights) to compute the nodes and weights for the Chebyshev quadratures. Notice that we have reordered the nodes to increase from left to right with index j. We store the constant weights in an array to allow us later to swap between Chebyshev and Legendre approximations. The Chebyshev nodes and weights are clearly much easier to compute than the Legendre quantities.

3.3 How to Evaluate Chebyshev Interpolants via the FFT

If we were to evaluate the discrete transform pair (1.141) and (1.142) we'd have to perform $O(N^2)$ operations to compute the value of the interpolant at each of the

Algorithm 27: *ChebyshevGaussLobattoNodesAndWeights:*

Procedure ChebyshevGaussLobattoNodesAndWeights
Input: N
for $j = 0$ **to** N **do**
  $x_j \leftarrow -\cos\left(\frac{j}{N}\pi\right)$
  $w_j \leftarrow \frac{\pi}{N}$
end
$w_0 \leftarrow w_0/2$
$w_N \leftarrow w_N/2$
return $\{x_j\}_{j=0}^N, \{w_j\}_{j=0}^N$
End Procedure ChebyshevGaussLobattoNodesAndWeights

nodes. We now show how to use FFT techniques to compute both the coefficients and the nodal values of the Chebyshev interpolant through the Gauss-Lobatto points. The trigonometric representation (1.85) makes FFT techniques natural for Chebyshev interpolants. Unfortunately, there is no equivalent fast transform for Legendre approximations.

3.3.1 The Fast Chebyshev Transform

We can evaluate discrete Chebyshev transforms with an FFT because they can be reduced to Discrete Cosine Transforms (DCT's). The Chebyshev polynomial interpolation of a function, f, in one space dimension is

$$F(x) = \sum_{k=0}^{N} \tilde{f}_k T_k(x). \tag{3.8}$$

We define the Chebyshev coefficients, $\tilde{f}_k$ so that $F(x_j) = f(x_j)$, using (1.141). For the specific case of Chebyshev polynomials, (1.141) is

$$\tilde{f}_k = \frac{\sum_{j=0}^{N} f_j T_k(x_j) w_j}{\sum_{j=0}^{N} T_k^2(x_j) w_j}, \tag{3.9}$$

where the $\{x_j\}_{j=0}^N$ are a set of either Gauss or Gauss-Lobatto quadrature points and the $\{w_j\}_{j=0}^N$ are the corresponding weights. This pair of formulas defines the *Discrete Chebyshev Transform.*

The most commonly used quadrature points in Chebyshev spectral methods are the Gauss-Lobatto points. When we evaluate the Chebyshev polynomials at those points,

$$T_k(x_j) = \cos(jk\pi/N). \tag{3.10}$$

When we substitute (3.10) and the values of the weights into (3.8) and (3.9), the Discrete Chebyshev Transform reduces at the Gauss-Lobatto points to the *Discrete Cosine Transform* (DCT)

$$\begin{aligned} \tilde{f}_k &= \frac{2}{N\bar{c}_k}\sum_{j=0}^{N}\frac{f_j}{\bar{c}_j}\cos\left(\frac{jk\pi}{N}\right), \quad k=0,1,\ldots,N, \\ f_j &= \sum_{k=0}^{N}\tilde{f}_k\cos\left(\frac{jk\pi}{N}\right), \quad j=0,1,\ldots,N, \end{aligned} \tag{3.11}$$

where

$$\bar{c}_j = \begin{cases} 2, & j=0,N, \\ 1, & j=1,2,\ldots,N-1. \end{cases} \tag{3.12}$$

We compute Discrete Cosine Transforms efficiently with the FFT. To develop an FFT algorithm for the DCT we first show that if the sequence to be transformed is even, the coefficients b_k in the real Fourier transform (2.34) vanish, leaving the cosine series. We will then use the algorithm that we developed for the real Fourier transform to compute the DCT, which we will then use to compute the discrete Chebyshev transform.

To develop the fast Chebyshev transform, we first show that the Fourier coefficients of a real, even sequence $\{\bar{f}_j\}_{j=0}^{2N-1}$ are real. Recall that

$$\begin{aligned} F_k &= \frac{1}{2N}\sum_{j=0}^{2N-1}\bar{f}_j e^{-\pi ijk/N}, \\ F_k^* &= \frac{1}{2N}\sum_{j=0}^{2N-1}\bar{f}_j e^{\pi ijk/N}. \end{aligned} \tag{3.13}$$

If the sequence $\{\bar{f}_j\}_{j=0}^{2N-1}$ is even, i.e., $\bar{f}_j = \bar{f}_{2N-j}$, then

$$F_k^* = \frac{1}{2N}\sum_{j=0}^{2N-1}\bar{f}_{2N-j}e^{\pi ijk/N}. \tag{3.14}$$

We now make the change of variable $l = 2N - j$ to see that

$$F_k^* = \frac{1}{2N}\sum_{l=2N}^{1}\bar{f}_l e^{\pi i(2N-l)k/N} = \frac{1}{2N}\sum_{l=1}^{2N}\bar{f}_l e^{-\pi ilk/N}e^{2\pi i} = F_k \tag{3.15}$$

since periodicity of the coefficients means that $F_0 = F_{2N}$.

Next, we show that the real Fourier transform of an even function is the cosine transform. From (2.34), the real transform of $\{\bar{f}_j\}_{j=0}^{2N-1}$ is

$$\bar{f}_j = \frac{a_0}{2} + \sum_{k=1}^{N-1} \left(a_k \cos\left(\frac{2\pi jk}{2N}\right) + b_k \sin\left(\frac{2\pi jk}{2N}\right)\right) + \frac{(-1)^j a_N}{2} \tag{3.16}$$

where $b_k = -2\,\mathrm{Im}(F_k) = 0$. Therefore,

$$\bar{f}_j = \frac{a_0}{2} + \sum_{k=1}^{N-1} a_k \cos\left(\frac{\pi jk}{N}\right) + \frac{(-1)^j a_N}{2} = \sum_{k=0}^{N} \frac{a_k}{\bar{c}_k} \cos\left(\frac{\pi jk}{N}\right), \tag{3.17}$$

which is the cosine transform.

The fact that the sequence needs to be even is not restrictive. We can create an even sequence from an arbitrary sequence $\{f_j\}_{j=0}^{N}$ by extending it to a sequence of length $2N$ as follows:

$$\begin{cases} \bar{f}_j = f_j, & j = 0, 1, \ldots, N, \\ \bar{f}_{2N-j} = f_j, & j = 1, 2, \ldots, N-1. \end{cases} \tag{3.18}$$

Doing so, however, computes the unneeded values of b_k, so twice as much work as necessary is being done. We will see how to cut that extra work right after we look at how to compute the forward transform.

The forward cosine transform is within a factor of $2/N$ of the backward transform (3.17). Recall that

$$a_k = 2\,\mathrm{Re}\,(F_k)\,, \quad k = 0, 1, \ldots, 2N-1 \tag{3.19}$$

and

$$F_k = \frac{1}{2N} \sum_{j=0}^{2N-1} \bar{f}_j e^{-ijk\pi/N}, \tag{3.20}$$

so

$$\begin{aligned} a_k &= \frac{2}{2N} \sum_{j=0}^{2N-1} \bar{f}_j\, \mathrm{Re}\left\{\cos\left(\frac{jk\pi}{N}\right) - i \sin\left(\frac{jk\pi}{N}\right)\right\} \\ &= \frac{1}{N} \sum_{j=0}^{2N-1} \bar{f}_j \cos\left(\frac{jk\pi}{N}\right). \end{aligned} \tag{3.21}$$

If we split the latter sum into two,

$$a_k = \frac{1}{N} \left\{ \bar{f}_0 + \sum_{j=1}^{N-1} \bar{f}_j \cos\left(\frac{jk\pi}{N}\right) + \sum_{j=N+1}^{2N-1} \bar{f}_j \cos\left(\frac{jk\pi}{N}\right) + (-1)^k \bar{f}_N \right\}. \tag{3.22}$$

By construction, $\bar{f}_j = \bar{f}_{2N-j} = f_j$ so if we again let $l = 2N - j$,

$$\sum_{j=N+1}^{2N-1} \bar{f}_j \cos\left(\frac{jk\pi}{N}\right) = \sum_{l=N-1}^{1} \bar{f}_{2N-l} \cos\left(\frac{(2N-l)\,k\pi}{N}\right) = \sum_{j=1}^{N-1} f_j \cos\left(\frac{jk\pi}{N}\right). \tag{3.23}$$

Therefore,

$$a_k = \frac{1}{N}\left\{ f_0 + 2\sum_{j=1}^{N-1} f_j \cos\left(\frac{jk\pi}{N}\right) + (-1)^k f_N \right\}, \tag{3.24}$$

or if we fold the endpoints into the sum,

$$a_k = \frac{2}{N}\sum_{j=0}^{N} \frac{f_j}{\bar{c}_j} \cos\left(\frac{jk\pi}{N}\right). \tag{3.25}$$

So we see that we can compute the backward transform (3.17) from the forward transform (3.25) simply by multiplying the results by $N/2$.

With a little more work, we can make the cosine transform more efficient by about a factor of two if we transform the even sequence so that the coefficients b_k that are returned by the real Fourier transform contain useful information. A method introduced by Dollimore generates the half sized array

$$e_j = \frac{1}{2}\left(\bar{f}_j + \bar{f}_{N-j}\right) - \left(\bar{f}_j - \bar{f}_{N-j}\right)\sin\frac{j\pi}{N}, \quad j = 0, 1, \ldots, N-1. \tag{3.26}$$

When we substitute for f_j using (3.17) and rearrange, we get the real transform

$$\begin{aligned} e_j &= \frac{a_0}{2} + \sum_{k=1}^{N-1}\left(a_{2k}\cos\frac{2\pi jk}{N} + (a_{2k+1} - a_{2k-1})\sin\frac{2\pi jk}{N}\right) + \frac{(-1)^j a_N}{2} \\ &= \frac{\bar{a}_0}{2} + \sum_{k=1}^{N-1}\left(\bar{a}_k \cos\left(\frac{2\pi jk}{N}\right) + \bar{b}_k \sin\left(\frac{2\pi jk}{N}\right)\right) + \frac{(-1)^j \bar{a}_N}{2}. \end{aligned} \tag{3.27}$$

Thus, we can use Algorithm 15 (ForwardRealFFT) to compute transform coefficients $\bar{a}_k$ and $\bar{b}_k$ of $\{e_j\}_{j=0}^{N-1}$. Then we make the correspondence to the coefficients of the cosine transform of the original sequence, $\{f_j\}_{j=0}^{N}$, through

$$\begin{cases} \bar{a}_k = a_{2k}, & k = 0, 1, \ldots, N/2, \\ \bar{b}_k = a_{2k+1} - a_{2k-1}, & k = 1, 2, \ldots, N/2 - 1. \end{cases} \tag{3.28}$$

Equation (3.28) says that we can compute the even coefficients of the cosine transform directly from the results of the forward real FFT. To get the odd coefficients we use the recursion relation

$$a_{2k+1} = a_{2k-1} + \bar{b}_k, \quad k = 1, 2, \ldots, N/2 - 1. \tag{3.29}$$

Algorithm 28: *FastCosineTransform:* The Cosine Transform Computed with the Real FFT

Procedure FastCosineTransform
Input: $\{f_j\}_{j=0}^{N}, \{w_j^+\}_{j=0}^{N-1}, \{C_j\}_{j=0}^{N}, \{S_j\}_{j=0}^{N}, s$
Uses Algorithms:
Algorithm 15 (ForwardRealFFT)
Comment: $C_j = \cos(j\pi/N)$, $S_j = \sin(j\pi/N)$
for $j = 0$ **to** $N-1$ **do**
  $e_j \leftarrow \frac{1}{2}\left(x_j + x_{N-j}\right) - S_j\left(x_j - x_{N-j}\right)$
end
$\left\{\{\bar{a}_k\}_{k=0}^{N/2}, \{\bar{b}_k\}_{k=0}^{N/2}\right\} \leftarrow \mathit{ForwardRealFFT}\left(\{e_j\}_{j=0}^{N-1}, \{w_j^+\}_{j=0}^{N-1}\right)$
for $k = 0$ **to** $N/2$ **do**
  $a_{2k} \leftarrow \bar{a}_k$
end
$a_1 \leftarrow f_0 - f_N$
for $j = 1$ **to** $N-1$ **do**
  $a_1 \leftarrow a_1 + 2C_j f_j$
end
$a_1 \leftarrow a_1/N$
for $k = 1$ **to** $N/2 - 1$ **do**
  $a_{2k+1} \leftarrow \bar{b}_k + a_{2k-1}$
end
if $s = \mathit{BACKWARD}$ **then**
  for $k = 0$ **to** N **do**
    $a_k \leftarrow N a_k/2$
  end
end
return $\{a_k\}_{k=0}^{N}$
End Procedure FastCosineTransform

To start the recursion for the odd indexed coefficients, we compute the first one directly by the real DFT

$$a_1 = \frac{f_0}{N} + \frac{2}{N}\sum_{j=1}^{N-1} f_j \cos\left(\frac{j\pi}{N}\right) + \frac{(-1)^1 f_N}{N}. \tag{3.30}$$

We now have all the relations, (3.26)–(3.30), necessary to implement the Fast Cosine Transform (FCT). We present the result in Algorithm 28 (FastCosineTransform). We could have chosen one of several ways to implement the transforms. If we need to compute many transforms of the same size, which is common in spectral method applications, then we would pre-compute both the trigonometric factors for the real transform Algorithm 15 (ForwardRealFFT) and the sine and cosine factors. We can compute and store them in a single array, using the periodicity of the trigonometric factors and taking their real and imaginary parts for the sine and cosine factors. For clarity, however, Algorithm 28 takes three arrays of the Fourier

Algorithm 29: *FastChebyshevTransform:* The Fast Chebyshev Transform using the Fast Cosine Transform

Procedure FastChebyshevTransform
Input: $\{f_j\}_{j=0}^{N}, \{w_j^+\}_{j=0}^{N-1}, \{C_j\}_{j=0}^{N}, \{S_j\}_{j=0}^{N}, s$
Uses Algorithms:
Algorithm 28 (FastCosineTransform)
Comment: $C_j = \cos(j\pi/N)$, $S_j = \sin(j\pi/N)$
for $j = 0$ **to** N **do**
  $g_j \leftarrow f_j$
end
if $s = BACKWARD$ **then**
  $g_0 \leftarrow 2g_0$
  $g_N \leftarrow 2g_N$
end
$\{a_k\}_{k=0}^{N} \leftarrow FastCosineTransform\left(\{g_j\}_{j=0}^{N}, \{w_j^+\}_{j=0}^{N-1}, \{C_j\}_{j=0}^{N}, \{S_j\}_{j=0}^{N}, s\right)$
if $s = FORWARD$ **then**
  $a_0 \leftarrow a_0/2$
  $a_N \leftarrow a_N/2$
end
return $\{a_k\}_{k=0}^{N}$
End Procedure FastChebyshevTransform

trigonometric factors plus the arrays of the cosine and sine factors. These would be computed in a FCT initialization routine that we do not show here. The algorithm also takes a parameter, s, so a single procedure contains the forward and backward transforms.

Finally, we use the Fast Cosine Transform to compute the Chebyshev transform. The only difference between the Chebyshev transform (3.11), the cosine transform (3.17), and (3.25) is how we have to treat the first and last elements of the sequence. Therefore we only need to perform a simple preprocessing to use the cosine transform to compute the Chebyshev transform, as we show in Algorithm 29 (FastChebyshevTransform).

3.4 How to Evaluate Polynomial Interpolants in Lagrange Form

Spectral polynomial interpolants are often computed in the nodal Lagrange form rather than the modal orthogonal polynomial expansion (1.137). The Lagrange form of the polynomial of degree N that interpolates a function at a set of points $\{(x_j, f_j)\}_{j=0}^{N}$ is

$$p_N(x) = \sum_{j=0}^{N} f_j \ell_j(x). \tag{3.31}$$

The functions $\ell_j(x)$ are again the *Lagrange interpolating polynomials*

$$\ell_j(x) = \prod_{\substack{i=0 \\ i \neq j}}^{N} \frac{x - x_i}{x_j - x_i}. \tag{3.32}$$

The important property of the interpolating polynomials that we will exploit many times when we derive nodal spectral methods is that $\ell_j(x_i) = \delta_{i,j}$.

Two alternate ways to write the Lagrange interpolation have gained attention in recent years. The first, which is called the "modified Lagrange interpolation", is

$$p_N(x) = \psi(x) \sum_{j=0}^{N} f_j \frac{w_j}{x - x_j} \tag{3.33}$$

where $\psi(x) = \prod_{i=0}^{N}(x - x_i)$ and

$$w_j = \frac{1}{\prod_{\substack{i=0 \\ i \neq j}}^{N} (x_j - x_i)}. \tag{3.34}$$

The equality

$$\psi(x) \sum_{j=0}^{N} \frac{w_j}{x - x_j} = 1 \tag{3.35}$$

allows us to write a third version known as the "Barycentric formula",

$$p_N(x) = \frac{\sum_{j=0}^{N} f_j \frac{w_j}{x - x_j}}{\sum_{j=0}^{N} \frac{w_j}{x - x_j}}. \tag{3.36}$$

Although (3.33) is theoretically better than (3.36), for the sets of interpolation nodes used in spectral methods where the spacing decreases towards the endpoints, the form (3.36) has similar rounding error properties.

From these two formulas, (3.34) and (3.36), we write two algorithms, one to compute the barycentric weights, w_j's, and the other to compute the interpolant at a point, x. Algorithm 30 (BarycentricWeights) implements (3.34) to compute the weights. Once we have computed and stored the weights, we will be able to compute the interpolant for any point x and any set of values $\{f_j\}_{j=0}^{N}$ using Algorithm 31 (LagrangeInterpolation), which implements (3.36). The advantage of the barycentric approach should be clear from these two algorithms. Once the weights are computed using Algorithm 30, the interpolant at any point x will take $O(N)$ floating point operations to compute using Algorithm 31.

If we want to interpolate repeatedly to a fixed set of nodes, say to a fine grid for plotting purposes or in multidimensional interpolation, we rewrite the operation as

Algorithm 30: *BarycentricWeights:* Weights for Lagrange Interpolation

Procedure BarycentricWeights
Input: $\{x_j\}_{j=0}^{N}$

for $j = 0$ **to** N **do**
  $w_j \leftarrow 1$
end
for $j = 1$ **to** N **do**
  for $k = 0$ **to** $j - 1$ **do**
    $w_k \leftarrow w_k \left(x_k - x_j\right)$
    $w_j \leftarrow w_j \left(x_j - x_k\right)$
  end
end
for $j = 0$ **to** N **do**
  $w_j \leftarrow 1/w_j$
end
return $\{w_j\}_{j=0}^{N}$
End Procedure BarycentricWeights

Algorithm 31: *LagrangeInterpolation:* Lagrange Interpolant from Barycentric Form

Procedure LagrangeInterpolation
Input: $x, \{x_j\}_{j=0}^{N}, \{f_j\}_{j=0}^{N}, \{w_j\}_{j=0}^{N}$
Uses Algorithms:
  Algorithm 139 (AlmostEqual)

numerator $\leftarrow 0$
denominator $\leftarrow 0$
for $j = 0$ **to** N **do**
  if *AlmostEqual*(x, x_j) **then return** f_j
  $t \leftarrow w_j/(x - x_j)$
  numerator $\leftarrow$ *numerator* $+ t * f_j$
  denominator $\leftarrow$ *denominator* $+ t$
end
return *numerator*/*denominator*
End Procedure LagrangeInterpolation

a matrix multiplication. To map between an original set of points, $\{x_j\}_{j=0}^{N}$ to a new set of points, $\{\xi_j\}_{j=0}^{M}$, (3.36) becomes

$$F_k = \frac{\sum_{j=0}^{N} \frac{w_j}{\xi_k - x_j} f_j}{\sum_{j=0}^{N} \frac{w_j}{\xi_k - x_j}} = \sum_{j=0}^{N} T_{kj} f_j \tag{3.37}$$

where

$$T_{kj} = \frac{\frac{w_j}{\xi_k - x_j}}{\sum_{j=0}^{N} \frac{w_j}{\xi_k - x_j}}. \tag{3.38}$$

Algorithm 32: *PolynomialInterpolationMatrix:* Matrix for Interpolation Between Two Sets of Points

Procedure PolynomialInterpolationMatrix
Input: $\{x_j\}_{j=0}^{N}, \{w_j\}_{j=0}^{N}, \{\xi_j\}_{j=0}^{M}$
Uses Algorithms:
Algorithm 139 (AlmostEqual)

for $k = 0$ **to** M **do**
  rowHasMatch $\leftarrow$ *false*
  for $j = 0$ **to** N **do**
    $T_{k,j} \leftarrow 0$
    if *AlmostEqual*(ξ_k, x_j) **then**
      rowHasMatch $\leftarrow$ *true*
      $T_{k,j} \leftarrow 1$
    end
  end
  if *rowHasMatch is false* **then**
    $s \leftarrow 0$
    for $j = 0$ **to** N **do**
      $t \leftarrow w_j / (\xi_k - x_j)$
      $T_{k,j} \leftarrow t$
      $s \leftarrow s + t$
    end
    for $j = 0$ **to** N **do**
      $T_{k,j} \leftarrow T_{kj} / s$
    end
  end
end
return $\{T_{i,j}\}_{i,j=0}^{M,N}$
End Procedure PolynomialInterpolationMatrix

We present Algorithm 32 (PolynomialInterpolationMatrix) to compute the interpolation matrix from $\{x_j\}_{j=0}^{N}$ to $\{\xi_j\}_{j=0}^{M}$. To compute the interpolant, we need only to modify the matrix multiplication algorithm, Algorithm 19 (MxVDerivative) to handle non-square matrices, giving Algorithm 33 (InterpolateToNewPoints).

Later in Sect. 4.7 we will see that the discontinuous Galerkin approximation requires the boundary values of the Lagrange interpolating polynomials at the endpoints, $\ell_j(\pm 1)$ using the Gauss points as the nodes. If we compare the two forms of the interpolant, (3.32) and (3.36), we see that

$$\ell_j(x) = \frac{w_j}{(x - x_j) \sum_{j=0}^{N} \frac{w_j}{x - x_j}}. \tag{3.39}$$

This relationship means that we can compute the Lagrange interpolating polynomials from the barycentric weights by a modification of Algorithm 31 (LagrangeInterpolation).

We present Algorithm 34 (LagrangeInterpolatingPolynomials) to show one way to compute the Lagrange interpolating polynomials. It first checks to see if the eval-

Algorithm 33: *InterpolateToNewPoints:* Interpolation Between Two Sets of Points by Matrix Multiplication

Procedure InterpolateToNewPoints
Input: $\{T_{i,j}\}_{i,j=0}^{M,N}$, $\{f_j\}_{j=0}^{N}$
for $i = 0$ **to** M **do**
  $t \leftarrow 0$
  for $j = 0$ **to** N **do**
    $t \leftarrow t + T_{i,j} * f_j$
  end
  $fInterp_i \leftarrow t$
end
return $\{fInterp_j\}_{j=0}^{M}$
End Procedure InterpolateToNewPoints

Algorithm 34: *LagrangeInterpolatingPolynomials:* $\ell_j(x)$

Procedure LagrangeInterpolatingPolynomials
Input: $x, \{x_j\}_{j=0}^{N}, \{w_j\}_{j=0}^{N}$
Uses Algorithms:
  Algorithm 139 (AlmostEqual)
$xMatchesNode \leftarrow false$
for $j = 0$ **to** N **do**
  $\ell_j \leftarrow 0.0$
  if $AlmostEqual(x, x_j)$ **then**
    $\ell_j \leftarrow 1.0$
    $xMatchesNode \leftarrow true$
  end
end
if $xMatchesNode$ **then return** $\{\ell_j\}_{j=0}^{N}$
$s \leftarrow 0$
for $j = 0$ **to** N **do**
  $t \leftarrow w_j/(x - x_j)$
  $\ell_j \leftarrow t$
  $s \leftarrow s + t$
end
for $j = 0$ **to** N **do**
  $\ell_j \leftarrow \ell_j/s$
end
return $\{\ell_j\}_{j=0}^{N}$
End Procedure LagrangeInterpolatingPolynomials

uation point is a node, x_j. If so, then we know that the polynomial is one at x_j and zero at all of the other nodes. If the evaluation point is not a node, then the algorithm evaluates equation (3.39).

Finally, we show how to compute interpolations of functions of two independent variables. Formally, we write the interpolant of a function $f(x, y)$ as the tensor

product

$$p_N(x,y) = \sum_{i=0}^{N} \sum_{j=0}^{N} f_{i,j} \ell_i(x) \ell_j(y). \tag{3.40}$$

We could evaluate this form directly using the Lagrange interpolating polynomials computed by Algorithm 34 (LagrangeInterpolatingPolynomials). In practice, however, direct evaluation can be slow.

A common need is to interpolate from one grid to a finer grid for plotting purposes. Plotting programs typically draw curves by assuming linear variations between points on the grid. Spectral methods are able to compute accurate solutions on coarse grids. When we plot solutions on these coarse grids, they often look worse than they really are. For this reason, we will usually interpolate spectral approximations to a fine grid before plotting.

We interpolate a polynomial $p_N(x,y)$ from a coarse $N^{old} \times M^{old}$ grid to a fine $N^{new} \times M^{new}$ grid in two stages using Algorithms 32 (PolynomialInterpolationMatrix) and 33 (InterpolateToNewPoints). The first stage is to compute one-dimensional interpolations from the original grid to a fine grid in one of the two coordinate directions, e.g. to a $N^{new} \times M^{old}$ grid

$$\bar{F}_{n,j} = \sum_{i=0}^{N^{old}} T_{ni}^{(x)} f_{i,j}, \quad n = 0, 1, \ldots, N^{new};\ j = 0, 1, \ldots, M^{old}. \tag{3.41}$$

The second stage is to interpolate from the intermediate grid to the final grid

$$F_{n,m} = \sum_{j=0}^{M^{old}} T_{mj}^{(y)} \bar{F}_{n,j}, \quad n = 0, 1, \ldots, N^{new};\ m = 0, 1, \ldots, M^{new}. \tag{3.42}$$

With the two stage approach, we need to compute an interpolation matrix only once for each direction, and the grid is interpolated to the new grid by a series of matrix-vector products. We show the procedure in Algorithm 35 (2DCoarseToFineInterpolation).

3.5 How to Evaluate Polynomial Derivatives

When we approximate a function by an interpolant, we approximate its derivatives by differentiating the interpolant

$$f'(x) \approx (I_N f(x))' = \sum_{j=0}^{N} f_j \ell_j'(x) = \sum_{k=0}^{N} \bar{f}_k^{(1)} \phi_k. \tag{3.43}$$

If we represent the interpolant of the function in Lagrange form, we use the first representation of the derivative that is written in terms of the Lagrange interpolating

Algorithm 35: *2DCoarseToFineInterpolation:* Interpolation from a Coarse to a Fine Grid in 2D

Procedure 2DCoarseToFineInterpolation
Input: $\{x_i\}_{i=0}^{N^{old}}$, $\{y_j\}_{j=0}^{M^{old}}$, $\{f_{i,j}\}_{i,j=0}^{N^{old},M^{old}}$
Input: $\{\xi_i\}_{i=0}^{N^{new}}$, $\{\eta_j\}_{j=0}^{M^{new}}$
Uses Algorithms:
Algorithm 32 (PolynomialInterpolationMatrix)
Algorithm 33 (InterpolateToNewPoints)
Algorithm 30 (BarycentricWeights)
$\{w_i\}_{i=0}^{N^{old}} \leftarrow BarycentricWeights(\{x_i\}_{i=0}^{N^{old}})$
$\{T_{i,j}\}_{i=0,j=0}^{N^{new},N^{old}} \leftarrow PolynomialInterpolationMatrix(\{x_j\}_{j=0}^{N^{old}}, \{w_j\}_{j=0}^{N^{old}}, \{\xi_j\}_{j=0}^{N^{new}})$
for $j = 0$ **to** M^{old} **do**
  $\{\bar{F}_{n,j}\}_{n=0}^{N^{new}} \leftarrow InterpolateToNewPoints(\{T_{i,j}\}_{i=0,j=0}^{N^{new},N^{old}}, \{f_{i,j}\}_{i=0}^{N^{old}})$
end
$\{w_j\}_{j=0}^{M^{old}} \leftarrow BarycentricWeights(\{y_j\}_{j=0}^{M^{old}})$
$\{T_{i,j}\}_{i=0,j=0}^{M^{new},M^{old}} \leftarrow PolynomialInterpolationMatrix(\{y_j\}_{j=0}^{M^{old}}, \{w_j\}_{j=0}^{M^{old}}, \{\eta_j\}_{j=0}^{M^{new}})$
for $n = 0$ **to** N^{new} **do**
  $\{F_{n,m}\}_{m=0}^{M^{new}} \leftarrow InterpolateToNewPoints(\{T_{i,j}\}_{i=0,j=0}^{M^{new},M^{old}}, \{\bar{F}_{n,m}\}_{m=0}^{M^{old}})$
end
return $\{F_{n,m}\}_{n,m=0}^{N^{new},M^{new}}$
End Procedure InterpolateToNewPoints

polynomials. We can evaluate the derivative at a set of mesh points as a matrix-vector multiplication, which is efficient for sufficiently small N. If N is large, transform methods are faster, if available. To use transform methods we use the second representation in (3.43), written as an orthogonal polynomial expansion.

3.5.1 Direct Evaluation of the Derivative

On some occasions we will need to compute the derivative of an interpolant at points other than the nodes. In those situations, it is reasonable to compute them directly. Fortunately, the barycentric form of the interpolant allows us to compute the derivative at a point in $O(N)$ operations.

If the evaluation point is not a node, then we can work directly with (3.36) to compute the derivative of the interpolant p_N. After we multiply through by the denominator of the right hand side of (3.36) and differentiate with respect to x,

$$p'_N(x)\sum_{j=0}^{N}\frac{w_j}{x-x_j} - p_N(x)\sum_{j=0}^{N}\frac{w_j}{(x-x_j)^2} = -\sum_{j=0}^{N}\frac{w_j f_j}{(x-x_j)^2}. \tag{3.44}$$

If we write (3.44) in terms of the derivative we get a formula that looks similar to the interpolation formula itself

$$p'_N(x) = \frac{\sum_{j=0}^{N} \frac{w_j}{x-x_j} \frac{p_N - f_j}{x-x_j}}{\sum_{j=0}^{N} \frac{w_j}{x-x_j}}. \tag{3.45}$$

At a node, the derivative simplifies to

$$p'_N(x_i) = -\frac{1}{w_i} \sum_{\substack{j=0 \\ j \neq i}}^{N} w_j \frac{f_i - f_j}{x_i - x_j}. \tag{3.46}$$

In those situations where we need to compute the derivative directly, we would use Algorithm 36 (LagrangeInterpolantDerivative), which implements (3.45) and (3.46).

Algorithm 36: *LagrangeInterpolantDerivative:* Direct Computation of the Polynomial Derivative in Barycentric Form

Procedure LagrangeInterpolantDerivative
Input: $x, \{x_j\}_{j=0}^{N}, \{f_j\}_{j=0}^{N}, \{w_j\}_{j=0}^{N}$
Uses Algorithms:
  Algorithm 139 (AlmostEqual)
  Algorithm 31 (LagrangeInterpolation)

```
atNode ← false
numerator ← 0
for j = 0 to N do
    if AlmostEqual(x, x_j) then
        atNode ← true
        p ← f_j
        denominator ← −w_j
        i ← j
    end
end
if atNode then
    for j = 0 to N do
        if j ≠ i then
            numerator ← numerator + w_j ∗ (p − f_j)/(x − x_j)
        end
    end
else
    denominator ← 0
    p ← LagrangeInterpolation(x, N, {x_j}_{j=0}^N, {f_j}_{j=0}^N, {w_j}_{j=0}^N)
    for j = 0 to N do
        t ← w_j/(x − x_j)
        numerator ← numerator + t ∗ (p − f_j)/(x − x_j)
        denominator ← denominator + t
    end
end
return p' ← numerator/denominator
```

End Procedure LagrangeInterpolantDerivative

3.5.2 Evaluation of Derivatives by Matrix Multiplication

One easy way to derive the values of the derivative of a polynomial interpolant at a set of nodes is to start with the Lagrange form

$$f'(x_i) \approx \mathscr{D} f_i \equiv (I_N f(x_i))' = \sum_{j=0}^{N} f_j \ell_j'(x_i) = \sum_{j=0}^{N} D_{ij} f_j, \quad i = 0, 1, \ldots, N. \tag{3.47}$$

We can therefore evaluate the derivative by matrix-vector multiplication, where $D_{ij} = \ell_j'(x_i)$ is the *derivative matrix*. In practice, we pre-compute and store the derivative matrix. The main issue is to mitigate the effects of roundoff error, which can become significant for high order derivatives and for large N.

The most general way to evaluate the derivative matrix is to form the derivatives of the Lagrange interpolating polynomials, as implied directly by (3.43). This works for any set of evaluation points and for any basis polynomials. The more likely situation, however, is that we will want the derivatives at the nodes, as occurs in (3.47). If we only need the nodal values of the derivatives, we can conveniently use the barycentric form, for which the derivative matrix is

$$D_{ij} = \frac{w_j}{w_i}\left[\frac{1}{x_i - x_j}\right], \quad i \neq j, \tag{3.48}$$

and the weights, w_j are given by (3.34). To compute the diagonal elements, we again use the negative sum trick, (2.43), to minimize the effects of rounding errors when we compute the derivative. Remember the remarks about (2.43), however. For large N we get the best results when the off diagonal terms are sorted and summed from smallest in magnitude to largest. For the sake of simplicity, however, we present Algorithm 37 (PolynomialDerivativeMatrix) without the sorting to compute the derivative matrix.

We note that in special situations, explicit forms for the derivative matrices exist. For example, if the nodes are the Chebyshev Gauss-Lobatto points, we could write the first derivative matrix as [10]

$$\begin{aligned} D_{ij} &= -\frac{1}{2}\frac{\bar{c}_i}{\bar{c}_j}\frac{(-1)^{i+j}}{\sin(\frac{(i+j)\pi}{2N})\sin(\frac{(i-j)\pi}{2N})}, \quad j \neq i, \\ D_{ii} &= -\frac{1}{2}\frac{x_i}{\sin^2(\frac{\pi i}{N})}, \quad i \neq 0, N, \\ D_{00} &= D_{NN} = \frac{2N^2+1}{6}, \end{aligned} \tag{3.49}$$

where $\bar{c}_j$ is defined in (3.12). However, the more general Algorithm 37 (PolynomialDerivativeMatrix) is more convenient to use for any set of basis functions.

Algorithm 37: *PolynomialDerivativeMatrix:* First Derivative Approximation Matrix

Procedure PolynomialDerivativeMatrix
Input: $\{x_j\}_{j=0}^{N}$
Uses Algorithms:
Algorithm 30 (BarycentricWeights)
$\{w_j\}_{j=0}^{N} \leftarrow BarycentricWeights(\{x_j\}_{j=0}^{N})$
for $i = 0$ **to** N **do**
  $D_{i,i} \leftarrow 0$
  for $j = 0$ **to** N **do**
    if $j \neq i$ **then**
      $D_{i,j} \leftarrow \frac{w_j}{w_i}\frac{1}{x_i - x_j}$
      $D_{i,i} \leftarrow D_{i,i} - D_{i,j}$
    end
  end
end
return $\{D_{i,j}\}_{i,j=0}^{N}$
End Procedure PolynomialDerivativeMatrix

We compute derivative matrices for high order derivatives in one of several ways. The recommended approach computes the matrices recursively

$$\begin{aligned} D_{ij}^{(m)} &= \frac{m}{x_i - x_j}\left(\frac{w_j}{w_i}D_{ii}^{(m-1)} - D_{ij}^{(m-1)}\right), \\ D_{ii}^{(m)} &= -\sum_{\substack{j=0 \\ j\neq i}}^{N} D_{ij}^{(m)}. \end{aligned} \tag{3.50}$$

Algorithm 38 (mthOrderPolynomialDerivativeMatrix) presents an implementation to compute the matrix elements of the mth derivative matrix.

Finally, once we have computed and stored the derivative matrix, we compute the derivative by matrix multiplication, Algorithm 19 (MxVDerivative).

3.5.3 Even-Odd Decomposition

We can speed up the matrix-vector computation of the spectral derivative at the Gauss and the Gauss-Lobatto by about a factor of two using an even-odd decomposition. The key observation is that

$$D_{i,j} = -D_{N-i,N-j}, \tag{3.51}$$

a fact that can be shown directly from the definition (3.48) and the fact that the Gauss and Gauss-Lobatto points satisfy $x_i = -x_{N-i}$.

Algorithm 38: *mthOrderPolynomialDerivativeMatrix:* Derivative Matrix for mth Order Derivatives

Procedure mthOrderPolynomialDerivativeMatrix
Input: m, $\{x_j\}_{j=0}^N$
Uses Algorithms:
Algorithm 30 (BarycentricWeights)
Algorithm 37 (PolynomialDerivativeMatrix)

$\{w_j\}_{j=0}^N \leftarrow BarycentricWeights(\{x_j\}_{j=0}^N)$
$\left\{D_{i,j}^{(m)}\right\}_{i,j=0}^N \leftarrow PolynomialDerivativeMatrix(\{x_j\}_{j=0}^N)$
if $m = 1$ **then return** $\left\{D_{i,j}^{(m)}\right\}_{i,j=0}^N$
for $k = 2$ **to** m **do**
  for $i = 0$ **to** N **do**
    $D_{i,i}^{(m)} \leftarrow 0$
    for $j = 0$ **to** N **do**
      if $j \neq i$ **then**
        $D_{i,j}^{(m)} \leftarrow \frac{k}{x_i - x_j}\left(\frac{w_j}{w_i} D_{i,i}^{(m)} - D_{i,j}^{(m)}\right)$
        $D_{i,i}^{(m)} \leftarrow D_{i,i}^{(m)} - D_{i,j}^{(m)}$
      end
    end
  end
end
return $\left\{D_{i,j}^{(m)}\right\}_{i,j=0}^N$
End Procedure mthOrderPolynomialDerivativeMatrix

To derive the even-odd decomposition, let us assume for the moment that the number of points, $N+1$, is even and define

$$\begin{aligned} e_j &= \frac{1}{2}\left(f_j + f_{N-j}\right), \\ o_j &= \frac{1}{2}\left(f_j - f_{N-j}\right), \end{aligned} \qquad j = 0, 1, \ldots, \lfloor N/2 \rfloor \tag{3.52}$$

to be the even and odd vectors, so that $f_j = e_j + o_j$. From their definitions, it is easy to see that $e_j = e_{N-j}$ and $o_j = -o_{N-j}$. Since matrix-vector multiplication is linear,

$$\mathscr{D} f = \mathscr{D}(e + o) = \mathscr{D} e + \mathscr{D} o. \tag{3.53}$$

The symmetry (3.51) allows us to decompose the matrix multiplication on the even and odd vectors into two sums of half the length. For instance,

$$\mathscr{D} e_i = \sum_{j=0}^{N} D_{ij} e_j = \sum_{j=0}^{\lfloor N/2 \rfloor} \left(D_{i,j} e_j + D_{i,N-j} e_{N-j}\right) = \sum_{j=0}^{\lfloor N/2 \rfloor} \left(D_{i,j} + D_{i,N-j}\right) e_j. \tag{3.54}$$

We only need to compute the values of $\mathscr{D}e_i$ for $i = 0, 1, \ldots, \lfloor N/2 \rfloor$, for

$$\mathscr{D}e_{N-i} = \sum_{j=0}^{\lfloor N/2 \rfloor} \left(-D_{i,j} e_{N-j} - D_{i,j} e_j \right) = - \sum_{j=0}^{\lfloor N/2 \rfloor} \left(D_{i,j} + D_{i,N-j} \right) e_j = -\mathscr{D}e_i. \tag{3.55}$$

Similarly,

$$\mathscr{D}o_i = \sum_{j=0}^{\lfloor N/2 \rfloor} \left(D_{i,j} - D_{i,N-j} \right) o_j \tag{3.56}$$

and

$$\mathscr{D}o_{N-i} = \mathscr{D}o_i. \tag{3.57}$$

To reconstruct the full derivative approximation we compute

$$\begin{aligned} \mathscr{D}f_j &= \mathscr{D}e_j + \mathscr{D}o_j, \quad j = 0, 1, \ldots, \lfloor N/2 \rfloor, \\ \mathscr{D}f_{N-j} &= -\mathscr{D}e_j + \mathscr{D}o_j, \quad j = 0, 1, \ldots, \lfloor N/2 \rfloor. \end{aligned} \tag{3.58}$$

Algorithm 39 (EOMatrixDerivative) presents a procedure for the even-odd decomposition, including the modification that adds the middle term when $N + 1$ is odd.

3.5.4 Evaluation by Transform Methods

Transform methods are more efficient for large N. We have all of the machinery necessary to build an algorithm to compute derivatives of Chebyshev approximations that use the FFT. The starting points are the relations (1.102) and (1.113) between the coefficients of the interpolant and the coefficients of the derivatives. The procedure is to compute the Chebyshev coefficients using the FFT, Algorithm 29 (FastChebyshevTransform), compute the coefficients of the derivative, Algorithm 5 (ChebyshevDerivativeCoefficients) and then transform back using Algorithm 29. We show this procedure in Algorithm 40 (FastChebyshevDerivative). Notice that, for speed, we assume the transform weights and sine and cosine factors are precomputed, stored, and available to the transform.

3.5.5 Performance of Various Polynomial Derivative Algorithms

We have developed four algorithms with which to compute the derivative of a polynomial interpolant at its nodes. The question is which of these is the fastest? We wish we could say with complete certainty, but the answer depends strongly on

Algorithm 39: *EOMatrixDerivative:* Computation of First Derivative by Even-Odd Decomposition

Procedure EOMatrixDerivative
Input: $\{D_{i,j}\}_{i,j=0}^{N}$, $\{f_j\}_{j=0}^{N}$
integer $M \leftarrow \lfloor (N+1)/2 \rfloor$
for $j = 0$ **to** M **do**
  $e_j \leftarrow (f_j + f_{N-j})/2$
  $o_j \leftarrow (f_j - f_{N-j})/2$
end
for $i = 0$ **to** $M-1$ **do**
  $\mathscr{D}e_i \leftarrow 0$
  $\mathscr{D}o_i \leftarrow 0$
  for $j = 0$ **to** $M-1$ **do**
    $\mathscr{D}e_i \leftarrow \mathscr{D}e_i + \left(D_{i,j} + D_{i,N-j}\right) * e_j$
    $\mathscr{D}o_i \leftarrow \mathscr{D}e_i + \left(D_{i,j} - D_{i,N-j}\right) * o_j$
  end
end
if $N+1$ *is odd* **then**
  for $i = 0$ **to** $M-1$ **do**
    $\mathscr{D}e_i \leftarrow \mathscr{D}e_i + D_{i,M} * e_M$
  end
  $\mathscr{D}e_M \leftarrow 0$
  for $j = 0$ **to** $M-1$ **do**
    $\mathscr{D}o_M \leftarrow \mathscr{D}o_M + \left(D_{M,j} - D_{M,N-j}\right) * o_j$
  end
end
for $j = 0$ **to** $M-1$ **do**
  $\mathscr{D}f_j \leftarrow \mathscr{D}e_j + \mathscr{D}o_j$
  $\mathscr{D}f_{N-j} \leftarrow -\mathscr{D}e_j + \mathscr{D}o_j$
end
if $N+1$ *is odd* **then**
  $\mathscr{D}f_M \leftarrow \mathscr{D}e_M + \mathscr{D}o_M$
end
return $\{\mathscr{D}f_j\}_{j=0}^{N}$
End Procedure EOMatrixDerivative

the number of points, the computer architecture, cache sizes, the code, the compiler, and the optimizer. For instance, in the past we found that hand-unrolled loops in the matrix-vector product routines produced significantly faster computations (cf. [9]). We have not found that to be the case on our current system. For production work, we recommend coding and testing several algorithms and implementations.

Figure 3.3 shows timing comparisons between direct implementations of the three algorithms 19 (MxVDerivative), 39 (EOMatrixDerivative), and 40 (FastChebyshevDerivative) without any special efforts to optimize them by hand, but with two levels of complier optimization. In each case, we computed the derivatives a million times and then divided the total time by that number. We see that for $N < 10$ there is little difference between the Even-Odd decomposition and the di-

Algorithm 40: *FastChebyshevDerivative:* Computation of the Derivative by the Fast Chebyshev Transform

Procedure FastChebyshevDerivative
Input: $\{f_j\}_{j=0}^{N}$
Uses Algorithms:
Algorithm 5 (ChebyshevDerivativeCoefficients)
Algorithm 29 (FastChebyshevTransform)

$\left\{\tilde{f}_k\right\}_{k=0}^{N} \leftarrow$
$FastChebyshevTransform\left(\{f_j\}_{j=0}^{N}, \{w_j\}_{j=0}^{N-1}, \{C_j\}_{j=0}^{N}, \{S_j\}_{j=0}^{N}, FORWARD\right)$
$\left\{\tilde{f}_k^{(1)}\right\}_{k=0}^{N} \leftarrow ChebyshevDerivativeCoefficients\left(\{\tilde{f}_k\}_{k=0}^{N}\right)$
$\{\mathscr{D} f_j\}_{j=0}^{N} \leftarrow$
$FastChebyshevTransform\left(\{\tilde{f}_k^{(1)}\}_{j=0}^{N}, \{w_j\}_{j=0}^{N-1}, \{C_j\}_{j=0}^{N}, \{S_j\}_{j=0}^{N}, BACKWARD\right)$
return $\{\mathscr{D} f_j\}_{j=0}^{N}$
End Procedure FastChebyshevDerivative

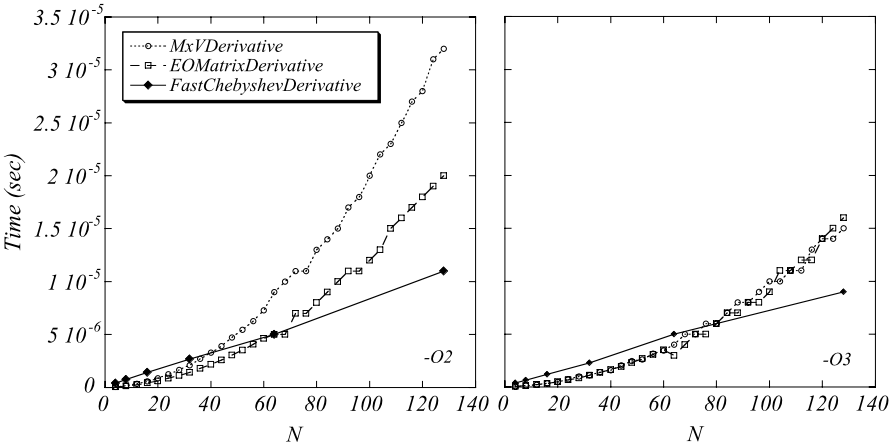

Fig. 3.3 A comparison of CPU times between three algorithms to compute a polynomial spectral derivative with two levels of optimization. Left: −O2. Right: −O3

rect matrix multiply in either graph. However by $N = 64$ and moderate optimization the even-odd decomposition is 1.7 times faster, which is consistent with the factor of two that we would expect by doing the decomposition. With more aggressive compiler optimization, however, there is virtually no difference between the speeds of the direct matrix multiply and even-odd decomposition algorithms. For our implementation, Algorithm 40 does not become competitive until $N = 64$. This could change significantly if we had used a highly tuned FFT.

Exercises

3.1 Reproduce Fig. 3.3 and compare the performance of the three algorithms, Algorithms 19 (MxVDerivative), 39 (EOMatrixDerivative), and 40 (FastChebyshevDerivative), for computing polynomial derivatives. Be sure to try different compiler optimization levels.

3.2 Plot and compare the error of the polynomial approximation of the derivative of the function $f(x) = \cos(2x)$ as a function of N using the three algorithms, Algorithms 19 (MxVDerivative), 39 (EOMatrixDerivative), and 40 (FastChebyshevDerivative). Include in your comparisons the Legendre and Chebyshev approximations. Compare also the errors with and without using the Negative Sum Trick to represent the derivative matrices.

3.3 Plot the locations of the nodes for both Chebyshev and Legendre Gauss and Gauss-Lobatto quadratures as a function of N for several values of N.

3.4 Derive the Even-Odd decomposition when N is odd.

3.5 The Chebyshev Gauss quadrature points have an important interpolation property that serves as a useful counter argument to the sometimes stated claim that interpolation at the Gauss points "wastes" points near the boundaries. The interpolation error $f(x) - P_N(x)$ is related to the derivatives of f and the nodes x_j by

$$f(x) - P_N(x) = \frac{f^{(N+1)}(\xi)}{(N+1)!} \prod_{j=0}^{N} (x - x_j).$$

The Gauss points are exactly those that minimize the maximum value of the polynomial factor of the interpolation error. (This result can be found in any elementary numerical analysis book.) Plot the interpolant of the function

$$f(x) = \frac{1}{1 + 25x^2}, \quad x \in [-1, 1]$$

for $N = 5, 10, 15, 20$ using the Gauss, Gauss-Lobatto and uniformly spaced points. Comment on your results. The effect seen when uniformly spaced points are used is called the *Runge Phenomenon*.

3.6 Develop and implement an algorithm to use Gauss quadrature to approximate integrals of the type

$$I = \int_{-1}^{1} \int_{-1}^{1} f(x, y) dx dy.$$

Modified versions of this algorithm will be used later in Chaps. 7 and 8.

Part II
Approximating Solutions of PDEs

Chapter 4
Survey of Spectral Approximations

Now that we know how to approximate functions, integrals and derivatives with high order orthogonal functions, we move to our ultimate goal and develop methods to approximate the solutions of partial differential equations (PDEs).

In this book, we will concentrate on the spectral approximation of three basic equations of mathematical physics, namely the potential equation

$$\nabla^2 \varphi = s, \tag{4.1}$$

the advection-diffusion equation

$$\varphi_t + \mathbf{q} \cdot \nabla \varphi = \nu \nabla^2 \varphi, \tag{4.2}$$

where $\mathbf{q}$ is some velocity field, and the wave equation

$$\varphi_{tt} - c^2 \nabla^2 \varphi = 0. \tag{4.3}$$

From the advection-diffusion equation we can immediately reduce to the scalar advection problem ($\nu = 0$) or the diffusion problem/heat equation ($\mathbf{q} = 0$). The form of the equations (4.1)–(4.3), the form in which the equations are usually written, is called the *strong* form.

The strong form of the equations may require the solutions to be more smooth than we want. For instance the temperature, φ, in a thin insulated rod of length, L, with variable thermal diffusivity and zero temperature specified at the ends is described by the initial-boundary value problem

$$\begin{cases} \varphi_t = \nu \varphi_{xx} + \nu_x \varphi_x, \quad x \in (0, L), \ t > 0, \\ \varphi(0, t) = \varphi(L, t) = 0, \\ \varphi(x, 0) = \varphi_0(x). \end{cases} \tag{4.4}$$

Its *classical solution* is the one that must be at least twice differentiable in space. The diffusivity, $\nu > 0$, must also be differentiable. We know physically, however, that at the joint between two materials with different thermal conductivities, ν will not be differentiable and a slope discontinuity will appear in the solution to make the heat flux, $f = \nu \varphi_x$, continuous. The piecewise smooth solution, which makes perfect physical sense, is not a classical solution to the heat equation. To include solutions with weaker smoothness constraints, we need to rewrite the equations.

To allow a larger class of solutions, we rewrite the PDE in a weak form. For example, if we write the right hand side of the PDE in (4.4) in the form $(\nu \varphi_x)_x$ and

D.A. Kopriva, *Implementing Spectral Methods for Partial Differential Equations,* Scientific Computation,

multiply by an arbitrary function ϕ and integrate over the domain, we find that the temperature also satisfies the equation

$$\int_0^L \varphi_t \phi dx = \int_0^L (\nu \varphi_x)_x \phi dx. \tag{4.5}$$

If ϕ is smooth enough, we can integrate the diffusion term by parts

$$\int_0^L \varphi_t \phi dx = \int_0^L (\nu \varphi_x)_x \phi dx = \nu \varphi \phi|_0^L - \int_0^L \nu \varphi_x \phi_x dx. \tag{4.6}$$

When we apply the boundary conditions, we are left with the *weak form* of the heat diffusion equation

$$\int_0^L \varphi_t \phi dx = -\int_0^L \nu \varphi_x \phi_x dx, \tag{4.7}$$

or using the inner product notation of the previous chapters,

$$(\varphi_t, \phi) = -(\nu \varphi_x, \phi_x). \tag{4.8}$$

The weak form of the heat equation requires only that the *weak solution* for the heat flux, $\nu \phi_x$, be square integrable, which is physically reasonable and less restrictive than requiring that ν be differentiable and φ be twice differentiable. It also implicitly defines the boundary conditions. On the other hand, if the solution is classical, then it satisfies both equations, for we can work backwards from (4.7), integrate by parts again, rearrange, and get

$$\int_0^L \left\{ \varphi_t - (\nu \varphi_x)_x \right\} \phi dx = 0, \tag{4.9}$$

provided that the boundary conditions are satisfied. Since the function ϕ is arbitrary and smooth enough, and the integral always vanishes, the quantity in braces must always vanish. In other words, the temperature satisfies the strong form of the equation, too.

We get a bonus from the weak form of the heat equation, for we can show immediately that the energy of the solution does not grow in time. Since ϕ is arbitrary, we can choose $\phi = \varphi$. Then (4.7) becomes

$$\int_0^L \varphi_t \varphi dx = -\int_0^L \nu \varphi_x \varphi_x dx. \tag{4.10}$$

We can rewrite the left hand side in terms of the energy, $\|\varphi\|$, if we pull the time derivative out of the integral. Furthermore, since $\nu > 0$, the energy satisfies

$$\frac{1}{2} \frac{d}{dt} \int_0^L \varphi^2 dx = -\int_0^L \nu (\varphi_x)^2 dx \leq 0. \tag{4.11}$$

Since the time derivative is always less than or equal to zero, it follows that the energy is always bounded by the initial energy,

$$\|\varphi\|_{L^2} \leq \|\varphi_0\|_{L^2} . \tag{4.12}$$

Spectral methods are high order techniques that we use to solve partial differential equations either in their strong form like (4.1)–(4.3) or in a weak form like (4.8). What sets spectral methods apart from others like finite difference methods (which start from the strong form of the equations) or finite element methods (which start from the weak form) is that to get a spectral method we approximate the solutions by high order orthogonal polynomial expansions. Whereas finite difference and finite element methods may be "high order" at orders two or three, spectral methods are run at orders up to the thousands. We have seen in Part I that orthogonal polynomial approximations can have very high convergence rates, which allow us to use fewer degrees of freedom for a desired level of accuracy.

We generate one of two types of spectral methods from the strong form of the equations. The most common method, known as *collocation*, looks much like a very high order finite difference method. To get a collocation method, we require that the PDE be satisfied at a set of grid, or more precisely, collocation points. We approximate derivatives at these grid points by the derivative of the nodal polynomial that interpolates the approximate solutions. Finite difference methods differ in that the derivative approximations are local to a grid point. Finite difference approximations to derivatives can be derived by differentiating a local low order polynomial interpolant at a point. We set the boundary conditions in a collocation method like we would in a finite difference method: We simply replace the grid point value at the boundary by the boundary condition. Another method known as the *penalty* method adds a term to the strong form of the equations to enforce the boundary conditions weakly. We will derive examples of collocation methods in Sects. 4.1 and 4.4. Penalty methods are covered in [14].

We also generate spectral methods from weak forms of the PDEs. The most common class of methods, known as *Galerkin* methods, look much like Galerkin finite element methods. To get a Galerkin method, we choose the functions ϕ, known as the *test functions*, from the same set of basis functions that we use to approximate the solution itself. The spectral Galerkin methods differ from finite element methods in that finite element methods use local functions as the test functions. Galerkin approximations naturally use the modal form to represent the solution. However, if we use a nodal representation of the solution and replace the integrals by Gauss quadratures, we can generate *nodal Galerkin* approximations. We derive examples of Galerkin spectral methods in Sects. 4.2, 4.5, 4.6, and 4.7. The *tau* method (so-called because the original had a parameter τ in it) is a modal method that starts from the weak form of an equation, but enforces boundary conditions differently than the Galerkin method. Tau methods are quickly remembered in [7].

In this chapter we will become familiar with spectral methods as we derive six approximations for the representative examples of the advection-diffusion and scalar advection equations in one space dimension. At the end of the chapter, after we

know what the methods are and how they are derived and implemented, we will regroup and make some sweeping generalizations about how to choose among them. In later chapters we will approximate and compute solutions to the three basic equations of mathematical physics that describe potentials, advection and diffusion, and the propagation of waves in two space dimensions.

4.1 The Fourier Collocation Method

The first spectral method that we will derive is known as the *Fourier spectral collocation* approximation. We will illustrate the derivation of the method by approximating the advection-diffusion equation with periodic boundary conditions,

$$\begin{aligned} &\varphi_t + \varphi_x = \nu\varphi_{xx}, \quad 0 < x < 2\pi,\ t > 0, \\ &\varphi(x,0) = \varphi_0(x), \quad 0 \le x \le 2\pi, \\ &\varphi(0,t) = \varphi(2\pi,t), \quad t \ge 0. \end{aligned} \tag{4.13}$$

For now, we will assume that $\nu > 0$ is a constant.

Since the problem is periodic, it is natural that we approximate the solution with a Fourier polynomial. To derive the collocation method, we approximate the solution as an interpolant, written in nodal form (1.56)

$$\varphi(x,t) \approx \Phi(x,t) = \sum_{n=0}^{N-1} \Phi_n(t) h_n(x), \tag{4.14}$$

where $\Phi_n(t) = \Phi(x_n,t)$ and the grid points are $x_n = 2\pi/N$. To determine the N unknowns, Φ_n, we need to find N independent equations.

We derive the collocation method like we derive a finite difference method. We find the equations necessary to determine the N unknowns by requiring that the approximate solution satisfy the differential equation at each of the N grid points,

$$\{\Phi_t + \Phi_x - \nu\Phi_{xx}\}|_{x_j} = 0, \quad j = 0, 1, \ldots, N-1. \tag{4.15}$$

Let's substitute for Φ with (4.14) and recall that $h_n(x_j) = \delta_{n,j}$ to get an ordinary differential equation to integrate in time for each of the grid point values Φ_j

$$\dot{\Phi}_j + \sum_{n=0}^{N-1} \Phi_n(t) h_n'(x_j) = \nu \sum_{n=0}^{N-1} \Phi_n(t) h_n''(x_j), \quad j = 0, 1, \ldots, N-1. \tag{4.16}$$

We now recognize that the two sums represent matrix-vector multiplications and so we rewrite this as

$$\dot{\Phi}_j + \mathscr{D}\Phi_i = \nu\mathscr{D}^2\Phi_j, \quad i = 0, 1, \ldots, N-1. \tag{4.17}$$

If we approximate $\mathscr{D}^2 \approx \mathscr{D}\mathscr{D}$ (see Problem 2.5), we get the collocation approximation

$$\dot{\Phi}_j = -\mathscr{D}\left(\Phi_j - \nu \mathscr{D}\Phi_j\right), \quad j = 0, 1, \ldots, N-1. \tag{4.18}$$

A finite difference approximation would also look like (4.17) if we wrote the finite differences as a matrix-vector multiplication. The standard second order approximation has a tri-diagonal matrix for the advection and diffusion terms. The matrices for the spectral collocation approximation are full.

Alternatively, we can use the modal form to represent the interpolant,

$$\Phi(x,t) = \sum_{k=-N/2}^{N/2} \frac{\tilde{\Phi}_k}{\bar{c}_k} e^{ikx}, \tag{4.19}$$

which gives the system of ordinary differential equations in terms of the coefficients

$$\dot{\Phi}_j + \sum_{k=-N/2}^{N/2} \frac{ik\tilde{\Phi}_k}{\bar{c}_k} e^{ikx_j} = \nu \sum_{k=-N/2}^{N/2} \frac{(-k^2)\tilde{\Phi}_k}{\bar{c}_k} e^{ikx_j}, \quad j = 0, 1, \ldots, N-1. \tag{4.20}$$

The two systems, (4.18) and (4.20) are not exactly equivalent, see Problem 2.5.

The approximation (4.14) automatically satisfies the boundary conditions, but we need an initial condition for each of the Φ_j to integrate (4.17) in time. The initial condition is usually taken to be the interpolant of the initial function φ_0, that is $\Phi_j(0) = \varphi_0(x_j)$, which has aliasing errors.

Like a finite difference method, the collocation method is easy to apply to general variable coefficient problems. Suppose, for instance, that the diffusion coefficient depends on the solution

$$\begin{aligned} &\varphi_t + \varphi_x = \left(\nu(\varphi)\varphi_x\right)_x, \quad x \in (0, 2\pi),\ t > 0, \\ &\varphi(x,0) = \varphi_0(x), \quad x \in (0, 2\pi), \\ &\varphi(0,t) = \varphi(2\pi, t), \quad t \geq 0. \end{aligned} \tag{4.21}$$

In the variable coefficient problem, we approximate the diffusive flux, $f \equiv \nu(\varphi)\varphi_x$, as a polynomial through its values at the collocation points,

$$F_j = \nu(\Phi_j)\mathscr{D}\Phi_j. \tag{4.22}$$

The approximation to the derivative of the diffusive flux is just the derivative of that interpolant, so the collocation approximation is

$$\begin{aligned} F_j &= \nu(\Phi_j)\mathscr{D}\Phi_j, \\ \dot{\Phi}_j + \mathscr{D}\Phi_j &= \mathscr{D}F_j, \quad j = 0, 1, \ldots, N-1, \\ \Phi_j(0) &= \varphi_0(x_j). \end{aligned} \tag{4.23}$$

Thus, collocation is an easy method to apply to variable coefficient problems, at the cost of introducing aliasing errors in the approximation of the diffusive flux and the initial condition.

We typically integrate the collocation approximation in time by some appropriate ODE solver. We compute the derivative approximations by one of the two representations, (1.73) or (1.74), depending on the size of N. If we use matrix multiplication, we compute the second derivative by applying the $\mathscr{D}$ matrix twice to the solution. If we use the DFT to compute the derivatives, then we compute the discrete coefficients from the mesh point values of the solution using the first equation of (1.69). We then form the coefficients of the first and second derivatives by multiplying each by ik and $-k^2$, respectively. We then evaluate the derivative at each mesh point by the second of the equations in (1.69).

4.1.1 How to Implement the Fourier Collocation Method

To implement the Fourier collocation approximation, we integrate the system of equations (4.18) in time with initial conditions $\Phi_j(0) = \varphi_0(x_j)$. Therefore, the first procedure that we implement computes the time derivative. We show an example in Algorithm 41 (FourierCollocationTimeDerivative), which computes the time derivative in the form

$$\begin{aligned} F_j &= \nu \mathscr{D} \Phi_j, \\ \dot{\Phi}_j &= \mathscr{D}\,(F - \Phi)_j\,, \end{aligned} \tag{4.24}$$

so it requires only two evaluations of the spatial derivative procedure instead of three. We can easily extend the procedure to a variable coefficient problem. The algorithm that we present uses matrix multiplication to compute the spatial derivative approximations. We could write a similar algorithm that uses the FFT, Algorithm 17 (FourierDerivativeByFFT).

Next, we need an algorithm to integrate of equations (4.24) in time. We usually integrate in time with a standard method for integrating systems of ODEs. The important consideration—besides accuracy, of course—is that the time integration method must be stable in time for this system of equations. If we use the Fourier transform, we see that the eigenvalues of the matrix $\nu D^2 - D$ are $\lambda_k = -(\nu k^2 + ik)$ for $k = -N/2 + 1, \ldots, N/2 - 1$. (See Problem 1.5.) Thus, we must choose a method whose region of absolute stability includes a portion of the left half of the complex plane. If we want to be able to compute pure advection problems where $\nu = 0$ and the eigenvalues of the system are purely imaginary, then we must choose an integrator whose region of absolute stability also includes the imaginary axis. Suitable methods include explicit third and higher order Runge-Kutta and Adams-Bashforth methods, or A-stable methods such as the backward Euler or trapezoidal rule (Crank-Nicolson). For thorough discussions of methods to integrate systems of ODEs, consult books on the subject like [17] or [3].

Algorithm 41: *FourierCollocationTimeDerivative:* The Fourier Collocation Time Derivative for the Advection-Diffusion Equation

Procedure FourierCollocationTimeDerivative
Input: $\{\Phi_j\}_{j=0}^{N-1}, \{D_{i,j}\}_{i,j=0}^{N-1}$
Uses Algorithms:
Algorithm 19 (MxVDerivative)
$\{F_j\}_{j=0}^{N-1} \leftarrow MxVDerivative(\{D_{i,j}\}_{i,j=0}^{N-1}, \{\Phi_j\}_{j=0}^{N-1})$
for $j = 0$ **to** $N-1$ **do**
  $F_j = \nu F_j - \Phi_j$
end
$\{\dot{\Phi}_j\}_{j=0}^{N-1} \leftarrow MxVDerivative(\{D_{i,j}\}_{i,j=0}^{N-1}, \{F_j\}_{j=0}^{N-1})$
return $\{\dot{\Phi}_j\}_{j=0}^{N-1}$
End Procedure FourierCollocationTimeDerivative

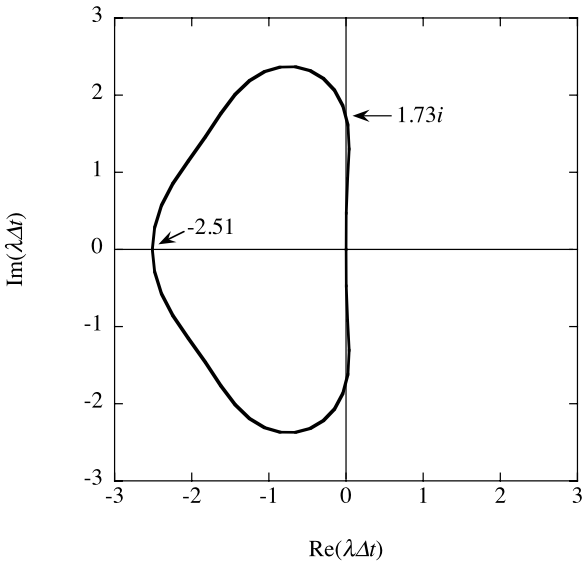

Fig. 4.1 Region of absolute stability for the third order Runge-Kutta method

As a concrete example, we will integrate the system (4.24) by Williamson's [25] third order low storage Runge-Kutta method. This method has the advantage of being appropriate for ν small or equal to zero, since its region of absolute stability includes the imaginary axis (Fig. 4.1). It requires only $2N$ levels of storage, which will be important later when we solve large systems of equations that result from multidimensional PDEs. Finally, the method is easy to implement.

To describe the low storage Runge-Kutta method, let's show how to integrate a generic system of ordinary differential equations, $\dot{u} = F(u,t)$. Let Δt be the time step and $t_n = n\Delta t$ be the current time. Next, let $U^n \approx u(t_n)$. Then we compute the

Algorithm 42: *CollocationStepByRK3:* Low-Storage Runge-Kutta Integration of the Fourier Collocation Approximation

Procedure CollocationStepByRK3
Input: $t_n, \Delta t, \{\Phi_j\}_{j=0}^{N-1}, \{D_{i,j}\}_{i,j=0}^{N-1}$
Uses Algorithms:
Algorithm 41 (FourierCollocationTimeDerivative)

```
for m = 1 to 3 do
    t ← t_n + b_m Δt
    {Φ̇_j}_{j=0}^{N-1} ← FourierCollocationTimeDerivative({Φ_j}_{j=0}^{N-1}, {D_{i,j}}_{i,j=0}^{N-1})
    for j = 0 to N − 1 do
        G_j ← a_m G_j + Φ̇_j
        Φ_j ← Φ_j + g_m Δt G_j
    end
end
return {Φ_j}_{j=0}^{N-1}
```

End Procedure CollocationStepByRK3

approximation at time t_{n+1} by

$$
\begin{aligned}
U &\leftarrow U^n, \\
G &\leftarrow F\left(U, t_n\right), \\
U &\leftarrow U + \frac{1}{3}\Delta t G, \\
G &\leftarrow -\frac{5}{9}G + F\left(U, t_n + \frac{1}{3}\Delta t\right), \\
U &\leftarrow U + \frac{15}{16}\Delta t G, \\
G &\leftarrow -\frac{153}{128}G + F\left(U, t_n + \frac{3}{4}\Delta t\right), \\
U^{n+1} &\leftarrow U + \frac{8}{15}\Delta t G.
\end{aligned}
\tag{4.25}
$$

In our context, we compute the time derivative, F by Algorithm 41 (FourierCollocationTimeDerivative) and implement the procedure in Algorithm 42 (CollocationStepByRK3). The procedure takes the current time, the time step and the solution at the current time and returns the solution at the next time level. To use the procedure we need to provide the coefficients of the method that we group in Table 4.1, and we assume that they have been pre-computed and stored. To save storage, we over-write the solution at time level n by the value at $n+1$. The array G is an intermediate storage array. Since the coefficients are not time dependent in this problem, Algorithm 42 uses neither the current time, t_n, nor its updates at each stage of the

Table 4.1 Coefficients for Williamson's 3rd order Runge-Kutta

m	a_m	b_m	g_m
1	0	0	1/3
2	−5/9	1/3	15/16
3	−153/128	3/4	8/15

Algorithm 43: *FourierCollocationDriver:* A Driver for the Fourier Collocation Approximation

Procedure FourierCollocationDriver
Input: N, N_T, T
Uses Algorithms:
Algorithm 18 (FourierDerivativeMatrix)
Algorithm 42 (CollocationStepByRK3)
$\{D_{ij}\}_{i,j=0}^{N-1} \leftarrow FourierDerivativeMatrix(N)$
$\Delta t \leftarrow T/N_T$
$t_n \leftarrow 0$
$\{\Phi_j\}_{j=0}^{N-1} \leftarrow InitialValues(N)$
for $n = 0$ **to** $N_T - 1$ **do**
  $\{\Phi_j\}_{j=0}^{N-1} \leftarrow CollocationStepByRK3(t_n, \Delta t, \{\Phi_j\}_{j=0}^{N-1}, \{D_{ij}\}_{i,j=0}^{N-1})$
  $t_n \leftarrow (n+1)\Delta t$
end
return $\{\Phi_j\}_{j=0}^{N-1}$
End Procedure FourierCollocationDriver

Runge-Kutta. We include them only to be complete, and so that we can easily modify the algorithm later to integrate more general systems.

Finally, we need a driver to integrate from the initial condition to the final time, T in N_T steps. We present an outline of such a procedure in Algorithm 43 (FourierCollocationDriver).

4.1.2 Benchmark Solution

To see how the Fourier Collocation method with third order Runge-Kutta integration in time behaves on the solution of the advection-diffusion equation, let's solve (4.13) for the initial condition

$$\varphi_0(x) = \frac{3}{5 - 4\cos(x)}. \tag{4.26}$$

We discussed this function in Sect. 1.3. It has the exact Fourier coefficients

$$\hat{\varphi}_{0,k} = 2^{-|k|}. \tag{4.27}$$

These coefficients decay exponentially fast with k, leading us to expect exponential convergence of the error if Δt is taken small enough. We compute the exact solution

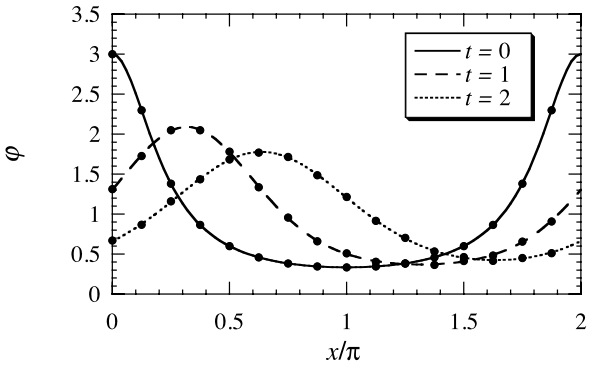

Fig. 4.2 Fourier collocation and exact solutions to the advection-diffusion equation. *Circles* represent the computed solutions at the collocation points; *Lines* represent the exact solutions

from its Fourier series,

$$\varphi = \sum_{k=-\infty}^{\infty} \hat{\varphi}_{0,k} e^{ik(x-t)-\nu k^2 t}, \tag{4.28}$$

so we can compute the error.

We'll test the collocation method with two approximations for the initial condition (4.26). The first is the normal interpolation approximation: We evaluate the initial condition as $\Phi_j(0) = \varphi_0(x_j)$, which as we have seen is the same thing as representing the polynomial approximation at the initial time by the interpolant, $\Phi(x,0) = I_N \varphi_0$. As we now know, this projection introduces aliasing errors. To see the effect of these aliasing errors, we also use the initial approximation $\Phi(x,0) = P_{N-1}\varphi_0$, which is free of aliasing errors.

Figure 4.2 compares the computed solutions with $N = 16$ to the exact solution at three times. Although the computed solution is plotted only at the collocation points, remember that we could have evaluated the interpolant representing these solutions at many points in the interval to get a more pleasing looking plot using either Algorithms 2 (FourierInterpolantFromModes) or 3 (FourierInterpolantFromNodes). Unlike finite difference approximations, the collocation solution (4.14) is defined everywhere in the interval.

Since we know the exact solution for this problem, we can compute the error of the approximation. We will measure the error in the *discrete norm*,

$$\|\varphi - \Phi\|_M^2 = \frac{2\pi}{M} \sum_{j=0}^{M-1} \big(\varphi(z_j, T) - \Phi(z_j, T)\big)^2 \tag{4.29}$$

with $z_j = 2j\pi/M$. The discrete norm is a spectrally accurate approximation to the true norm, (1.13). To see this, recall that for a square integrable function, f, $\tilde{f}_k = (f, e^{ikx})_M/2\pi$ and $\hat{f}_k = (f, e^{ikx})/2\pi$. If we set $k = 0$ then (1.71) says that

$$2\pi\, \tilde{f}_0 = \frac{2\pi}{M} \sum_{j=0}^{M-1} f\left(\frac{2j\pi}{M}\right) = \int_0^{2\pi} f(x)dx + 2\pi \sum_{\substack{p=-\infty \\ p\neq 0}}^{\infty} \hat{f}_{pM}. \tag{4.30}$$

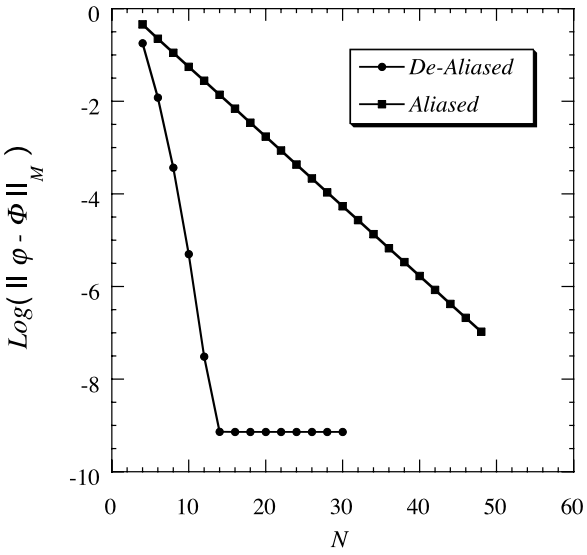

Fig. 4.3 Error decay for the Fourier Collocation approximation of the advection-diffusion equation

So the quadrature error depends only on the smoothness of the integrand through the rate of decay of the Fourier coefficients.

We show the decay of the error for the Fourier collocation approximation of the advection-diffusion equation in Fig. 4.3. To produce the graph, we chose $\Delta t = 1.25 \times 10^{-3}$ and computed the error at $M = 100$ points. For both projections of the initial conditions—with and without aliasing—the convergence rate is exponential. A least squares fit of the solution with aliasing shows that the slope is approximately -0.1506. In this problem the convergence rate is significantly faster if we remove the initial aliasing errors. This is because the initial truncation error is damped by the exponential factor $e^{-\nu k^2 t}$ (4.28). On the other hand, the aliases of the high frequency modes (starting with the $N/2$ mode) are damped as if they are low frequency modes, i.e., slowly in time. From (4.27) and (1.72) we expect that dominant error to decay as $2^{-N/2}$, or, in other words, with a slope of $-\text{Log}(2)/2 = -0.1505$ on the semi-log plot. Although it is true that the aliased solution is significantly worse than the de-aliased one, the convergence rate is still exponential in the number of degrees of freedom. Doubling the number of degrees of freedom decreases the error by three orders of magnitude.

4.2 The Fourier Galerkin Method

Our next example is a *Fourier Spectral Galerkin* method. To allow us to make direct comparisons to the collocation approximation, we will again approximate the initial boundary-value problem (4.13) for the advection-diffusion equation.

The derivation of the Galerkin approximation starts from a weak form of the PDE. If we multiply the PDE in (4.13) by the complex conjugate of any periodic

function ϕ and integrate over $[0, 2\pi]$, we see that the solution also satisfies

$$\int_0^{2\pi} \{\varphi_t + \varphi_x - \nu\varphi_{xx}\}\phi^* dx = 0. \tag{4.31}$$

When we expand the sum and use inner product notation for the integrals, we get the weak form

$$(\varphi_t, \phi) + (\varphi_x, \phi) = (\nu\varphi_{xx}, \phi)\,. \tag{4.32}$$

Let's look at the integral on the right. If we integrate it by parts,

$$(\nu\varphi_{xx}, \phi) = \nu\varphi_x\phi|_0^{2\pi} - (\nu\varphi_x, \phi_x)\,. \tag{4.33}$$

Then we can write the weak form of the advection-diffusion equation as

$$(\varphi_t, \phi) + (\varphi_x, \phi) = \nu\varphi_x\phi|_0^{2\pi} - (\nu\varphi_x, \phi_x)\,. \tag{4.34}$$

We now set out to get an approximate solution to (4.34). Since the problem is periodic, we again choose the Fourier basis with which to approximate the solution and approximate the solution by a Fourier polynomial of degree N, this time written in modal form

$$\varphi(x,t) \approx \Phi(x,t) = \sum_{n=-N/2}^{N/2} \hat{\Phi}_n(t) e^{inx}. \tag{4.35}$$

We see that there are $N+1$ degrees of freedom (the $\hat{\Phi}_n$ values) for us to find.

The Galerkin approximation determines the degrees of freedom by requiring that the approximate solution also satisfies (4.34) for ϕ being each of the $N+1$ basis functions, e^{ikx}, $k = -N/2, \ldots, N/2$. That is, the Galerkin approximation is the solution Φ that satisfies

$$\left(\Phi_t, e^{ikx}\right) + \left(\Phi_x, e^{ikx}\right) = \nu\Phi_x e^{ikx}\big|_0^{2\pi} - \left(\nu\Phi_x, \left(e^{ikx}\right)_x\right), \quad k = -N/2, \ldots, N/2. \tag{4.36}$$

Since Φ is a Fourier polynomial and its derivative is too, the flux, $\nu\Phi_x$ is periodic. Therefore, the boundary terms vanish and the approximate solution satisfies

$$\left(\Phi_t, e^{ikx}\right) + \left(\Phi_x, e^{ikx}\right) = -\left(\nu\Phi_x, \left(e^{ikx}\right)_x\right), \quad k = -N/2, \ldots, N/2. \tag{4.37}$$

We now find the coefficients $\hat{\Phi}_n$. When we substitute for Φ into (4.37),

$$\begin{aligned} &\sum_{n=-N/2}^{N/2} \dot{\hat{\Phi}}_n \left(e^{inx}, e^{ikx}\right) + \sum_{n=-N/2}^{N/2} in\hat{\Phi}_n \left(e^{inx}, e^{ikx}\right) \\ &\quad = -\sum_{n=-N/2}^{N/2} (in)\,(-ik)\,\hat{\Phi}_n \left(e^{inx}, e^{ikx}\right), \quad k = -N/2, \ldots, N/2. \end{aligned} \tag{4.38}$$

Orthogonality of the basis functions leaves us with

$$\dot{\hat{\Phi}}_k = -\left(ik + k^2\right)\hat{\Phi}_k, \quad k = -N/2, \ldots, N/2. \tag{4.39}$$

We see that the equations for the coefficients that the Galerkin approximation gives us are exactly the equations that we would derive by Fourier series (cf. Sect. 1.1).

As in the collocation approximation, the Galerkin approximation gives us a system of ordinary differential equations to integrate in time. The boundary conditions are satisfied because the basis functions do so individually. To integrate in time, we only need initial conditions. For that, we use truncation, $\Phi(x, 0) = P_N \varphi_0$. Therefore,

$$\hat{\Phi}_k(0) = \hat{\varphi}_{0,k}, \quad k = -N/2, \ldots, N/2. \tag{4.40}$$

Truncation ensures that there are no aliasing errors in the initial coefficients of the approximate solution.

Like the collocation approximation and a finite element approximation, the Fourier Galerkin approximation defines the solution at every point in space. At any time, we can compute the solution for any point in space using the sum in (4.35).

4.2.1 How to Implement the Fourier Galerkin Method

To implement the method, we need procedures to compute the time derivatives (4.39), to integrate that system of equations, and to evaluate the solution from the coefficients using (4.35). The first of these is straightforward; To compute the time derivatives of the coefficients, we can use Algorithm 44 (AdvectionDiffusionTimeDerivative), which takes the array of coefficients and returns the values of $\dot{\hat{\Phi}}_k$ using (4.39).

To start the time integration, we need the exact Fourier coefficients, $\hat{\varphi}_{0,k}$. Finding these is the hardest part of the whole procedure since we must compute the integrals in (1.19). As we saw in Sects. 1.5 and 1.6, we can approximate them as accurately as we like using the DFT, but for right now, let us assume that we have analytical expressions for the Fourier coefficients of the initial condition.

Algorithm 44: *AdvectionDiffusionTimeDerivative*: Advection-Diffusion Time Derivative for Fourier Galerkin

Procedure AdvectionDiffusionTimeDerivative
Input: $\{\hat{\Phi}_k\}_{k=-N/2}^{N/2}$
for $k = -N/2$ **to** $N/2$ **do**
  $\dot{\hat{\Phi}}_k \leftarrow -\left(ik + \nu k^2\right)\hat{\Phi}_k$
end
return $\{\dot{\hat{\Phi}}_k\}_{k=-N/2}^{N/2}$
End Procedure AdvectionDiffusionTimeDerivative

Algorithm 45: *FourierGalerkinStep:* Take One Time Step of the Fourier Galerkin Method

Procedure FourierGalerkinStep
Input: t_n, Δt, $\{\hat{\Phi}_k\}_{k=-N/2}^{N/2}$
Uses Algorithms:
Algorithm 44 (AdvectionDiffusionTimeDerivative)
for $m = 1$ **to** 3 **do**
  $t \leftarrow t_n + b_m \Delta t$
  $\{\dot{\hat{\Phi}}_k\}_{k=-N/2}^{N/2} \leftarrow AdvectionDiffusionTimeDerivative(\{\hat{\Phi}_k\}_{k=-N/2}^{N/2})$
  for $k = -N/2$ **to** $N/2$ **do**
    $G_k \leftarrow a_m G_k + \dot{\hat{\Phi}}_k$
    $\hat{\Phi}_k \leftarrow \hat{\Phi}_k + g_m \Delta t G_k$
  end
end
return $\{\hat{\Phi}_k\}_{k=-N/2}^{N/2}$
End Procedure FourierGalerkinStep

Algorithm 46: *EvaluateFourierGalerkinSolution:* Direct Synthesis of the Fourier Galerkin Solution

Procedure EvaluateFourierGalerkinSolution
Input: x, $\{\hat{\Phi}_k\}_{k=-N/2}^{N/2}$
$\Phi \leftarrow 0$
for $k = -N/2$ **to** $N/2$ **do**
  $\Phi \leftarrow \Phi + \hat{\Phi}_k e^{ikx}$
end
return Φ
End Procedure EvaluateFourierGalerkinSolution

We usually integrate the system of equations (4.39) in time with a standard method for integrating systems of ODEs. We will again use Williamson's third order low storage Runge-Kutta method. Algorithm 45 (FourierGalerkinStep) gives a version of the Runge-Kutta method tailored to integrate the Fourier-Galerkin approximation. As before, the coefficients of the method are given in Table 4.1.

Next, we need a way to evaluate the solution as a function of position for plotting and analysis purposes. In other words, we need an implementation of (4.35). We can implement the synthesis of the solution efficiently if we use the Fast Fourier Transform. (See Problem 4.3.) Algorithm 46 (EvaluateFourierGalerkinSolution) computes the sum directly. Note that although the coefficients $\hat{\Phi}_k$ are complex, the synthesized approximation, Φ, is real.

Finally, we need a driver to run it all. Algorithm 47 (FourierGalerkinDriver) provides an outline for code that we need to integrate the Fourier Galerkin approximation of the advection-diffusion equation from the initial condition to time T in N_T

Algorithm 47: *FourierGalerkinDriver:* A Driver for the Fourier Galerkin Approximation

Procedure FourierGalerkinDriver
Input: N, N_T, T, N_{out}
Uses Algorithms:
Algorithm 45 (FourierGalerkinStep)
Algorithm 46 (EvaluateFourierGalerkinSolution)

$\Delta t \leftarrow T/N_T$
$t_n \leftarrow 0$
$\{\hat{\Phi}_k\}_{k=-N/2}^{N/2} \leftarrow InitialCoefficients(N)$
for $n = 0$ **to** $M - 1$ **do**
  $\{\hat{\Phi}_k\}_{k=-N/2}^{N/2} \leftarrow FourierGalerkinStep(t_n, \Delta t, \{\hat{\Phi}_k\}_{k=-N/2}^{N/2})$
  $t_n \leftarrow (n+1)\Delta t$
end
$\Delta x \leftarrow 2\pi/N_{out}$
for $j = 0$ **to** N_{out} **do**
  $x_j \leftarrow j\Delta x$
  $\Phi_j \leftarrow EvaluateFourierGalerkinSolution(x_j, \{\hat{\Phi}_k\}_{k=-N/2}^{N/2})$
end
return $\{\Phi_j\}_{j=0}^{N_{out}}$
End Procedure FourierGalerkinDriver

steps. After it computes the coefficients at the final time, it re-constructs (synthesizes) the solution at a set of N_{out} points in the interval $[0, 2\pi]$.

Before we present results showing how accurate the Fourier Galerkin method is, let's derive the expected error under the assumption that we integrate the system (4.39) exactly in time. The first thing to notice is that the ODEs for the coefficients are exactly the ODEs for the exact solution. Thus,

$$\begin{aligned} \varphi(x,t) - \Phi(x,t) &= \sum_{|k|=N/2+1}^{\infty} \hat{\varphi}_k(t) e^{ikx} \\ &= \sum_{|k|=N/2+1}^{\infty} \hat{\varphi}_{0,k} e^{-(ik+\nu k^2)t} e^{ikx} \end{aligned} \tag{4.41}$$

so that

$$\|\varphi - \Phi\|^2 = \sum_{|k|=N/2+1}^{\infty} \left|\hat{\varphi}_{0,k}\right|_k^2 e^{-2\nu k^2 t} \leq \|\varphi_0 - P_N \varphi_0\|^2 . \tag{4.42}$$

Therefore, the Fourier Galerkin method is spectrally accurate, with an error that depends on how rapidly the coefficients of the initial condition decay with wavenumber. The last term on the right of (4.42) shows that the error depends directly on how well the Fourier truncation operator approximates the initial condition.

4.2.2 Benchmark Solution

As our benchmark problem for the Fourier Galerkin approximation to the advection-diffusion equation, let's solve (4.13) again for the initial condition (4.26) so that we know both the exact solution and the Fourier collocation solutions.

Figure 4.4 shows the computed solution plotted at $N_{out} = 50$ points at four times for $N = 16$, i.e., 17 degrees of freedom, for the diffusion coefficient $\nu = 0.2$. Clearly visible is the movement of the initial profile to the right with strong effects of diffusion that serve to eliminate the short wavelengths.

We show a plot of the error as a function of N and Δt in Fig. 4.5. In that figure we see two important characteristics of the error: First, the total error decays very rapidly (exponentially) then stalls when it is dominated by the time integration error. That time integration error is $O(\Delta t^3)$ for the third order Runge-Kutta method,

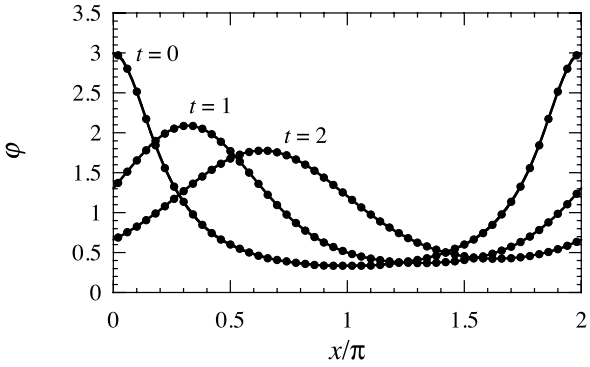

Fig. 4.4 Solution of the advection-diffusion equation by Fourier Galerkin at three times. The computed solutions, reconstructed at 50 points, are marked by *circles*. *Lines* are used to mark the exact solutions

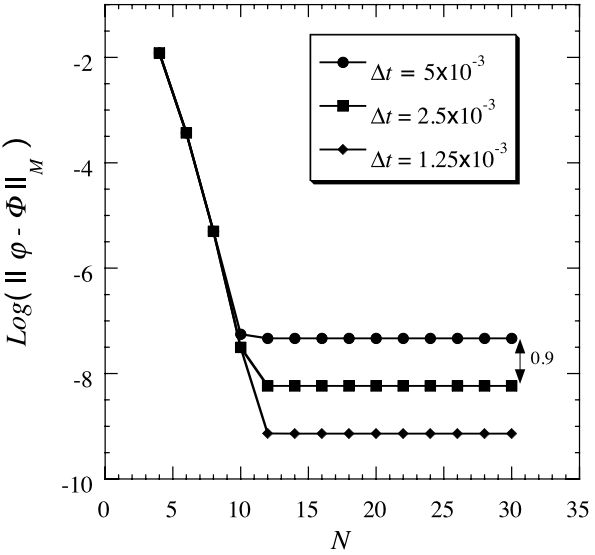

Fig. 4.5 Error in the Fourier Galerkin approximation at $t = 2$ as a function of N and Δt

so as Δt is decreased by a factor of two, the error decreases by a factor of eight, whose logarithm is $\log(8) = 0.90$. We can also compare to the collocation solution and see that the error for the dealiased collocation solution and the Galerkin approximation are the same. In fact, we can show that for this linear problem, the two approximations are equivalent (Problem 4.6).

4.3 Nonlinear and Product Terms

In Sects. 4.1 and 4.2 we derived the collocation and Fourier Galerkin approximations to the constant coefficient linear advection-diffusion equation. For that problem, the approximations were equivalent, provided that the collocation method is started with the Fourier truncation rather than interpolation approximation of the initial condition. They are not equivalent for variable coefficient or nonlinear problems because of aliasing.

In this section we examine a model that better represents the advection terms in equations such as the incompressible Navier-Stokes equations. The model is the Burgers equation,

$$u_t + uu_x = \nu u_{xx}. \tag{4.43}$$

We approximate the linear second derivative term the same as before with each of the two methods, collocation and Galerkin. The nonlinear term introduces new difficulties, so in this section we will discuss the Burgers equation with $\nu = 0$.

4.3.1 The Galerkin Approximation

If we follow the development of Sect. 4.2 to derive the Galerkin approximation to (4.43) with $\nu = 0$, we approximate the solution u with the polynomial U of the form (4.35) and compute the projections

$$\left(U_t + UU_x, e^{imx}\right) = 0, \quad m = -N/2, \ldots, N/2. \tag{4.44}$$

The coefficients therefore satisfy the equations

$$\frac{d\hat{U}_m}{dt} + \widehat{(UU_x)}_m = 0, \quad m = -N/2, \ldots, N/2, \tag{4.45}$$

where the

$$\widehat{(UU_x)}_m = \frac{1}{2\pi}\int_0^{2\pi} UU_x e^{-imx} dx \tag{4.46}$$

are Fourier coefficients of the product.

We now relate the Fourier coefficients of the product, $\widehat{(UU_x)}_m$, to the Fourier coefficients $\hat{U}_k$ of the approximate solution, U. In fact, we can derive the relationship more generally for the Fourier coefficients of the product of two functions. To be

more general, then, let's let V and W be Fourier polynomials of degree less than or equal to N. The Fourier coefficients of the product VW are related to the Fourier coefficients $\hat{V}$ and $\hat{W}$ by

$$\begin{aligned}\widehat{(VW)}_m &= \frac{1}{2\pi}\int_0^{2\pi}\left(\sum_{q=-N/2}^{N/2}\hat{V}_q e^{iqx}\right)\left(\sum_{p=-N/2}^{N/2}\hat{W}_p e^{ipx}\right)e^{-imx}dx \\ &= \sum_p\sum_q \hat{V}_q\hat{W}_p\left[\frac{1}{2\pi}\int_0^{2\pi} e^{i(q+p-m)x}dx\right] \\ &= \sum_p\sum_q \hat{V}_q\hat{W}_p\delta_{p+q,m}. \end{aligned} \tag{4.47}$$

For convenience, let us define new *padded* coefficients

$$\hat{\hat{V}}_k = \begin{cases}\hat{V}_k, & |k| \le N/2, \\ 0, & |k| > N/2,\end{cases} \tag{4.48}$$

so the coefficients of the product are related to the padded coefficients $\hat{\hat{V}}$ and $\hat{\hat{W}}$ by

$$\widehat{(VW)}_m = \sum_{p=-N/2}^{N/2}\hat{\hat{W}}_p\hat{\hat{V}}_{m-p}. \tag{4.49}$$

To get the coefficients in (4.45), let U correspond to V and W correspond to U_x. Then $\hat{V}_p = \hat{U}_p$ and $\hat{W}_p = ip\hat{U}_p$ so the system of equations to be integrated for the coefficients of the approximate solution (4.45) is

$$\frac{d\hat{U}_m}{dt} + \sum_{p=-N/2}^{N/2} ip\hat{U}_p\hat{U}_{m-p} = 0, \quad m = -N/2, \ldots, N/2, \tag{4.50}$$

under the implicit definition that $\hat{U}_p \equiv 0$ for $|p| > N/2$.

We see in (4.50) that the presence of the nonlinear advective term in (4.43) couples the coefficients of the solution. (Compare this with the linear equation, (4.39), where the coefficients are not coupled.) Furthermore, they are not generally equal to the Fourier coefficients of the exact product. For two square integrable functions u and v, the exact coefficients of the product are

$$\begin{aligned}\widehat{(vw)}_m &= \frac{1}{2\pi}\int_0^{2\pi}\left(\sum_{q=-\infty}^{\infty}\hat{v}_q e^{iqx}\right)\left(\sum_{p=-\infty}^{\infty}\hat{w}_p e^{ipx}\right)e^{-imx}dx \\ &= \sum_{n=-\infty}^{\infty}\sum_{m=-\infty}^{\infty}\hat{v}_q\hat{w}_p\left(\frac{1}{2\pi}\int_0^{2\pi} e^{i(q+p-m)}dx\right) \\ &= \sum_{p=-\infty}^{\infty}\hat{w}_p\hat{v}_{m-p}, \end{aligned} \tag{4.51}$$

so that the coefficients of the exact solution satisfy the system of equations

$$\frac{d\hat{u}_m}{dt} + \sum_{p=-\infty}^{\infty} ip\hat{u}_p\hat{u}_{m-p} = 0, \quad m = -\infty, \ldots, \infty. \tag{4.52}$$

The coefficients of the approximate and exact solutions match for wavenumber $m \leq N/2$ for the *linear* problem, (4.39), but don't for the nonlinear problem (4.44). Even for the low order coefficients, $\hat{U}_m \neq \hat{u}_m$.

One important property of the Galerkin approximation is that we can show directly that it is stable. Stable approximations are those for which we can bound the energy of the solution by an amount proportional to the initial energy,

$$\|U\|_{L^2} \leq Ke^{\alpha t} \|U_0\|_{L^2} \tag{4.53}$$

for some constants K and α independent of t and N. Stability says that the approximate solution cannot blow up over finite time. Equation (4.44), by way of the first part of Problem 4.7, implies that

$$(U_t + UU_x, U) = 0. \tag{4.54}$$

If we write the inner products directly as integrals, then we see that periodicity guarantees that

$$\frac{1}{2}\frac{d}{dt}\int_0^{2\pi} |U|^2\, dx = -\int_0^{2\pi} UU_xU dx = -\frac{1}{3}\int_0^{2\pi} \left(U^3\right)_x dx = -\left.\frac{U^3}{3}\right|_0^{2\pi} = 0. \tag{4.55}$$

Therefore, if we integrate exactly in time the energy of the approximation is

$$\int_0^{2\pi} |U|^2\, dx = const = \int_0^{2\pi} |U_0|^2\, dx \tag{4.56}$$

or

$$\|U\|_{L^2} = \|U_0\|_{L^2}. \tag{4.57}$$

4.3.2 How to Compute the Convolution Sum

The sum, (4.49), that we use to compute the coefficients of the Galerkin approximation of the nonlinear advection term is called the *convolution sum*. We could compute it directly using Algorithm 48 (DirectConvolutionSum). Since the convolution sum requires essentially $N+1$ multiplications for each of the $N+1$ coefficients in (4.50), the total work is $O(N^2)$. For large N, this work can be unacceptably large, especially since we will show that we can use the FFT instead and reduce the work to $O(N \operatorname{Log} N)$. For this reason, we only use Algorithm 48 when N is small, or for testing purposes.

Algorithm 48: *DirectConvolutionSum:* Direct (Slow) Computation of the Convolution Sum

Procedure DirectConvolutionSum
Input: $\{\hat{V}_k\}_{k=-N/2}^{N/2}, \{\hat{W}_k\}_{k=-N/2}^{N/2}$
for $k = -N/2$ **to** $N/2$ **do**
  $\widehat{(VW)}_k \leftarrow 0$
  for $p = MAX(-N/2, k - N/2)$ **to** $MIN(N/2, N/2 + k)$ **do**
    $\widehat{(VW)}_k \leftarrow \widehat{(VW)}_k + \hat{V}_{k-p} * \hat{W}_p$
  end
end
return $\{\widehat{(VW)}_k\}_{k=-N/2}^{N/2}$
End Procedure DirectConvolutionSum

We will use the FFT to reduce the work required to compute the sum (4.49) when N is large. The basic idea is to represent the product $V(x)W(x)$ by an *interpolant* $Q(x) = I_M(V(x)W(x))$ of sufficiently high order, M, that there are no aliasing errors. We compute the mesh point values of the interpolant and its coefficients efficiently with the FFT.

To compute the mesh point values of the interpolant of VW, we use the DFT and the Fourier coefficients of V and W to form a set of mesh point values at $M \geq N$ mesh points $y_j = 2\pi j/M$,

$$\begin{aligned} V_j &= \sum_{k=-M/2}^{M/2-1} \hat{\hat{V}}_k e^{iky_j}, \\ W_j &= \sum_{k=-M/2}^{M/2-1} \hat{\hat{W}}_k e^{iky_j}, \end{aligned} \qquad j = 0, 1, \ldots, M-1. \tag{4.58}$$

The padded coefficients are defined in (4.48). From V_j and W_j we compute the product $Q_j = V_j W_j$ for $j = 0, 1, \ldots, M-1$. On this grid, the values of Q define a Fourier polynomial of degree M, whose coefficients are defined by the discrete Fourier transform

$$\tilde{Q}_k = \frac{1}{M} \sum_{j=0}^{M-1} Q_j e^{-iky_j}, \quad k = -M/2, \ldots, M/2 - 1. \tag{4.59}$$

From the aliasing formula, (1.72), and the convolution sum (4.49), the coefficients of the interpolant and the exact coefficients of the product are related by

$$\tilde{Q}_k = \widehat{(VW)}_k + \sum_{\substack{q=-\infty \\ q\neq 0}}^{\infty} \widehat{(VW)}_{k+qM} = \widehat{(VW)}_k + \sum_{\substack{q=-\infty \\ q\neq 0}}^{\infty} \sum_{p=-N/2}^{N/2} \hat{\hat{W}}_p \hat{\hat{V}}_{k+qM-p}. \tag{4.60}$$

With a proper choice of M, we can eliminate the last (aliasing) term in (4.60) so that $\tilde{Q}_k = \widehat{(VW)}_k$. Remember that $\hat{\hat{V}}_m = 0$ for $m > N/2$. Thus, if M is chosen so that $|k + M - p| > N/2$ for $|k|, |p| \leq N/2$, then the aliasing term vanishes for all $q \neq 0$. The worst case occurs when $k = -N/2$ and $p = N/2$, which leads to

$$\begin{aligned} -\frac{N}{2} - \frac{N}{2} + M &> \frac{N}{2} \\ \implies \quad M &> \frac{3N}{2}. \end{aligned} \tag{4.61}$$

So if the product in physical space is computed with at least $3N/2 + 2$ points (M has to be even according to (4.58)), the interpolant $Q(x)$ through the values at those points matches the product UV exactly, and the Fourier interpolation coefficients $\tilde{Q}_k$ match $\widehat{(VW)}_k$. The only difference is that for large N we can evaluate the coefficients $\tilde{Q}_k$ using FFTs much faster than we can evaluate $\widehat{(VW)}_k$ by direct summation.

Remember that the application that is giving us the convolution sum is the approximation of the uu_x term in the equation and therefore V and W correspond to U and U_x. Then, again, $\hat{V}_p = \hat{U}_p$ and $\hat{W}_p = ip\hat{U}_p$. As we discussed in Sect. 1.7, the $-N/2$ mode of the derivative doesn't appear in the sum, so we account for that by setting $W_{-N/2} = 0$. For the approximation of nonlinear advection, then, the worst case above is $k = -N/2 + 1$, which leads to the more commonly reported result that $M \geq 3N/2$.

If we use the fast convolution sum with $M < 3N/2$, in particular $M = N$, then (4.60) shows that there will be aliasing errors introduced into the approximation. Our stability proof doesn't tell us anything about what happens with aliasing present, however experience has shown that the approximation of the Burgers equation aliasing errors is unstable then $\nu = 0$. For more on the approximation of the Burgers equation and the issues regarding aliasing errors, see Chap. 3 of [7].

To develop the fast convolution sum algorithm, let us assume that we already have the coefficients $\hat{V}_k$ and $\hat{W}_k$ stored in the sequence $-N/2, \ldots, N/2$, as would be natural in the Galerkin approximation (cf. 4.50). The coefficients must be padded and then re-ordered so that the FFT can be used, as we described in Sect. 2.1.1. Then we use the inverse FFT to compute the sums in (4.58). This is an order $M \log M$ operation. Once we have the values of Q_j at each grid point (an order M operation), we use the forward FFT to compute the coefficients of the product (4.59), which is another $M \log M$ operation. The procedure is finished when we reorder the coefficients to $-N/2, \ldots, N/2$. Overall, we see that the transformation requires order $M \log M$ operations vs. order N^2 operations for the direct sum. This is significant since M needs only to be 50% larger than N. We present the procedure for the fast convolution sum in Algorithm 49 (FastConvolutionSum). Since our FFT (Algorithm 8) requires $M = 2^p$, we have padded the coefficients to double the size. A more general FFT would allow us to decrease the size of M. Also, Algorithm 49 recomputes the trigonometric factors for the FFT. For highest efficiency, these should be precomputed and stored.

Algorithm 49: *FastConvolutionSum:* Computation of the Convolution Sum with the FFT

Procedure FastConvolutionSum
Input: $N, \{\hat{V}_k\}_{k=-N/2}^{N/2}, \{\hat{W}_k\}_{k=-N/2}^{N/2}$
Uses Algorithms:
Algorithm 7 (InitializeFFT)
Algorithm 8 (Radix2FFT)

$M = 2N$
for $k = -M/2$ **to** $M/2$ **do**
  $\tilde{\hat{V}}_k \leftarrow 0$
  $\tilde{\hat{W}}_k \leftarrow 0$
end
for $k = 0$ **to** $N/2$ **do**
  $\tilde{\hat{V}}_k \leftarrow \hat{V}_k$
  $\tilde{\hat{W}}_k \leftarrow \hat{V}_k$
end
for $k = -1$ **to** $-N/2$ *Step* -1 **do**
  $\tilde{\hat{V}}_{M+k} \leftarrow \hat{V}_k$
  $\tilde{\hat{W}}_{M+k} \leftarrow \hat{V}_k$
end
$\{w_k\}_{k=0}^{M-1} \leftarrow InitializeFFT(M, BACKWARD)$
$\{V_j\}_{j=0}^{M-1} \leftarrow Radix2FFT(\{\tilde{\hat{V}}_k\}_{k=0}^{M-1}, \{w_k\}_{k=0}^{M-1})$
$\{W_j\}_{j=0}^{M-1} \leftarrow Radix2FFT(\{\tilde{\hat{W}}_k\}_{k=0}^{M-1}, \{w_k\}_{k=0}^{M-1})$
for $j = 0$ **to** $M - 1$ **do**
  $Q_j \leftarrow V_j W_j$
end
$\{w_k\}_{k=0}^{M-1} \leftarrow InitializeFFT(M, FORWARD)$
$\{\tilde{Q}_k\}_{k=0}^{M-1} \leftarrow Radix2FFT(\{Q_j\}_{j=0}^{M-1}, \{w_k\}_{k=0}^{M-1})$
for $k = 0$ **to** $N/2$ **do**
  $\widehat{(VW)}_k \leftarrow \tilde{Q}_k$
end
for $k = -1$ **to** $-N/2$ *Step* -1 **do**
  $\widehat{(VW)}_k \leftarrow \tilde{Q}_{M+k}$
end
return $\{\widehat{VW}_k\}_{k=-N/2}^{N/2}$
End Procedure FastConvolutionSum

4.3.3 The Collocation Approximation

The collocation method approximates the nonlinear term $v = uu_x$ by a polynomial interpolant. We get different interpolants, however, depending on how we write the nonlinear term. For instance, for smooth enough solutions, the following equations

are equivalent:

$$\begin{aligned} &(i) \quad u_t + uu_x = 0, \\ &(ii) \quad u_t + f_x = 0, \\ &(iii) \quad u_t + \tfrac{2}{3} f_x + \tfrac{1}{3} uu_x = 0, \end{aligned} \tag{4.62}$$

where $f = \frac{1}{2}u^2$ is the flux. The collocation approximations, however, differ. The issues regarding the Fourier collocation approximation of the equations in (4.62) are the same that we encounter with finite difference approximations.

The most natural approximation would be to evaluate the nonlinear term at each grid point using form (i). That is, we would compute the value V_j at each grid point by

$$V_j = U_j \mathscr{D} U_j. \tag{4.63}$$

We can write the polynomial that this represents in terms of the interpolation operator, I_N

$$V = I_N \left(U U' \right). \tag{4.64}$$

The product of the two polynomials of degree N that represent u and its derivative is a polynomial degree $2N$. When we project that polynomial on a grid of N points by the interpolation projection we introduce an aliasing error.

An alternative is to use form (ii), evaluate $F_j = \frac{1}{2}U_j^2$ and to approximate that form by

$$V_j = \mathscr{D} F_j. \tag{4.65}$$

Computationally, this means that we square the grid point values of U and differentiate the polynomial that results. In terms of the interpolation projection, the polynomial that passes through the V_j at the grid points for this approximation is

$$V = \frac{1}{2} \left(I_N U^2 \right)'. \tag{4.66}$$

Both forms introduce aliasing errors, but are not equivalent since differentiation and interpolation do not commute. The second form (4.65), however, is conservative in the sense that the semi-discrete approximation satisfies

$$\frac{d}{dt} \int_0^{2\pi} U dx = 0, \tag{4.67}$$

whereas the first one does not.

We approximate the final form, (iii) by the combination of the two previous approximations,

$$V_j = \frac{2}{3} \mathscr{D} F_j + \frac{1}{3} U_j \mathscr{D} U_j. \tag{4.68}$$

The situation with collocation approximations is the same as with finite difference approximations. We get little guidance on how to choose among the various

collocation approximations *a priori*. The third form, especially, takes twice as much work to compute, so we might wonder why anyone would be interested in it at all. All three approximations have aliasing errors. We can show stability of the third form, however.

To show that the approximation with the third form for the nonlinear term

$$\frac{d}{dt}U_j + \frac{2}{3}\mathscr{D}F_j + \frac{1}{3}U_j\mathscr{D}U_j = 0, \quad j = 0, 1, \ldots, N-1 \tag{4.69}$$

is stable, we multiply each equation by $2\pi U_j/N$ and sum over j

$$\frac{2\pi}{N}\sum_{j=0}^{N-1} U_j\frac{d}{dt}U_j + \frac{2}{3}\frac{2\pi}{N}\sum_{j=0}^{N-1} U_j\mathscr{D}F_j + \frac{1}{3}\frac{2\pi}{N}\sum_{j=0}^{N-1} U_j^2\mathscr{D}U_j = 0. \tag{4.70}$$

We recognize the sums as the discrete inner products, so another way to write (4.70) is

$$\left(\frac{d}{dt}U, U\right)_N + \frac{2}{3}\left(\mathscr{D}F, U\right)_N + \frac{1}{3}\left(U^2, \mathscr{D}U\right)_N = 0. \tag{4.71}$$

Since U is a Fourier polynomial of degree less than or equal to N, the first discrete inner product is exact so we can write it as

$$\left(\frac{d}{dt}U, U\right)_N = \left(\frac{d}{dt}U, U\right) = \frac{1}{2}\frac{d}{dt}\|U\|^2. \tag{4.72}$$

The second inner product is also exact because $\mathscr{D}F = (I_N F)'$, so

$$\left(\mathscr{D}F, U\right)_N = \left(\frac{d}{dx}I_N\frac{U^2}{2}, U\right). \tag{4.73}$$

When we integrate by parts and use the fact that all the functions are periodic,

$$\frac{1}{2}\left(\frac{d}{dx}I_N U^2, U\right) = -\frac{1}{2}\left(I_N U^2, \frac{d}{dx}U\right) = -\frac{1}{2}\left(U^2, \frac{d}{dx}U\right)_N. \tag{4.74}$$

The last inner product in (4.71) is not exact because U^2 is not a Fourier polynomial of degree less than or equal to N. However, when we replace the second term in (4.71) with (4.74), the second and third terms cancel leaving us with

$$\frac{1}{2}\frac{d}{dt}\|U\|^2 = 0. \tag{4.75}$$

Therefore the approximation is stable, because (4.75) means that

$$\|U\| = \|U_0\|. \tag{4.76}$$

If we use the FFT to compute the spatial derivatives in the collocation approximation to the nonlinear Burgers equation then the work is overall $N \log N$, just like

the Galerkin approximation with the fast convolution sum. However, to guarantee that the approximation is stable, we need to compute two FFT derivatives per time step.

4.4 Polynomial Collocation Methods

When the boundary conditions are not periodic, we switch to orthogonal polynomial approximations. In this section, we will derive polynomial collocation approximations to the diffusion and advection equations separately.

4.4.1 Approximation of the Diffusion Equation

We first show how to approximate the solution to the initial-boundary-value problem to the diffusion equation

$$\begin{cases} \varphi_t = \varphi_{xx}, \quad -1 < x < 1, \\ \varphi(x,0) = \varphi_0(x), \quad -1 \le x \le 1, \\ \varphi(-1,t) = \varphi(1,t) = 0 \end{cases} \tag{4.77}$$

with polynomial collocation.

The development of the polynomial collocation method follows that of the Fourier Collocation method (Sect. 4.1). We approximate the solution φ by the polynomial interpolant

$$\varphi(x,t) \approx \Phi(x,t) = \sum_{n=0}^{N} \tilde{\Phi}_n(t)\phi_n(x) = \sum_{j=0}^{N} \Phi_j(t)\ell_j(x), \tag{4.78}$$

where the interpolation points, x_j, are now the nodes of the Gauss-Lobatto quadrature (1.129). The collocation method is nodal, with the fundamental unknowns taken to be the Φ_j's, so we will use the second, Lagrange, representation for the polynomial.

Next, we substitute the approximate solution, Φ, into the PDE and require that it satisfy the equation at the nodes. This requirement generates the system of ODEs for the nodal values of the solution

$$\dot{\Phi}|_{x_j} - \Phi_{xx}|_{x_j} = 0, \quad j = 1, 2, \ldots, N-1. \tag{4.79}$$

We use the boundary conditions to set the values at the end points, $\Phi_0 = \Phi_N = 0$. When we substitute for Φ with its nodal representation,

$$\sum_{i=0}^{N} \dot{\Phi}_i \ell_i(x_j) - \sum_{i=0}^{N} \Phi_i \ell_i''(x_j) = 0. \tag{4.80}$$

Algorithm 50: *CollocationStepByRK3:* Low Storage Runge-Kutta Integration of a Polynomial Collocation Approximation

Procedure CollocationStepByRK3
Input: t_n, Δt, $\{\Phi_j\}_{j=0}^N$, $\{D_{ij}\}_{i,j=0}^N$
Input: Procedures: *TimeDerivative*, g^L, g^R
for $m = 1$ **to** 3 **do**
  $t = t_n + b_m \Delta t$
  $\{\dot{\Phi}_j\}_{j=0}^N = TimeDerivative(\{\Phi_j\}_{j=0}^N, \{D_{ij}\}_{i,j=0}^N)$
  for $j = 0$ **to** N **do**
    $G_j = a_m G_j + \dot{\Phi}_j$
    $\Phi_j = \Phi_j + g_m \Delta t G_j$
  end
end
$\Phi_0 = g^L(t + \Delta t)$; $\Phi_N = g^R(t + \Delta t)$
return $\{\Phi_j\}_{j=0}^N$
End Procedure CollocationStepByRK3

We can simplify the system of ODEs because $\ell_i(x_j) = \delta_{i,j}$. With the simplification, the system of equations that we integrate in time becomes

$$\begin{cases} \dot{\Phi}_j = \sum_{i=0}^N \Phi_i \ell''(x_j) = D^{(2)} \Phi_j, \quad j = 1, 2, \ldots, N-1, \\ \Phi_0 = \Phi_N = 0, \end{cases} \tag{4.81}$$

where $D^{(2)}$ is the second derivative matrix that we defined in Sect. 3.5.

We integrate the system of ODEs (4.81) in time with an ODE integrator. If we use an implicit time integrator such as the trapezoidal rule to integrate the system (4.81), the matrices to invert will be full. For one-dimensional problems such as (4.77), inverting the matrix directly, say by an LU decomposition (Appendix D.1.2), is not an excessive cost. Nevertheless, the full matrices that arise in the collocation method do lead to greater cost per time step than a finite difference method.

We will use the explicit third order Runge-Kutta method that we already presented in Sect. 4.1 to integrate the system (4.81) in time. Algorithm 50 (CollocationStepByRK3) presents a generic procedure that takes as input the approximate solution, Φ^n, at time t_n, the derivative matrix D, which in this case is the second derivative matrix, and two procedures that compute the time derivatives at the interior and the boundaries. The time step routine returns the array of the approximate solution at time t_{n+1}.

Our choice of the explicit Runge-Kutta method to integrate the system in time limits the size of the time step, which depends on the eigenvalues of the second derivative matrix, which are real and grow with N. We do not have analytical representations for the eigenvalues of the polynomial derivative matrices, however. In general, they must be computed, say by the DGEEV eigenvalue routine in LAPACK. Figure 4.6 plots the magnitude of the maximum eigenvalue as a function of N for the Chebyshev and Legendre second derivative approximations for Dirichlet boundary conditions. This means we plot the eigenvalues for the matrix $D^{(2)}$ with the first

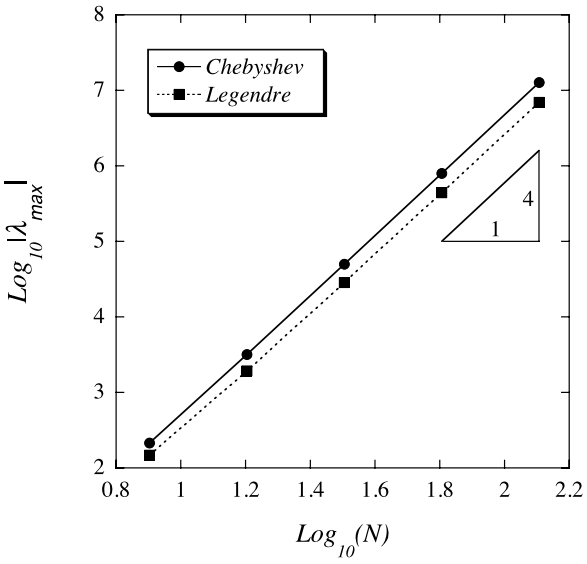

Fig. 4.6 Growth as N^4 of the maximum eigenvalue of the polynomial collocation second derivative matrix

and last rows and columns removed to account for the boundary conditions. We see from this graph that the maximum eigenvalue grows as N^4 for both the Chebyshev and the Legendre approximations. The maximum eigenvalue for the Legendre matrix, which we see grows approximately as $0.025N^4$, is approximately one half that of the Chebyshev matrix, which grows approximately as $0.047N^4$.

We determine the maximum time step from the largest eigenvalue. The limit on the time step for the third order Runge-Kutta method along the real axis is approximately $\lambda_{max}\Delta t = -2.51$. Thus, the maximum time step for the Legendre (4.81) approximation is approximately twice as large as the time step for the Chebyshev approximation, but in both cases $\Delta t \sim N^{-4}$.

4.4.2 How to Implement the Methods

We illustrate how to integrate the system from the initial to the final time in Algorithm 51 (LegendreCollocation), which includes the integrator and the time derivative procedures. The integrator first computes the Gauss-Lobatto points and weights, using Algorithm 25 (LegendreGaussLobattoNodesAndWeights). Only the nodes are needed for the collocation approximation. To change the approximation to Chebyshev, we would replace Algorithms 25 with 26 (ChebyshevGaussNodesAndWeights). The procedure then computes the second order derivative matrix using Algorithm 38 (mthOrderPolynomialDerivativeMatrix). Initialization is completed by computing the time step and evaluating the initial condition at the collocation points, which is illustrated by the user-supplied procedure *InitialValues*. The procedure then integrates the solution in time using Algorithm 50 (CollocationStepByRK3).

Algorithm 51: *LegendreCollocation:* Drivers for Legendre Collocation Approximation

Procedure LegendreCollocationIntegrator
Input: N, N_T, N_{out}, T
Uses Algorithms:
Algorithm 25 (LegendreGaussLobattoNodesAndWeights)
Algorithm 38 (mthOrderPolynomialDerivativeMatrix)
Algorithm 50 (CollocationStepByRK3)
Algorithm 30 (BarycentricWeights)
Algorithm 32 (PolynomialInterpolationMatrix)
Algorithm 33 (InterpolateToNewPoints)
Algorithm 19 (MxVDerivative)

$\left\{\{x_j\}_{j=0}^N, \{w_j\}_{j=0}^N\right\} \leftarrow LegendreGaussLobattoNodesAndWeights(N)$
$\left\{D_{ij}^2\right\}_{i,j=0}^N \leftarrow mthOrderPolynomialDerivativeMatrix(2, \{x_j\}_{j=0}^N)$
$\Delta t \leftarrow T/N_T$
$t_n = 0$
$\{\Phi_j\}_{j=0}^N = InitialValues(\{x_j\}_{j=0}^N)$
for $n = 0$ **to** $N_T - 1$ **do**
  $\{\Phi_j\}_{j=0}^N \leftarrow CollocationStepByRK3\left(t_n, \Delta t, \left\{\Phi_j^n\right\}_{j=0}^N, \left\{D_{ij}^2\right\}_{i,j=0}^N, TDerivative, g^L, g^R\right)$
  $t_n = (n+1)\Delta t$
end

for $j = 0$ **to** N_{out} **do**
  $X_j \leftarrow -1 + 2 * j/N_{out}$
end
$\{w_j\}_{j=0}^N \leftarrow BarycentricWeights(\{x_j\}_{j=0}^N)$
$\{T_{ij}\}_{i=0,j=0}^{N_{out},N} \leftarrow PolynomialInterpolationMatrix\left(\left\{x_j\right\}_{j=0}^N, \left\{w_j\right\}_{j=0}^N, \left\{X_j\right\}_{j=0}^{N_{out}}\right)$
$\left\{\Phi_j^I\right\}_{j=0}^{N_{out}} \leftarrow InterpolateToNewPoints\left(\left\{T_{ij}\right\}_{i=0,j=0}^{N_{out},N}, \left\{\Phi_j\right\}_{j=0}^N\right)$
return $\left\{\Phi_j^I\right\}_{j=0}^{N_{out}}$
End Procedure LegendreCollocationIntegrator

Procedure TDerivative
Input: $\left\{\Phi_j\right\}_{j=0}^N, \left\{D_{i,j}\right\}_{i,j=0}^N$
$\left\{\dot{\Phi}_j\right\}_{j=0}^N \leftarrow MxVDerivative\left(\left\{D_{i,j}\right\}_{i,j=0}^N, \left\{\Phi_j\right\}_{j=0}^N\right)$
return $\left\{\dot{\Phi}_j\right\}_{j=0}^N$
End Procedure TDerivative

At the conclusion of time integration loop, the calculation is finished, and the solution at the final time can be returned for plotting or other purposes. However, to illustrate the use of the interpolation algorithms that we derived in Sect. 3.4, the procedure returns the solution at the final time interpolated onto a uniformly distributed set of points, $\{X_j\}_{j=0}^K$. To interpolate the solution, the procedure computes the Barycentric weights for the Legendre Gauss-Lobatto collocation points using Algorithm 30 (BarycentricWeights). From the collocation points, it computes the interpolation weights and the new points. Finally, it computes the interpolation matrix, T, using Algorithm 32 (PolynomialInterpolationMatrix). (If these three steps

were done before the time integration, we could interpolate intermediate results, and the use of the interpolation matrix approach would be more cost effective.) Finally, Algorithm 33 (InterpolateToNewPoints) performs the actual interpolation, and the interpolated solutions are returned for plotting or printing.

Also included in Algorithm 51 (LegendreCollocation) are three routines to compute the time derivatives at the interior and boundary points. For the diffusion equation, time derivative is simply $D^2\Phi$, according to (4.81). For this reason, we evaluate the time derivative using Algorithm 19 (MxVDerivative). If we were using the Chebyshev instead of the Legendre approximation, and N was large enough, we would compute the time derivative by FFT techniques by applying Algorithm 40 (FastChebyshevDerivative). The time derivatives at the boundary are particularly simple in this example. The vanishing Dirichlet conditions mean that the time derivatives of the solution at the boundaries are zero, and $g^L(t) = g^R(t) = 0$.

4.4.3 Benchmark Solution

As a benchmark problem for Algorithm 51 (LegendreCollocation), we present results for the solution

$$\varphi(x,t) = \sin[\pi\,(x+1)]\,e^{-k^2\pi^2 t}, \tag{4.82}$$

which satisfies the boundary conditions. Figure 4.7 shows the computed solution, its interpolant and the exact solution for the Legendre collocation approximation at time $t = 0.1$ and $N = 12$. Enough time steps were taken so that the time integration error was negligibly small. We see that even on the very coarse grid, the nodal values

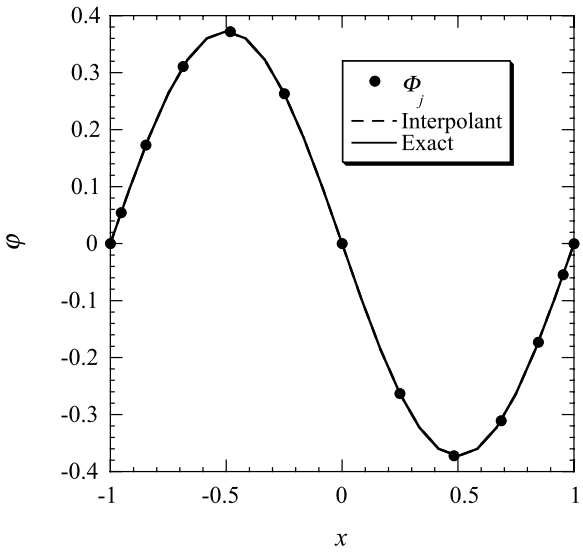

Fig. 4.7 Computed, exact, and interpolated solutions for the Legendre collocation approximation to the diffusion equation, (4.77)

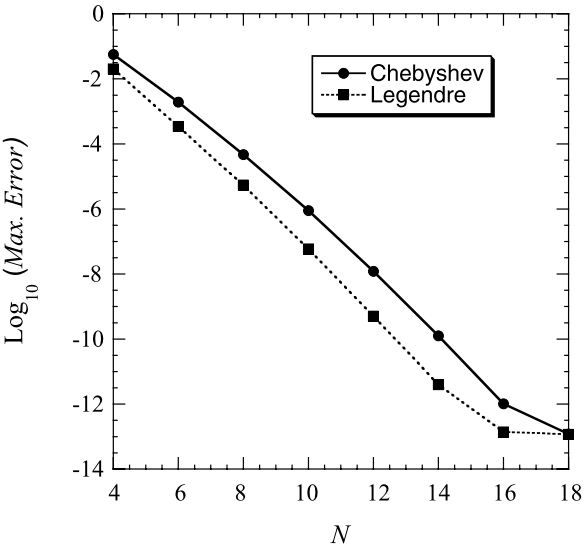

Fig. 4.8 Convergence of the maximum errors for Chebyshev and Legendre collocation approximations to the diffusion equation, (4.77)

of the computed solution and the interpolant are accurate to within plotting accuracy. To be quantitative about the error, we plot the error convergence as a function of N in Fig. 4.8 for both the Legendre and Chebyshev approximations. We see that for both polynomial bases, the error decays exponentially fast until rounding error is reached. For this particular problem, the Legendre approximation is more accurate than the Chebyshev approximation.

4.4.4 Approximation of Scalar Advection

The collocation approximation is equally easy to apply to the problem of scalar advection,

$$\begin{cases} \varphi_t + \varphi_x = 0, & -1 < x < 1, \\ \varphi(x,0) = \varphi_0(x), & -1 \le x \le 1, \\ \varphi(-1,t) = g(t), & t > 0. \end{cases} \tag{4.83}$$

For this equation, it is

$$\begin{cases} \dot{\Phi}_j + (D\Phi)_j = 0, & j = 1, 2, \ldots, N, \\ \Phi_0 = g(t), \end{cases} \tag{4.84}$$

where D is the first derivative matrix that we derived in Sect. 3.5.2 and that is computed with Algorithm 37 (PolynomialDerivativeMatrix). We can still integrate the

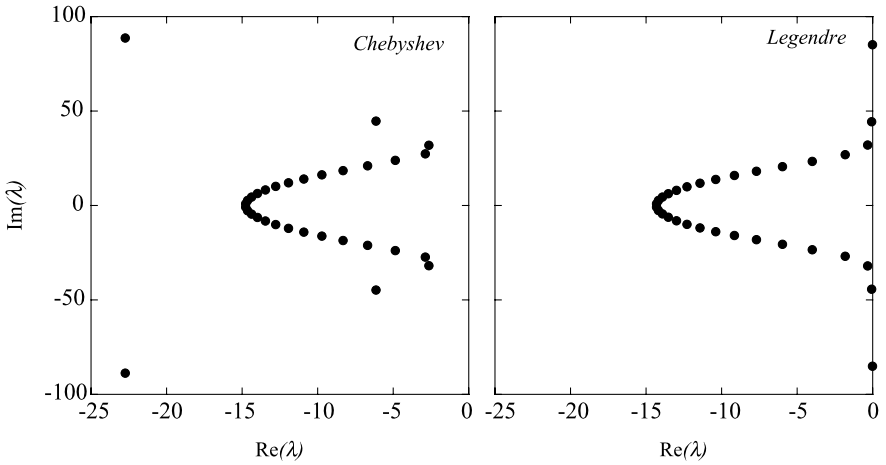

Fig. 4.9 Distribution of the eigenvalues for the Chebyshev and Legendre collocation first derivative matrices when $N = 32$

system of equations in time with Algorithm 51 (LegendreCollocation), with two simple modifications. We must change the derivative matrix and apply the boundary condition only to the left boundary in Algorithm 50.

The fact that we do not need to derive special approximations at or near the boundaries is an often noted advantage of spectral collocation methods over high order finite difference methods. At the left, only the boundary condition needs to be specified. At the right, no conditions at all need to be applied, independent of the order of the approximation. Contrast this with a high order finite difference method where special biased stencils must be derived near both ends of the interval. When solving large, complex problems, this advantage should not be underappreciated.

Since we will again use an explicit third order Runge-Kutta method to integrate in time, there will be a limit on the size of the time step that depends on the size of the eigenvalues of the derivative matrix. The Fourier transform tells us that the eigenvalues of the exact first derivative operator are purely imaginary, and we expect the eigenvalues of the derivative matrix to approximate this fact at the least. Analytic representations for the eigenvalues of the polynomial derivative matrices are not known, so we must again find the eigenvalues numerically.

Figure 4.9 shows the computed eigenvalues of the collocation first derivative matrices for $N = 32$. The eigenvalues have large imaginary parts and some negative real parts. The presence of the real parts indicates that the approximations are dissipative, that is, some energy is lost as the computation proceeds. Dissipation is important for the stability of variable coefficient and nonlinear problems, so its presence here is not necessarily a problem. We see also that the structure of the eigenvalues differs between the Chebyshev and Legendre approximations. The largest eigenvalues of the Legendre approximation are very near the imaginary axis, while the corresponding eigenvalues of the Chebyshev approximation have significantly larger real parts.

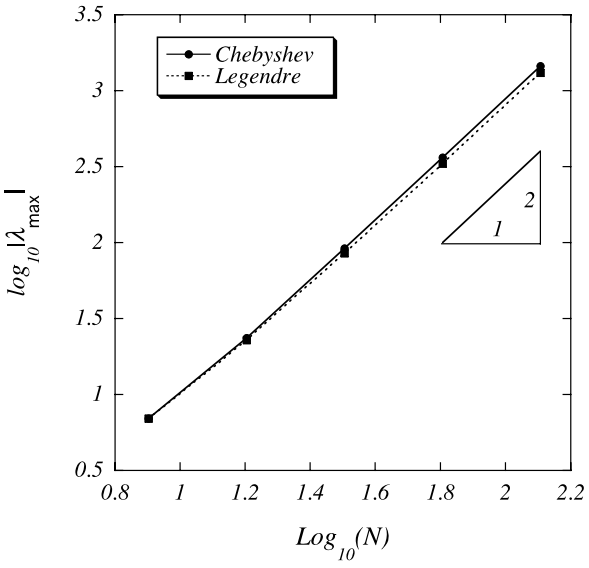

Fig. 4.10 Second order (N^2) growth of the maximum eigenvalue for the collocation first derivative matrices

The time step will be limited by the size and location in the complex plane of the largest eigenvalue. We show the growth of the magnitude of the largest eigenvalue in Fig. 4.10 for both the Chebyshev and Legendre approximations. We see from the graph that the largest eigenvalue grows asymptotically as $O(N^2)$. The largest eigenvalue is smaller for the Legendre approximation, but the difference is not nearly as large as in the diffusion example. Overall, we see that $\lambda_{max} \sim 0.09N^2$ for the Chebyshev approximation and $\lambda_{max} \sim 0.08N^2$ for the Legendre.

For the approximation to be stable in time, $\lambda_{max}\Delta t$ must fall within the region of absolute stability of the time integration method. It is more difficult to find the exact limit on Δt for the spectral polynomial advection approximations than for Fourier or centered finite difference approximations because the largest eigenvalue does not lie exactly on the imaginary axis. At best, we can only find a reasonable approximation to the largest time step that the approximation can take. Since we know that the eigenvalues grow as $Const \times N^2$, and since we know the constant approximately from Fig. 4.10, we can estimate the maximum time steps for the Chebyshev and Legendre approximations by $\Delta t \approx 1.73C/N^2$ where we find C so that the eigenvalues fall within the region of absolute stability. The value 1.73 comes from the fact that the region of absolute stability of the third order Runge-Kutta crosses the imaginary axis at $1.73i$. Figure 4.11 shows the location of $\lambda\Delta t$ for Δt approximated this way with $C = 11 \approx 1/0.09$ for the Chebyshev approximation and $C = 10$ for the Legendre. We see that with these choices for C, $\lambda\Delta t$ falls within the region for all eigenvalues λ when N is between 8 and 32. Figure 4.11 shows that the choice of C is conservative for the Chebyshev approximation. The time step could be increased by about 50% for $N > 16$. The estimate for Δt is just adequate for the Legendre approximation. We conclude from this exercise that the Chebyshev approximation can take a slightly larger time step than the Legendre approximation

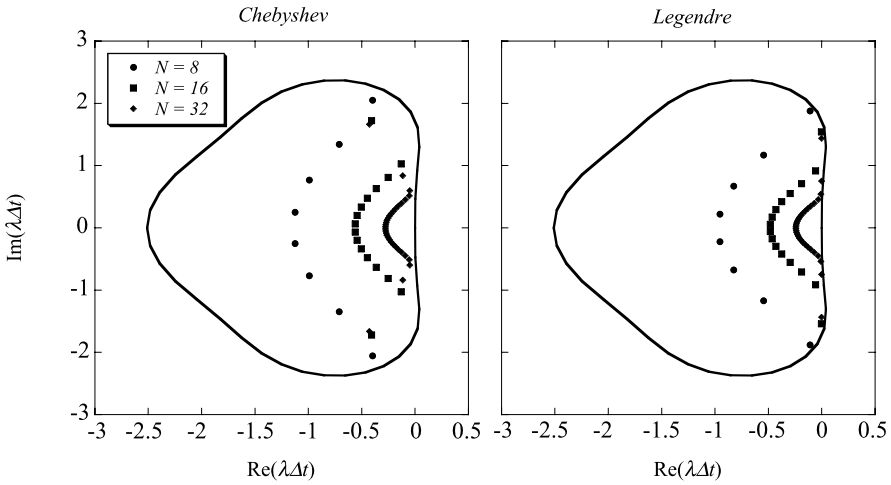

Fig. 4.11 Footprints of the eigenvalues of the Chebyshev and Legendre first derivative matrices for $N = 8, 16, 32$ scaled by $const \times 1.73/N^2$, relative to the region of absolute stability of the third order Runge-Kutta method. Stability requires $\lambda \Delta t$ to lie within the enclosed region

when used in conjunction with the third order Runge-Kutta method for the time integration.

We defer a benchmark solution until we have derived a discontinuous Galerkin approximation to the advection problem. At that point, we will compare three methods designed to solve the scalar advection equation in one space dimension.

4.5 The Legendre Galerkin Method

We can also derive polynomial Galerkin methods to PDEs. We introduce the Legendre Galerkin method by way of the heat equation,

$$\begin{cases} \varphi_t = \varphi_{xx}, \quad -1 < x < 1, \\ \varphi(x,0) = \varphi_0(x), \quad -1 \le x \le 1, \\ \varphi(-1,t) = \varphi(1,t) = 0. \end{cases} \tag{4.85}$$

Recall from the introduction to this chapter that the spectral Galerkin approximation has two basic characteristics. The first is that the solution is approximated by a polynomial of degree N,

$$\varphi \approx \Phi = \sum \hat{\Phi}_k(t)\phi_k, \tag{4.86}$$

where the basis functions ϕ_k individually satisfy the boundary conditions. The second is that the expansion coefficients, $\hat{\Phi}_k(t)$, are determined from a weak form of the equation.

Whereas the Fourier basis functions are periodic, which makes them suitable for problems with periodic boundary conditions, the Legendre polynomials (or the Chebyshev polynomials, for that matter) do not necessarily satisfy the boundary conditions for a given problem. In fact, from the recursion of the Legendre polynomials (1.82) we see that $L_k(1) = 1$ and $L_k(-1) = (-1)^k$, so they are not immediately suitable for the problem (4.85).

The first task is to find suitable polynomial basis functions that satisfy the boundary conditions. For the trivial boundary conditions in (4.85), several choices are possible, but for efficient computation, the polynomials

$$\phi_k(x) = \frac{1}{\sqrt{4k+6}}\left[L_k(x) - L_{k+2}(x)\right] \tag{4.87}$$

are particularly useful. A simple calculation shows that the ϕ_k satisfy the boundary conditions and that the approximation

$$\Phi = \sum_{k=0}^{N-2} \hat{\Phi}_k \phi_k \tag{4.88}$$

is a polynomial of degree N.

Next, we must find a set of equations for the coefficients, $\hat{\Phi}_k$. Since we are deriving a Galerkin approximation, we get them from the weak form of the diffusion equation

$$(\Phi_t, \phi_n) + \left(\Phi_x, \phi_n'\right) = 0, \quad n = 0, 1, \ldots, N-2. \tag{4.89}$$

If we now substitute the expansion for Φ,

$$\sum_{k=0}^{N-2} \dot{\hat{\Phi}}_k \left(\phi_k, \phi_n\right) + \sum_{k=0}^{N-2} \hat{\Phi}_k \left(\phi_k', \phi_n'\right) = 0. \tag{4.90}$$

Formally, this is a system of ordinary differential equations for the vector of the coefficients. That is, if we define two matrices M and S with $M_{kn} = (\phi_k, \phi_n)$ and $S_{kn} = (\phi_k', \phi_n')$, then

$$M\dot{\hat{\mathbf{\Phi}}} = -S\hat{\mathbf{\Phi}}, \tag{4.91}$$

where $\hat{\mathbf{\Phi}} = [\hat{\Phi}_0\ \hat{\Phi}_1\ \ldots\ \hat{\Phi}_{N-2}]^T$.

By choosing the basis functions (4.87), we make the matrices M and S particularly simple. The matrix S turns out to be the identity matrix, for the recursion formula for the derivatives of the Legendre polynomials, (1.83), implies that

$$L_{k+2}' - L_k' = (2k+3)\, L_{k+1}, \tag{4.92}$$

or

$$L_{k+1} = \frac{1}{(2k+3)}\left(L_{k+2}' - L_k'\right) = -\frac{2}{\sqrt{4k+6}}\phi_k'. \tag{4.93}$$

Thus,

$$\left(\phi_k', \phi_n'\right) = \frac{4k+6}{4}\left(L_{k+1}, L_{n+1}\right) = \delta_{k,n}. \tag{4.94}$$

Next, by direct calculation of the inner product (ϕ_k, ϕ_n), we find that the elements of the matrix, M, are given by

$$(\phi_k, \phi_n) = \alpha_k \alpha_n \left\{\beta_k \delta_{kn} + \gamma_n \delta_{k+2,n} + \mu_n \delta_{k,n+2}\right\} \tag{4.95}$$

where

$$\alpha_n = \frac{1}{\sqrt{4n+6}}, \qquad \gamma_n = -\frac{2}{2n+1}, \qquad \mu_n = -\frac{2}{2n+5}, \qquad \beta_n = -\left(\gamma_n + \mu_n\right). \tag{4.96}$$

Thus, the matrix M is pentadiagonal.

The fact that the jth row of M, as seen in (4.95), has non-zero elements in the jth, $j+2$nd and $j-2$nd columns means that the even and odd coefficients are decoupled. To take advantage of this decoupling, let us define new vectors of the even and odd indexed coefficients, namely, let

$$\begin{aligned} \hat{\Phi}_j^e &= \hat{\Phi}_{2j}, \quad j = 0, 1, \ldots, \left\lfloor \frac{N-2}{2} \right\rfloor, \\ \hat{\Phi}_j^o &= \hat{\Phi}_{2j+1}, \quad j = 0, 1, \ldots, \left\lfloor \frac{N-2+1}{2} \right\rfloor - 1. \end{aligned} \tag{4.97}$$

Now, from (4.95), the decoupling is represented in the fact that

$$(\phi_{2k}, \phi_{2n}) = \alpha_{2k}\alpha_{2n}\left\{\beta_{2k}\delta_{2k,2n} + \gamma_{2n}\delta_{2(k+1),n} + \mu_n \delta_{k,2(n+1)}\right\} \tag{4.98}$$

and

$$(\phi_{2k+1}, \phi_{2n}) = 0. \tag{4.99}$$

Thus,

$$\left(\dot{\Phi}, \phi_{2j}\right) = \beta_{2j}\alpha_{2j}^2 \dot{\hat{\Phi}}_j^e + \gamma_{2j}\alpha_{2(j-1)}\dot{\hat{\Phi}}_{j-1}^e + \mu_{2j}\alpha_{2(j+1)}\alpha_{2j}\dot{\hat{\Phi}}_{j+1}^e. \tag{4.100}$$

We can get a similar relation for the odd indexed coefficients. After decoupling, the systems of ODEs that the even and odd coefficients satisfy are

$$T^e \dot{\hat{\Phi}}^e = \hat{\Phi}^e, \qquad T^o \dot{\hat{\Phi}}^o = \hat{\Phi}^o, \tag{4.101}$$

where $T^{e,o}$ are now the tridiagonal matrices

$$T = \mathrm{diag}(l_j, d_j, u_j). \tag{4.102}$$

The diagonal, subdiagonal, and superdiagonal vectors of T are

$$\begin{aligned} d_j &= \alpha_{2j+p}^2 \beta_{2j+p}, \quad j = 0, 1, \ldots, \left\lfloor \frac{N-2+p}{2} \right\rfloor - p, \\ l_j &= \gamma_{2j} \alpha_{2j} \alpha_{2(j-1)}, \quad j = 1, 2, \ldots, \left\lfloor \frac{N-2+p}{2} \right\rfloor - p, \\ u_j &= \mu_{2j} \alpha_{2j} \alpha_{2(j+1)}, \quad j = 0, 1, \ldots, \left\lfloor \frac{N-2+p}{2} \right\rfloor - p - 1, \end{aligned} \tag{4.103}$$

where $p = 0$ for the even coefficients and $p = 1$ for the odds.

The matrices T are symmetric. They are also diagonally dominant, which allows us to use the Thomas algorithm (Algorithm 141) without pivoting to solve the systems (4.101).

We create the initial conditions for the systems given in (4.101) from the exact Legendre coefficients of the initial condition $\varphi_0(x)$. The Galerkin method approximates the initial coefficients by the orthogonal polynomial truncation of the initial condition

$$\Phi_0 = P_N \varphi_0 = \sum_{k=0}^{N} (\hat{\varphi}_0)_k L_k(x). \tag{4.104}$$

To get the coefficients $\hat{\Phi}_k^0$ from the Legendre coefficients, we take the orthogonal projection

$$\sum_{k=0}^{N-2} \hat{\Phi}_k^0 (\phi_k, \phi_m) = \sum_{k=0}^{N} (\hat{\varphi}_0)_k (L_k, \phi_m), \quad m = 0, 1, \ldots, N-2. \tag{4.105}$$

We have already derived the inner products on the left. The results are in (4.95). The relation between the ϕ_k and the L_k (4.87) implies that

$$(L_k, \phi_m) = -\alpha_m \gamma_m \delta_{km} + \mu_m \alpha_m \delta_{k,m+2}. \tag{4.106}$$

Thus, the inner products on the left side of (4.105) form a tri-diagonal matrix, and the inner products on the right form a bi-diagonal matrix. Again, the even and odd indexed coefficients decouple, so that we can find the initial coefficients also by solving two tri-diagonal matrix systems.

As before, when we developed the Fourier Galerkin method in Sect. 4.2, the approximation of nonlinear and non-constant coefficient problems is more complicated because of the need to evaluate the integrals analytically. The polynomial Galerkin method has the additional difficulty of needing to construct the basis functions so that they satisfy the boundary conditions. Nevertheless, we see that the Galerkin approximation can be made very efficient with work rivaling a second order finite difference method, yet with exponential convergence of the error. For more general problems with variable coefficients and more complex boundary conditions,

however, it would be useful to replace integrals with quadrature that will retain spectral accuracy for the approximation. Approximation with quadratures is the topic of the following two sections.

4.5.1 How to Implement the Method

Let us now develop the algorithms that we need to implement the Legendre Galerkin method. We present procedures to construct the modified Legendre basis in Algorithm 52 (ModifiedLegendreBasis) and to reconstruct the solution in Algorithm 53 (EvaluateLegendreGalerkinSolution).

We use the procedure in Algorithm 54 (InitTMatrix) to compute the matrix T. We assume in that procedure that functions to evaluate the coefficients α, β and γ by way of (4.96) are available. The procedure also explicitly uses the fact that the matrices are symmetric.

We show how to compute the initial coefficients $\hat{\Phi}_k$ in Algorithm 55 (ModifiedCoefsFromLegendreCoefs). For efficiency, we compute the three vectors for each of the tri-diagonal matrices once and store them.

Next, we need to integrate the systems of ODEs for the even and odd coefficients in time. Since the systems are tri-diagonal, it is convenient to use the second order,

Algorithm 52: *ModifiedLegendreBasis:* The Legendre Basis Modified to Vanish at Endpoints

Procedure ModifiedLegendreBasis
Input: k, x
Uses Algorithms:
Algorithm 20 (LegendrePolynomial)

$\phi_k \leftarrow LegendrePolynomial(k, x) - LegendrePolynomial(k+2, x)$; $\phi_k \leftarrow \phi_k / \sqrt{4k+6}$
return ϕ_k
End Procedure ModifiedLegendreBasis

Algorithm 53: *EvaluateLegendreGalerkinSolution:* Synthesis of the Legendre Galerkin Solution

Procedure EvaluateLegendreGalerkinSolution
Input: $N, x, \{\hat{\Phi}_k\}_{k=0}^{N-2}$
Uses Algorithms:
Algorithm 52 (ModifiedLegendreBasis)

$U \leftarrow 0$
for $k = 0$ **to** $N-2$ **do**
  $\Phi \leftarrow \Phi + \hat{\Phi}_k * ModifiedLegendreBasis(k, x)$
end
return Φ
End Procedure EvaluateLegendreGalerkinSolution

Algorithm 54: *InitTMatrix:* Legendre Galerkin Tridiagonal Matrix

Procedure initTMatrix
Input: N, p
for $j = 0$ **to** N **do**
  $d_j \leftarrow \beta_{2j+p} * \alpha^2_{2j+p}$
end
for $j = 1$ **to** N **do**
  $l_j \leftarrow \gamma_{2j+p} * \alpha_{2j+p} * \alpha_{2(j-1)+p}$
  $u_{j-1} \leftarrow l_j$
end
return $\left\{l_j\right\}_{j=2}^{N}, \left\{d_j\right\}_{j=1}^{N}, \left\{u_j\right\}_{j=1}^{N-1}$
End Procedure initTMatrix

Algorithm 55: *ModifiedCoefsFromLegendreCoefs:* Computing the Modified Legendre Coefficients from Legendre Coefficients

Procedure ModifiedCoefsFromLegendreCoefs
Input: $\left\{\hat{\varphi}_k\right\}_{k=0}^{N}$
Uses Algorithms:
  Algorithm 141 (TriDiagonalSolve)
  Algorithm 54 (initTMatrix)
`/* Even index coefficients */`
$M \leftarrow \lfloor (N-2)/2 \rfloor$
$\left\{\left\{l_j\right\}_{j=1}^{M}, \left\{d_j\right\}_{j=0}^{M}, \left\{u_j\right\}_{j=0}^{M-1}\right\} \leftarrow initTMatrix(M, 0)$
for $j = 0$ **to** M **do**
  $rhs_j \leftarrow \mu_{2j} * \alpha_{2j} * \hat{\varphi}_{2j+2} - \alpha_{2j} * \gamma_{2j} * \hat{\varphi}_{2j}$
end
$\left\{b_j\right\}_{j=0}^{M} \leftarrow TriDiagonalSolve(M, \{l_j\}_{j=1}^{M}, \{d_j\}_{j=0}^{M}, \{u_j\}_{j=0}^{M-1}, \{rhs_j\}_{j=0}^{M})$
for $j = 0$ **to** M **do**
  $\hat{\Phi}_{2j} \leftarrow b_j$
end
`/* Odd index coefficients */`
$M \leftarrow \lfloor (N-2+1)/2 \rfloor - 1$
$\left\{\left\{l_j\right\}_{j=1}^{M}, \left\{d_j\right\}_{j=0}^{M}, \left\{u_j\right\}_{j=0}^{M-1}\right\} \leftarrow initTMatrix(M, 1)$
for $j = 0$ **to** M **do**
  $rhs_j \leftarrow \mu_{2j+1} * \alpha_{2j+1} * \hat{\varphi}_{2j+3} - \alpha_{2j+1} * \gamma_{2j+1} * \hat{\varphi}_{2j+1}$
end
$\left\{b_j\right\}_{j=0}^{M} \leftarrow TriDiagonalSolve(M, \{l_j\}_{j=1}^{M}, \{d_j\}_{j=0}^{M}, \{u_j\}_{j=0}^{M-1}, \{rhs_j\}_{j=0}^{M})$
for $j = 0$ **to** M **do**
  $\hat{\Phi}_{2j+1} \leftarrow b_j$
end
return $\left\{\hat{\Phi}_k\right\}_{k=0}^{N-2}$
End Procedure ModifiedCoefsFromLegendreCoefs

implicit trapezoidal rule in time,

$$T\frac{\hat{\boldsymbol{\Phi}}^{n+1}-\hat{\boldsymbol{\Phi}}^{n}}{\Delta t}=-\frac{\hat{\boldsymbol{\Phi}}^{n+1}+\hat{\boldsymbol{\Phi}}^{n}}{2}, \tag{4.107}$$

which we rewrite as

$$\left(T+\frac{\Delta t}{2}I\right)\hat{\boldsymbol{\Phi}}^{n+1}=\left(T-\frac{\Delta t}{2}I\right)\hat{\boldsymbol{\Phi}}^{n}, \tag{4.108}$$

for each of the even and odd indexed coefficient vectors. What we see is that we can take one time step of the Legendre Galerkin method for the constant coefficient heat equation for the same amount of work as a standard, second order Crank-Nicolson finite difference scheme.

Algorithm 56 (LegendreGalerkinStep) implements the time-stepping procedure. It takes the coefficients at time level n and returns the values at the new time level, $n+1$. For the sake of clarity, this algorithm re-computes the tri-diagonal matrices. For efficiency those coefficients should be pre-computed and stored.

Finally, the computation of the Galerkin approximation requires a driver to integrate from the initial to the final times. We can easily modify Algorithm 47 (FourierGalerkinDriver) to drive the Legendre Galerkin approximation by substituting the routines in Algorithms 55 (ModifiedCoefsFromLegendreCoefs), 56 (LegendreGalerkinStep) and 53 (EvaluateLegendreGalerkinSolution) for the procedures *InitialCoefficients*, *FourierGalerkinStep*, and *EvaluateFourierGalerkinSolution*.

4.6 The Nodal Continuous Galerkin Method

An approximation that is intermediate between the Galerkin method and the collocation method starts with the Galerkin formulation, but replaces integrals with Gauss quadratures. The result is a nodal method that is significantly easier to implement than the Galerkin method. It is also the foundation of the Spectral Element Method that we discuss in Chap. 8. In the context of our discussions above, we make the following choices: We will use the Galerkin weak form of the equation and Legendre expansions to keep the weight function constant. We will represent the solution in nodal form and use a quadrature approximation to approximate the integrals that arise. It is possible to integrate analytically or to use exact quadrature with a nodal representation of the solution, but such methods are not commonly used in practice. For this reason, we will refer to the method presented here as the *nodal continuous Galerkin method*.

Our description of the approximation will again be in the context of the diffusion equation, (4.77), as our model problem

$$\begin{cases}\varphi_t=\varphi_{xx}, & -1<x<1,\\ \varphi(x,0)=\varphi_0(x), & -1\le x\le 1,\\ \varphi(-1,t)=\varphi(1,t)=0, & t>0.\end{cases} \tag{4.109}$$

Algorithm 56: *LegendreGalerkinStep:* Take One Time Step by Trapezoidal Rule

Procedure LegendreGalerkinStep;
Input: Δt, $\left\{\hat{\Phi}_k^n\right\}_{k=0}^{N-2}$
Uses Algorithms:;
Algorithm 141 (TriDiagonalSolve);
Algorithm 54 (initTMatrix);

`/* Even index coefficients */`

$M \leftarrow \lfloor (N-2)/2 \rfloor$;
$\left\{\{l_j\}_{j=1}^{M}, \{d_j\}_{j=0}^{M}, \{u_j\}_{j=0}^{M-1}\right\} \leftarrow initTMatrix(M, 0)$
$rhs_0 \leftarrow (d_0 - \Delta t/2) * \hat{\Phi}_0^n + u_0 * \hat{\Phi}_2^n$;
for $j = 1$ **to** $M-1$ **do**
  $rhs_j \leftarrow l_j * \hat{\Phi}_{2(j-1)}^n + (d_j - \Delta t/2) * \hat{\Phi}_{2j}^n + u_j * \hat{\Phi}_{2(j+1)}^n$;
end
$rhs_M \leftarrow (d_M - \Delta t/2) * \hat{\Phi}_{2M}^n + l_M * \hat{\Phi}_{2(M-1)}^n$;
for $j = 0$ **to** M **do**
  $d_j \leftarrow d_j + \Delta t/2$;
end
$\left\{\hat{\Phi}_j\right\}_{j=0}^{M} \leftarrow TriDiagonalSolve(M, \{l_j\}_{j=1}^{M}, \{d_j\}_{j=0}^{M}, \{u_j\}_{j=0}^{M-1}, \{rhs_j\}_{j=0}^{M})$;
for $j = 0$ **to** M **do**
  $\hat{\Phi}_{2j}^{n+1} \leftarrow \hat{\Phi}_j$;
end

`/* Odd index coefficients */`

$M \leftarrow \lfloor (N-2+1)/2 \rfloor - 1$;
$\left\{\{l_j\}_{j=1}^{M}, \{d_j\}_{j=0}^{M}, \{u_j\}_{j=0}^{M-1}\right\} \leftarrow initTMatrix(M, 1)$
$rhs_0 \leftarrow (d_0 - \Delta t/2) * \hat{\Phi}_1^n + u_0 * \hat{\Phi}_3^n$;
for $j = 1$ **to** $M-1$ **do**
  $rhs_j \leftarrow l_j * \hat{\Phi}_{2(j-1)+1}^n + (d_j - \Delta t/2) * \hat{\Phi}_{2j+1}^n + u_j * \hat{\Phi}_{2(j+1)+1}^n$;
end
$rhs_M \leftarrow (d_M - \Delta t/2) * \hat{\Phi}_{2M+1}^n + l_M * \hat{\Phi}_{2(M-1)+1}^n$;
for $j = 0$ **to** M **do**
  $d_j \leftarrow d_j + \Delta t/2$;
end
$\left\{\hat{\Phi}_j\right\}_{j=0}^{M} \leftarrow TriDiagonalSolve(M, \{l_j\}_{j=1}^{M}, \{d_j\}_{j=0}^{M}, \{u_j\}_{j=0}^{M-1}, \{rhs_j\}_{j=0}^{M})$;
for $j = 0$ **to** M **do**
  $\hat{\Phi}_{2j+1}^{n+1} \leftarrow \hat{\Phi}_j$;
end
return $\left\{\hat{\Phi}_k^{n+1}\right\}_{k=0}^{N-2}$;
Procedure LegendreGalerkinStep;

Since we are deriving a Galerkin approximation, we start with the weak form of the PDE,

$$(\varphi_t, \phi) + (\varphi_x, \phi_x) = 0 \tag{4.110}$$

for any sufficiently smooth ϕ that satisfies the boundary conditions.

As usual, we approximate the solution by a polynomial of degree N. We will take advantage of Gauss-Lobatto quadratures to approximate integrals, so it is convenient to use the nodal Lagrange form with nodes at the Gauss-Lobatto points. Thus, we approximate

$$\varphi(x,t) \approx \Phi(x,t) = \sum_{j=0}^{N} \Phi_j(t)\ell_j(x). \tag{4.111}$$

To have the approximation satisfy the boundary conditions, we set $\Phi_0 = \Phi_N = 0$.

So far, we have re-specified the Legendre Galerkin approximation of the previous section. Now we will take advantage of flexibility given to us by the nodal representation of a polynomial. Since Φ satisfies (4.110) for any function ϕ, it must satisfy the condition for all linear combinations of the basis functions,

$$\phi = \sum_{k=0}^{N} b_n \phi_n(x) \tag{4.112}$$

with arbitrary coefficients b_n. Since ϕ is an Nth order polynomial we can write it, too, in the equivalent Lagrange form,

$$\phi = \sum_{j=0}^{N} \phi_j \ell_j(x), \tag{4.113}$$

where the nodal values ϕ_j are arbitrary, except that $\phi_0 = \phi_N = 0$ to ensure that ϕ satisfies the boundary conditions.

To get the nodal Galerkin approximation, we approximate the integrals in (4.110) rather than evaluate them analytically. We replace them by Legendre Gauss-Lobatto quadrature (Sect. 1.11), which we write as

$$(\Phi_t, \phi)_N + (\Phi_x, \phi_x)_N = 0. \tag{4.114}$$

It is worth noting that the second discrete inner product in (4.114) is exact. The original integral is the product of two polynomials of degree $N-1$, so the product is a polynomial of degree $2N-2$. The Gauss Lobatto quadrature is exact for polynomials of degree $2N-1$ or less, so there is no error going from the continuous to the discrete. The first inner product is not exact, since the integrand is a polynomial of degree $2N$. It does have a spectrally small quadrature error associated with it, however.

Let us examine the two discrete inner products in (4.114) separately. When we replace the polynomials by their Lagrange representations, the first discrete inner product becomes

$$(\Phi_t, \phi)_N = \sum_{j=0}^{N} w_j \left(\sum_{n=0}^{N} \dot{\Phi}_n \ell_n(x_j) \sum_{m=0}^{N} \phi_m \ell_m(x_j) \right). \tag{4.115}$$

Since $\ell_i(x_j) = \delta_{i,j}$, the sums reduce all the way to

$$(\Phi_t, \phi)_N = \sum_{j=0}^{N} \dot{\Phi}_j \phi_j w_j. \tag{4.116}$$

The second quadrature expands partially to

$$(\Phi_x, \phi_x)_N = \sum_{j=0}^{N} w_j \Phi'_j \left(\sum_{m=0}^{N} \phi_m \ell'_m(x_j) \right) = \sum_{m=0}^{N} \phi_m \left(\sum_{j=0}^{N} w_j \Phi'_j \ell'_m(x_j) \right). \tag{4.117}$$

If we rename the indices $m \leftarrow j$ and $j \leftarrow k$, add the time derivative term, and rearrange, (4.114) becomes

$$\sum_{j=0}^{N} \left[\dot{\Phi}_j w_j + \sum_{k=0}^{N} w_k \Phi'_k \ell'_j(x_k) \right] \phi_j = 0. \tag{4.118}$$

Since the ϕ_j are linearly independent, the coefficient of each ϕ_j must be zero, so

$$\dot{\Phi}_j w_j + \sum_{k=0}^{N} w_k \Phi'_k \ell'_j(x_k) = 0, \quad j = 1, 2, \ldots, N-1. \tag{4.119}$$

Notice that the end points, $j = 0$ and $j = N$ are not included, since $\phi_0 = \phi_N = 0$ to satisfy the boundary conditions. We specify the unknowns at those points by the boundary conditions.

We complete the approximation (4.119) by expanding the derivative Φ'. The derivative of the interpolant is

$$\Phi'_k = \sum_{n=0}^{N} \Phi_n \ell'_n(x_k), \tag{4.120}$$

so we can write the sum in (4.119) as

$$\sum_{k=0}^{N} w_k \Phi'_k \ell'_j(x_k) = \sum_{k=0}^{N} w_k \sum_{n=0}^{N} \Phi_n \ell'_n(x_k) \ell'_j(x_k) = \sum_{n=0}^{N} \Phi_n \left(\sum_{k=0}^{N} w_k \ell'_n(x_k) \ell'_j(x_k) \right). \tag{4.121}$$

But $\ell'_n(x_k) = D_{kn}$ is the polynomial derivative matrix, so the system of ODEs for the solution unknowns is

$$\begin{aligned} \dot{\Phi}_j + \sum_{n=0}^{N} \hat{G}_{jn} \Phi_n = 0, \quad & j = 1, 2, \ldots, N-1, \\ \Phi_0 = \Phi_N = 0, & \end{aligned} \tag{4.122}$$

Algorithm 57: *CGDerivativeMatrix:* Matrix for Legendre Galerkin Approximation

Procedure CGDerivativeMatrix
Input: N
Uses Algorithms:
Algorithm 25 (LegendreGaussLobattoNodesAndWeights)
Algorithm 37 (PolynomialDerivativeMatrix)
$\left\{\{x_j\}_{j=0}^N, \{w_j\}_{j=0}^N\right\} \leftarrow LegendreGaussLobattoNodesAndWeights(N)$
$\{D_{i,j}\}_{i,j=0}^N \leftarrow PolynomialDerivativeMatrix(N, \{x_j\}_{j=0}^N)$
for $j = 0$ **to** N **do**
  for $n = 0$ **to** N **do**
    $s \leftarrow 0$
    for $k = 0$ **to** N **do**
      $s \leftarrow s + D_{k,n} * D_{l,j} * w_k$
    end
    $G_{j,n} \leftarrow s$
  end
end
return $\{G_{i,j}\}_{i,j=0}^N$
End Procedure CGDerivativeMatrix

if we define the matrix $\hat{G}$ by

$$\hat{G}_{jn} = \frac{1}{w_j} G_{jn} = \frac{1}{w_j} \sum_{k=0}^{N} D_{kn} D_{kj} w_k. \tag{4.123}$$

In matrix form, we can represent $\hat{G}$ by $\hat{G} = W^{-1} D^T W D$, where W is the diagonal matrix of the Gauss-Lobatto quadrature weights. In finite element parlance, the matrix $D^T W D$ is the *stiffness matrix* and the matrix W is the *mass matrix*. The matrix $\hat{G}$, therefore, is the product of the inverse of the mass matrix times the stiffness matrix. One advantage of the discrete orthogonality of the basis functions, we see, is that the mass matrix is diagonal, and hence trivial to invert. We present a procedure to compute the matrix G generated in Algorithm 57 (CGDerivativeMatrix).

4.6.1 How to Implement the Method

In practical terms, the nodal Galerkin approximation (4.122) looks the same as the collocation approximation (4.81), differing only by the matrix D being replaced by the matrix G. For this reason, we can implement it with Algorithm 51 (LegendreCollocation) simply by replacing the calls to *LegendreGaussLobattoNodesAndWeights* and *mthOrderPolynomialDerivativeMatrix* with the call to *CGDerivativeMatrix* in Algorithm 57. In fact, the resemblance of the two methods goes beyond form. It can be shown that the elements of the matrix $\hat{G}$ for $1 \leq i, j \leq N-1$ are identical to

those elements of the Legendre collocation method, making the two approximations for the constant coefficient Dirichlet problems identical [7].

One advantage of the Galerkin formulation, however, comes with the ease with which we can impose other boundary conditions, such as Neumann conditions. To specify a Neumann boundary condition, say $\varphi_x(1) = \sigma$, we start with the weak form of the equation after integration by parts, but before setting the boundary conditions

$$(\varphi_t, \phi) - \varphi_x \phi|_{-1}^{1} + (\varphi_x, \phi_x) = 0.$$

We replace the solution derivative by σ and then proceed with the approximation to get

$$(\Phi_t, \phi)_N - \sigma\phi(1) + (\Phi_x, \phi_x)_N = 0. \tag{4.124}$$

When we follow the same procedure that leads to (4.122), we get the system of ordinary differential equations to integrate in time

$$\begin{aligned} \dot{\Phi}_j + \sum_{n=0}^{N} \hat{G}_{jn}\Phi_n = 0, \quad j = 1, 2, \ldots, N-1, \\ \dot{\Phi}_N + \sum_{n=0}^{N} \hat{G}_{Nn}\Phi_n - \frac{\sigma}{w_N} = 0, \\ \Phi_0 = 0. \end{aligned} \tag{4.125}$$

4.6.2 Benchmark Solution

To examine the performance of the nodal continuous Galerkin approximation, we revisit the example of Sect. 4.1.1, where we solved the heat equation with Chebyshev and Legendre collocation for the exact solution

$$u(x,t) = \sin[\pi\,(x+1)]\, e^{-k^2\pi^2 t}. \tag{4.126}$$

Figure 4.12 displays the convergence of the error for the nodal Galerkin method along with the Chebyshev and Legendre collocation errors shown previously in Fig. 4.8. For this problem, we see that there is no difference in the maximum error of the two Legendre approximations, since for this problem the two approximations are identical.

4.7 The Nodal Discontinuous Galerkin Method

The last spectral approximation that we develop uses the *Discontinuous Galerkin* formulation, which is particularly well-suited to solve wave propagation problems.

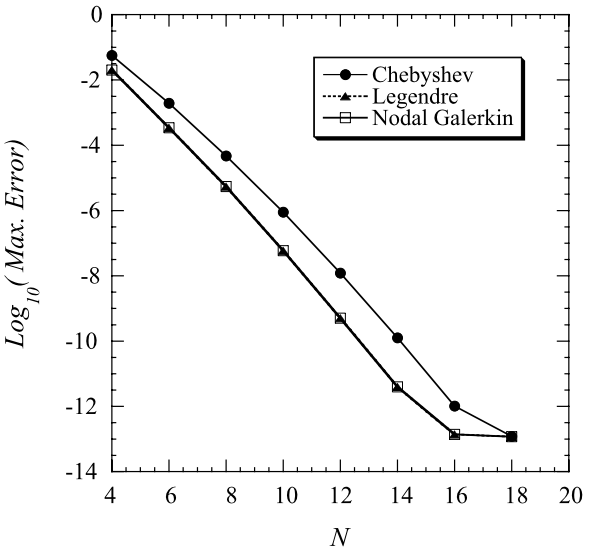

Fig. 4.12 Convergence of the maximum errors for Chebyshev and Legendre collocation approximations to the diffusion equation, (4.77), compared with the errors for the Nodal Galerkin approximation

We will use the method later in Chaps. 5, 7 and 8 to approximate the solutions of systems of conservation laws, which include systems that describe gas dynamics, ocean waves, and electrodynamics.

To motivate the formulation, we approximate the first order initial boundary value problem for the advection equation (4.83) again. Since we are developing a Galerkin approximation, we rewrite the PDE in weak form,

$$(\varphi_t, \phi) + (\varphi_x, \phi) = 0 \tag{4.127}$$

for any function ϕ. Unlike the continuous Galerkin approximation of the previous section, we will not require ϕ to satisfy the boundary condition on the left. This is what distinguishes the discontinuous from the continuous formulation.

Since we are developing a nodal method, we approximate the solution by a polynomial of degree N and represent the polynomial in the nodal, Lagrange form

$$\varphi(x,t) \approx \Phi(x,t) = \sum_{j=0}^{N} \Phi_j(t)\ell_j(x). \tag{4.128}$$

We defer the question of what points to use as the nodes until we have derived the final weak form of the equation that the approximation will satisfy.

We find the $N+1$ equations that we need to determine the Φ_j values by the usual Galerkin conditions. We require that Φ satisfy the weak form (4.127) for any ϕ that is a polynomial of degree N

$$(\Phi_t, \phi) + (\Phi_x, \phi) = 0. \tag{4.129}$$

To get the actual set of $N+1$ equations, we write the polynomial ϕ in nodal Lagrange form

$$\phi = \sum_{j=0}^{N} \phi_j \ell_j(x), \tag{4.130}$$

and understand that when we say "for any", we mean that the nodal values ϕ_j are arbitrary and linearly independent. When we substitute (4.130) for ϕ in (4.129) and rearrange, we see that Φ satisfies

$$\sum_{j=0}^{N} \left\{ \left(\Phi_t, \ell_j\right) + \left(\Phi_x, \ell_j\right) \right\} \phi_j = 0.$$

We get our $N+1$ equations because the ϕ_j are independent, for each factor must vanish individually leaving us with

$$\left(\Phi_t, \ell_j\right) + \left(\Phi_x, \ell_j\right) = 0, \quad j = 0, 1, \ldots, N. \tag{4.131}$$

We still have to enforce the boundary condition. To do that, we first separate the boundary contributions from the interior contributions by integrating the inner product that contains the spatial derivatives by parts

$$\left(\Phi_t, \ell_j\right) + \Phi \ell_j \big|_{-1}^{1} - \left(\Phi, \ell_j'\right) = 0. \tag{4.132}$$

Unlike when we use the continuous Galerkin method, the boundary terms do not vanish because $\ell_j(\pm 1)$ does not necessarily vanish. Nor is the approximation, Φ, required to satisfy the boundary conditions exactly.

We enforce the boundary condition for the discontinuous Galerkin approximation of the advection equation weakly. We replace the boundary term in (4.132) at the left boundary by the boundary condition

$$\left(\Phi_t, \ell_j\right) + \left\{ \Phi(1,t)\ell_j(1) - g(t)\ell_j(-1) \right\} - \left(\Phi, \ell_j'\right) = 0. \tag{4.133}$$

Since there is no boundary condition on the right, we do nothing there.

Finally, we find the equations that the Φ_j satisfy. We insert the representation (4.128) into the weak form (4.133) to get

$$\sum_{n=0}^{N} \dot{\Phi}_n \left(\ell_n, \ell_j\right) + \left\{ \Phi(1,t)\ell_j(1) - g(t)\ell_j(-1) \right\} - \sum_{n=0}^{N} \Phi_n \left(\ell_n, \ell_j'\right) = 0. \tag{4.134}$$

Rather than evaluate the integral inner products analytically, we approximate them by quadrature,

$$\sum_{n=0}^{N} \dot{\Phi}_n \left(\ell_n, \ell_j\right)_N + \left\{ \Phi(1,t)\ell_j(1) - g(t)\ell_j(-1) \right\} - \sum_{n=0}^{N} \Phi_n \left(\ell_n, \ell_j'\right)_N = 0. \tag{4.135}$$

It is at this point that we make the decision on what quadrature to use and where to place the nodes. The obvious choices for the quadrature are to use either the Legendre Gauss or the Legendre Gauss-Lobatto approximations. The Gauss quadrature would ensure that the discrete inner product, $(\ell_n, \ell_j)_N$ is exact, since the Lagrange interpolating polynomials are of degree N. If we use the Gauss-Lobatto quadrature, a quadrature error is introduced. Both types of quadratures have been used in practice.

Each quadrature has its advantages and disadvantages. The advantage of the Lobatto points is that the boundary term $\Phi(1,t) = \Phi_N(t)$. If we use the Gauss points, then we must evaluate the interpolant, (4.128), to get the boundary value. On the other hand, the Gauss points give us $(\ell_n, \ell_j)_N = (\ell_n, \ell_j)$ and

$$\left(\ell_n, \ell_j\right)_N = \sum_{k=0}^{N} \ell_n(x_k)\, \ell_j(x_k)\, w_k = w_j \delta_{n,j}. \tag{4.136}$$

Notice that the situation differs from that of the continuous Galerkin approximation of the previous section. We used the Gauss-Lobatto points there because the approximate solution had to satisfy the boundary conditions. Since the boundary conditions do not have to be satisfied (exactly) with the discontinuous Galerkin approximation, we have the opportunity to use the higher precision Gauss quadrature.

We will choose the Legendre Gauss points as the nodes, which are interior to the interval. With that choice made, we can derive the final approximation. We use (4.136) to simplify the time derivative term, while

$$\left(\ell_n, \ell'_j\right)_N = \sum_{k=0}^{N} \ell_n(x_k)\, \ell'_j(x_k)\, w_k = \ell'_j(x_n)\, w_n. \tag{4.137}$$

The simplifications give us the equations satisfied by the nodal values of the approximate solution,

$$\dot{\Phi}_j = -\left\{ \Phi(1,t) \frac{\ell_j(1)}{w_j} - g(t) \frac{\ell_j(-1)}{w_j} + \sum_{n=0}^{N} \hat{D}_{jn} \Phi_n \right\}, \quad j = 0, 1, \ldots, N, \tag{4.138}$$

where

$$\hat{D}_{jn} = -\frac{D_{nj} w_n}{w_j}, \tag{4.139}$$

and $D_{nj} = \ell'_j(x_n)$ is the transpose of the standard derivative matrix, computed with Algorithm 37 (PolynomialDerivativeMatrix). We find the boundary value of the solution that we need in (4.138) from the interpolant

$$\Phi(1,t) = \sum_{j=0}^{N} \Phi_j \ell_j(1). \tag{4.140}$$

We can compute boundary value simply by a dot product if we compute the coefficients $\ell_j(1)$ once and store them.

It is possible to integrate (4.133) by parts a second time. Then the equation looks more like a penalty method

$$\left(\Phi_t, \ell_j\right) + \left[\Phi(-1,t) - g(t)\right]\ell_j(-1) + \left(\Phi', \ell_j\right) = 0, \tag{4.141}$$

where a correction based on the difference between the value at the left boundary and the boundary condition is added to the equation to weakly impose the boundary condition. For linear problems like this and on the square, there is no difference. For nonlinear equations and equations on curvilinear domains, the quadrature errors differ between the two, making the approximations slightly different. We will only develop algorithms for which the equation is integrated by parts once.

4.7.1 How to Implement the Method

We are now ready to construct algorithms to compute the discontinuous Galerkin first derivative approximation. Since it is most efficient to pre-compute the arrays $\ell_j(\pm 1)$ and $\hat{D}_{jn}$ and store them, we wrap the discontinuous Galerkin derivative operator into a class, Algorithm 58 (NodalDiscontinuousGalerkin), that groups both procedures and data. We use the constructor to compute and store these quantities as member data. The derivative computation, *ComputeDGDerivative*, evaluates the quantities in the braces in (4.138).

Algorithm 59 (NodalDiscontinuousGalerkin:Construct) implements a constructor for the discontinuous Galerkin approximation and is built from algorithms that we have already developed. It first computes the Legendre Gauss nodes the quadrature weights via Algorithm 23 (LegendreGaussNodesAndWeights). The barycentric weights, which we use to compute the Lagrange interpolating polynomials (Sect. 3.4), are evaluated next using Algorithm 30 (BarycentricWeights). The two arrays of Lagrange interpolating polynomials that are evaluated at the endpoints are then computed with Algorithm 34 (LagrangeInterpolatingPolynomials). Finally, the

Algorithm 58: *NodalDiscontinuousGalerkin:* A Discontinuous Galerkin Class Definition

Class NodalDiscontinuousGalerkin
Data:
$N, \left\{\hat{D}_{i,j}\right\}_{i,j=0}^{N}, \left\{\ell_j(-1)\right\}_{j=0}^{N}, \left\{\ell_j(1)\right\}_{j=0}^{N}, \left\{w_j\right\}_{j=0}^{N}$
$\left\{\Phi_j\right\}_{j=0}^{N}$; `// Solution array`
Procedures:
Construct(N); `// Algorithm 59`
DGDerivative$(\Phi^L, \Phi^R, \{\Phi_j\}_{j=0}^{N})$; `// Algorithm 60`
DGTimeDerivative(t); `// Algorithm 61`
End Class NodalDiscontinuousGalerkin

Algorithm 59: *NodalDiscontinuousGalerkin:Construct:* Constructor for the Discontinuous Galerkin Class

Procedure Construct
Input: N
Uses Algorithms:
Algorithm 23 (LegendreGaussNodesAndWeights)
Algorithm 37 (PolynomialDerivativeMatrix)
Algorithm 34 (LagrangeInterpolatingPolynomials)
Algorithm 30 (BarycentricWeights)

$this.N \leftarrow N$
$\left\{\left\{x_j\right\}_{j=0}^{N}, this.\left\{w_j\right\}_{j=0}^{N}\right\} \leftarrow LegendreGaussNodesAndWeights(N)$
$\left\{w_j^B\right\}_{j=0}^{N} \leftarrow BarycentricWeights(\left\{x_j\right\}_{j=0}^{N})$
$this.\left\{\ell_j(-1)\right\}_{j=0}^{N} \leftarrow LagrangeInterpolatingPolynomials\left(-1.0, \left\{x_j\right\}_{j=0}^{N}, \left\{w_j^B\right\}_{j=0}^{N}\right)$
$this.\left\{\ell_j(1)\right\}_{j=0}^{N} \leftarrow LagrangeInterpolatingPolynomials\left(1.0, \left\{x_j\right\}_{j=0}^{N}, \left\{w_j^B\right\}_{j=0}^{N}\right)$
$\left\{D_{ij}\right\}_{i,j=0}^{N} \leftarrow PolynomialDerivativeMatrix(\left\{x_j\right\}_{j=0}^{N})$
for $j = 0$ **to** N **do**
  for $i = 0$ **to** N **do**
    $this.\hat{D}_{i,j} \leftarrow -D_{j,i} * this.w_j / this.w_i$
  end
end
End Procedure Construct

Algorithm 60: *NodalDiscontinuousGalerkin:DGDerivative:* First Spatial Derivative via the Galerkin Approximation

Procedure ComputeDGDerivative
Input: $\Phi^L, \Phi^R, \left\{\Phi_j\right\}_{j=0}^{N}$
Uses Algorithms:
Algorithm 19 (MxVDerivative)

$\left\{\Phi'_j\right\}_{j=0}^{N} \leftarrow MxVDerivative(\{this.\hat{D}_{ij}\}_{i,j=0}^{N}, \{\Phi_j\}_{j=0}^{N})$
for $j = 0$ **to** N **do**
  $\Phi'_j \leftarrow \Phi'_j + (\Phi^R * this.\ell_j(1) - \Phi^L * this.\ell_j(-1))/this.w_j$
end
return $\left\{\Phi'_j\right\}_{j=0}^{N}$
End Procedure ComputeDGDerivative

procedure computes the derivative matrix $\hat{D}$, defined by (4.139), from the standard polynomial derivative matrix, D.

We show how to compute the approximation of the first derivative in Algorithm 60 (NodalDiscontinuousGalerkin:DGDerivative). It implements the quantity in the braces in (4.138). To be more general, the procedure takes the interior state as input, represented by the array of Φ_j's, and the left and right boundary states to compute the approximation of the derivative. In this way, we can accommodate

Algorithm 61: *NodalDiscontinuousGalerkin:DGTimeDerivative:* Time Derivative via the Discontinuous Galerkin Approximation

Procedure DGTimeDerivative
Input: t
Uses Algorithms:
Algorithm 60 (NodalDiscontinuousGalerkin:DGDerivative)
if $c > 0$ **then**
  $\Phi^L \leftarrow g(t)$
  $\Phi^R \leftarrow InterpolateToBoundary(\{this.\Phi_j\}_{j=0}^N, \{this.\ell_j(1)\}_{j=0}^N)$
else
  $\Phi^R \leftarrow g(t)$
  $\Phi^L \leftarrow InterpolateToBoundary(\{this.\Phi_j\}_{j=0}^N, \{this.\ell_j(-1)\}_{j=0}^N)$
end
$\{\dot{\Phi}_j\}_{j=0}^N \leftarrow -c * this.DGDerivative(\Phi^L, \Phi^R, \{this.\Phi_j\}_{j=0}^N)$
return $\{\dot{\Phi}_j\}_{j=0}^N$
End Procedure DGTimeDerivative

Procedure InterpolateToBoundary
Input: $\{\Phi_j\}_{j=0}^N, \{\ell_j\}_{j=0}^N$
$interpolatedValue \leftarrow 0$
for $j = 0$ **to** N **do**
  $interpolatedValue \leftarrow interpolatedValue + \ell_j * \Phi_j$
end
return *interpolatedValue*
End Procedure InterpolateToBoundary

problems with positive or negative wavespeeds. We place the logic to decide which boundary needs to be specified in the time derivative procedure.

We implement the time derivative algorithm next. It uses the spatial derivative procedure to implement the right hand side of (4.138). To be more general, we will write the time derivative to solve the equation

$$\varphi_t + c\varphi_x = 0, \tag{4.142}$$

where the wave speed c might be positive or negative. If c is positive, the boundary condition is applied at the left, and the solution is interpolated to the right, as was the situation in (4.138). If c is negative, the boundary condition is applied to the right and the solution is interpolated to the left. Thus, to compute the time derivative, the procedure must test the wavespeed and send the appropriate values for Φ^L and Φ^R to the spatial derivative routine.

We implement the time derivative in Algorithm 61 (NodalDiscontinuousGalerkin:DGTimeDerivative). It first computes the boundary values of the solution according to the sign of the wavespeed. To compute the boundary condition, we assume the presence of a user supplied function $g(t)$. To compute the value of the solution at the other boundary, the procedure evaluates the interpolant using *InterpolateToBoundary*. *InterpolateToBoundary* is nothing but a dot product of the

Algorithm 62: *DGStepByRK3:* Low Storage Runge-Kutta Integration of a Nodal Discontinuous Galerkin Approximation

Procedure DGStepByRK3
Input: t_n, Δt
Input: dg ; `// Of type NodalDiscontinuousGalerkin`
for $m = 1$ **to** 3 **do**
  $t \leftarrow t_n + b_m \Delta t$
  $\left\{\dot{\Phi}_j\right\}_{j=0}^{N} \leftarrow dg.TimeDerivative(t)$
  for $j = 0$ **to** N **do**
    $G_j \leftarrow a_m G_j + \dot{\Phi}_j$
    $dg.\Phi_j \leftarrow dg.\Phi_j + g_m \Delta t G_j$
  end
end
return dg
End Procedure DGStepByRK3

vectors represented by the array of solution values and the array of Lagrange interpolating polynomials. For large N we could replace it by a BLAS xDot routine (Appendix C). Once the boundary values of the solution are determined, the time derivative procedure passes them and the solution array to *ComputeDGDerivative* to compute the space derivative. The final step multiplies the space derivative by the wavespeed to compute the time derivative.

With the spatial discretization and time derivatives in hand, we turn to the integration in time. Once again we use a third order Runge-Kutta to integrate in time. Algorithm 62 (DGStepByRK3) is a modification of the third order method of Algorithm 50 (CollocationStepByRK3). The two differ because in the discontinuous Galerkin approximation, the boundary conditions are applied weakly as part of the spatial derivative approximation. This difference makes the discontinuous Galerkin approximation simpler to implement than collocation methods for systems of equations. (We will cover systems of equations at length in Sect. 5.4.) We can construct a driver for the time integration by a straightforward modification of the collocation driver, Algorithm 51 (LegendreCollocation).

Again, we must look at the distribution and growth of the eigenvalues of the derivative matrix and see how it fits within the region of absolute stability of the time integration method. We show an example of the distribution of the eigenvalues in Fig. 4.13 for $N = 32$. We notice that the distribution differs from those of the Chebyshev and Legendre collocation matrices (Fig. 4.9). The maximum eigenvalues occur near the real axis, which indicates very strong dissipation in those modes. A plot of the magnitude of the maximum eigenvalue as a function of N, Fig. 4.14, also shows a difference from what we have seen before. For $N \leq 32$, the growth of the maximum eigenvalue is linear, and only becomes quadratic for larger values of N. Since the imaginary part of the largest eigenvalue is small, we can use the known real limit of the region of absolute stability (-2.51 on Fig. 4.1 for the third order Runge-Kutta method) to determine the maximum time step. From a curve fit for the growth of the maximum eigenvalue and the stability limit for the Runge-Kutta, we

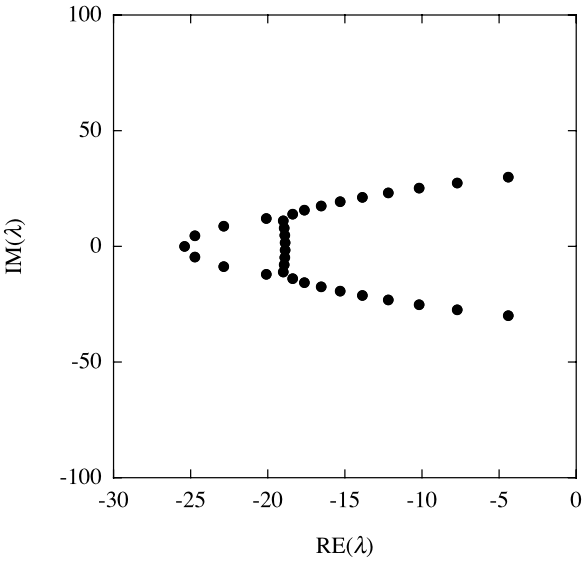

Fig. 4.13 Eigenvalues of the nodal discontinuous Galerkin first derivative for $N = 32$

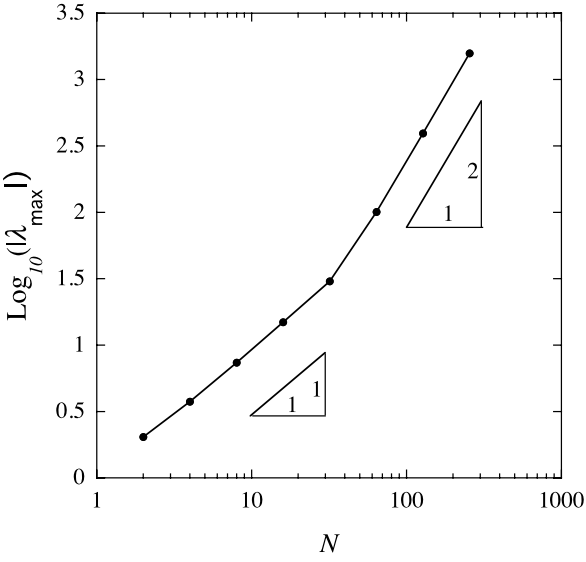

Fig. 4.14 Growth of the magnitude of the largest eigenvalue for the nodal discontinuous Galerkin first derivative

can get an approximate upper limit on the time step of

$$\Delta t \leq \begin{cases} \frac{2.5}{N}, & N < 32, \\ \frac{38 \times 2.51}{N^2}, & N \geq 32. \end{cases} \tag{4.143}$$

4.7.2 Benchmark Solution

As a benchmark, we solve the problem (4.83) with the initial condition $\varphi_0(x) = e^{-\ln(2)(x+1)^2/\sigma^2}$, with $\sigma = 0.2$. We show the computed and exact solutions at three times in Fig. 4.15. The computed solution there is for $N = 36$, and we chose $\Delta t = 1.5 \times 10^{-4}$ to ensure the temporal errors were small relative to the spatial errors. The wave propagates with little loss of amplitude or spreading. It moves with the correct speed so there is no noticeable shift between the exact and computed solutions.

It is natural to compare the performance of the discontinuous Galerkin approximation with the polynomial collocation methods that we developed in Sect. 4.4.4. We show one such comparison in Table 4.2, which lists the logarithms of the maximum errors for the discontinuous Galerkin approximation along with the errors of the Chebyshev and Legendre collocation approximations. What we see is that

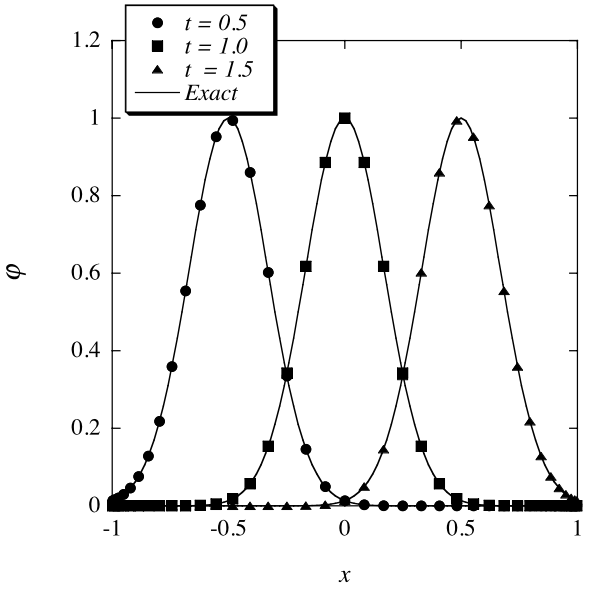

Fig. 4.15 Computed and exact solutions at three times for the nodal discontinuous Galerkin approximation of the advection equation

Table 4.2 Logarithm of maximum errors for three methods

N	Chebyshev	Legendre	Discontinuous Galerkin
16	−2.0	−1.8	−2.0
20	−2.7	−2.4	−3.0
24	−3.8	−3.4	−4.0
28	−4.8	−4.3	−5.2
32	−6.1	−5.4	−6.6
36	−7.3	−6.7	−7.9
40	−8.7	−8.0	−9.0

for the advection problem, the Chebyshev collocation method bests the Legendre collocation, but that the discontinuous Galerkin approximation has the best error overall.

4.8 Summary and Some Broad Generalizations

Now that we have seen six examples of spectral approximation methods, let's get some perspective that will guide our choice of methods throughout the remainder of this book. The fundamental characteristic across all spectral methods is that they use orthogonal polynomial approximations with which to approximate the solution of a PDE. The differences between them come from the flexibility that we have to decide how to determine the degrees of freedom, written either as the polynomial coefficients or the values at a set of nodes. Flexibility allows us to tailor the approximation to our needs, but the availability of too many choices can also be a detriment. We would like some guiding principals to help us make a choice among all the possibilities. We get those guiding principals from an understanding of the tradeoffs that we make in each of the approximations. Whether those tradeoffs are worth making is a matter of one's goals, and is subject to opinion, which we freely profess in this section.

The first choice we have to make is what basis functions to use. For periodic problems, the Fourier basis is the clear choice. For problems in bounded domains, we will choose between the Legendre and Chebyshev polynomials. How we choose between the two depends more on practical matters than on which one has better approximation properties. Each will give exponentially accurate solutions for smooth enough solutions. The family of orthogonal polynomials is much larger than three, however, and for some special circumstances, e.g. on infinite intervals, there are alternatives. For most practical applications one of Fourier, Legendre or Chebyshev will be sufficient. For other choices, see [4].

The next choice is how to determine the degrees of freedom. In this chapter we concentrated on two commonly used methods: Collocation and Galerkin. We derive collocation methods from a strong form of the equations. We start Galerkin approximations from an integral, weak form of the equations.

Collocation methods, like finite difference methods, are derived from a strong form of the PDE and share many of the same advantages and disadvantages. Foremost is that they are easy to derive and to implement for a wide class of problems—constant coefficient, variable coefficient and nonlinear. Since collocation methods require the solution to satisfy the PDE at a set of grid points, they are naturally nodal approximations and will have aliasing errors even for constant coefficient problems. A main tradeoff is that there is little formal mathematical guidance on how to derive a stable approximation or how to implement boundary conditions. The latter makes it difficult to extend collocation methods to complex geometries or to systems of equations. Like finite difference methods, collocation methods are most easily applied to geometries that we can map onto a simple square or cube. Within those

constraints, however, collocation methods will give spectrally accurate approximations. For this reason, we will develop implementations of collocation algorithms for square and mapped geometries in Chaps. 5 and 7. For a collocation method, we choose between Legendre and Chebyshev approximations mostly on how many modes we need, since we can use fast transforms to compute the derivatives with Chebyshev methods.

Galerkin spectral methods, like finite element methods, are derived from a weak form of the equations. They are less easily derived than collocation methods, but the formulation naturally leads to stable approximations and gives guidance on how to implement boundary conditions. Galerkin methods can be either nodal or modal. Modal approximations can be significantly more accurate than nodal approximations, depending on the problem. They are much harder to derive and more complex to implement, however, particularly for variable coefficient, nonlinear, or multidimensional problems. Exceptions include special nonlinearities like we saw for the Burgers equation, or special geometries like cylindrical coordinates. However, within those constraints, the modal Galerkin method is the method of choice if ultimate accuracy is required. We will not develop modal Galerkin methods further in this book. For further study, see [21].

Nodal spectral Galerkin methods are intermediate between collocation and Galerkin methods. They start from a weak form of the equations, but replace hard to evaluate integrals by quadrature. Because they start from a weak form of the equations, Legendre approximations are almost always used today to avoid having to deal with a variable weight function in the inner products. The Galerkin approach gives guidance on how to implement boundary conditions. We have seen, however, that the nodal approximations are just as easy to implement as collocation methods. In fact, we will see in the following chapters that we can often convert algorithms from collocation approximations to nodal Galerkin approximations with one line changes. For some equations, particularly for systems of conservation laws, they are much easier to implement than collocation, especially in multiple space dimensions. Like finite element methods, nodal Galerkin formulations are easily extendable to solve problems in complex geometries, which we will do in Chap. 8. Whereas modal Galerkin methods are the methods of choice when ultimate accuracy is required, we choose nodal Galerkin methods to solve problems in complex geometries.

Exercises

4.1 Rewrite Algorithm 41 (FourierCollocationTimeDerivative) to use the FFT rather than matrix-vector multiplication to compute the spatial derivatives.

4.2 Show how to use quadrature to compute the initial Fourier coefficients to start the integration of a Fourier Galerkin method.

4.3 Use padding and the FFT to create a faster implementation of Algorithm 46 (EvaluateFourierGalerkinSolution).

4.4 Implement Algorithms 44–47 and compute the solution for the initial condition (4.26) when $\nu = 0$. Show analytically that the energy, $\|u\|_{L^2}$ should remain constant, and verify it numerically when Δt is small enough. (To guarantee that the time differencing scheme does not decrease the energy, we should use a time integration method like the trapezoidal rule.)

4.5 Verify numerically on several test problems that the discrete norm (4.29) is spectrally accurate for periodic functions.

4.6 Show algebraically that for the linear, constant coefficient advection-diffusion equation the Fourier collocation and Fourier Galerkin methods are equivalent.

4.7 Use (4.37) to show that

$$(\Phi_t, V) + (\Phi_x, V) = -(\nu \Phi_x, V_x)$$

for any

$$V = \sum_{k=-N/2}^{N/2} b_k e^{ikx}.$$

Use this result to show that the Fourier Galerkin approximation is *stable*, that is,

$$\|\Phi\| \leq \|\Phi_0\|$$

if integrated exactly in time.

4.8 Apply the Fourier Galerkin method to the initial condition

$$S(x) = \begin{cases} 1, & 0 \leq x \leq \pi/2, \\ 0, & \pi/2 < x < \pi \end{cases} \tag{4.144}$$

and explain the behavior of the numerical solution.

4.9 Verify the conservation result (4.67). Hint: $\int_0^{2\pi} U dx = (U, 1)$.

4.10 The debate over the importance of aliasing in a spectral calculation goes back to the early days in the development of spectral methods. In this problem you will solve the KdV equation

$$u_t + 2\pi \frac{\partial}{\partial x}\left(u + \frac{1}{2}u^2\right) + \frac{1}{2}\lambda_D^2 (2\pi)^3 \frac{\partial^3 u}{\partial x^3} = 0. \tag{4.145}$$

This equation has the exact (traveling wave) solution

$$u(x,t) = u_0 + \Delta u \,\mathrm{sech}^2\left[\frac{1}{2\pi\lambda_D}\sqrt{\frac{\Delta u}{6}}(x - ct - 4\pi)\right],$$
$$c = 2\pi\left(1 + u_0 + \frac{1}{3}\Delta u\right), \tag{4.146}$$
$$u_0 = -2\lambda_D\sqrt{6\Delta u}\tanh\left(\frac{1}{\lambda_D}\sqrt{\frac{\Delta u}{24}}\right)$$

on $(-\infty, \infty)$. The parameters are $\lambda_D = 0.01$ and $\Delta u = 0.2$.

Implement the Fourier-Galerkin method to integrate (4.145) to time $t = 1$ by modifying Algorithms 44–47 to include the nonlinear term and the third order derivative term. Although the initial solution is defined on the infinite interval, it is nonzero to within rounding errors only in a small region, making the periodic extension sufficiently smooth for all intents and purposes. Use the fast convolution transform, Algorithm 49, to evaluate the convolution sum. Do the computation with ($M = N$) and without ($M \geq 3N/2$) aliasing. Examine the spectra ($|\hat{U}_k|$ as a function of k) of the two solutions and comment on the differences between them. Do you see spectral accuracy as the order, N, increases?

4.11 Verify the discretization, (4.125).

Chapter 5
Spectral Approximation on the Square

It is simplest, though not always of much practical interest, to describe spectral methods on the square domain $(x, y) \in [-1, 1] \times [-1, 1]$. Once the ideas are understood for the simplest of geometries, they can be extended to solve PDEs on more complex geometries by using mappings (Chap. 7), multidomain methods (Chap. 8), or both. We will illustrate the development of spectral approximations for three canonical problems in mathematical physics: The solution of steady potentials, transport with and without diffusion, and wave propagation. These physical processes are modeled by the Poisson equation on the square with Dirichlet boundary conditions, the advection-diffusion equation, the scalar advection equation, and systems of conservation laws.

5.1 Approximation of Functions in Multiple Space Dimensions

In multiple space dimensions, spectral methods use expansion functions that are tensor products of the one dimensional functions that we used in preceding chapters. Spectral methods have the same representations used to derive separation of variables solutions of PDEs.

In two space dimensions, for example, the Fourier truncation approximation is

$$P_{NM} f = \sum_{n=-N/2}^{N/2} \sum_{m=-M/2}^{M/2} \hat{f}_{nm} e^{-inx} e^{-imy}. \tag{5.1}$$

We find the Fourier coefficients with the inner product

$$(u, v) = \int_0^{2\pi} \int_0^{2\pi} u(x, y)\, v^*(x, y)\, dxdy. \tag{5.2}$$

That is,

$$\hat{f}_{nm} = \frac{1}{(2\pi)^2} \left(f, e^{i(nx+my)} \right). \tag{5.3}$$

We define the Fourier interpolant in two space dimensions similarly,

$$I_{NM} f = \sum_{n=-N/2}^{N/2} \sum_{m=M/2}^{M/2} \frac{\tilde{f}_{nm}}{\bar{c}_n \bar{c}_m} e^{-inx} e^{-imy} = \sum_{j=0}^{N-1} \sum_{k=0}^{M-1} f_{j,k} h_j(x)\, h_k(y). \tag{5.4}$$

D.A. Kopriva, *Implementing Spectral Methods for Partial Differential Equations,* Scientific Computation,

As in one space dimension, we compute the discrete coefficients from the two dimensional discrete inner product, which is now

$$(u, v)_{NM} = \frac{(2\pi)^2}{NM} \sum_{j=0}^{N-1} \sum_{k=0}^{M-1} u\left(x_j, y_k\right) v^*\left(x_j, y_k\right), \tag{5.5}$$

so that

$$\tilde{f}_{nm} = \frac{1}{NM} \sum_{j=0}^{N-1} \sum_{k=0}^{M-1} f\left(x_j, y_k\right) e^{-inx_j} e^{-imy_k}. \tag{5.6}$$

The tensor product form is convenient for computation, since we can evaluate the double sum by a series of sums along each direction separately. If we define an intermediate array

$$\bar{f}_n\left(y_k\right) = \frac{1}{N} \sum_{j=0}^{N-1} f\left(x_j, y_k\right) e^{-inx_j}, \quad n = -N/2, \ldots, N/2-1;\ k = 0, \ldots, N-1, \tag{5.7}$$

then

$$\begin{aligned} \tilde{f}_{nm} &= \frac{1}{M} \sum_{k=0}^{M-1} \bar{f}_n\left(y_k\right) e^{-imy_k}, \\ & n = -N/2, \ldots, N/2-1;\ m = -M/2, \ldots, M/2-1. \end{aligned} \tag{5.8}$$

Likewise, the polynomial truncation approximation is

$$P_{NM} f(x) = \sum_{n=0}^{N} \sum_{m=0}^{M} \hat{f}_{nm} \phi_n(x) \phi_m(y), \tag{5.9}$$

where we compute the coefficients using the two dimensional weighted inner product

$$\hat{f}_{nm} = \frac{(f, \phi_n \phi_m)_w}{\|\phi_n \phi_m\|_w^2} = \frac{\int\int f(x, y) \phi_n(x) \phi_m(y)\, w(x) w(y) dx dy}{\int\int \phi_n^2(x) \phi_m^2(y)\, w(x) w(y) dx dy}. \tag{5.10}$$

We can choose to write the polynomial interpolant in two dimensions either in terms of the discrete coefficients or the equivalent Lagrange form

$$I_{NM} f(x) = \sum_{n=0}^{N} \sum_{m=0}^{M} \tilde{f}_{nm} \phi_n(x) \phi_m(y) = \sum_{j=0}^{N} \sum_{k=0}^{M} f_{j,k} \ell_j(x) \ell_k(y). \tag{5.11}$$

Like the Fourier coefficients, we compute the discrete polynomial coefficients from a sequence of one dimensional transforms. For example, if we compute the

sums in the x direction to get the intermediate values

$$\bar{f}_n(y_k) = \frac{1}{\|\phi_n\|_N^2} \sum_{j=0}^{N} f_{j,k}\phi_n(x_j)w_j, \quad n = 0, 1, \ldots, N;\ k = 0, 1, \ldots, M, \tag{5.12}$$

the two dimensional discrete coefficients are

$$\tilde{f}_{nm} = \frac{1}{\|\phi_m\|_M^2} \sum_{k=0}^{M} \bar{f}_n(y_k)\,\phi_m(y_k)\,w_k, \quad n = 0, 1, \ldots, N;\ m = 0, 1, \ldots, M. \tag{5.13}$$

The tensor product representation of the solution makes it easy to use different approximations in different coordinate directions. If the problem is non-periodic in the x direction and periodic in y, for example, we could write

$$P_{NM} f(x, y) = \sum_{j=0}^{N} \sum_{k=0}^{M} f_{j,k}\ell_j(x)\,h_k(y). \tag{5.14}$$

Tensor products also make mixed representations possible. For instance, the following polynomial is modal in the x direction and nodal in the y direction

$$P_{NM} f(x, y) = \sum_{k=-N/2}^{N/2} \sum_{j=0}^{M} \hat{f}_{k,j} e^{ikx} \ell_j(y), \tag{5.15}$$

where $\hat{f}_{k,j}$ is the kth Fourier coefficient at the point y_j.

In summary, the tensor product approximation of functions makes spectral methods efficient at high order because we can evaluate multidimensional approximations as sequences of one dimensional approximations. The ability to mix representations and basis functions makes spectral methods flexible.

5.2 Potential Problems on the Square

The first PDE that we will approximate on the square describes potential problems such as the steady state temperature distribution with a heat source. It is the Poisson equation with Dirichlet boundary conditions

$$\begin{cases} \nabla^2\varphi = \varphi_{xx} + \varphi_{yy} = s(x, y), \quad (x, y) \in (-1, 1) \times (-1, 1), \\ \varphi(x, -1) = 0, \quad -1 \le x \le 1, \\ \varphi(1, y) = 0, \quad -1 \le y \le 1, \\ \varphi(x, 1) = 0, \quad -1 \le x \le 1, \\ \varphi(-1, y) = 0, \quad -1 \le y \le 1. \end{cases} \tag{5.16}$$

Although we have specified that the potential, φ, vanish along the boundaries, we can specify any continuous potential distribution with only simple modifications to the approximations.

5.2.1 The Collocation Approximation

The simplest spectral approximation to derive for the Poisson equation is the collocation method. In two space dimensions, we lay a grid of points, (x_i, y_j), on the square and approximate the solution by a polynomial interpolant represented by the solution values, $\Phi_{i,j}$, at those points. Since the boundary conditions in (5.16) are not periodic in either direction, Legendre or Chebyshev polynomial approximations are appropriate. Since we want the solutions at the boundaries as well as in the interior, we choose the grid points to be the tensor product of the Gauss-Lobatto quadrature points (Sect. 1.11). For Chebyshev polynomial approximations, recall that these points are simply

$$\left(x_i, y_j\right) = \left(-\cos\frac{i\pi}{N}, -\cos\frac{j\pi}{N}\right), \quad i, j = 0, 1, \ldots, N. \tag{5.17}$$

For simplicity of exposition, we will assume that number of grid points is the same in each direction, but this is not necessary in practice. Note that we have reversed the order of the points so that the (x, y) values of the nodes increase as the indices i, j increase. To get a Legendre approximation, we would use Algorithm 25 (LegendreGaussLobattoNodesAndWeights) to compute the grid points.

To derive the collocation approximation, we approximate the potential $\varphi(x, y)$ and the forcing term $s(x, y)$ by polynomials Φ and S written in the second, i.e. Lagrange, form interpolant in (5.11),

$$\begin{aligned} \Phi(x, y) &= \sum_{i,j=0}^{N} \Phi_{i,j} \ell_i(x) \ell_j(y), \\ S(x, y) &= \sum_{i,j=0}^{N} s\left(x_i, y_j\right) \ell_i(x) \ell_j(y). \end{aligned} \tag{5.18}$$

To find the equations for the grid point values $\Phi_{i,j}$ we require that Φ satisfies the PDE at the interior points

$$\left(\frac{\partial^2 \Phi}{\partial x^2} + \frac{\partial^2 \Phi}{\partial x^2} - S\right)\Big|_{x_i, y_j} = 0, \quad i, j = 1, 2, \ldots, N-1. \tag{5.19}$$

The second derivative of the polynomial interpolant is

$$\frac{\partial^2 \Phi}{\partial x^2} = \frac{\partial^2}{\partial x^2} \sum_{k,l=0}^{N} \Phi_{k,l} \ell_k(x) \ell_l(y) = \sum_{k,l=0}^{N} \Phi_{k,l} \ell_k''(x) \ell_l(y). \tag{5.20}$$

By construction (Sect. 1.12), $\ell_l(y_j) = \delta_{j,l}$. Therefore, when we evaluate the second derivative at the grid points,

$$\left.\frac{\partial^2 \Phi}{\partial x^2}\right|_{i,j} = \sum_{k=0}^{N} \Phi_{k,j} \ell_k''(x_i) = \sum_{k=0}^{N} D_{i,k}^{(2),x} \Phi_{k,j}, \tag{5.21}$$

where $D_{ik}^{(2),x}$ is the second order spectral derivative matrix (Sect. 3.4), which we compute with Algorithm 38 (mthOrderPolynomialDerivativeMatrix). We derive a similar formula for the second derivative in the y direction. Finally, since $S(x_i, y_j) = s(x_i, y_j) = s_{i,j}$, the interior points satisfy the equations

$$\sum_{k=0}^{N} D_{ik}^{(2),x} \Phi_{k,j} + \sum_{k=0}^{N} D_{jk}^{(2),y} \Phi_{i,k} = s_{i,j}, \quad i, j = 1, 2, \ldots, N-1. \tag{5.22}$$

We will also express (5.22) in shorthand notation,

$$\nabla_N^2 \Phi_{ij} = s_{i,j}. \tag{5.23}$$

The left side of (5.23) is the *action of the discrete spectral Laplace operator.* The full equation (5.23) is the spectral collocation approximation of the Poisson equation.

The values of $\Phi_{i,j}$ along the boundaries are all that remain for us to specify. In the collocation method, as in a finite difference method, we set the approximate solution along the boundary to be its boundary value, i.e,

$$\Phi_{i,j} = 0, \quad \begin{cases} i = 0, N; \ j = 0, 1, \ldots, N, \\ j = 0, N; \ i = 0, 1, \ldots, N. \end{cases} \tag{5.24}$$

Equations (5.22) and (5.24) form a linear system of equations that we must solve for $\Phi_{i,j}$.

We can easily extend the collocation method to variable coefficient equations like

$$\nabla \cdot (\nu \nabla \varphi) = s. \tag{5.25}$$

For instance, suppose that the diffusivity depends on the potential so that $\nu = \nu(\varphi)$. We approximate the components of the flux, $\mathbf{f} = (f, g) = (\nu \varphi_x, \nu \varphi_y)$ also by polynomials of degree N. For instance

$$f\left(x_i, y_j\right) = \nu\left(\varphi\right) \varphi_x\big|_{x_i, y_j} \approx \nu\left(\Phi_{i,j}\right) \sum_{k=0}^{N} D_{ik}^{x} \Phi_{k,j} = \nu\left(\Phi_{i,j}\right) \Phi_x\left(x_i, y_j\right) = F_{i,j}. \tag{5.26}$$

We compute $D_{ik}^{x} = \ell_k'(x_i)$ using Algorithm 37 (PolynomialDerivativeMatrix). A similar formula holds for $g = \nu \varphi_y$. Then the interior approximation for the collo-

cation method is

$$
\begin{aligned}
F_{i,j} &= \nu\left(\Phi_{i,j}\right) \sum_{k=0}^{N} D^x_{ik} \Phi_{k,j}, \quad i,j = 0,1,2,\ldots,N, \\
G_{i,j} &= \nu\left(\Phi_{i,j}\right) \sum_{k=0}^{N} D^y_{jk} \Phi_{i,k}, \quad i,j = 0,1,2,\ldots,N, \\
\sum_{k}^{N} D^x_{ik} F_{k,j} &+ \sum_{k}^{N} D^y_{jk} G_{i,k} = s_{i,j}, \quad i,j = 1,2,\ldots,N-1
\end{aligned}
\tag{5.27}
$$

and we apply the Dirichlet boundary conditions as in (5.24).

We can also apply the collocation method to problems with Neumann boundary conditions. The formulation of (5.27) makes it easy to see how. Suppose we replace the boundary condition along $x = 1$ in (5.16) by the Neumann condition $\varphi_x(1, y) = b(y)$. Then to compute the flux, F, along the boundary, we simply replace the derivative there by the boundary condition. We compute the interior fluxes as we did for the Dirichlet problem. The system to be solved therefore becomes

$$
\begin{aligned}
F_{i,j} &= \nu\left(\Phi_{i,j}\right) \sum_{k=0}^{N} D^x_{ik} \Phi_{k,j}, \quad i = 0,1,\ldots,N-1;\ j = 0,1,\ldots,N, \\
F_{N,j} &= \nu\left(\Phi_{N,j}\right) b(y_j), \quad j = 1,\ldots,N-1, \\
G_{i,j} &= \nu\left(\Phi_{i,j}\right) \sum_{k=0}^{N} D^y_{jk} \Phi_{i,k}, \quad i,j = 0,1,2,\ldots,N, \\
\sum_{k=0}^{N} D^x_{i,k} F_{k,j} &- \sum_{k=0}^{N} D^y_{j,k} G_{i,k} = s_{i,j}, \quad i = 1,2,\ldots,N;\ j = 1,\ldots,N-1.
\end{aligned}
\tag{5.28}
$$

To specify the remaining degrees of freedom, namely the values of the solution along the other boundaries, we set them equal to their boundary values.

5.2.1.1 How to Implement the Collocation Approximation

To implement the collocation approximation, let us first introduce a structure of type *Nodal2DStorage* that we will use many times to group data needed by nodal spectral methods such as collocation. We use the structure to store the x and y locations of the collocation points, called ξ and η in the class, the quadrature weights, plus the derivative matrices that we may need. Since none of these quantities change during the course of a calculation, we only need to compute them once at the start. We show the structure in Algorithm 63 (Nodal2DStorage). Usually we only allocate and compute those quantities that we need for a particular approximation.

We encapsulate the collocation approximation of the potential problem in a class, too. The class stores an array for the solution and the source, plus an instance of the

Algorithm 63: *Nodal2DStorage:* Storage for a Nodal Spectral Method

Structure Nodal2DStorage
Data:
N, M
$\{\xi_i\}_{i=0}^{N}, \{\eta_j\}_{j=0}^{M}$; `// Gauss(-Lobatto) points`
$\{w_i^{(\xi)}\}_{i=0}^{N}, \{w_j^{(\eta)}\}_{j=0}^{M}$; `// Gauss(-Lobatto) weights`
$\left\{D_{i,j}^{\xi}\right\}_{i,j=0}^{N}, \left\{D_{i,j}^{\eta}\right\}_{i,j=0}^{M}$; `// First Derivative Matrices`
$\left\{D_{i,j}^{(2),\xi}\right\}_{i,j=0}^{N}, \left\{D_{i,j}^{(2),\eta}\right\}_{i,j=0}^{M}$; `// Second Derivative Matrices`
End Structure Nodal2DStorage

Algorithm 64: *NodalPotentialClass:* A Class for the Potential Problem on the Square

Class NodalPotentialClass
Uses Algorithms:
Algorithm 63 (Nodal2DStorage)
Data:
spA ; `// Of type Nodal2DStorage`
$\left\{\Phi_{i,j}\right\}_{i,j=0}^{N,M}$; `// Solution`
$\left\{s_{i,j}\right\}_{i,j=0}^{N,M}$; `// Source`
$\{mask_i\}_{i=1}^{4}$
Procedures:
Construct(N, M); `// Algorithm 65`
LaplacianOnTheSquare$(\{U_{i,j}\}_{i,j=0}^{N,M})$; `// Algorithm 66`
MatrixAction$(\{U_{i,j}\}_{i,j=0}^{N,M})$; `// Algorithm 68`
End Class NodalPotentialClass

structure Nodal2DStorage to store the necessary spectral approximation data. The quantities that we need to store are generic to all nodal spectral methods for the potential equation, so we present Algorithm 64 (NodalPotentialClass) as an implementation. The class includes an array called *mask*, which we will describe presently, to manage boundary conditions. We must also define at least two procedures. The first is to construct the nodes, weights and derivative matrices. The other is to compute the approximation of the Laplace operator, (5.22). We include in the class a procedure to compute the matrix action, which we will use for the iterative solution of the system of equations, (5.23).

We specify the choice of polynomial and approximation type in the constructor for the NodalPotentialClass. Algorithm 65 (NodalPotentialClass:Construct), for instance, shows a constructor for the Chebyshev collocation approximation. It computes the second derivative matrices by way of Algorithm 38 (mthOrderPolynomialDerivativeMatrix) with $m = 2$ and stores them in the second derivative matrix storage of the Nodal2DStorage structure. The first derivative matrices are not needed for the Poisson problem on the square, so they are not computed. We easily change

Algorithm 65: *NodalPotentialClass:Construct:* Constructor for the Chebyshev Collocation Approximation of the Potential Problem

Procedure Construct
Input: N, M
Uses Algorithms:
Algorithm 38 (mthOrderPolynomialDerivativeMatrix)
Algorithm 27 (ChebyshevGaussLobattoNodesAndWeights)

$this.spA.N \leftarrow N$; $this.spA.M \leftarrow M$
$\left\{this.spA.\{\xi_i\}_{i=0}^{N}, this.spA.\left\{w_i^{(\xi)}\right\}_{i=0}^{N}\right\} \leftarrow ChebyshevGaussLobattoNodesAndWeights(N)$
$this.spA.\left\{D_{i,j}^{(2),\xi}\right\}_{i,j=0}^{N} \leftarrow mthOrderPolynomialDerivativeMatrix(2, this.spA.\{\xi_j\}_{j=0}^{N})$
Repeat for η (y) direction...
End Procedure Construct

Algorithm 66: *NodalPotentialClass:LaplacianOnTheSquare:* Collocation Approximation to the Laplace Operator

Procedure LaplacianOnTheSquare
Input: $\left\{U_{i,j}\right\}_{i,j=0}^{N,M}$
Uses Algorithms:
Algorithm 19 (MxVDerivative)

$N \leftarrow this.spA.N$; $M \leftarrow this.spA.M$
for $j = 0$ **to** M **do**
$\left\{\left.\frac{\partial^2 U}{\partial x^2}\right|_{i,j}\right\}_{i=0}^{N} \leftarrow MxVDerivative\left(this.spA.\left\{D_{i,j}^{(2),\xi}\right\}_{i,j=0}^{N}, \left\{U_{i,j}\right\}_{i=0}^{N}\right)$
end
for $i = 0$ **to** N **do**
$\left\{\left.\frac{\partial^2 U}{\partial y^2}\right|_{i,j}\right\}_{j=0}^{M} \leftarrow MxVDerivative\left(this.spA.\left\{D_{i,j}^{(2),\eta}\right\}_{i,j=0}^{N}, \left\{U_{i,j}\right\}_{j=0}^{M}\right)$
end
for $j = 0$ **to** M **do**
for $i = 0$ **to** N **do**
$\nabla_N^2 U_{i,j} \leftarrow \left.\frac{\partial^2 U}{\partial x^2}\right|_{i,j} + \left.\frac{\partial^2 U}{\partial y^2}\right|_{i,j}$
end
end
return $\left\{\nabla_N^2 U_{i,j}\right\}_{i,j=0}^{N,M}$
End Procedure LaplacianOnTheSquare

the approximation to a Legendre method if we replace the calls to ChebyshevGaussLobattoNodesAndWeights with calls to Algorithm 25 (LegendreGaussLobattoNodesAndWeights).

We implement the action of the discrete Laplace operator (5.22) in Algorithm 66 (NodalPotentialClass:LaplacianOnTheSquare). It computes the matrix-vector mul-

tiplication by way of Algorithm 19 (MxVDerivative) so that we can use it for either Chebyshev or Legendre collocation approximations. Otherwise, for Chebyshev collocation we could use the Fast Chebyshev Transform. Notice that the algorithm makes the tensor product nature of the approximation explicit by the fact that the derivatives are computed row-by-row and column-by-column in the grid. Therefore the procedure computes the derivatives by passing array slices to the matrix-vector multiply routine. (Remember, we denote the passing of a slice of a two dimensional array, $\{U_{i,j}\}_{i,j=0}^{N,M}$, by $\{U_{i,j}\}_{i=0}^{N}$ for slices along columns and $\{U_{i,j}\}_{j=0}^{M}$ along rows. See Appendix A.)

To enforce the boundary conditions, we introduce the concept of an array *mask* function that we use selectively to set parts of an array to zero. We use the mask function to set the residual and solution values to zero along the boundaries for Dirichlet boundary conditions. We could also use them to set boundary fluxes to zero for Neumann boundary conditions. Mask functions provide a simple way to eliminate boundary points in iterative solvers. We will use them many times. To allow some flexibility, let us number the four sides of the square counter-clockwise starting with the boundary along $y = 0$. Let us then define an array $\{mask_k\}_{k=1}^{4}$. If a mask value $mask_k$ is true it will signal us to zero the boundary values of an array along side k. To ensure the mask is always available, we store it as a member array of the nodal approximation class in Algorithm 64 (NodalPotentialClass). We then use Algorithm 67 (MaskSides) to mask an input array as desired.

Finally, we introduced a procedure *MatrixAction* in Algorithm 64 (NodalPotentialClass). This is a function that we will use when we solve the linear system of equations (5.22) for the solution unknowns with an iterative method. For the potential problem, the matrix action is the function *LaplacianOnTheSquare* with boundary points masked as necessary, as we show in Algorithm 68 (NodalPotentialClass:MatrixAction).

5.2.1.2 How to Solve the Linear System

Equations (5.22) plus (5.24) form a linear system of equations that we need to solve for the $\Phi_{i,j}$. If the system is small enough, we can solve the system by a direct solver through a variant of Gauss elimination. Unlike the typical second order finite difference approximation, however, the system of equations represented by (5.22) is not pentadiagonal, but full. The system is neither diagonally dominant nor symmetric. Balancing the practical difficulties these properties create is the rapid convergence property of a spectral method; for smooth source and boundary conditions, the approximation error will converge much more quickly than the more easily solved finite difference approximation.

In practice, we will most likely solve the system defined by (5.22) plus (5.24) by an iterative technique. The topic of iterative solution of linear systems of equations is, of course, a huge one in the field of numerical linear algebra that we cannot fully survey here. Instead, we will describe representative algorithms that are appropriate for spectral collocation approximations and do not require significant amounts of extra storage.

Algorithm 67: *MaskSides:* Set Boundary Values to Zero According to a Mask Function

Procedure MaskSides
Input: $\{U_{ij}\}_{i,j=0}^{N,M}$, $\{mask_i\}_{i=1}^{4}$

if $mask_1 = true$ **then**
  for $i = 0$ **to** N **do**
    $U_{i,0} \leftarrow 0$
  end
end
if $mask_2 = true$ **then**
  for $j = 0$ **to** M **do**
    $U_{N,j} \leftarrow 0$
  end
end
if $mask_3 = true$ **then**
  for $i = 0$ **to** N **do**
    $U_{i,M} \leftarrow 0$
  end
end
if $mask_4 = true$ **then**
  for $j = 0$ **to** M **do**
    $U_{0,j} \leftarrow 0$
  end
end
return $\{U_{ij}\}_{i,j=0}^{N,M}$
End Procedure MaskSides

Algorithm 68: *NodalPotentialClass:MatrixAction:* Collocation Approximation to the Laplace Operator

Procedure MatrixAction
Input: $\{U_{ij}\}_{i,j=0}^{N,M}$
Uses Algorithms:
  Algorithm 64 (NodalPotentialClass)
  Algorithm 66 (NodalPotentialClass:LaplacianOnTheSquare)
  Algorithm 67 (MaskSides)

$N \leftarrow this.spA.N$; $M \leftarrow this.spA.M$
$\{action_{i,j}\}_{i,j=0}^{N,M} \leftarrow this.LaplacianOnTheSquare(\{U_{ij}\}_{i,j=0}^{N,M})$
$\{action_{i,j}\}_{i,j=0}^{N,M} \leftarrow MaskSides(\{action_{i,j}\}_{i,j=0}^{N,M}, this.\{mask_k\}_{k=1}^{4})$
return $\{action_{i,j}\}_{i,j=0}^{N,M}$
End Procedure MatrixAction

5.2.1.3 Direct Solution of the Equations

Direct solution of the system of equations represented by (5.22) is probably the simplest, particularly if one has an efficient direct solver already available. In

Appendix D.1.2, for example, we derive Algorithm 142 (LUFactorization) that we can use to solve a linear system by LU factorization. Fortunately, the LAPACK project [2] has made efficient and portable routines available for use with Fortran95/77 and C/C++. A Java binding is also available. There is little reason to write the direct solver oneself if one uses a programming language for which LAPACK bindings are available.

The main work on our part is to put the pointwise representation of the system, (5.22), into the standard matrix system form $A\mathbf{x} = \mathbf{y}$. We will generalize (5.22) at this point to allow N modes in the x direction and M modes in the y direction. To re-write the system, we start with the fact that the boundary values (for Dirichlet boundary conditions) are known. Thus, we shuffle them onto the right hand side of the equation

$$\begin{aligned}
&\sum_{k=1}^{N-1} D_{ik}^{(2),x} \Phi_{k,j} + \sum_{k=1}^{M-1} D_{ik}^{(2),y} \Phi_{i,k} \\
&\quad = s_{i,j} - D_{i0}^{(2),x} \Phi_{0,j} - D_{iN}^{(2),x} \Phi_{N,j} - D_{j0}^{(2),y} \Phi_{i,0} - D_{jM}^{(2),y} \Phi_{i,M} \\
&\quad \equiv RHS_{i,j}, \quad i = 1, 2, \ldots, N-1; \; j = 1, 2, \ldots, N-1. \qquad (5.29)
\end{aligned}$$

We must then arrange the two-dimensional array $RHS_{i,j}$ in the form of a vector array, $\{RHS_n\}_{n=1}^{L}$, where $L = (N-1) \times (M-1)$.

It is natural to store the matrix A either by rows or columns in the grid, depending on whether a language like C (rows) or Fortran (columns) is used. In either case, we make a mapping $n = index(i, j)$ between the location on the grid, i, j and the location in the array, n, which are

$$n = index(i, j) \equiv \begin{cases} i + (j-1)(N-1) & columnwise/Fortran, \\ j + (i-1)(M-1) & rowwise/C. \end{cases} \qquad (5.30)$$

We form RHS on the grid by Algorithm 69 (CollocationRHSComputation).

The next step is to construct the actual matrix, A, represented by the summations on the left of (5.29). To get the matrix entries, let us write (5.29) for the nth $= index(i, j)$ row,

$$\begin{aligned}
&D_{i1}^{(2),x} \Phi_{1,j} + D_{i2}^{(2),x} \Phi_{2,j} + \cdots + D_{i(N-1)}^{(2),x} \Phi_{N-1,j} \\
&\quad + D_{j1}^{(2),y} \Phi_{i,1} + D_{j2}^{(2),y} \Phi_{i,2} + \cdots + D_{j(M-1)}^{(2),y} \Phi_{i,M-1} = RHS_n, \\
&\quad i = 1, 2, \ldots, N-1, \; j = 1, 2, \ldots, M-1. \qquad (5.31)
\end{aligned}$$

The entry in the mth column of A is the coefficient of the mth value of Φ, stored according to the index function. For example, the grid location $(1, j)$ corresponds to the vector location $m = index(1, j)$. The coefficient of $\Phi_{1,j}$ in row i corresponds to the matrix element $A_{index(i,j),index(1,j)}$. When we look at (5.31), we see that two entries of the unknown solution appear in each row where $index(i, k) = index(k, j)$.

Algorithm 69: *CollocationRHSComputation:* Right Hand Side Construction for Direct Solution of the Collocation Equations

Procedure CollocationRHSComputation
Input: *npc* `// Instance of NodalPotentialClass`
Uses Algorithms:
Algorithm 64 (NodalPotentialClass)

$N \leftarrow npc.spA.N$
$M \leftarrow npc.spA.M$
$L \leftarrow (N-1) \times (M-1)$
for $j = 1$ **to** $M-1$ **do**
  for $i = 1$ **to** $N-1$ **do**
    $n \leftarrow index(i, j)$
    $RHS_n \leftarrow npc.s_{i,j} - npc.spA.D^{\xi}_{i,0} * npc.\Phi_{0,j} - npc.spA.D^{\xi}_{i,N} * npc.\Phi_{N,j} - npc.spA.D^{\eta}_{j,0} * npc.\Phi_{i,0} - npc.spA.D^{\eta}_{j,M} * npc.\Phi_{i,M}$
  end
end
return $\{RHS_n\}_{n=0}^{L}$
End Procedure CollocationRHSComputation

The matrix elements for those include both $D^{(2),x}$ and $D^{(2),y}$ values. All other rows include one or the other of $D^{(2),x}$ and $D^{(2),y}$. When we match terms, we find the matrix elements of the global collocation matrix

$$\begin{aligned} A_{index(i,j),index(k,j)} &= D^{(2),x}_{ik}, \quad k = 1, 2, \ldots, N-1;\ k \neq i, \\ A_{index(i,j),index(i,k)} &= D^{(2),y}_{jk}, \quad k = 1, 2, \ldots, M-1;\ k \neq j, \\ A_{index(i,j),index(i,j)} &= D^{(2),x}_{ii} + D^{(2),y}_{jj}. \end{aligned} \tag{5.32}$$

Algorithm 70 (LaplaceCollocationMatrix) implements these formulas.

Clearly, the construction of the matrix requires the storage of its $L = (N-1) \times (M-1)$ components. For large grids, this storage can be impractically large, making iterative solvers more appropriate. For systems of small size, however, solution by a direct solver is easy to implement. A performance comparison will wait until we have described the iterative solution procedure.

5.2.1.4 Iterative Solution of the Equations

Iterative solution is typically preferred for large systems of equations for two reasons. First, storage requirements can be significantly less than for a direct solver since we do not need to store the entire matrix. Instead, we only need the matrix A through its matrix-vector action on an iterate. We will not need to construct the matrix explicitly as we did above. Second, a particular application may not require the solution to be iterated to machine accuracy, which can reduce the cost. For those

Algorithm 70: *LaplaceCollocationMatrix:* Matrix Construction for Direct Solution of the Collocation Approximation for the Poisson Problem

Procedure LaplaceCollocationMatrix
Input: *npc* `// Instance of NodalPotentialClass`
Uses Algorithms:
Algorithm 64 (NodalPotentialClass)

$N \leftarrow npc.spA.N$; $M \leftarrow npc.spA.M$
$L \leftarrow (N-1) \times (M-1)$
for $m = 1$ **to** L **do**
  for $n = 1$ **to** L **do**
    $A_{n,m} \leftarrow 0$
  end
end
for $j = 1$ **to** $M-1$ **do**
  for $i = 1$ **to** $N-1$ **do**
    $n = index(i, j)$
    for $k = 1$ **to** $N-1$ **do**
      $m \leftarrow index(k, j)$
      $A_{n,m} \leftarrow npc.spA.D^{(2),\xi}_{i,k}$
    end
    for $k = 1$ **to** $M-1$ **do**
      $m \leftarrow index(i, k)$
      $A_{n,m} \leftarrow A_{n,m} + npc.spA.D^{(2),\eta}_{j,k}$
    end
  end
end
return $\left\{A_{n,m}\right\}_{n,m=1}^{L}$
End Procedure LaplaceCollocationMatrix

who do not have a background with iterative methods for the solution of linear systems, we give a quick introduction in Appendix D.2.

Of the many types of iterative solvers, we must choose one that is appropriate for the system of equations to be solved. The system of equations that the collocation approximation generates is not symmetric, so many classical iterative methods, including the Conjugate Gradient method are not appropriate. In this section, we will use the Bi-CGSTAB algorithm to solve the nonsymmetric system. We list that algorithm in Appendix D.2.

The goal of the iterative solver is to drive the iteration residual to zero at each collocation point. For the collocation approximation to the potential problem (5.22), we make the association $Ax \leftrightarrow \nabla_N^2 \Phi$ so that the matrix action applied to a set of grid point values $\{U_{i,j}\}_{i,j=0}^{N}$ is the left hand side of (5.22)

$$\nabla_N^2 U_{ij} \equiv \sum_{k=0}^{N} D_{ik}^{(2),x} U_{k,j} + \sum_{k=0}^{N} D_{jk}^{(2),y} U_{i,k}. \tag{5.33}$$

Algorithm 71: *Residual:* Residual for a Polynomial Collocation Approximation to the Potential Equation on the Square

Procedure Residual
Input: *npc* ; `// NodalPotentialClass`
Uses Algorithms:
Algorithm 64 (NodalPotentialClass)
Algorithm 66 (LaplacianOnTheSquare)
Algorithm 67 (MaskSides)
Algorithm 140 (BLAS_Level1)

$N = npc.spA.N$; $M = npc.spA.M$; $L \leftarrow (N+1) \times (M+1)$
$\{r\}_{i,j=0}^{N,M} \leftarrow npc.LaplacianOnSquare(npc.\{\Phi_{i,j}\}_{i,j=0}^{N,M})$
$\{r_{i,j}\}_{i,j=0}^{N,M} \leftarrow BLAS\_SCAL(L, -1, \{r_{i,j}\}_{i,j=0}^{N,M}, 1)$
$\{r_{i,j}\}_{i,j=0}^{N,M} \leftarrow BLAS\_AXPY(L, 1, npc.\{s_{i,j}\}_{i,j=0}^{N,M}, 1, \{r_{i,j}\}_{i,j=0}^{N,M}, 1)$
$\{r_{i,j}\}_{i,j=0}^{N,M} \leftarrow MaskSides(\{r_{i,j}\}_{i,j=0}^{N,M}, npc.\{mask_i\}_{i=1}^{4})$
return $\{r_{i,j}\}_{i,j=0}^{N,M}$
End Procedure Residual

The iteration residual for Dirichlet boundary conditions is therefore

$$r_{ij} = s_{ij} - \nabla_N^2 U_{ij}, \quad i, j = 1, 2, \ldots, N-1 \tag{5.34}$$

in the interior of the domain. Along the boundaries, Dirichlet conditions ensure that the residual vanishes. For Neumann conditions, we will want to include boundary residuals.

To compute the residual for the polynomial collocation approximation, we use Algorithm 66 (LaplacianOnSquare) and the source term stored in the collocation approximation class Algorithm 64 (NodalPotentialClass). Algorithm 71 (Residual) shows how to compute the residual (5.34) for a collocation approximation to the potential equation. For efficiency, we use BLAS Level 1 procedures to perform the basic whole array operations instead of directly using loops. We discuss the BLAS operations in Appendix C. For Dirichlet conditions, all elements of the mask array will be set to *true*.

5.2.1.5 A Finite Difference Preconditioner

Before we describe how to implement the Bi-CGSTAB iteration procedure, we note that a preconditioner, H, that approximates the matrix, A, is almost always used to accelerate convergence. (See Appendix D.2 a short discussion of preconditioning.) In many ways it is an art to develop suitable preconditioners, and we could use one of a variety of approximations. We could work directly with the matrix, say by using the diagonal of the original, as in a Jacobi method. The present context of a spectral approximation to a PDE allows a different approach. The preconditioner, H, comes from an alternative, yet easier to solve approximation to the original differential

equation. Possibilities include finite difference or finite element approximations to the original equations. Both have been used as preconditioners for spectral methods. Finite difference methods are easy to apply on the square, so we will describe the finite difference approximation first. We will derive a finite element preconditioner later in Sect. 5.2.2.3 for the nodal Galerkin approximation that we could use here just as well.

The finite difference preconditioner uses a low order and more easily invertible approximation to the original equations as an approximation to the spectral approximations. Since the collocation points used by the spectral approximation are not uniform, we must derive a finite difference approximation that takes this nonuniformity into account.

To derive the standard second order (when on a uniform grid) centered approximation to the second derivative, we take the derivative of a quadratic polynomial through three points at x_{j-1}, x_j, x_{j+1}, that have spacing $\Delta x_j = x_j - x_{j-1}$. We write the quadratic polynomial that interpolates a solution u at these points in Lagrange form as

$$I_2 u = \frac{(x - x_j)(x - x_{j+1})}{\Delta x_j(\Delta x_j + \Delta x_{j+1})} u_{j-1} - \frac{(x - x_{j-1})(x - x_{j+1})}{\Delta x_j \Delta x_{j+1}} u_j + \frac{(x - x_j)(x - x_{j-1})}{\Delta x_{j+1}(\Delta x_j + \Delta x_{j+1})} u_{j+1}. \tag{5.35}$$

The second derivative approximation is therefore

$$(I_2 u)''(x_j) = \frac{2}{\Delta x_j(\Delta x_j + \Delta x_{j+1})} u_{j-1} - \frac{2}{\Delta x_j \Delta x_{j+1}} u_j + \frac{2}{\Delta x_{j+1}(\Delta x_j + \Delta x_{j+1})} u_{j+1}. \tag{5.36}$$

When the spacing is uniform, $(I_2 u)''(x_j)$ reduces to the usual second order centered approximation to the second derivative.

In two space dimensions, we add the second derivative in the y direction to the finite difference operator to get

$$(H_{FD} u)_{i,j} = A_{ij} u_{i,j} + B_{ij} u_{i-1,j} + C_{ij} u_{i,j-1} + E_{ij} u_{i+1,j} + F_{ij} u_{i,j+1}, \tag{5.37}$$

where

$$\begin{aligned} A_{ij} &= -2\left(\frac{1}{\Delta x_i \Delta x_{i+1}} + \frac{1}{\Delta y_j \Delta y_{j+1}}\right), \\ B_{ij} &= \frac{2}{\Delta x_i(\Delta x_i + \Delta x_{i+1})}, \qquad C_{ij} = \frac{2}{\Delta y_j(\Delta y_j + \Delta y_{j+1})}, \\ E_{ij} &= \frac{2}{\Delta x_{i+1}(\Delta x_i + \Delta x_{i+1})}, \qquad F_{ij} = \frac{2}{\Delta y_{j+1}(\Delta y_j + \Delta y_{j+1})}. \end{aligned} \tag{5.38}$$

The matrix associated with the finite difference operator (5.37) is pentadiagonal, which is still more complex to invert than we would like. Rather than solve a pentadiagonal system directly, say by LU factorization (Appendix D.1.2), we approximate it by an incomplete LU factorization (ILU) approximation, $H_{ILU} = \hat{L}\hat{U}$ that approximates H_{FD} by the product of a lower triangular matrix $\hat{L}$ and an upper triangular matrix $\hat{U}$. The advantage of this product, as we will see, is that we can solve the system with minimal storage and simply by a forward followed by a backward elimination sweep.

We find the matrix entries for the triangular matrices $\hat{L}$ and $\hat{U}$ by matching the entries of H_{ILU} to H_{FD}. The individual actions of the lower and upper triangular matrices are

$$\begin{aligned}\left(\hat{L}\mathbf{u}\right)_{i,j} &= a_{ij}u_{i,j} + b_{ij}u_{i-1,j} + c_{ij}u_{i,j-1},\\ \left(\hat{U}\mathbf{u}\right)_{i,j} &= u_{i,j} + e_{ij}u_{i+1,j} + f_{ij}u_{i,j+1}.\end{aligned} \tag{5.39}$$

When multiplied together, the action of the lower and upper triangular matrices is

$$\begin{aligned}\hat{L}\left(\hat{U}\mathbf{u}\right)_{i,j} = a_{ij}&\left(u_{i,j} + e_{ij}u_{i+1,j} + f_{ij}u_{i,j+1}\right)\\ &+ b_{ij}\left(u_{i-1,j} + e_{i-1,j}u_{ij} + f_{i-1j}u_{i-1,j+1}\right)\\ &+ c_{ij}\left(u_{i,j-1} + e_{i,j-1}u_{i+1,j-1} + f_{i,j-1}u_{ij}\right).\end{aligned} \tag{5.40}$$

When we gather the coefficients of the $u_{i,j}$'s and match them to the coefficients of H_{FD} (5.37), we get the off-diagonal entries

$$\begin{aligned}c_{ij} &= C_{ij}, \qquad b_{ij} = B_{ij},\\ e_{ij} &= E_{ij}/a_{ij}, \qquad f_{ij} = F_{ij}/a_{ij}.\end{aligned} \tag{5.41}$$

The diagonal entry match gives

$$A_{ij} = a_{ij} + b_{ij}e_{i-1,j} + c_{ij}f_{i,j-1}, \tag{5.42}$$

which leaves two entries,

$$b_{ij}f_{i-1,j}u_{i+1,j-1} + c_{ij}e_{i,j-1}u_{i+1,j-1} \tag{5.43}$$

without matching terms in H_{FD}. To ensure that the approximation has the same row sum as the original (which often makes a better preconditioner), we add the additional off-diagonal terms to the diagonal entry

$$A_{ij} = a_{ij} + b_{ij}e_{i-1,j} + c_{ij}f_{i,j-1} + b_{ij}f_{i-1,j} + c_{ij}e_{i,j-1}. \tag{5.44}$$

Therefore, the diagonal entry in the lower tri-diagonal matrix, $\hat{L}$ is

$$a_{ij} = A_{ij} - \left(b_{ij}e_{i-1,j} + c_{ij}f_{i,j-1} + b_{ij}f_{i-1,j} + c_{ij}e_{i,j-1}\right). \tag{5.45}$$

With the coefficients matched, we write the actions of the lower and upper tridiagonal matrices without most of the intermediate variables as

$$\begin{aligned}\left(\hat{L}\mathbf{u}\right)_{ij} &= a_{ij}u_{ij} + B_{ij}u_{i-1,j} + C_{ij}u_{i,j-1},\\ \left(\hat{U}\mathbf{u}\right)_{ij} &= u_{ij} + \frac{E_{ij}}{a_{ij}}u_{i+1,j} + \frac{F_{ij}}{a_{ij}}u_{i,j+1}.\end{aligned} \tag{5.46}$$

The diagonal entries, a_{ij}, must be computed recursively, for

$$a_{ij} = A_{ij} - \frac{B_{ij}E_{i-1,j}}{a_{i-1,j}} - \frac{C_{ij}F_{i,j-1}}{a_{i,j-1}} - \frac{B_{ij}F_{i-1,j}}{a_{i-1,j}} - \frac{C_{ij}E_{i,j-1}}{a_{i,j-1}}. \tag{5.47}$$

To get the starting values of a_{ij}, we note that the preconditioned problem requires the solution of a system

$$\left(\hat{L}\hat{U}\mathbf{u}\right)_{i,j} = R_{i,j}, \tag{5.48}$$

which we break into two stages—a forward and then a backward elimination. If we call $w_{i,j} = (\hat{U}\mathbf{u})_{i,j}$, then

$$\left(\hat{L}\mathbf{w}\right)_{i,j} = R_{i,j} \tag{5.49}$$

is the lower triangular problem. Written out, the lower triangular problem is

$$a_{ij}w_{i,j} + B_{ij}w_{i-1,j} + C_{ij}w_{i,j-1} = R_{ij}, \quad i = 1, 2, \ldots, N-1;\ j = 1, 2, \ldots, M-1. \tag{5.50}$$

Boundary values of w, namely $w_{0,j}$ and $w_{j,0}$ that occur when $i = 1$ and $j = 1$ are moved to the right hand side of the equation, so to compute a_{ij} we take $B_{1,j} = 0$ and $C_{i,1} = 0$. Thus,

$$\begin{aligned}a_{11} &= A_{11},\\ a_{1j} &= A_{1j} - \frac{C_{1j}F_{1,j-1}}{a_{1,j-1}} - \frac{C_{1j}E_{1,j-1}}{a_{1,j-1}}, \quad j = 2, 3, \ldots, M-1,\\ a_{i1} &= A_{i1} - \frac{B_{i1}E_{i-1,1}}{a_{i-1,1}} - \frac{B_{i1}F_{i-1,1}}{a_{i-1,1}}, \quad i = 2, 3, \ldots, N-1.\end{aligned} \tag{5.51}$$

For all other points, we use (5.47).

It is convenient to encapsulate the data and procedures for the preconditioner into a class. At the minimum, this class should store the diagonal coefficients since they must be computed recursively. We will compromise between storage and execution speed and compute the off-diagonal coefficients on-the-fly rather than store them for each point. A prototype class for the preconditioner is shown in Algorithm 72 (FDPreconditioner).

The constructor for the class computes the grid spacing and the diagonal coefficients, as shown in procedure *Construct* in Algorithm 73 (FDPreconditioner: Constructor). We do not show the procedures to compute the coefficients, A–F, since they are simply direct applications of (5.38).

Algorithm 72: *FDPreconditioner:* A Class for a Finite Difference Preconditioner

Class FDPreconditioner
Data:
$N, M, \{a_{i,j}\}_{i,j=1}^{N,M}, \{dx_i\}_{i=1}^{N}, \{dy_j\}_{j=1}^{M}$
Procedures:
Construct$(N, M, \{x_i\}_{i=1}^{N}, \{y_j\}_{j=1}^{M})$; `// Algorithm 73`
$A(i, j)$; $B(i, j)$; $C(i, j)$; $E(i, j)$; $F(i, j)$; `// Equation (5.38)`
Solve$(\{R_{ij}\}_{i,j=0}^{N,M})$; `// Algorithm 74`
End Class FDPreconditioner

Algorithm 73: *FDPreconditioner:Construct:* Constructor for the Finite Difference Preconditioner on the Square

Procedure Construct
Input: $N, M, \{x_i\}_{i=0}^{N}, \{y_j\}_{j=0}^{M}$
$this.N \leftarrow N$; $this.M \leftarrow M$
for $i = 1$ **to** N **do**
  $this.dx_i \leftarrow x_i - x_{i-1}$
end
for $j = 1$ **to** M **do**
  $this.dy_j \leftarrow y_j - y_{j-1}$
end
$this.a_{1,1} \leftarrow this.A(1, 1)$
for $i = 2$ **to** $N - 1$ **do**
  $this.a_{i,1} \leftarrow this.A(i, 1) - \frac{this.B(i,1) * this.E(i-1,1)}{this.a_{i-1,1}} - \frac{this.B(i,1) * this.F(i-1,1)}{this.a_{i-1,1}}$
end
for $j = 2$ **to** $M - 1$ **do**
  $this.a_{1,j} \leftarrow this.A(1, j) - \frac{this.C(1,j) * this.F(1,j-1)}{this.a_{1,j-1}} - \frac{this.C(1,j) * this.E(1,j-1)}{this.a_{1,j-1}}$
  for $i = 2$ **to** $N - 1$ **do**
    $this.a_{i,j} \leftarrow this.A(i, j) - \frac{this.B(i,j) * this.E(i-1,j)}{this.a_{i-1,j}} - \frac{this.C(i,j) * this.F(i,j-1)}{this.a_{i,j-1}} - \frac{this.B(i,j) * this.F(i-1,j)}{this.a_{i-1,j}} - \frac{this.C(i,j) * this.E(i,j-1)}{this.a_{i,j-1}}$
  end
end
End Procedure Construct

The last main procedure is *Solve*, which solves the system $H_{ILU}\mathbf{z} = \mathbf{r}$. For this, we use a modification of the LU solver, procedure *LUSolve*, presented in Algorithm 142 (LUFactorization).

The first stage of the ILU solver is the forward substitution on the lower triangular part of the system. We have already written the lower triangular matrix problem in

pointwise form in (5.46). With the boundary conditions $B_{1,j} = 0$ and $C_{i,1} = 0$,

$$\begin{aligned} w_{11} &= R_{11}/a_{11}, \\ w_{i1} &= \left(R_{i1} - B_{i1} w_{i-1,1}\right)/a_{i1}, \\ w_{1,j} &= \left(R_{ij} - C_{1j} w_{1,j-1}\right)/a_{1j}, \end{aligned} \tag{5.52}$$

and the interior point values are computed by

$$w_{i,j} = \left(R_{i,j} - B_{ij} w_{i-1,j} - C_{ij} w_{i,j-1}\right)/a_{ij}. \tag{5.53}$$

We make similar arguments to develop the backward substitution for the upper triangular part, $(\hat{U}u)_{i,j} = w_{i,j}$, namely

$$\begin{aligned} u_{N-1,M-1} &= w_{N-1,M-1}, \\ u_{i,M-1} &= w_{i,M-1} - \frac{E_{i,M-1}}{a_{i,M-1}} u_{i+1,M-1}, \\ u_{N-1,j} &= w_{N-1,j} - \frac{F_{N-1,j}}{a_{N-1,j}} u_{N-1,j+1}, \end{aligned} \tag{5.54}$$

while the interior point values are

$$u_{i,j} = w_{i,j} - \frac{E_{ij}}{a_{ij}} u_{i+1,j} - \frac{F_{ij}}{a_{ij}} u_{i,j+1}. \tag{5.55}$$

Algorithm 74 (FDPreconditioner:Solve) implements (5.52)–(5.55) to solve the preconditioned system, assuming that the coefficients A–F are computed on the fly. It takes a right hand side array, R, and the coefficients of the diagonal of $\hat{L}$ computed in Algorithm 73 (FDPreconditioner), and returns the solution to the system.

As we will soon see, the effect of preconditioning is significant. Although the use of preconditioning adds significant complexity to the solution procedure, and can be avoided for small problems, it should be considered a must to solve large scale potential problems.

5.2.1.6 How to Construct the Iterative Potential Solver

The purpose of the iterative solver is to find the solution $\{\Phi_{ij}\}_{i,j=0}^{N,M}$ so that the iteration residual, (5.34) vanishes, or in practice a norm of the residual is less than some specified tolerance. As we said earlier, we will use the Bi-CGSTAB solver that we list in Appendix D.2, since the system of equations is not symmetric. For our particular implementation, we will write the solver almost completely in terms of Level 1 BLAS (Appendix C) operations.

Algorithm 74: *FDPreconditioner:Solve:* Solver for the ILU Preconditioner $H_{ILU}\mathbf{u} = \mathbf{R}$

Procedure Solve
Input: $\{R_{ij}\}_{i,j=0}^{N,M}$
$N = this.N$; $M = this.M$
$w_{1,1} \leftarrow R_{1,1}/this.a_{1,1}$
for $i = 2$ **to** $N-1$ **do**
  $w_{i,1} \leftarrow \left(R_{i,1} - this.B(i,1) * w_{i-1,1}\right)/this.a_{i,1}$
end
for $j = 2$ **to** $M-1$ **do**
  $w_{1,j} \leftarrow \left(R_{1,j} - this.C(1,j) * w_{1,j-1}\right)/this.a_{1,j}$
  for $i = 2$ **to** $N-1$ **do**
    $w_{i,j} \leftarrow \left(R_{i,j} - this.B(i,j) * w_{i-1,j} - this.C(i,j) * w_{i,j-1}\right)/this.a_{i,j}$
  end
end
$u_{N-1,M-1} \leftarrow w_{N-1,M-1}$
for $i = N-2$ **to** 1 **Step** -1 **do**
  $u_{i,M-1} \leftarrow w_{i,M-1} - \dfrac{this.E(i,M-1)}{this.a_{i,M-1}} u_{i+1,M-1}$
end
for $j = M-2$ *to* 1 **Step** -1 **do**
  $u_{N-1,j} \leftarrow w_{N-1,j} - \dfrac{this.F(N-1,j)}{this.a_{N-1,j}} u_{N-1,j+1}$
  for $i = N-2$ **to** 1 **Step** -1 **do**
    $u_{i,j} \leftarrow w_{i,j} - \dfrac{this.E(i,j)}{this.a_{i,j}} u_{i+1,j} - \dfrac{this.F(i,j)}{this.a_{i,j}} u_{i,j+1}$
  end
end
return $\{u_{i,j}\}_{i,j=0}^{N,M}$
End Procedure Solve

We show our implementation of the Bi-CGSTAB method for the polynomial spectral collocation approximation in Algorithm 75 (Bi-CGSSTABSolve). The input is the maximum number of iterations to be allowed, N_{it}, and the tolerance *TOL* for convergence. It also takes an instance of the spatial approximation, in this case defined by Algorithm 64 (NodalPotentialClass), and an instance of the preconditioner, e.g. Algorithm 72 (FDPreconditioner).

Finally, we need a driver to solve the potential problem. The driver must perform tasks like compute the source array, construct the spatial approximation, call the solver, and set the boundary conditions. We present an example driver for the Chebyshev collocation approximation in Algorithm 76 (CollocationPotentialDriver). We include calls to a source value function and to an external routine that sets the boundary values for Dirichlet conditions that need to be user supplied. We have the driver initialize the mask array, here initialized for Dirichlet conditions on all four sides.

Algorithm 75: *BiCGSSTABSolve:* BiCGStab Iterative Solver for Nodal Spectral Methods

Procedure BiCGSSTABSolve
Input: N_{it}, TOL
Input: npc ; `// NodalPotentialClass instance`
Input: H ; `// Preconditioner instance, e.g. FDPreconditioner`
Uses Algorithms:
Algorithm 64 (NodalPotentialClass)
Algorithm 71 (Residual)
Algorithm 74 (Solve)
Algorithm 140 (BLAS_Level1)

$N \leftarrow npc.spA.N$; $M \leftarrow npc.spA.M$; $L \leftarrow (N+1) \times (M+1)$
$\rho \leftarrow 1$; $\alpha \leftarrow 1$; $omega \leftarrow 1$
$\{r\}_{i,j=0}^{N,M} \leftarrow Residual(npc)$
$\{\bar{r}_{ij}\}_{i,j=0}^{N,M} \leftarrow BLAS\_COPY(L, \{r_{ij}\}_{i,j=0}^{N,M}, 1, \{\bar{r}_{ij}\}_{i,j=0}^{N,M}, 1)$
for $k = 1$, N_{it} **do**
  $\hat{\rho} \leftarrow \rho$
  $\rho \leftarrow BLAS\_DOT(L, \{\bar{r}_{ij}\}_{i,j=0}^{N,M}, 1, \{r_{ij}\}_{i,j=0}^{N,M}, 1)$
  $\beta \leftarrow \rho\alpha/(\hat{\rho}\omega)$
  $\{p_{ij}\}_{i,j=0}^{N,M} \leftarrow BLAS\_AXPY(L, -\omega, \{v_{ij}\}_{i,j=0}^{N,M}, 1, \{p_{ij}\}_{i,j=0}^{N,M}, 1)$
  $\{p_{ij}\}_{i,j=0}^{N,M} \leftarrow BLAS\_SCAL(L, beta, \{p_{ij}\}_{i,j=0}^{N,M}, 1)$
  $\{p_{ij}\}_{i,j=0}^{N,M} \leftarrow BLAS\_AXPY(L, 1, \{r_{ij}\}_{i,j=0}^{N,M}, 1, \{p_{ij}\}_{i,j=0}^{N,M}, 1)$
  $\{y_{ij}\}_{i,j=0}^{N,M} \leftarrow H.Solve(\{p_{ij}\}_{i,j=0}^{N,M})$
  $\{v_{ij}\}_{i,j=0}^{N,M} \leftarrow npc.MatrixAction(\{y_{ij}\}_{i,j=0}^{N,M})$
  $\alpha \leftarrow \rho/BLAS\_DOT(L, \{\bar{r}_{ij}\}_{i,j=0}^{N,M}, 1, \{v_{ij}\}_{i,j=0}^{N,M}, 1)$
  $\{s_{ij}\}_{i,j=0}^{N,M} \leftarrow BLAS\_COPY(L, \{r_{ij}\}_{i,j=0}^{N,M}, 1, \{s_{ij}\}_{i,j=0}^{N,M}, 1)$
  $\{s_{ij}\}_{i,j=0}^{N,M} \leftarrow BLAS\_AXPY(L, -\alpha, \{v_{ij}\}_{i,j=0}^{N,M}, 1, \{s_{ij}\}_{i,j=0}^{N,M}, 1)$
  $\{z_{ij}\}_{i,j=0}^{N,M} \leftarrow H.Solve(\{s_{ij}\}_{i,j=0}^{N,M})$
  $\{t_{ij}\}_{i,j=0}^{N,M} \leftarrow npc.MatrixAction(\{z_{ij}\}_{i,j=0}^{N,M})$
  $\omega \leftarrow$
  $BLAS\_DOT(L, \{t_{ij}\}_{i,j=0}^{N,M}, 1, \{s_{ij}\}_{i,j=0}^{N,M}, 1)/BLAS\_DOT(L, \{t_{ij}\}_{i,j=0}^{N,M}, 1, \{t_{ij}\}_{i,j=0}^{N,M}, 1)$
  $npc.\{\Phi_{ij}\}_{i,j=0}^{N,M} \leftarrow BLAS\_AXPY(L, \alpha, \{y_{ij}\}_{i,j=0}^{N,M}, 1, npc.\{\Phi_{ij}\}_{i,j=0}^{N,M}, 1)$
  $npc.\{\Phi_{ij}\}_{i,j=0}^{N,M} \leftarrow BLAS\_AXPY(L, \omega, \{z_{ij}\}_{i,j=0}^{N,M}, 1, npc.\{\Phi_{ij}\}_{i,j=0}^{N,M}, 1)$
  $\{r_{ij}\}_{i,j=0}^{N,M} \leftarrow BLAS\_COPY(L, \{s_{ij}\}_{i,j=0}^{N,M}, 1, \{r_{ij}\}_{i,j=0}^{N,M}, 1)$
  $\{r_{ij}\}_{i,j=0}^{N,M} \leftarrow BLAS\_AXPY(L, -\omega, \{t_{ij}\}_{i,j=0}^{N,M}, 1, \{r_{ij}\}_{i,j=0}^{N,M}, 1)$
  if $BLAS\_NRM2(L, \{r_{ij}\}_{i,j=0}^{N,M}, 1) < TOL$ **then** Exit
end
return npc
End Procedure BiCGSSTABSolve

Algorithm 76: *CollocationPotentialDriver:* Driver for a Polynomial Collocation Approximation to the Potential on the Square

Procedure Main
Input: N, M, N_{it}, TOL
Uses Algorithms:
Algorithm 64 (NodalPotentialClass)
Algorithm 65 (NodalPotentialClass:Construct)
Algorithm 72 (FDPreconditioner)
Algorithm 73 (FDPreconditioner:Construct)
Algorithm 75 (BiCGStabSolve)
Derived Types: NodalPotentialClass: *npc*, FDPreconditioner: H

$npc.Construct(N, M)$
for $j = 0$ **to** M **do**
  for $i = 0$ **to** N **do**
    $npc.s_{i,j} \leftarrow SourceValue(npc.spA.\xi_i, npc.spA.\eta_j)$
  end
end
$npc.\{mask_k\}_{k=1}^{4} \leftarrow \{true, true, true, true\}$
$npc.\{\Phi_{ij}\}_{i,j=0}^{N,M} \leftarrow SetBoundaryValues(npc.\{\Phi_{ij}\}_{i,j=0}^{N,M})$
$H.Construct(N, M, npc.spA.\{\xi_i\}_{i=0}^{N}, npc.spA.\{eta_j\}_{j=0}^{M})$
$npc \leftarrow BiCGStabSolve(N_{it}, TOL, npc, H)$
Output results, etc.
End Procedure Main

5.2.1.7 Benchmark Solution

We have so far derived two collocation approximations—Chebyshev and Legendre—and two solvers—direct and iterative. Our benchmark solution compares the performance of the approximations and solvers. We will solve the simple model boundary value problem

$$\begin{cases} \nabla^2\varphi = \varphi_{xx} + \varphi_{yy} = -8\pi^2 \cos(2\pi x)\sin(2\pi y), & (x, y) \in (-1, 1) \times (-1, 1), \\ \varphi(x, -1) = 0, & -1 \le x \le 1, \\ \varphi(1, y) = \sin(2\pi y), & -1 \le y \le 1, \\ \varphi(x, 1) = 0, & -1 \le x \le 1, \\ \varphi(-1, y) = \sin(2\pi y), & -1 \le y \le 1, \end{cases} \tag{5.56}$$

which has the analytical solution $\varphi = \cos(2\pi x)\sin(2\pi y)$. We show the Chebyshev collocation solution for this problem in Fig. 5.1 for $N = M = 64$. We are interested in the accuracy and convergence behavior, especially the differences between the Chebyshev and Legendre approximations. We also need to see how effective the solvers are for the solution of the linear systems.

We show the logarithm of the maximum errors for the Chebyshev and Legendre approximations in Table 5.1. We see that the errors decay exponentially fast. Doubling the number of points in each direction causes the error to drop by a factor

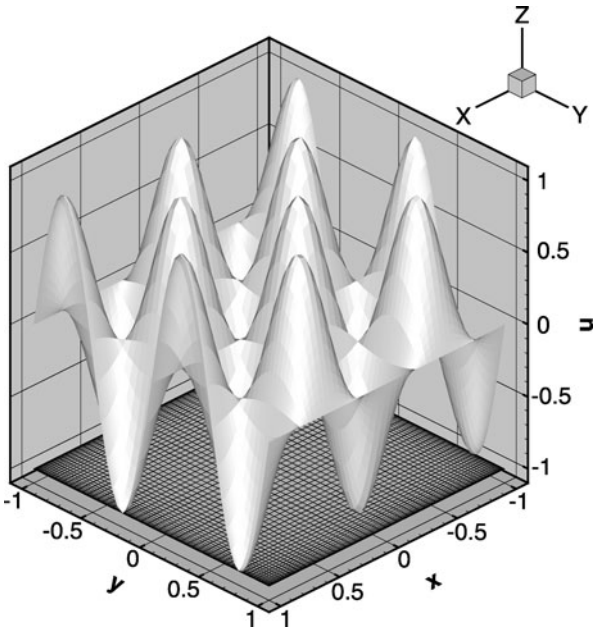

Fig. 5.1 Solution and grid for the Chebyshev collocation approximation of the steady potential in a square with a sinusoidal heat source

Table 5.1 Logarithm of maximum errors for collocation approximations to (5.56)

N	8	12	16	20	24
Chebyshev	−1.5375	−3.8044	−6.6535	−9.8774	−13.411
Legendre	−1.8658	−4.1584	−7.1440	−10.451	−14.020

of approximately one thousand. Finally, we see that the Legendre approximation is "slightly" more accurate by about a factor of three.

Next, let's examine the performance of the direct and iterative solvers. To find the solutions with the direct solver, we used Algorithms 69 (CollocationRHSComputation) and 70 (LaplaceCollocationMatrix) to set up the matrix system. We then used Algorithm 142 (LUFactorization) to solve the system. For comparison, we also used the LAPACK routine DGETRF to perform the LU decomposition on the matrix and its companion DGETRS to solve the system. To solve the system iteratively, we used Algorithm 75 (BiCGSSTABSolve) with the BLAS routines of 140 (BLAS_Level1).

We show the performance of the Bi-CGSTAB iterative solver in Fig. 5.2, which plots the logarithm of the norm

$$\|r\| = \sqrt{\sum_{i,j=0}^{N,M} (r_{ij})^2} \tag{5.57}$$

for the Chebyshev collocation approximation with $N = 72$ and an initial iterate $\Phi = 0$. Figure 5.2 clearly shows the need to precondition. The preconditioned iter-

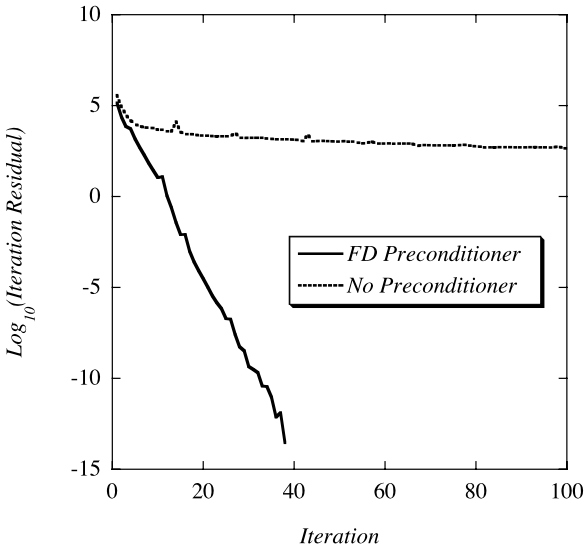

Fig. 5.2 Iteration convergence of the Bi-CGSTAB iterative solver for the Chebyshev collocation approximation to the steady potential in a square with a source when $N = 72$

Table 5.2 Number of iterations for the Bi-CGSTAB solver with finite difference preconditioning

N	8	16	32	64	72
Chebyshev	12	18	24	36	39
Legendre	13	17	24	34	39

Table 5.3 Storage requirements (MB) for the direct and iterative solvers

N	8	16	32	64
Direct	0.02	0.41	7.4	126.1
Iterative	0.004	0.014	0.052	0.2

ation converges to near machine precision in only 39 iterations. Since the condition number of the system increases with the size of the matrix, the number of iterations increases with the size. Table 5.2 shows the number of iterations as a function of N for both the Chebyshev and Legendre approximations.

Ultimately, we make the decision to use a direct or an iterative solver on the memory and CPU time usage of each. Memory considerations give the advantage to the iterative solvers. Table 5.3 shows the actual number of megabytes needed for the direct and iterative solvers when we used double precision arithmetic. The memory needs of the direct solver grow as N^4. For the iterative solver, we traded some storage in return for some increased computation time by only storing the diagonal of the finite difference preconditioner. Even with additional storage for the off-diagonal terms, the memory requirements for the iterative solver grow only as N^2. It is clear that the storage requirements of the direct solver become prohibitive as the system gets larger.

Table 5.4 CPU time (s) for the direct and BiCGStab solvers

N	Algorithm 142 (Decomposition)	Algorithm 142 (Solve)	LAPACK (Decomposition)	LAPACK (Solve)	BiCGStab
16	0.01	0	0	0	0.002
24	0.16	0.0007	0.04	0	0.004
48	30.0	0.05	1.73	0.02	0.026
64	203	0.14	9.83	0.09	0.064
72	421	0.3	20.1	0.15	0.09

CPU comparisons of the direct vs. preconditioned iterative solvers also favor the iterative solution of the equations. Table 5.4 shows timings for the LU decomposition and solve for the direct solvers, along with the time to converge the iterative solver. The first thing that we notice is that we should use the LAPACK routines to solve the system, not our Algorithm 142 (LUFactorization). The comparison is not totally fair, however, since the LAPACK computations used a vendor supplied optimized version of LAPACK that automatically ran in parallel on eight CPU cores. But even if we account for a factor of eight, the LAPACK routines are still significantly better. If we add the decomposition and solve time, the (parallel) direct solver is more efficient than the iterative solver only for $N < 24$. However, if the decomposition needs only to be done once, such as part of a time dependent problem (see, e.g., Sect. 5.3) then we may be able to amortize the cost of the factorization. At $N = 24$, for instance, it becomes less expensive to use the direct solver if the same system is solved more than six times. At $N \geq 48$, however, the direct solver is never more efficient. Our conclusion that the iterative solution is the better choice is strengthened by the fact that the iterative solver used Algorithm 75 (BiCGSSTABSolve) and the unoptimized BLAS routines based on Algorithm 140 (BLAS_Level1) and Algorithm 19 (MxVDerivative) rather than one of the faster alternatives.

5.2.2 The Nodal Galerkin Approximation

Recall that the Galerkin approximation uses an alternative set of constraints to find the degrees of freedom. It starts from a weak form of the equation to determine the unknowns that define the polynomial, Φ, that approximates the potential, φ. To get the weak form, we multiply the equation by a smooth function $\phi(x, y)$ that satisfies the boundary conditions, multiply by the weight function appropriate to the polynomial in which we expand the solution, and integrate over the domain

$$\int_{-1}^{1}\int_{-1}^{1}\left(s - \nabla^2\varphi\right)\phi\,(x, y)\,w(x, y)dxdy = 0. \tag{5.58}$$

We then apply Green's identity to re-write (5.58).

For smooth enough functions u and v Green's first identity is

$$\int_{-1}^{1}\int_{-1}^{1} v\nabla^2 u dxdy = \int_{\partial} v\nabla u \cdot \hat{n} dS - \int_{-1}^{1}\int_{-1}^{1} \nabla u \cdot \nabla v dxdy. \tag{5.59}$$

Here, ∂ represents the boundary of the square, $\hat{n}$ represents the outward facing normal, and dS is the associated surface differential. To simplify the use of Green's identity, it is convenient to choose an approximation for which the weight function $w = 1$. This implies that we want to use a Legendre approximation. We see that the presence of the weight function in (5.58) highlights a difference between the Legendre or the Chebyshev approximation that we don't see in the collocation approximation. In the Legendre approximation, the weight function is unity. For Chebyshev approximations, $w(x, y) = 1/\sqrt{(1-x^2)(1-y^2)}$.

If we use the Legendre weight, we get the weak form of the potential equation

$$\int_{\partial} \phi\nabla\varphi \cdot \hat{n} dS - \int_{-1}^{1}\int_{-1}^{1} \nabla\phi \cdot \nabla\varphi dxdy = \int_{-1}^{1}\int_{-1}^{1} s\phi dxdy \tag{5.60}$$

when we apply (5.59) to (5.58). Since ϕ satisfies the boundary conditions, $\phi = 0$ along the boundary, and the boundary integral in (5.60) vanishes, leaving the final form of the equation

$$-\int_{-1}^{1}\int_{-1}^{1} \nabla\phi \cdot \nabla\varphi dxdy = \int_{-1}^{1}\int_{-1}^{1} s\phi dxdy. \tag{5.61}$$

As before, to get the Galerkin approximation, we replace φ by its polynomial approximation Φ, the source s by its polynomial approximation, S, and take ϕ to be any polynomial of the same degree as Φ that satisfies the boundary conditions. Then the Galerkin approximation for the potential problem on the square is

$$-\int_{-1}^{1}\int_{-1}^{1} \left\{ \frac{\partial\phi}{\partial x}\frac{\partial\Phi}{\partial x} + \frac{\partial\phi}{\partial y}\frac{\partial\Phi}{\partial y} \right\} dxdy = \int_{-1}^{1}\int_{-1}^{1} \phi S dxdy. \tag{5.62}$$

The Galerkin approximation is not limited to constant coefficient problems. To approximate the variable coefficient problem

$$\nabla \cdot (\nu\nabla\varphi) = s \tag{5.63}$$

that we already considered in (5.25), we simply change the weak form (5.61) to

$$-\int_{-1}^{1}\int_{-1}^{1} \nabla\phi \cdot (\nu\nabla\varphi)\, dxdy = \int_{-1}^{1}\int_{-1}^{1} s\phi dxdy. \tag{5.64}$$

We complete the approximation when we find the system of equations that the degrees of freedom satisfy. Since we are deriving the nodal Galerkin approximation,

we write Φ in nodal form

$$\Phi(x, y) = \sum_{i,j=0}^{N} \Phi_{i,j} \ell_i(x) \ell_j(y). \tag{5.65}$$

The boundary conditions tell us that $\Phi_{i,j} = 0$ along the boundaries so there are a total of $(N-1)^2$ degrees of freedom to determine. To find the equations, we use the fact that ϕ is now any polynomial of the same degree as the solution that satisfies the boundary conditions. Therefore we can write ϕ also in the nodal form

$$\phi(x, y) = \sum_{i,j=0}^{N} \phi_{ij} \ell_i(x) \ell_j(y) \tag{5.66}$$

for any nodal values $\phi_{i,j}$, $i, j = 1, 2, \ldots, N-1$ with $\phi_{i,j} = 0$ on the boundaries. When we substitute for ϕ in (5.62) and rearrange,

$$\begin{aligned} &-\sum_{i,j=1}^{N-1} \phi_{i,j} \left\{ \int_{-1}^{1} \int_{-1}^{1} \left[\ell_j(y) \frac{\partial \ell_i}{\partial x} \frac{\partial \Phi}{\partial x} + \ell_i(x) \frac{\partial \ell_j}{\partial y} \frac{\partial \Phi}{\partial y} \right] dxdy \right\} \\ &\quad = \sum_{i,j=1}^{N-1} \phi_{i,j} \left\{ \int_{-1}^{1} \int_{-1}^{1} \ell_i(x) \ell_j(y) S dxdy \right\}. \end{aligned} \tag{5.67}$$

Since the $\phi_{i,j}$ are arbitrary and hence linearly independent, the coefficients of each must match, which gives us the $(N-1)^2$ equations

$$\begin{aligned} &-\int_{-1}^{1} \int_{-1}^{1} \left[\ell_j(y) \frac{\partial \ell_i}{\partial x} \frac{\partial \Phi}{\partial x} + \ell_i(x) \frac{\partial \ell_j}{\partial y} \frac{\partial \Phi}{\partial y} \right] dxdy \\ &\quad = \int_{-1}^{1} \int_{-1}^{1} \ell_i(x) \ell_j(y) S dxdy, \quad i, j = 1, 2, \ldots, N-1. \end{aligned} \tag{5.68}$$

To get equations for the unknown grid point values, $\Phi_{i,j}$, we replace Φ in (5.68) with (5.65) and change the independent indices

$$\begin{aligned} &-\int_{-1}^{1} \int_{-1}^{1} \left\{ \ell_j(y) \ell_i'(x) \left(\sum_{n,m=0}^{N} \Phi_{n,m} \ell_m(y) \ell_n'(x) \right) dxdy \right\} \\ &\quad -\int_{-1}^{1} \int_{-1}^{1} \left\{ \ell_i(x) \ell_j'(y) \left(\sum_{n,m=0}^{N} \Phi_{n,m} \ell_m'(y) \ell_n(x) \right) dxdy \right\} \\ &\quad = \int_{-1}^{1} \int_{-1}^{1} \ell_i(x) \ell_j(y) S dxdy, \quad i, j = 1, 2, \ldots, N-1. \end{aligned} \tag{5.69}$$

We then swap the orders of the summations and integrals, and gather the coefficients of the unknowns

$$
\begin{aligned}
&-\sum_{n,m=0}^{N} \Phi_{n,m}\left[\int_{-1}^{1}\int_{-1}^{1} \ell_n'(x)\ell_i'(x)\ell_m(y)\ell_j(y)dxdy\right] \\
&\quad -\sum_{n,m=0}^{N} \Phi_{n,m}\left[\int_{-1}^{1}\int_{-1}^{1} \ell_i(x)\ell_n(x)\ell_j'(y)\ell_m'(y)dxdy\right] \\
&\quad =\int_{-1}^{1}\int_{-1}^{1} \ell_i(x)\,\ell_j(y)\,S dxdy, \quad i,j=1,2,\ldots,N-1. \qquad (5.70)
\end{aligned}
$$

Equation (5.70) plus the boundary values defines a linear system for the interior values of the $\Phi_{i,j}$ with coefficients given by the integrals within the square brackets.

Although we could try to evaluate the integrals to derive the coefficients exactly, we are developing a nodal Galerkin approximation with a quadrature approximation of the integrals. We will therefore replace the integrals with Gauss-Lobatto quadratures. Quadrature simplifies the computation of the coefficients, retains spectral accuracy, easily extends to variable coefficient problems like (5.64), and will be used as the foundation of the spectral element method. With the quadrature approximation, the discrete orthogonality (Sect. 1.11) of the Lagrange interpolating polynomials causes the integral over the source term to reduce to

$$
\begin{aligned}
\int_{-1}^{1}\int_{-1}^{1} \ell_i(x)\,\ell_j(y)\,S dxdy &\approx \sum_{n,m=0}^{N} w_n w_m \ell_i(x_n)\,\ell_j(y_m)\,S(x_n,y_m) \\
&= w_i w_j S(x_i,y_j). \qquad (5.71)
\end{aligned}
$$

We also find the coefficients on the left of (5.70) by replacing the integrals with quadrature. For the first term on the left, we have the approximation

$$
\begin{aligned}
&\int_{-1}^{1}\int_{-1}^{1} \ell_n'(x)\ell_i'(x)\ell_m(y)\ell_j(y)dxdy \\
&\quad =\left(\int_{-1}^{1} \ell_m(y)\ell_j(y)dy\right)\left(\int_{-1}^{1} \ell_n'(x)\ell_i'(x)dx\right) \\
&\quad \approx \left(\delta_{j,m} w_m\right)\left(\sum_{k=0}^{N} \ell_n'(x_k)\ell_i'(x_k)w_k\right). \qquad (5.72)
\end{aligned}
$$

As before (e.g. Sect. 3.5.2), let us call

$$
D_{nk}^{(x)} = \ell_n'(x_k) \qquad (5.73)
$$

and define the symmetric matrix

$$G_{in}^{(x)} = G_{ni}^{(x)} = \sum_{k=0}^{N} D_{nk}^{(x)} D_{ik}^{(x)} w_k. \tag{5.74}$$

Therefore, we approximate the first term in (5.70) by

$$\sum_{n,m=0}^{N} \Phi_{n,m} \left[\int_{-1}^{1} \int_{-1}^{1} \ell_n'(x) \ell_i'(x) \ell_m(y) \ell_j(y) dx dy \right] \approx \sum_{n=0}^{N} \Phi_{n,j} w_j^{(y)} G_{in}^{(x)}. \tag{5.75}$$

Similarly, we approximate the second term on the left of (5.70) by

$$\sum_{n,m=0}^{N} \Phi_{n,m} \left[\int_{-1}^{1} \int_{-1}^{1} \ell_i(x) \ell_n(x) \ell_j'(y) \ell_m'(y) dx dy \right] \approx \sum_{m=0}^{N} \Phi_{i,m} w_i^{(x)} G_{jm}^{(y)}. \tag{5.76}$$

When we put the two together, we have the nodal Galerkin approximation

$$-\left\{ \sum_{k=0}^{N} w_j^{(y)} G_{ik}^{(x)} \Phi_{k,j} + w_i^{(x)} G_{jk}^{(y)} \Phi_{i,k} \right\} = w_i^{(x)} w_j^{(y)} S_{i,j}, \quad i, j = 1, 2, \ldots, N-1. \tag{5.77}$$

We will represent the quantity on the left by $(\nabla^2 \Phi, \ell_i \ell_j)_N$.

Equation (5.77) looks much like the collocation approximation, (5.22), with one important difference. In the collocation approximation the coefficient matrix is not symmetric, whereas the Galerkin coefficient matrix, G, clearly is. The symmetry allows us to use the popular and efficient Conjugate Gradient method (Appendix D.2) to solve the system iteratively. To maintain this symmetry, we do not divide the system by the coefficients of the mass matrix (which is clearly diagonal) represented by the product $w_i^{(x)} w_j^{(y)}$, as we did in the approximation of the time dependent one-dimensional problem in (4.122).

5.2.2.1 How to Implement the Nodal Galerkin Approximation

We have already developed the machinery we need to implement the approximation to the Laplace operator; we only need to modify existing algorithms. To implement the nodal Galerkin method, we reuse the nodal approximation class Algorithm 64 (NodalPotentialClass). We have to change the constructor, the Laplace operator approximation and the driver. The new implementations are:

- *Create a nodal Galerkin constructor*. To change the constructor, Algorithm 65 (NodalPotentialClass:Construct), we note that the second derivative matrices in the Nodal2DStorage structure must now store the matrices G_{ik} defined by (5.74) and computed by Algorithm 57 (CGDerivativeMatrix). The quadrature weights and nodes must be computed by Algorithm 25 (LegendreGaussLobattoNodesAndWeights).

Algorithm 77: *LaplacianOnTheSquare:* Nodal Galerkin Approximation to the Laplace Operator

Procedure LaplacianOnTheSquare
Input: $\{U_{ij}\}_{i,j=0}^{N,M}$
Uses Algorithms:
Algorithm 19 (MxVDerivative)
$N \leftarrow this.spA.N$; $M \leftarrow this.spA.M$
for $j = 0$ **to** M **do**
  $\{Uxx_{ij}\}_{i=0}^{N} \leftarrow MxVDerivative\big(this.spA.\{D_{ij}^{(2),\xi}\}_{i,j=0}^{N}, \{U_{ij}\}_{i=0}^{N}\big)$
  for $i = 0$ **to** N **do**
    $Uxx_{ij} \leftarrow this.spA.w_j^{(\eta)} * Uxx_{ij}$
  end
end
for $i = 0$ **to** N **do**
  $\{Uyy_{ij}\}_{j=0}^{M} \leftarrow MxVDerivative\big(this.spA.\{D_{ij}^{(2),\eta}\}_{i,j=0}^{N}, \{U_{ij}\}_{j=0}^{M}\big)$
end
for $j = 0$ **to** M **do**
  for $i = 0$ **to** N **do**
    $\big(\nabla^2 U, \ell_i \ell_j\big)_N \leftarrow -Uxx_{ij} - this.spA.w_i^{(\xi)} * Uyy_{ij}$
  end
end
return $\{\big(\nabla^2 U, \ell_i \ell_j\big)_N\}_{i,j=0}^{N,M}$
End Procedure LaplacianOnTheSquare

- *Replace the algorithm to approximate the Laplacian.* We also need to replace Algorithm 66 (LaplacianOnTheSquare), to implement the nodal Galerkin approximation. We show the nodal Galerkin version in Algorithm 77 (NodalGalerkinLaplacian). Don't forget that those second derivative arrays now store the matrices G that we define in (5.74).
- *Modify the source term to compute the residual.* The residual for the nodal Galerkin approximation is

$$\begin{aligned} r_{ij} &= s_{ij} w_i^{(x)} w_j^{(y)} + \sum_{k=0}^{N} w_j^{(y)} G_{ik}^{(x)} \Phi_{k,j} + w_i^{(x)} G_{jk}^{(y)} \Phi_{i,k} \\ &\equiv s_{ij} w_i^{(x)} w_j^{(y)} - \big(\nabla^2 \Phi, \ell_i \ell_j\big)_N. \end{aligned} \tag{5.78}$$

We see, then, that we can reuse Algorithms 71 (Residual) and 68 (NodalPotentialClass:MatrixAction) if we store the quantity $s_{ij} w_i^{(x)} w_j^{(y)}$ in the space that we allotted for the source terms in Algorithm 64 (NodalPotentialClass). So to solve the nodal Galerkin approximation iteratively, we modify the driver Algorithm 76 (CollocationPotentialDriver) to store the modified source term.

Thus we see that although the derivation is quite different, the implementation of the nodal Galerkin approximation is virtually identical to the collocation approximation. From a programming point of view, then, there is no reason to prefer collocation over this approximation to solve the Poisson equation on the square.

5.2.2.2 Direct Solution of the Equations

Given the practical similarity of the nodal Galerkin method to the collocation approximation, it should be no surprise that we can solve the system directly with only simple modifications to Algorithms 69 (CollocationRHSComputation) and 70 (LaplaceCollocationMatrix). To compute the right hand side, we must account for the weight functions and replace the derivative matrices. Thus, we must replace (5.29) by

$$\begin{aligned}
&\sum_{k=1}^{N-1} w_j^{(y)} G_{ik}^{(x)} \Phi_{k,j} + \sum_{k=1}^{M-1} w_i^{(x)} G_{jk}^{(y)} \Phi_{i,k} \\
&\quad = w_i^{(x)} w_j^{(y)} s_{i,j} - w_j^{(y)} G_{i0}^{(x)} \Phi_{0,j} - w_j^{(y)} G_{iN}^{(x)} \Phi_{N,j} \\
&\qquad - w_i^{(x)} G_{j0}^{(y)} \Phi_{i,0} - w_i^{(x)} G_{jM}^{(y)} \Phi_{i,M} \\
&\quad \equiv RHS_{ij}, \quad i = 1, 2, \ldots, N-1;\ j = 1, 2, \ldots, M-1,
\end{aligned} \tag{5.79}$$

which we implement by modifying Algorithm 69 (CollocationRHSComputation).

Similarly, we replace the matrix elements in (5.32) by

$$\begin{aligned}
A_{n(i,j),m(k,j)} &= w_j^{(y)} G_{ik}^{(x)}, \quad k = 1, 2, \ldots, N-1;\ k \neq i, \\
A_{n(i,j),m(i,k)} &= w_i^{(x)} G_{jk}^{(y)}, \quad k = 1, 2, \ldots, M-1;\ k \neq j, \\
A_{n(i,j),m(i,j)} &= w_j^{(y)} G_{ii}^{(x)} + w_i^{(x)} G_{jj}^{(y)}.
\end{aligned} \tag{5.80}$$

With these equations, we modify Algorithm 70 (LaplaceCollocationMatrix) to represent the nodal Galerkin approximation.

Nevertheless, our tests of the direct solver for the collocation approximation lead us to expect that the direct solution of the system will not be competitive with an iterative solver except for small systems.

5.2.2.3 Iterative Solution of the Equations

The symmetry of the Galerkin approximation enables us to use the popular Conjugate Gradient method (Appendix D.2) to solve the system of equations (5.77) iteratively. As with the collocation approximation, it is usually necessary to precondition the system. We will therefore derive a preconditioner before we implement the solver.

5.2.2.4 A Finite Element Preconditioner

As is typical of spectral approximations, the system of equations represented by (5.77) needs to be preconditioned for an iterative technique to be most effective. For the Conjugate Gradient method, the preconditioner must be a symmetric approximation to the matrix. Because of the non-uniform grid generated by the Gauss-Lobatto points, the finite difference preconditioner that we developed for the collocation approximation is not symmetric, and hence we should not use it with the Conjugate Gradient method.

A finite element preconditioner, starting from the same Galerkin weak form that we used to derive the spectral approximation, can satisfy the symmetry requirements. To derive the finite element approximation, we approximate the solution by local bi-linear interpolants that we form on quadrilateral elements whose four corners are grid points, as we show in Fig. 5.3.

To derive the finite element approximation, it is convenient to map the rectangular element to the unit square by the (affine) transformation

$$\begin{aligned} x &= x_i + \Delta x_i \xi, \\ y &= y_i + \Delta y_j \eta, \end{aligned} \tag{5.81}$$

where $\Delta x = x_{i+1} - x_i$ and $\Delta y = y_{j+1} - y_j$. (A more general discussion of mappings and their effect on the approximations is the topic of the next chapter.) In

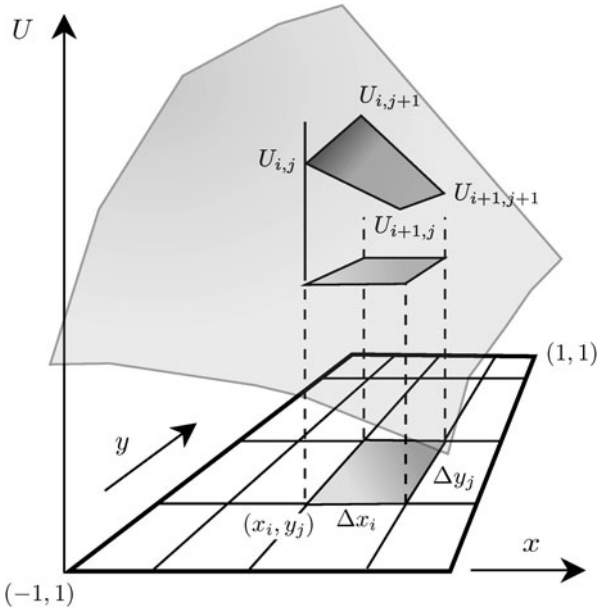

Fig. 5.3 Linear finite element approximation on a quadrilateral element created from four points on the grid

terms of the mapped coordinates, we approximate a function, U, by the bi-linear interpolant on the element

$$\begin{aligned} U(\xi,\eta) &= U_{i,j}\psi_{0,0} + U_{i+1,j}\psi_{1,0} + U_{i+1,j+1}\psi_{1,1} + U_{i,j+1}\psi_{0,1} \\ &= \sum_{k,l=0}^{1} U_{i+k,j+l}\psi_{k,l}, \end{aligned} \tag{5.82}$$

where $U_{i,j}$ is the value of the function at the point (i, j) and the basis functions are the bi-linear functions that vanish at all corners but one:

$$\begin{aligned} \psi_{0,0} &= (1-\xi)(1-\eta), \\ \psi_{1,0} &= \xi(1-\eta), \\ \psi_{1,1} &= \xi\eta, \\ \psi_{0,1} &= (1-\xi)\eta. \end{aligned} \tag{5.83}$$

We will write these four basis functions in a compact, although cryptic, form

$$\psi_{k,l}(\xi,\eta) = \big(1-k-(-1)^k\xi\big)\big(1-l-(-1)^l\eta\big). \tag{5.84}$$

We make the finite element preconditioner approximate the spectral operator by having it approximate the same equation, (5.61). Since we will soon make a change of variables to (ξ, η), let us rename the gradient operator in the original (x, y) variables to be ∇_x. Then the contribution from each element to the stiffness integral on the left is

$$\int_{y_j}^{y_{j+1}}\int_{x_i}^{x_{i+1}} \nabla_x U\cdot\nabla_x\psi\, dxdy = \Delta x_i\Delta y_j\int_0^1\int_0^1 \nabla_x U\cdot\nabla_x\psi\, d\xi d\eta, \tag{5.85}$$

or, when we substitute for U from (5.82),

$$\sum_{k,l=0}^{1} U_{i+k,j+l}\left\{\Delta x_i\Delta y_j\int_0^1\int_0^1 \nabla_x\psi_{k,l}\cdot\nabla_x\psi_{n,m}d\xi d\eta\right\}, \quad n,m=0,1. \tag{5.86}$$

We represent this sum as a *local stiffness matrix* multiplication of the grid point values of U

$$\begin{bmatrix} S_{00}^{00} & S_{10}^{00} & S_{01}^{00} & S_{11}^{00} \\ S_{00}^{10} & S_{10}^{10} & S_{01}^{10} & S_{11}^{10} \\ S_{00}^{01} & S_{10}^{01} & S_{01}^{01} & S_{11}^{01} \\ S_{00}^{11} & S_{10}^{11} & S_{01}^{11} & S_{11}^{11} \end{bmatrix} \begin{bmatrix} U_{i,j} \\ U_{i+1,j} \\ U_{i,j+1} \\ U_{i+1,j+1} \end{bmatrix}, \tag{5.87}$$

where

$$S_{kl}^{nm} = -\Delta x_i\Delta y_j\int_0^1\int_0^1 \nabla_x\psi_{k,l}\cdot\nabla_x\psi_{n,m}d\xi d\eta. \tag{5.88}$$

(Note that the matrix is symmetric by virtue of the product in the integrand.) To be consistent with standard matrix notation, we write the local stiffness matrix with entries $\hat{S}_{pq} = S_{kl}^{nm}$ with $p = n + 2m + 1$ and $q = k + 2l + 1$.

The gradients in the integrands transform easily under the affine transformation, specifically

$$\nabla_x \psi_{k,l} = \frac{\partial \psi_{k,l}}{\partial \xi}\frac{\partial \xi}{\partial x}\hat{x} + \frac{\partial \psi_{k,l}}{\partial \eta}\frac{\partial \eta}{\partial y}\hat{y} = \frac{1}{\Delta x}\frac{\partial \psi_{k,l}}{\partial \xi}\hat{x} + \frac{1}{\Delta y}\frac{\partial \psi_{k,l}}{\partial \eta}\hat{y}, \tag{5.89}$$

where

$$\begin{aligned} \frac{\partial \psi_{k,l}}{\partial \xi} &= -(-1)^k\big(1 - l - (-1)^l \eta\big), \\ \frac{\partial \psi_{k,l}}{\partial \eta} &= -(-1)^l\big(1 - k - (-1)^k \xi\big). \end{aligned} \tag{5.90}$$

We show how to compute the components of the gradient in the procedure *Psi_Xi* and *Psi_Eta* in Algorithm 78 (ApproximateFEMStencil).

Rather than evaluate the integrals in (5.88) exactly, Canuto et al. [7] report that a linear approximation to the integrands yields a better preconditioner for the spectral Galerkin approximation. Therefore, we will approximate the integrands

$$g_{kl}^{nm}(\xi, \eta) = \nabla_x \psi_{k,l} \cdot \nabla_x \psi_{n,m} = \frac{1}{\Delta x^2}\frac{\partial \psi_{k,l}}{\partial \xi}\frac{\partial \psi_{n,m}}{\partial \xi} + \frac{1}{\Delta y^2}\frac{\partial \psi_{k,l}}{\partial \eta}\frac{\partial \psi_{n,m}}{\partial \eta} \tag{5.91}$$

with a bilinear function

$$g(\xi, \eta) = (1 - \xi)(1 - \eta)g(0, 0) + \xi(1 - \eta)g(1, 0) + \xi\eta g(1, 1) + (1 - \xi)\eta g(0, 1). \tag{5.92}$$

We easily evaluate the integral of the bilinear approximation

$$\int_0^1 \int_0^1 g_{kl}^{nm}(\xi, \eta) d\xi d\eta = \frac{1}{4}\left(g_{kl}^{nm}(0, 0) + g_{kl}^{nm}(1, 0) + g_{kl}^{nm}(1, 1) + g_{kl}^{nm}(0, 1)\right) \tag{5.93}$$

and find that it is simply the average of the values at the four corners. Thus, the local stiffness matrix entries for the approximate finite element preconditioner are

$$S_{kl}^{nm} = \frac{\Delta x_i \Delta y_j}{4}\left(g_{kl}^{nm}(0, 0) + g_{kl}^{nm}(1, 0) + g_{kl}^{nm}(1, 1) + g_{kl}^{nm}(0, 1)\right). \tag{5.94}$$

To compute the local stiffness matrix we use the procedure *LocalStiffnessMatrix* in Algorithm 78 (ApproximateFEMStencil).

We now have to sum the contributions from each element to compute the global stiffness matrix. Stiffness summation is a procedure that is standard in finite element texts, and is applicable to unstructured and structured grids. However, since the grid here is regular, the approximation is local, and since we have already derived a useful finite difference solver, we will compute the stiffness summation explicitly to derive a local stencil, much like the finite difference stencil in (5.37).

Algorithm 78: *ApproximateFEMStencil:* Computing the Approximate Finite Element Stencil on the Square

Procedure *StencilCoefficients*
Input: $i, j, \{x_k\}_{k=0}^{N}, \{y_k\}_{k=0}^{M}$
for $m = 0$ **to** 1 **do**
  for $n = 0$ **to** 1 **do**
    $p \leftarrow n + 2m + 1$
    $\left\{{}^{p}\hat{S}_{kl}\right\}_{k,l=1}^{4} \leftarrow LocalStiffnessMatrix(x_{i-n+1} - x_{i-n}, y_{j-n+1} - y_{j-n})$
  end
end
for $m = 0$ **to** 1 **do**
  for $n = 0$ **to** 1 **do**
    $p \leftarrow n + 2m + 1$
    for $k = -n$ **to** $-n + 1$ **do**
      for $l = -m$ **to** $-m + 1$ **do**
        $C_{kl} \leftarrow C_{kl} + {}^{p}\hat{S}_{pq}$
      end
    end
  end
end
return $\{C_{kl}\}_{k,l=-1}^{1}$
End Procedure *StencilCoefficients*

Procedure *LocalStiffnessMatrix*
Input: $\Delta x, \Delta y$
for $m = 0$ **to** 1 **do**
  for $n = 0$ **to** 1 **do**
    $q \leftarrow n + 2 * m + 1$
    for $l = 0$ **to** 1 **do**
      for $k = 0$ **to** 1 **do**
        $p \leftarrow k + 2 * l + 1$
        $t \leftarrow 0$
        for $s = 0$ **to** 1 **do**
          for $r = 0$ **to** 1 **do**
            $t \leftarrow t + Psi\_Xi(k, l, s) * Psi\_Xi(n, m, s)/\Delta x^2 + Psi\_Eta(k, l, r) * Psi\_Eta(n, m, r)/\Delta y^2$ // R1
          end
        end
        $\hat{S}_{p,q} \leftarrow \Delta x \Delta y * t/4$
      end
    end
  end
end
return $\{S_{kl}\}_{k,l=1}^{4}$
End Procedure *LocalStiffnessMatrix*

Procedure *Psi_Xi*
Input: k, l, η
$\frac{\partial \psi_{kl}}{\partial \xi} \leftarrow -(-1)^k \left(1 - l - (-1)^l \eta\right)$
return $\partial \psi / \partial \xi_{kl}(\eta)$
End Procedure *Psi_Xi*

Procedure *Psi_Eta*
Input: k, l, ξ
$\frac{\partial \psi_{kl}}{\partial \eta} \leftarrow -(-1)^l \left(1 - k - (-1)^k \xi\right)$
return $\partial \psi / \partial \xi_{kl}(\eta)$
End Procedure *Psi_Eta*

The contribution to a point (i, j) comes from the four finite elements around it, $E^{i,j}$, $E^{i-1,j}$, $E^{i-1,j-1}$, $E^{i,j-1}$ and corresponds to the basis function in each element that is non-zero at the point (i, j). To distinguish between the local stiffness matrices between the four elements, we'll add a superscript to them. The contributions to the stencil are

$$\begin{aligned}
&\sum_{l=0}^{1}\sum_{k=0}^{1} {}^{(i,j)}S_{kl}^{00}U_{i+k,j+l} + \sum_{l=0}^{1}\sum_{k=-1}^{0} {}^{(i-1,j)}S_{k+1,l}^{10}U_{i+k,j+l} \\
&\quad + \sum_{l=-1}^{0}\sum_{k=-1}^{0} {}^{(i-1,j-1)}S_{k+1,l+1}^{11}U_{i+k,j+l} + \sum_{l=-1}^{0}\sum_{k=0}^{1} {}^{(i,j-1)}S_{k,l+1}^{01}U_{i+k,j+l} \\
&= \sum_{l=-1}^{1}\sum_{k=-1}^{1} C_{kl}^{(i,j)}U_{i+k,j+l}. \qquad (5.95)
\end{aligned}$$

We collapse the four sums on the left of (5.95) to get

$$\sum_{m=0}^{1}\sum_{n=0}^{1}\sum_{l=-m}^{-m+1}\sum_{k=-n}^{-n+1} {}^{(i-n,j-m)}S_{k+n,l+m}^{nm}U_{i+k,j+l} = \sum_{l=-1}^{1}\sum_{k=-1}^{1} C_{kl}^{(i,j)}U_{i+k,j+l}, \qquad (5.96)$$

which defines the stencil coefficients $C_{kl}^{(i,j)}$ for the grid point (i, j). For the full linear finite element approximation, this represents a nine-point stencil. The approximate finite element method is only a five point stencil, since the coefficients turn out to be zero at the four corner points.

We compute the stencil coefficients from the four matrices $\hat{S}_{pq}$ that correspond to the four values of n and m. We will store those as a four dimensional array of 4×4 matrices and denote these arrays by ${}^{p}\hat{S}_{pq}$ where, as before, $p = n + 2m + 1$, but because the subscripts have changed, $q = (k + n) + 2(l + m) + 1$. The procedure *StencilCoefficients* in Algorithm 78 (ApproximateFEMStencil) implements the construction of the stencil coefficients.

Finally, we discuss how to solve the preconditioned system. As we describe in Appendix D.2, the preconditioned Conjugate Gradient method performs a system solve of the form $Hz = r$ during each iteration. As always, there are tradeoffs to consider between computational time and storage costs. The solution of the system, which is pentadiagonal, is standard in finite element texts. In roughly reverse order of the amount of extra storage required, a direct solution of the system without taking into account the sparsity of the matrix requires a large amount of memory and is computationally very expensive. Banded direct solvers are possible. We can use approximations to these solvers, like the ILU solver that we implemented in Sect. 5.2.1.4 in the context of the finite difference preconditioner. Finally, we can use an iterative solver for the preconditioner (with it's own preconditioner, for the Cat in the Hat fans.)

We can equally apply the ILU solution of the preconditioned system described in Algorithm 74 (Solve) to the approximate finite element preconditioner on the grid. We need only make the correspondence between the coefficients A–F in the finite difference approximation and the coefficients C in the finite element approximation, $A = C_{00}$, etc., and make the appropriate changes.

An alternative to the ILU solver for the preconditioner's pentadiagonal system that can turn out to be almost as fast, yet requires less coding, is to use a single sweep of the symmetric successive overrelaxation method (SSOR) as an approximation to the solution of the preconditioner. Of course, we could use the SSOR to solve the system exactly by iterating to convergence. This would minimize the number of iterations of the (outer) Conjugate Gradient solver, at the price of substantial increased CPU time per iteration. Using a single sweep of the SSOR will increase the number of iterations in the outer solver, but the single sweep is fast. If we do not iterate the preconditioner to convergence, we must use SSOR rather than the SOR since the preconditioner must be symmetric overall. We must also carefully choose a value for the acceleration parameter in the interval $[1, 2)$ that gives fastest convergence—a definite disadvantage of using SSOR to solve the preconditioner. On the plus side, the iterative solver is widely applicable, can be used for preconditioners that have other than pentadiagonal matrices, and is easy to code. The SSOR iteration consists of two loops, one running forward through the grid, and the other backwards. For reference, we show the SSOR sweep in Algorithm 79 (SSORSweep). We can either compute the stencil coefficients C on the fly or once during the construction of the iterative solver and save them. Finally, we note that we could just as well have used the SSOR solver with the finite difference preconditioner for the collocation equations instead of the ILU solver.

5.2.2.5 Construction of the PCG Solver

Finally, we implement the Conjugate Gradient solver of Appendix D.2 using the BLAS-1 algorithms of Appendix C to solve the nodal Galerkin approximation. We show this modification in Algorithm 80 (ConjugateGradientSolve). The algorithm takes an instance of a NodalPotentialClass, Algorithm 64, constructed as we have described above to represent a nodal Galerkin approximation. The solver also takes an instance of a preconditioner, which we model after the finite difference preconditioner of Algorithms 72 (FDPreconditioner)–74 (Solve). We represent the solver for the preconditioner as the ILU preconditioner of Algorithm 74 (Solve), which we must modify to compute the approximate finite element method shown in Algorithm 78 (ApproximateFEMStencil). We could swap the preconditioner's solver with the SSOR solver of Algorithm 79 (SSORSweep). We make a tradeoff between storage and computation cost by deciding to store or not to store the stencil coefficients when we modify Algorithm 72 (FDPreconditioner). Finally, all our comments in the discussion of the BiCGStab algorithm about boundary conditions apply equally to Algorithm 80 (ConjugateGradientSolve).

Algorithm 79: *SSORSweep:* SSOR Sweep for the Finite Element Preconditioner

Procedure *SSORSweep*
Input: $\{r_{i,j}\}_{i,j=0}^{N,M}$, ω
for $j = 0$ **to** M **do**
  for $i = 0$ **to** N **do**
    $z_{i,j} \leftarrow 0$
  end
end
for $j = 1$ **to** $M - 1$ **do**
  for $i = 1$ **to** $N - 1$ **do**
    $s = 0$
    for $k = -1$ **to** 1 **do**
      for $l = -1$ **to** 1 **do**
        $s \leftarrow s + C_{kl}^{ij} * z_{i+k,j+l}$
      end
    end
    $z_{i,j} \leftarrow z_{i,j} + \omega * (r_{i,j} - s)/C_{00}^{ij}$
  end
end
for $j = M - 1$ **to** 1 **step** -1 **do**
  for $i = N - 1$ **to** 1 **step** -1 **do**
    $s \leftarrow 0$
    for $k = -1$ **to** 1 **do**
      for $l = -1$ **to** 1 **do**
        $s \leftarrow s + C_{kl}^{ij} * z_{i+k,j+l}$
      end
    end
    $z_{i,j} \leftarrow z_{i,j} + \omega * (r_{i,j} - s)/C_{00}^{ij}$
  end
end
return $\{z_{i,j}\}_{i,j=0}^{N,M}$
End Procedure *SSORSweep*

5.2.2.6 Benchmark Solution

We reconsider the boundary value problem (5.56) to benchmark the performance of the nodal Galerkin approximation with a preconditioned Conjugate Gradient solver. As before, we are interested in the accuracy of the approximation and the effectiveness of the solver and preconditioners. The issues with the direct solver have not changed, except that we could reduce the storage by half by using a Cholesky factorization of the matrix. Therefore, we will only discuss the performance of the iterative solver.

We show the maximum errors of the nodal Galerkin solution as a function of $N = M$ in Table 5.5. These errors correspond to those of the collocation approximation shown in Table 5.1. Again, we see that the error decays exponentially fast. (The differences between successive entries for the logarithm of the error are ap-

Algorithm 80: *PreconditionedConjugateGradientSolve:* Conjugate Gradient Iterative Solver for Nodal Spectral Methods

Procedure PreconditionedConjugateGradientSolve
Input: N_{it}, TOL
Input:
npc `// NodalPotentialClass instance`
H `// AFEMPreconditioner instance`
Uses Algorithms:
Algorithm 77 (NodalGalerkinLaplacian)
Algorithm 74 (Solve- modified for AFEM)
Algorithm 140 (BLAS_Level1)

$N \leftarrow npc.spA.N$; $M \leftarrow npc.spA.M$; $L \leftarrow (N+1) \times (M+1)$
$\{r_{i,j}\}_{i,j=0}^{N,M} \leftarrow Residual(npc)$
$\{z_{i,j}\}_{i,j=0}^{N,M} \leftarrow H.Solve(\{r_{i,j}\}_{i,j=0}^{N,M})$
$\{v_{i,j}\}_{i,j=0}^{N,M} \leftarrow BLAS\_COPY(L, \{z_{i,j}\}_{i,j=0}^{N,M}, 1, \{v_{i,j}\}_{i,j=0}^{N,M}, 1)$
$c \leftarrow BLAS\_DOT(L, \{r_{i,j}\}_{i,j=0}^{N,M}, 1, \{z_{i,j}\}_{i,j=0}^{N,M}, 1)$
for $k = 1,\ N_{it}$ **do**
  $\{z_{i,j}\}_{i,j=0}^{N,M} \leftarrow npc.MatrixAction(\{v_{i,j}\}_{i,j=0}^{N,M})$
  $\omega \leftarrow c/BLAS\_DOT(L, \{v_{i,j}\}_{i,j=0}^{N,M}, 1, \{z_{i,j}\}_{i,j=0}^{N,M}, 1)$
  $npc.\{\Phi_{i,j}\}_{i,j=0}^{N,M} \leftarrow BLAS\_AXPY(L, \omega, \{v_{i,j}\}_{i,j=0}^{N,M}, 1, npc.\{\Phi_{i,j}\}_{i,j=0}^{N,M}, 1)$
  $\{r_{i,j}\}_{i,j=0}^{N,M} \leftarrow BLAS\_AXPY(L, -\omega, \{z_{i,j}\}_{i,j=0}^{N,M}, 1, \{r_{i,j}\}_{i,j=0}^{N,M}, 1)$
  if $BLAS\_NRM2(L, \{r_{i,j}\}_{i,j=0}^{N,M}, 1) \leq TOL$ **then** Exit
  $\{z_{i,j}\}_{i,j=0}^{N,M} \leftarrow H.Solve(\{r_{i,j}\}_{i,j=0}^{N,M})$
  $d \leftarrow BLAS\_DOT(L, \{r_{i,j}\}_{i,j=0}^{N,M}, 1, \{z_{i,j}\}_{i,j=0}^{N,M}, 1)$
  $\{v_{i,j}\}_{i,j=0}^{N,M} \leftarrow BLAS\_SCAL(L, d/c, \{v_{i,j}\}_{i,j=0}^{N,M}, 1,)$
  $\{v_{i,j}\}_{i,j=0}^{N,M} \leftarrow BLAS\_AXPY(L, 1.0, \{z_{i,j}\}_{i,j=0}^{N,M}, 1, \{v_{i,j}\}_{i,j=0}^{N,M}, 1)$
  $c \leftarrow d$
end
return npc
End Procedure PreconditionedConjugateGradientSolve

Table 5.5 Logarithm of maximum errors for nodal Galerkin method to (5.56)

N	8	12	16	20	24
$\text{Log}_{10}(Error)$	−1.87	−4.16	−7.14	−10.45	−14.0

proximately equal.) Doubling the number of points decreases the error by about a factor of one thousand. We see also that the nodal Galerkin approximation has the same errors as the Legendre collocation approximation, and hence is still slightly better than the Chebyshev method.

We next examine the performance of the iterative solver. Table 5.6 shows the number of iterations and the CPU time for the ILU, SSOR solver, and without preconditioning. For the SSOR solver, we experimented with the parameter $\omega \in [1, 2)$

Table 5.6 Performance comparison for preconditioned conjugate gradient solution of the nodal Galerkin approximation

N	ILU		SSOR		None	
	Iterations	Time	Iterations	Time	Iterations	Time
16	30	0.001	34	0.001	57	0.001
24	36	0.002	39	0.003	107	0.005
48	48	0.017	49	0.020	303	0.084
64	55	0.041	59	0.048	465	0.280
72	58	0.057	63	0.068		

to get the best performance. One thing we notice is that the Conjugate Gradient method converges without preconditioning much better than does the BiCGStab method applied to the collocation approximation. The second is that for small systems with $N \leq 16$ it is possible not to use preconditioning at all. However, for larger values of N up to $N = 72$ the preconditioned iteration takes up to a factor of seven less time than the Conjugate Gradient method alone. We see little difference between the time to converge using the ILU and SSOR solvers for the preconditioner.

Finally, it is worth comparing the efficiency of the nodal Galerkin method with collocation. As we have seen, the errors for this test problem are comparable to the Legendre collocation method. The number of iterations is larger with the Conjugate Gradient solver, but for larger N, the nodal Galerkin method takes less time.

5.3 Approximation of Time Dependent Advection-Diffusion

The transport of the concentration of a substance such as a pollutant in a river by a velocity field $\mathbf{q} = u\hat{x} + v\hat{y}$ is described by an advection-diffusion equation

$$\begin{aligned} \frac{\partial \varphi}{\partial t} + \mathbf{q} \cdot \nabla \varphi &= \nu \nabla^2 \varphi, \quad (x, y) \in (-1, 1) \times (-1, 1), \\ \varphi(x, y, 0) &= \varphi_0(x, y), \quad (x, y) \in [-1, 1] \times [-1, 1]. \end{aligned} \tag{5.97}$$

The addition of appropriate boundary conditions on all four sides of the square domain completes the description of the problem.

In this section we derive both the collocation and nodal Galerkin approximations to the advection-diffusion equation. Each is straightforward to apply on the square. In both cases, our derivations will re-use and build on the work that we completed in Sects. 5.2.1 and 5.2.2, where we developed the approximations to the diffusion operator.

5.3.1 The Collocation Approximation

We first derive the collocation approximation. Since we have already approximated the diffusion term in Sect. 5.2.1, we concentrate on the transport term, $\mathbf{q} \cdot \nabla \varphi$, for

constant velocity $\mathbf{q}$. We approximate that by differentiating the interpolant of the approximate solution

$$\mathbf{q}\cdot\nabla\varphi\approx\mathbf{q}\cdot\nabla\Phi=u\Phi_x+v\Phi_y=u\sum_{n,m}^{N}\Phi_{n,m}\ell_n'(x)\ell_m(y)+v\sum_{n,m}^{N}\Phi_{n,m}\ell_n(x)\ell_m'(y). \tag{5.98}$$

We then evaluate the approximation at the collocation points, where it simplifies to

$$\begin{aligned}\mathbf{q}\cdot\nabla\Phi_{i,j}&=u\sum_{n}^{N}\Phi_{n,m}\ell_n'(x_i)+v\sum_{m}^{N}\Phi_{n,m}\ell_m'(y_j)\\&=u\sum_{n}^{N}D_{in}^{(x)}\Phi_{n,j}+v\sum_{m}^{N}D_{jm}^{(y)}\Phi_{i,m}.\end{aligned} \tag{5.99}$$

Therefore, the collocation approximation to the advection-diffusion equation for Dirichlet boundary conditions is

$$\frac{d\Phi_{i,j}}{dt}=\nu\nabla_N^2\Phi_{i,j}-\mathbf{q}\cdot\nabla_N\Phi_{i,j},\quad i,j=1,2,\ldots,N, \tag{5.100}$$

where

$$\mathbf{q}\cdot\nabla_N\Phi_{i,j}=u\sum_{n=0}^{N}D_{in}^{(x)}\Phi_{n,j}+v\sum_{m=0}^{N}D_{jm}^{(y)}\Phi_{i,m} \tag{5.101}$$

and

$$\nabla_N^2\Phi_{i,j}=\sum_{k=0}^{N}D_{i,k}^{(2),x}\Phi_{k,j}+\sum_{k=0}^{N}D_{j,k}^{(2),y}\Phi_{i,k}, \tag{5.102}$$

which we repeat from (5.22)–(5.23).

5.3.2 The Nodal Galerkin Approximation

We follow the now familiar steps to derive the Galerkin approximation of the transport term. We write a weak form of the advection term

$$(\mathbf{q}\cdot\nabla\varphi,\phi)=\int_{-1}^{1}\int_{-1}^{1}\mathbf{q}\cdot\nabla\varphi\phi dxdy, \tag{5.103}$$

replace the solution φ by its approximation Φ, and replace ϕ by an arbitrary polynomial of the same degree as Φ that vanishes on the boundaries. We then use the

fact that the nodal values of ϕ are arbitrary and linearly dependent to get the approximation of the advection term

$$\left(\mathbf{q}\cdot\nabla\Phi,\ell_i\ell_j\right)=\int_{-1}^{1}\int_{-1}^{1}\mathbf{q}\cdot\nabla\Phi\ell_i(x)\ell_j(y)dxdy. \tag{5.104}$$

When we substitute the Lagrange representation of the approximate solution, (5.98), and rearrange, we get

$$\begin{aligned}\left(\mathbf{q}\cdot\nabla\Phi,\ell_i\ell_j\right)=u\sum_{n,m=0}^{N}\Phi_{n,m}\left[\int_{-1}^{1}\int_{-1}^{1}\ell_n'(x)\ell_m(y)\ell_i(x)\ell_j(y)dxdy\right]\\ +v\sum_{n,m=0}^{N}\Phi_{n,m}\left[\int_{-1}^{1}\int_{-1}^{1}\ell_n(x)\ell_m'(y)\ell_i(x)\ell_j(y)dxdy\right].\end{aligned} \tag{5.105}$$

Finally, we replace the integrals in (5.105) by Legendre Gauss-Lobatto quadratures. As usual, the equations simplify since

$$\int_{-1}^{1}\ell_n(x)\ell_i(x)dx\approx\sum_{k=0}^{N}\ell_n(x_k)\ell_i(x_k)w_k^{(x)}=\delta_{n,i}w_i^{(x)} \tag{5.106}$$

and so forth. With these substitutions, we have our approximation of the transport term

$$\int_{-1}^{1}\int_{-1}^{1}\mathbf{q}\cdot\nabla\varphi\phi dxdy\rightarrow w_i^{(x)}w_j^{(y)}\left[u\sum_{n=0}^{N}D_{in}^{(x)}\Phi_{n,j}+v\sum_{m=0}^{N}D_{jm}^{(y)}\Phi_{i,m}\right]. \tag{5.107}$$

We see, therefore, that for the advection-diffusion problem with constant coefficients, the nodal Galerkin approximation of the transport term with the integrals replaced by Gauss-Lobatto quadrature is just the collocation approximation multiplied by the quadrature weights.

To derive the nodal Galerkin approximation of the advection-diffusion equation, we must include the time derivative term. When we replace the integrals by Gauss-Lobatto quadratures, we reduce the time derivative term simply to

$$\int_{-1}^{1}\int_{-1}^{1}\frac{d\varphi}{dt}\phi dxdy\rightarrow w_i^{(x)}w_j^{(y)}\frac{d\Phi_{i,j}}{dt}. \tag{5.108}$$

Therefore, we write the nodal Galerkin approximation as

$$w_i^{(x)}w_j^{(y)}\frac{d\Phi_{i,j}}{dt}=\nu\left(\nabla^2\Phi,\ell_i\ell_j\right)_N-\left(\mathbf{q}\cdot\nabla\Phi,\ell_i\ell_j\right)_N, \tag{5.109}$$

where

$$\left(\mathbf{q}\cdot\nabla\Phi,\ell_i\ell_j\right)_N=w_i^{(x)}w_j^{(y)}\left[u\sum_{n}^{N}D_{in}^{(x)}\Phi_{n,m}+v\sum_{m}^{N}D_{jm}^{(y)}\Phi_{n,m}\right], \tag{5.110}$$

and we repeat the diffusion term (5.77)

$$\left(\nabla^2\Phi, \ell_i\ell_j\right)_N = \sum_{k=0}^{N} w_j^{(y)} G_{ik}^{(x)} \Phi_{k,j} + w_i^{(x)} G_{jk}^{(y)} \Phi_{i,k}. \tag{5.111}$$

Note that we could trivially divide both sides of the Galerkin approximation by the product of the quadrature weights to get an equation that is identical to the Legendre collocation approximation. If we want to integrate the equation explicitly, that is exactly what we would do. If we want to integrate the equation implicitly, however, we will get a symmetric linear system that we can solve with the Conjugate Gradient method if we don't divide by the weights.

To complete the spatial approximation, we need to implement boundary conditions. Dirichlet conditions, which prescribe the concentration, Φ, along the boundaries, are appropriate for the advection-diffusion problem. We could apply Neumann conditions instead to specify the flux. We showed how to set both types of boundary conditions in Sects. 4.4 and 4.6 for one dimensional problems. We showed how to extend the ideas presented there to two dimensions in Sects. 5.2.1 and 5.2.2.

5.3.3 Time Integration

Let us now address the time integration of the approximations (5.100) and (5.109). From Chap. 4 we know that the approximation of the diffusion terms is much more stiff than the transport terms. Specifically, we have seen in Sects. 4.4.1 and 4.4.4 that the eigenvalues of the spatial approximation of the diffusion terms grow as $O(N^4)$ compared to the $O(N^2)$ for the advection terms. Unless diffusion is small compared to advection, it is usually necessary to integrate the equations implicitly. Often, only the diffusion terms are integrated implicitly; the advection terms are integrated explicitly. The result is a semi-implicit method.

In this section, we will show how to use a semi-implicit method to integrate the approximations to the advection-diffusion equation in time. Semi-implicit methods are commonly used by the incompressible flow community to integrate approximations of the Navier Stokes equations. They are particularly useful in problems where diffusion dominates advection. Specifically, we'll implement a *linear multistep method* that uses an implicit third order backward differentiation (BDF) method for the diffusion terms and an explicit third order extrapolation method for the advection. The third order BDF method has an absolute stability region that includes the entire negative real axis, which makes it unconditionally stable for the diffusion terms, whose eigenvalues are real ([7], Sect. 7.3.1). The extrapolation method has been derived so that it uses information from the same previous time steps as the BDF approximation. Overall, the time step is limited by the less stiff advection terms, rather than the more stiff diffusion terms.

The third order semi-implicit time integration approximation applied to the collocation approximation (5.100) in space is

$$\begin{aligned}
&\Phi_{i,j}^{n+1} - \frac{6\nu\Delta t}{11}\nabla_N^2\Phi_{i,j}^{n+1} \\
&\quad = \frac{1}{11}\left(18\Phi_{i,j}^{n} - 9\Phi_{i,j}^{n-1} + 2\Phi_{i,j}^{n-2}\right) \\
&\qquad - \frac{6\Delta t}{11}\left(3\mathbf{q}\cdot\nabla_N\Phi_{i,j}^{n} - 3\mathbf{q}\cdot\nabla_N\Phi_{i,j}^{n-1} + \mathbf{q}\cdot\nabla_N\Phi_{i,j}^{n-2}\right).
\end{aligned} \tag{5.112}$$

When we apply it to the nodal Galerkin approximation it is

$$\begin{aligned}
&w_i^{(x)}w_j^{(y)}\Phi_{i,j}^{n+1} - \frac{6\nu\Delta t}{11}\left(\nabla^2\Phi_{i,j}^{n+1}, \ell_i\ell_j\right)_N \\
&\quad = \frac{1}{11}\left(18\Phi_{i,j}^{n} - 9\Phi_{i,j}^{n-1} + 2\Phi_{i,j}^{n-2}\right) \\
&\qquad - \frac{6\Delta t}{11}\left(3\left(\mathbf{q}\cdot\nabla\Phi_{i,j}^{n}, \ell_i\ell_j\right)_N - 3\left(\mathbf{q}\cdot\nabla\Phi_{i,j}^{n-1}, \ell_i\ell_j\right)_N\right. \\
&\qquad \left. + \left(\mathbf{q}\cdot\nabla\Phi_{i,j}^{n-2}, \ell_i\ell_j\right)_N\right).
\end{aligned} \tag{5.113}$$

In both cases we must solve a linear system of equations that arises from the terms on the left of the equals sign at every time step. Furthermore, we see that we need to store the solution and the transport terms at three time values, t_n, t_{n-1} and t_{n-2}.

Iterative solvers like those we implemented in the previous section are now particularly attractive to solve the systems represented by the left sides of (5.112) and (5.113) when compared to direct solvers. First, we have the previous time value to serve as a good initial iterate at each time step, so the number of iterations per solve is significantly reduced. Also, since there is now time discretization error at every step, we only have to iterate until the iteration error is smaller than the time integration error. We do not have to iterate the residual to machine zero, so that reduces the number of iterations we need per time step even more.

To use the BiCGStab iterative scheme that we implemented in Algorithm 75 (BiCGSSTABSolve), we rewrite the collocation approximation as

$$\left(I - \frac{6\nu\Delta t}{11}\nabla_N^2\right)\Phi_{i,j} = RHS_{i,j}, \tag{5.114}$$

where

$$\begin{aligned}
RHS_{i,j} &= \frac{1}{11}\left(18\Phi_{i,j}^{n} - 9\Phi_{i,j}^{n-1} + 2\Phi_{i,j}^{n-2}\right) \\
&\quad - \frac{6\Delta t}{11}\left(3\mathbf{q}\cdot\nabla_N\Phi_{i,j}^{n} - 3\mathbf{q}\cdot\nabla_N\Phi_{i,j}^{n-1} + \mathbf{q}\cdot\nabla_N\Phi_{i,j}^{n-2}\right).
\end{aligned} \tag{5.115}$$

We then drive a norm of the residual

$$r_{i,j} = RHS_{i,j} - \Phi_{i,j} + \frac{6\nu\Delta t}{11}\nabla_N^2\Phi_{i,j} \tag{5.116}$$

to be less than some tolerance.

We get a similar system to solve when we apply the time discretization to the nodal Galerkin approximation (5.113). The residual that we must make small at each time step is now

$$r_{i,j} = \left(RHS_{i,j} - w_i^{(x)}w_j^{(y)}\Phi_{i,j}\right) + \frac{6\nu\Delta t}{11}\left(\nabla^2\Phi, \ell_i\ell_j\right)_N, \tag{5.117}$$

where

$$\begin{aligned} RHS_{i,j} = {} & \frac{w_i^{(x)}w_j^{(y)}}{11}\left(18\Phi_{i,j}^n - 9\Phi_{i,j}^{n-1} + 2\Phi_{i,j}^{n-2}\right) \\ & - \frac{6\Delta t}{11}\left\{3\left(\mathbf{q}\cdot\nabla_N\Phi^n, \ell_i\ell_j\right)_N - 3\left(\mathbf{q}\cdot\nabla_N\Phi^{n-1}, \ell_i\ell_j\right)_N \right. \\ & \left. + \left(\mathbf{q}\cdot\nabla_N\Phi^{n-2}, \ell_i\ell_j\right)_N\right\}. \end{aligned} \tag{5.118}$$

The multistep method does have the disadvantage that it requires two steps beyond the initial condition to be computed before it can be used. Although we could use one of several approaches to create these two values, the simplest is to integrate the first two time steps with an explicit Runge-Kutta, such as the (matching) third order approximation that we have already used in Algorithm 50 (*CollocationStepByRK3*). Once we complete those steps and store the results, we switch over to the multistep integration.

5.3.4 How to Implement the Approximations

To implement the polynomial collocation and nodal Galerkin approximations, (5.112) and (5.113), we need storage for three time levels of the solution and the transport terms. We also need procedures to construct the linear systems to be solved at each time step, which are represented by (5.114) and its equivalent for the nodal Galerkin method.

5.3.4.1 Multilevel Time Storage

We need to store the solution and transport terms at the current and two previous time levels to compute the right hand sides, (5.115) or (5.118). It appears that three time levels of the transport and four of the solution need to be saved. However, once the quantities in (5.115) or (5.118) have been computed, the solution and transport

term $\mathbf{q} \cdot \nabla \Phi$ at time level $n-2$ are no longer needed. We reuse that storage space for the solution at the new time level.

We use indirect addressing to minimize both storage and the need to copy arrays from time step to time step. One way to implement indirect addressing is to create pointers to the arrays. Those pointers are swapped to point to the desired array at each time step. Another way is to create an array of integers that store an array index for a larger storage array. Since the former is trivial to implement, we will discuss how to do the latter.

To illustrate the use of an index array, suppose that we store the potential in a three-dimensional array $\{\Phi^k_{i,j}\}^{N,N;0}_{i,j=0;k=-2}$, which stores the solution values at $\Phi^{n+k}_{i,j}$. We create an integer pointer array to express that organization of the storage: $\{p_k\}^0_{k=-2}$ with $p_{-2} = -2$, $p_{-1} = -1$, and $p_0 = 0$. We would then access the solution array for time level $n+k$ indirectly by $\Phi^{(p_k)}_{i,j}$. At the end of a time step, what was time level $n-1$ becomes $n-2$ and what was time level n becomes $n-1$. We store the current level n where the no longer needed $n-2$ values were stored by setting

$$\begin{aligned} tmp &\leftarrow p_{-2}, \\ p_{-2} &\leftarrow p_{-1}, \\ p_{-1} &\leftarrow p_0, \\ p_0 &\leftarrow tmp. \end{aligned} \tag{5.119}$$

5.3.4.2 The Advection-Diffusion Class

To address the needs that we have just outlined, we expand Algorithm 64 (NodalPotentialClass) to organize the storage and procedures to integrate the advection-diffusion equation on the square. We present that new class in Algorithm 81 (NodalAdvDiffClass). We have added storage for the solution and transport terms at the three time steps, the indirect addressing pointer, and, of course, the physical parameters that describe the problem. New procedures in the class compute the transport terms, the right hand side and the residual. We reuse Algorithms 66 (LaplacianOnTheSquare) and 77 (LaplacianOnTheSquare) to compute the diffusion term, depending on which approximation we choose.

As with the potential approximation, we specify the choice of polynomial in the constructor. Algorithm 65 (NodalAdvDiffClass:Construct), for instance, shows a constructor for the Chebyshev collocation approximation. It computes the second derivative matrices by way of Algorithm 38 (mthOrderPolynomialDerivativeMatrix) with $m = 2$ and stores them in the second derivative matrix storage of the Nodal2DStorage structure. We now need the first derivative matrices and compute them using Algorithm 37 (PolynomialDerivativeMatrix). We easily change the approximation to a Legendre method if we replace the calls to ChebyshevGaussLobattoNodesAndWeights with calls to Algorithm 25 (LegendreGaussLobattoNodesAndWeights). To change to the nodal Galerkin approximation, we again note that

Algorithm 81: *NodalAdvDiffClass:* A Class for the Advection-Diffusion Problem on the Square

Class NodalAdvDiffClass
Uses Algorithms:
Algorithm 63 (Nodal2DStorage)
Data:
u, v, ν ; // advection speeds and diffusion coefficient
spA ; // Of type Nodal2DStorage
$\left\{\Phi_{i,j}^{k}\right\}_{i,j=1;k=-2}^{N,M;0}$; // Solution at three time steps
$\left\{transport_{i,j}^{k}\right\}_{i,j=0;k=-2}^{N,M;0}$; // Advection terms at three time steps
$\left\{RHS_{i,j}\right\}_{i,j=0}^{N,M}$; // Right hand side for implicit solve
$\{mask_i\}_{i=1}^{4}$; // Boundary condition mask
$\{p_k\}_{k=-2}^{0}$; // Time step pointer
Procedures:
$Construct(N, M, u, v, \nu)$; // Algorithm 82
$LaplacianOnTheSquare(\{U_{ij}\}_{i,j=0}^{N,M})$; // Algorithms 66 or 77
$Transport(k)$; // Algorithm 83
$ExplicitRHS(\Delta t)$; // Algorithm 84
$MatrixAction(\{U_{ij}\}_{i,j=0}^{N,M}, \Delta t)$; // Algorithm 85
$Residual(\Delta t)$; // Algorithm 86
End Class NodalAdvDiffClass

the second derivative matrices in the Nodal2DStorage structure store the matrices G_{ik}, which are computed by Algorithm 57 (CGDerivativeMatrix). The quadrature weights and nodes are the Legendre values, which we compute by Algorithm 25.

5.3.4.3 The Transport Terms

The next procedure we implement computes the transport terms. The implementation of the transport approximation is similar to that of the diffusion terms, as seen in Algorithm 83 (NodalAdvDiffClass:Transport). Notice that Algorithm 83 is the same whether we use Chebyshev or Legendre collocation. To change it to compute the nodal Galerkin approximation, the transport term needs only to be modified according to (5.110), that is, we only need to multiply by the local quadrature weight values.

5.3.4.4 The Iterative Solver

To solve the systems (5.114) or (5.117) by the BiCGStab (Algorithm 75) or Conjugate Gradient (Algorithm 80) iterative methods, we need to implement the residual computation and the MatrixAction procedures. Notice that a slight modification

Algorithm 82: *NodalAdvDiffClass:Construct:* Constructor for the Chebyshev Collocation Approximation of the Advection-Diffusion Problem

Procedure Construct
Input: N, M, u, v, ν
Uses Algorithms:
Algorithm 38 (mthOrderPolynomialDerivativeMatrix)
Algorithm 27 (ChebyshevGaussLobattoNodesAndWeights)
Algorithm 37 (PolynomialDerivativeMatrix)

$this.spA.N \leftarrow N$; $this.spA.M \leftarrow M$
$this.u \leftarrow u$; $this.v \leftarrow v$; $this.\nu \leftarrow \nu$
$\left\{\{this.spA.\xi_i\}_{i=0}^{N}, \left\{this.spA.w_i^{(\xi)}\right\}_{i=0}^{N}\right\} \leftarrow ChebyshevGaussLobattoNodesAndWeights(N)$
$\left\{\left\{this.spA.\eta_j\right\}_{j=0}^{N}, \left\{this.spA.w_j^{(\eta)}\right\}_{j=0}^{N}\right\} \leftarrow ChebyshevGaussLobattoNodesAndWeights(M)$
$this.spA.\left\{D_{ij}^{(2),\xi}\right\}_{i,j=0}^{N} \leftarrow mthOrderPolynomialDerivativeMatrix(N, 2, this.spA.\{\xi_i\}_{i=0}^{N})$
$this.spA.\left\{D_{ij}^{(2),\eta}\right\}_{i,j=0}^{M} \leftarrow mthOrderPolynomialDerivativeMatrix(M, 2, this.spA.\{\eta_j\}_{j=0}^{M})$
$this.spA.\left\{D_{ij}^{\xi}\right\}_{i,j=0}^{N} \leftarrow PolynomialDerivativeMatrix(N, this.spA.\{\xi_i\}_{i=0}^{N})$
$this.spA.\left\{D_{ij}^{\eta}\right\}_{i,j=0}^{M} \leftarrow PolynomialDerivativeMatrix(M, this.spA.\{\eta_j\}_{j=0}^{M})$
$this.\{p_k\}_{k=-2}^{0} \leftarrow \{-2, -1, 0\}$
End Procedure Construct

Algorithm 83: *NodalAdvDiffClass:Transport:* Approximation to $\mathbf{q} \cdot \nabla \Phi$

Procedure Transport
Input: k
Uses Algorithms:
Algorithm 19 (MxVDerivative)
Algorithm 67 (MaskSides)

$N \leftarrow this.spA.N$; $M \leftarrow this.spA.M$
for $j = 0$ **to** M **do**
  $\left\{\Phi x_{i,j}\right\}_{i=0}^{N} \leftarrow MxVDerivative\left(this.spA.\left\{D_{i,j}^{\xi}\right\}_{i,j=0}^{N,N}, this.\left\{\Phi_{i,j}^{k}\right\}_{i=0}^{N}\right)$
end
for $i = 0$ **to** N **do**
  $\left\{\Phi y_{i,j}\right\}_{j=0}^{M} \leftarrow MxVDerivative\left(this.spA.\left\{D_{i,j}^{\eta}\right\}_{i,j=0}^{M,M}, this.\left\{\Phi_{i,j}^{k}\right\}_{j=0}^{M}\right)$
end
for $j = 0$ **to** M **do**
  for $i = 0$ **to** N **do**
    $this.transport_{i,j}^{k} \leftarrow this.u * \Phi x_{i,j} + this.v * \Phi y_{i,j}$
    [For nodal Galerkin add:
    $this.transport_{i,j}^{k} \leftarrow this.spA.w_i^{(\xi)} * this.spA.w_j^{(\eta)} * this.transport_{i,j}^{k}$]
  end
end
$this.\left\{transport_{i,j}^{k}\right\}_{i,j=0}^{N,M} \leftarrow MaskSides\left(this.\left\{transport_{i,j}^{k}\right\}_{i,j=0}^{N,M}, this.\{mask_n\}_{n=1}^{4}\right)$
End Procedure Transport

Algorithm 84: *NodalAdvDiffClass:ExplicitRHS:* Explicit Part of the BDF Approximation of the Advection-Diffusion Equation

Procedure AdvDiffExplicitRHS
Input: Δt
Uses Algorithms:
Algorithm 67 (MaskSides)

$n \leftarrow this.p_0$; $nm1 \leftarrow this.p_{-1}$; $nm2 \leftarrow this.p_{-2}$
for $j = 0$ **to** M **do**
  for $i = 0$ **to** N **do**
    $this.RHS_{i,j} \leftarrow \frac{1}{11}\left(18 * this.\Phi^{n}_{i,j} - 9 * this.\Phi^{nm1}_{i,j} + 2 * this.\Phi^{nm2}_{i,j}\right)$
    [For nodal Galerkin, add:
    $this.RHS_{i,j} \leftarrow this.spA.w^{(\xi)}_{i} * this.spA.w^{(\eta)}_{j} * this.RHS_{i,j}$]
    $this.RHS_{i,j} \leftarrow this.RHS_{i,j} -$
    $\frac{6\Delta t}{11}\left(3 * this.transport^{n}_{i,j} - 3 * this.transport^{nm1}_{i,j} + this.transport^{nm2}_{i,j}\right)$
  end
end
$this.\{RHS_{i,j}\}^{N,M}_{i,j=0} \leftarrow MaskSides(this.\{RHS_{i,j}\}^{N,M}_{i,j=0}, this.\{mask_k\}^{4}_{k=1})$
End Procedure AdvDiffExplicitRHS

needs to be made to the two solvers to allow the time step, Δt to be passed to the Residual and MatrixAction procedures.

To compute the residual, (5.116) or (5.117), we need to have the *RHS* array from the current and previous time step levels available. Algorithm 84 (NodalAdvDiffClass:AdvDiffExplicitRHS), for instance, computes the right hand side of the collocation approximation (5.115). To convert the procedure to the nodal Galerkin approximation, the solution terms need to be multiplied by the quadrature weights, according to (5.118), as shown. The matrix action is computed in Algorithm 85 (NodalAdvDiffClass:MatrixAction). Finally, we compute the residual from the RHS and the matrix action in Algorithm 86 (NodalAdvDiffClass:Residual) for the collocation approximation. Note that we could use BLAS level 1 routines to replace many of the loops.

For best performance, preconditioning should be applied to the system (5.114). Fortunately, it is easy to modify the preconditioners that we have already been derived for the Laplace operator. For instance, the finite difference preconditioner, (5.37), when applied to (5.114) becomes

$$(H_{FD}u)_{ij} = \hat{A}_{ij}u_{ij} + \hat{B}_{ij}u_{i-1,j} + \hat{C}_{ij}u_{i,j-1} + \hat{E}_{ij}u_{i+1,j} + \hat{F}_{ij}u_{i,j+1}, \quad (5.120)$$

where

$$\hat{A}_{ij} = 1 - \frac{6\nu\Delta t}{11}A_{ij} \quad (5.121)$$

and

$$\hat{B}_{ij} = -\frac{6\nu\Delta t}{11}B_{ij}, \quad \text{etc.} \quad (5.122)$$

Algorithm 85: *NodalAdvDiffClass:MatrixAction:* Matrix Action for the BDF Approximation of the Advection-Diffusion Equation

Procedure MatrixAction
Input: Δt, $\left\{U_{i,j}\right\}_{i,j=0}^{N,M}$
Uses Algorithms:
Algorithm 66 or 77 (LaplacianOnTheSquare)
Algorithm 67 (MaskSides)

$\left\{\nabla_N^2 U_{ij}\right\}_{i,j=0}^{N,M} \leftarrow this.LaplacianOnSquare(\{U_{ij}\}_{i,j=0}^{N,M})$
for $j = 0$ **to** M **do**
  for $i = 0$ **to** N **do**
    $action_{i,j} = U_{i,j} - \frac{6\nu\Delta t}{11}\nabla_N^2 U_{i,j}$
    [For nodal Galerkin, use:
    $action_{i,j} = this.spA.w_i^{(\xi)} * this.spA.w_j^{(\eta)} * U_{i,j} - \frac{6\nu\Delta t}{11}\nabla_N^2 U_{i,j}$]
  end
end
$\{action_{i,j}\}_{i,j=0}^{N,M} \leftarrow MaskSides(\{action_{i,j}\}_{i,j=0}^{N,M}, this.\{mask_k\}_{k=1}^{4})$
return $\{action_{i,j}\}_{i,j=0}^{N,M}$
End Procedure MatrixAction

Algorithm 86: *NodalAdvDiffClass:Residual:* Iteration Residual for the BDF Approximation of the Advection-Diffusion Equation

Procedure Residual
Input: Δt, $\left\{U_{i,j}\right\}_{i,j=0}^{N,M}$
Uses Algorithms:
Algorithm 85 (NodalAdvDiffClass:MatrixAction)
Algorithm 67 (MaskSides)

$\{action_{i,j}\}_{i,j=0}^{N,M} \leftarrow this.MatrixAction(\{U_{ij}\}_{i,j=0}^{N,M}, \Delta t)$
for $j = 0$ **to** M **do**
  for $i = 0$ **to** N **do**
    $r_{i,j} = this.RHS_{i,j} - action_{i,j}$
  end
end
$\{r_{i,j}\}_{i,j=0}^{N,M} \leftarrow MaskSides(\{r_{i,j}\}_{i,j=0}^{N,M}, this.\{mask_k\}_{k=1}^{4})$
return $\{r_{i,j}\}_{i,j=0}^{N,M}$
End Procedure Residual

Similar modifications change the finite element preconditioner for use with the Conjugate Gradient iteration.

The need to precondition (5.114) is not as critical, however, as it is for the solution of the potential problem. As we said, a good initial guess is available, so the residual starts small. More importantly, the presence of the Δt factor means that the system is not as stiff. Clearly, for small Δt the matrix on the left approaches the

identity matrix, which has a condition number of one. Only for very large Δt does the system become badly ill-conditioned. Very large values won't occur in practice since the overall time step is limited by the advection time step, and so $\Delta t \sim 1/N^2$. The condition number of the system to be solved will therefore grow only as $O(N^2)$ instead of $O(N^4)$. Since relatively few iterations are needed for the less stiff system, the benefits of preconditioning will be diminished. Nevertheless, for large N and tight iteration tolerances, preconditioning can reduce the number of iterations per time step significantly enough to make it worthwhile to code.

5.3.4.5 Multistep Time Integration

Lastly, we need to implement an algorithm to evaluate the time stepping procedure (5.112). The procedure must first compute the right hand side via Algorithm 84 (ExplicitRHS) and then update the pointers via (5.119) to shift the arrays and to make the storage that is no longer needed available to the new solution and transport terms. The boundary conditions on the solution are then set for the new time, which will be used when the system is solved. Once that is done, the transport terms are computed via Algorithm 83 (Transport).

One time step of the multistep method (5.112) is implemented in Algorithm 87 (MultistepIntegration). The steps taken within that algorithm are identical if the nodal Galerkin approximation is used, except that we would use the Conjugate Gradient solver Algorithm 80 (PreconditionedConjugateGradientSolve). A bound-

Algorithm 87: *MultistepIntegration:* One Step of the Linear Multistep Integration of the Advection-Diffusion Equation

Procedure MultistepIntegration
Input:
advDiff `// NodalAdvDiffClass instance`
H `// Preconditioner instance`
Δt
Uses Algorithms:
Algorithm 84 (NodalAdvDiffClass:ExplicitRHS)
Algorithm 83 (NodalAdvDiffClass:Transport)
Algorithm 75 (BiCGSSTABSolve)

$advDiff.\left\{RHS_{i,j}\right\}_{i,j=0}^{N,M} \leftarrow advDiff.ExplicitRHS(\Delta t)$
$tmp \leftarrow advDiff.p_{-2}$
$advDiff.p_{-2} \leftarrow advDiff.p_{-1}$
$advDiff.p_{-1} \leftarrow advDiff.p_0$
$advDiff.p_0 \leftarrow tmp$
$advDiff.\left\{\Phi_{i,j}^{p_0}\right\}_{i,j=0}^{N,M} \leftarrow SetBoundaryConditions(t+\Delta t, advDiff)$
$advDiff \leftarrow BiCGSSTABSolve(N_{it}, TOL, advDiff, H)$
$advDiff.\left\{transport_{i,j}^{p_0}\right\}_{i,j=0}^{N,M} \leftarrow advDiff.Transport(p_0)$
return *advDiff*
End Procedure MultistepIntegration

ary condition routine must be supplied to compute the values of the solution along Dirichlet boundaries.

Since the semi-implicit scheme (5.112) is not self starting, we must integrate the first two steps with an explicit method. Two additional procedures are needed. We must

- *Modify the explicit time integration Algorithm 50 (CollocationStepByRK3).* Algorithm 50, which implements the third order Runge-Kutta method for collocation with Dirichlet boundary conditions, is appropriate to compute the first two time steps. It has the same order of accuracy in time as the multistep method, although matching the order precisely is not critical. After all, the two steps to be computed will have to be taken using the explicit diffusion limited time step whose size is $O(N^{-4})$, which is much smaller that the advective time step ($O(N^{-2})$ that limits the explicit part of the multistep method. Algorithm 50 was presented for one dimensional problems, so it needs to be extended to act on doubly, rather than singly, dimensioned arrays.
- *Implement a time derivative procedure for fully explicit integration.* Algorithm 50 embeds the spatial approximations in the algorithm that implements the *TimeDerivative* function. The time derivative function will now evaluate either equation (5.100) or (5.109), depending on which spatial approximation we choose. The time derivative function merely has to call the procedures to compute the transport and diffusion terms, i.e. Algorithms 83 (Transport) and 66 (LaplacianOnTheSquare), so we will not write it explicitly here. As we have mentioned before, the weight functions are divided out of (5.109) when we use the explicit time integrator.

5.3.5 Benchmark Solution: Advection and Diffusion of a Spot in a Uniform Flow

To benchmark the advection-diffusion solver, we compute the approximate solution to the advection-diffusion equation, (5.97), with initial and Dirichlet boundary conditions chosen so that the exact solution is

$$\varphi(x, y, t) = \frac{1}{4t+1} e^{-\frac{((x-ut-x_0)^2+(y-vt-y_0)^2)}{\nu(4t+1)}}. \tag{5.123}$$

This solution describes a circular patch that is advected at a constant speed $u\hat{x} + v\hat{y}$ while it diffuses. Specific parameters for the benchmark solutions will be $u = v = 0.8$, $\nu = 0.01$, and $x_0 = y_0 = -0.5$.

We present contours of the exact and Legendre collocation solutions in Fig. 5.4 for two times $t = 0.5$ and $t = 1.5$. We computed these solutions with $N = M = 28$ and $\Delta t = 3.9 \times 10^{-4}$, then interpolated them to 70 points in each direction using Algorithm 35 (2DCoarseToFineInterpolation). In Fig. 5.5, we plot the values of the exact solutions and the computed solutions interpolated to 70 uniformly spaced points along the line $y = x$.

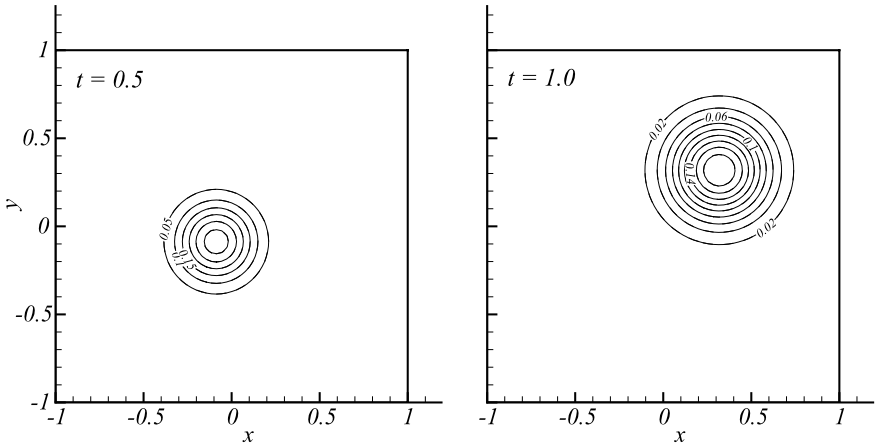

Fig. 5.4 Advection and diffusion of a Gaussian spot by Legendre collocation with $N = 28$. Exact contours drawn with *dashed lines* are indistinguishable from the *solid contours* of the computed solutions

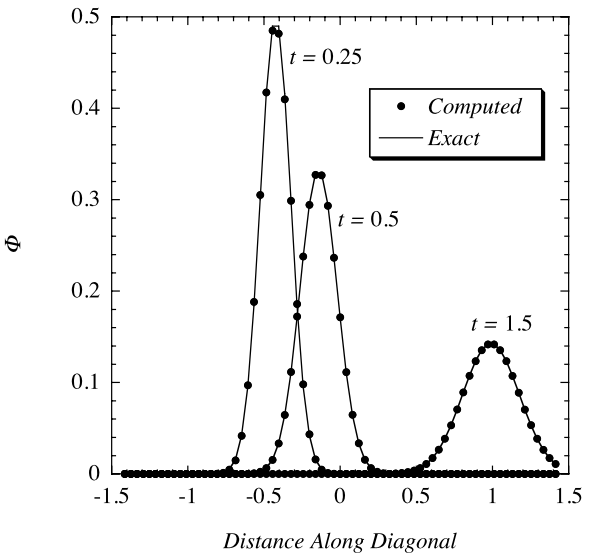

Fig. 5.5 Comparison of computed and exact solutions at three times for the advection and diffusion of a Gaussian spot by Legendre collocation with $N = M = 28$, interpolated to 70 points along the line $y = x$

As a point of comparison, at time $t = 1.25$, the maximum error of the Legendre collocation approximation is approximately 2×10^{-4}. Contrast this with the well-known second order Lax-Wendroff finite difference method, that with 26,000 degrees of freedom (thirty times the number of degrees of freedom in this spectral approximation) the maximum error is still one hundred times larger at 2×10^{-2}.

5.4 Approximation of Wave Propagation Problems

The basic model for wave propagation is, of course, the wave equation. In its most recognizable form, the wave equation is

$$\frac{\partial^2 p}{\partial t^2} - c^2 \left(p_{xx} + p_{yy}\right) = 0. \tag{5.124}$$

In this context, the variable p might represent the acoustic pressure in an otherwise quiescent gas and c would be the sound speed. In other applications, it might represent the electric field with c corresponding to the speed of light, or it could represent the height of water in a shallow tank, where c is the gravity wave speed.

Rather than solve the second order wave equation directly, we will re-write it as a system of three first order equations. (In actuality, the wave equation is the derived form. The system of first order equations is closer to the original description of the phenomena.) We can then use the first order system of equations as a model for more complex systems such as Maxwell's equations used in electromagnetics, the Euler gas dynamics equations, which describe inviscid fluid flow, or the shallow water equations, which are used in meteorology and oceanography simulations.

To convert the wave equation to a system of first order equations, let

$$\begin{aligned} u_t &= -p_x, \\ v_t &= -p_y. \end{aligned} \tag{5.125}$$

(As one might suspect from the notation, u and v correspond to the components of the velocity in a fluid flow.) If we assume that the order of mixed partial derivatives does not matter, then

$$\frac{\partial^2 p}{\partial t^2} + c^2 \left((u_x)_t + (v_y)_t\right) = 0. \tag{5.126}$$

With the proper initial conditions,

$$p_t + c^2 \left(u_x + v_y\right) = 0. \tag{5.127}$$

We find the system of equations by grouping the equations for the pressure and two velocity components

$$\begin{bmatrix} p \\ u \\ v \end{bmatrix}_t + \begin{bmatrix} 0 & c^2 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} p \\ u \\ v \end{bmatrix}_x + \begin{bmatrix} 0 & 0 & c^2 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} p \\ u \\ v \end{bmatrix}_y = 0 \tag{5.128}$$

or

$$\mathbf{q}_t + B\mathbf{q}_x + C\mathbf{q}_y = 0. \tag{5.129}$$

Finally, since B and C are constant, we bring them inside the derivatives to create

$$\mathbf{q}_t + \mathbf{f}_x + \mathbf{g}_y = 0, \tag{5.130}$$

where $\mathbf{f} = B\mathbf{q}$ and $\mathbf{g} = C\mathbf{q}$. This is known as the divergence or *conservation law* form since it is nothing but

$$\mathbf{q}_t + \nabla \cdot \mathscr{F} = 0 \tag{5.131}$$

for the vector flux $\mathscr{F} = \mathbf{f}\hat{x} + \mathbf{g}\hat{y}$. The term conservation law follows from the fact that the differential equation is what we get when we apply the divergence theorem to the integral conservation law,

$$\frac{d}{dt}\int_V \mathbf{q} dV = -\int_S \mathscr{F} \cdot \hat{n} dS, \tag{5.132}$$

which states that the change in the total amount of $\mathbf{q}$ in an arbitrary volume V is equal to the total amount passing through the surface of that volume per unit time.

The defining feature of the wave equation is that it has plane wave type solutions. It will be crucial to understand this fact later to develop boundary conditions. These plane wave solutions have the form

$$\mathbf{q}(\mathbf{x}, t) = \mathbf{a} f\left(\frac{\mathbf{k} \cdot \mathbf{x}}{|\mathbf{k}|} - \gamma t\right) \tag{5.133}$$

for any function f. The vector $\mathbf{k} = k_x\hat{x} + k_y\hat{y}$ is the wavevector, its magnitude $k = |\mathbf{k}|$ is the wavenumber, γ is the wave speed, and the vector $\mathbf{a}$ gives the amplitudes of the three components of the solution, p, u, v. To find the dispersion relation, which is the relationship between k and γ for the plane wave to be a solution of the differential equation, we substitute (5.133) into the differential equation, (5.128), and assume f is smooth to get the algebraic relation

$$\left(-\gamma\mathbf{a} + \frac{k_x}{k}B\mathbf{a} + \frac{k_y}{k}C\mathbf{a}\right) f'\left(\frac{\mathbf{k} \cdot \mathbf{x}}{k} + \gamma t\right) = 0. \tag{5.134}$$

Since this holds for any f, the parameters γ, $\mathbf{k}$ and $\mathbf{a}$ must satisfy

$$\left(\frac{k_x}{k}B + \frac{k_y}{k}C\right)\mathbf{a} = \gamma\mathbf{a}. \tag{5.135}$$

In other words, to have a plane wave solution of the form (5.133), γ must be an eigenvalue of the matrix

$$A = \frac{k_x}{k}B + \frac{k_y}{k}C \tag{5.136}$$

and $\mathbf{a}$ is the eigenvector associated with the eigenvalue γ.

To find the wave speeds associated with the system (5.130), we must therefore find the eigenvalues of

$$A = \alpha B + \beta C = \begin{bmatrix} 0 & \alpha c^2 & \beta c^2 \\ \alpha & 0 & 0 \\ \beta & 0 & 0 \end{bmatrix}, \tag{5.137}$$

where the constants $\alpha \equiv k_x/k$ and $\beta \equiv k_y/k$ satisfy $\alpha^2 + \beta^2 = 1$. The characteristic equation for the eigenvalues of A is

$$-\gamma^3 + \alpha^2 c^2 \gamma + \beta^2 c^2 \gamma = 0 \tag{5.138}$$

so $\gamma = 0$ and $\gamma = \pm\sqrt{\alpha^2 + \beta^2}c = \pm c$. Thus, the system admits two waves that move with speed $\pm c$ along any wavevector in the plane and another that is stationary. (In gas dynamics, the waves that move with speed $\pm c$ are called acoustic or sound waves. The wave that doesn't move is the vorticity wave. It is stationary only because there is no mean flow in this example, otherwise the vorticity wave moves with the fluid. Similar analogies can be made with other systems of equations.)

The eigenvectors, $\mathbf{a}$, associated with the eigenvalues $\gamma = 0, \pm c$ give the relationship between the components of the plane waves. Those three right eigenvectors are

$$\mathbf{a}_0 = \begin{bmatrix} 0 \\ \beta \\ -\alpha \end{bmatrix}, \qquad \mathbf{a}_{\pm c} = \begin{bmatrix} \frac{1}{2} \\ \pm\frac{\alpha}{2c} \\ \pm\frac{\beta}{2c} \end{bmatrix}. \tag{5.139}$$

(To continue the fluid dynamics analogy, note that the acoustic waves have pressure and velocity components, but the vorticity waves, for which vorticity is defined as the curl of the velocity, has no pressure component.)

5.4.1 The Nodal Discontinuous Galerkin Approximation

Although spectral collocation methods have been developed and used to solve systems of conservation laws, it is by far more convenient to implement boundary conditions for the discontinuous Galerkin formulation. So in what follows, we will describe only that method and refer to the book by Canuto et al. [7] for a discussion of other approximations.

To be somewhat general, we will derive the approximation of the wave equation in the form of the conservation law, (5.131), on the reference square $[-1, 1] \times [-1, 1]$ with outward normal $\hat{n}$ and boundary Γ made up of four segments $\Gamma_i, i = 1, 2, 3, 4$ as we sketch in Fig. 5.6.

Once again, we approximate the solution and fluxes by polynomials

$$\begin{aligned} \mathbf{q} \approx \mathbf{Q} &= \sum_{n=0}^{N} \sum_{m=0}^{N} \mathbf{Q}_{n,m} \ell_n(x) \ell_m(y), \\ \mathscr{F} \approx \mathbf{F} &= \sum_{n=0}^{N} \sum_{m=0}^{N} \left(\mathbf{F}_{n,m} \hat{x} + \mathbf{G}_{n,m} \right) \ell_n(x) \ell_m(y), \end{aligned} \tag{5.140}$$

where $\mathbf{F}_{n,m}\hat{x} + \mathbf{G}_{n,m}\hat{y} = B\mathbf{Q}_{n,m}\hat{x} + C\mathbf{Q}_{n,m}\hat{y}$. When we substitute the approximations into the weak form of the differential equation, and project onto the basis

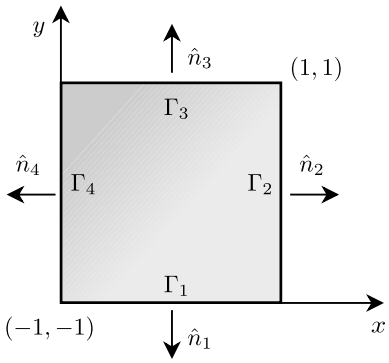

Fig. 5.6 The reference square with normals and boundary curves

functions $\phi_{ij} = \ell_i(x)\ell_j(y)$ (cf. Sect. 4.7)

$$\left(\mathbf{Q}_t, \phi_{ij}\right) + \left(\nabla \cdot \mathbf{F}, \phi_{ij}\right) = 0. \tag{5.141}$$

The next step is to apply Green's identity to the second integral

$$\left(\nabla \cdot \mathbf{F}, \phi_{ij}\right) = \int_{-1}^{1} \phi_{ij} \nabla \cdot \mathbf{F} dx dy = \int_{\Gamma} \phi_{ij} \mathbf{F} \cdot \hat{n} d\Gamma - \int_{-1}^{1} \mathbf{F} \cdot \nabla \phi_{ij} dx dy. \tag{5.142}$$

In the discontinuous Galerkin method, the boundary conditions are weakly enforced when we apply them to the boundary integral. For the moment, we will denote the fact that we apply the boundary conditions as part of the flux in the boundary integrals by $\mathbf{F}^* \cdot \hat{n}$ replacing $\mathbf{F} \cdot \hat{n}$. We'll determine exactly what that replacement is after we finish the spatial approximation. When we make that replacement for the boundary flux, the approximation satisfies

$$\left(\mathbf{Q}_t, \phi_{ij}\right) + \int_{\Gamma} \phi_{ij} \mathbf{F}^* \cdot \hat{n} d\Gamma - \int_{-1}^{1} \mathbf{F} \cdot \nabla \phi_{ij} dx dy = 0. \tag{5.143}$$

The final stages of the approximation procedure are to choose the locations of the nodes, approximate the integrals by quadrature, and simplify the results. For the reasons argued in Sect. 4.7, we choose Legendre Gauss quadrature to approximate the integrals and use the tensor product of the Legendre Gauss quadrature points as the nodes. Those nodes we represent as circles in Fig. 5.7. To approximate the boundary integrals we again choose the nodes to be the Legendre Gauss quadrature points along the boundaries (marked by squares in Fig. 5.7) and use Gauss quadrature to approximate the integrals. This choice will make the computation of the boundary fluxes efficient. To derive the approximation, we will examine each integral in (5.143) separately.

The first integral in (5.143), $\left(\mathbf{Q}_t, \phi_{ij}\right)$, simplifies to a single point value when we apply Gauss quadrature. With Gauss quadrature and nodes taken as the Gauss

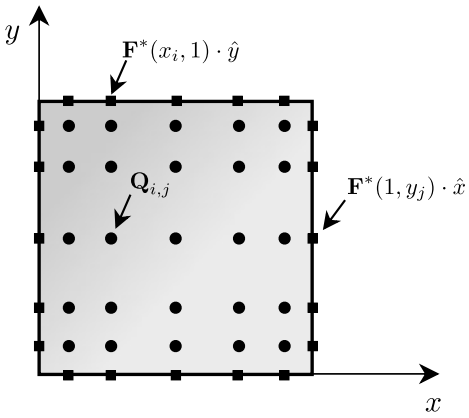

Fig. 5.7 Location of nodes for the discontinuous Galerkin approximation of the wave equation. *Circles* represent nodes where the solution is approximated and are located at the two dimensional Legendre Gauss quadrature points. *Squares* represent nodes where the boundary fluxes are approximated, and are located at the Legendre Gauss quadrature points along the boundary curves

points,

$$\begin{aligned}\left(\mathbf{Q}_t, \phi_{ij}\right) &= \int_{-1}^{1} \frac{d\mathbf{Q}(x,y)}{dt} \ell_i(x)\ell_j(y)dxdy \\ &= \sum_{k=0}^{N}\sum_{l=0}^{N} \frac{d\mathbf{Q}(x_k, y_l)}{dt} \ell_i(x_k)\ell_j(y_l) w_k^{(x)} w_l^{(y)}. \end{aligned} \tag{5.144}$$

Because the product in the integrand is a polynomial of degree $2N$ in each direction, the quadrature is exact. The fact that $\ell_j(x_k) = \delta_{k,j}$, etc., simplifies the double sum to

$$\left(\mathbf{Q}_t, \phi_{ij}\right) = \frac{d\mathbf{Q}_{i,j}}{dt} w_i^{(x)} w_j^{(y)}. \tag{5.145}$$

We'll skip over the boundary integral in (5.143) for the moment and approximate the last integral next. After we substitute $\phi_{ij} = \ell_i(x)\ell_j(y)$ and expand the vector dot product,

$$\int_{-1}^{1} \mathbf{F} \cdot \nabla \phi_{ij} dxdy = \int_{-1}^{1} \left\{ \mathbf{F}(x,y)\ell_i'(x)\ell_j(y) + \mathbf{G}(x,y)\ell_i(x)\ell_j'(y) \right\} dxdy. \tag{5.146}$$

We then replace the integrals each by Gauss quadrature, which again is exact because $\mathbf{F}$ and $\mathbf{G}$ are linear functions of $\mathbf{Q}$ and the integrands are polynomials of degree at most N in each direction. The Kronecker delta property of the Lagrange interpolating polynomials once again simplifies the summations. With quadrature and simplifications,

$$\int_{-1}^{1} \mathbf{F} \cdot \nabla \phi_{ij} dxdy = \sum_{k=0}^{N} \mathbf{F}_{k,j} \ell_i'(x_k) w_k^{(x)} w_j^{(y)} + \sum_{k=0}^{N} \mathbf{G}_{i,k} \ell_j'(y_k) w_i^{(x)} w_k^{(y)}. \tag{5.147}$$

Notice again that the tensor product approximation decouples the derivatives in the two space directions. When we use our definition for the polynomial derivative ma-

trix, we write the last integral in (5.143) as

$$\int_{-1}^{1} \mathbf{F} \cdot \nabla \phi_{ij} dx dy = \sum_{k=0}^{N} \mathbf{F}_{k,j} D_{ki}^{(x)} w_k^{(x)} w_j^{(y)} + \sum_{k=0}^{N} \mathbf{G}_{i,k} D_{jk}^{(y)} w_i^{(x)} w_k^{(y)}. \tag{5.148}$$

Finally, we approximate the boundary integral in (5.143). We break it into four pieces, along the four sides of the square, as

$$\begin{aligned}\int_{\Gamma} \phi_{i,j} \mathbf{F}^* \cdot \hat{n} d\Gamma = &\int_{-1}^{1} \ell_i(x) \ell_j(-1) \mathbf{F}^* \cdot \left(-\hat{y}\right) dx + \int_{-1}^{1} \ell_i(1) \ell_j(y) \mathbf{F}^* \cdot \hat{x} dy \\ &+ \int_{-1}^{1} \ell_i(x) \ell_j(1) \mathbf{F}^* \cdot \hat{y} dx + \int_{-1}^{1} \ell_i(-1) \ell_j(y) \mathbf{F}^* \cdot \left(-\hat{x}\right) dy.\end{aligned} \tag{5.149}$$

Then we approximate the integrals by Gauss quadrature, which are exact, too. For example,

$$\begin{aligned}\int_{-1}^{1} \ell_i(x) \ell_j(-1) \mathbf{F}^* \cdot \hat{y} dx &= \sum_{k=0}^{N} \ell_i(x_k) \ell_j(-1) \mathbf{F}^*(x_k, -1) \cdot \hat{y} w_k^{(x)} \\ &= \mathbf{F}^*(x_i, -1) \cdot \hat{y} \ell_j(-1) w_i^{(x)}.\end{aligned} \tag{5.150}$$

(Compare this with the boundary terms in (4.138).) After we apply quadrature to each of the segments in the boundary integral, that integral becomes

$$\begin{aligned}\int_{\Gamma} \phi_{i,j} \mathbf{F}^* \cdot \hat{n} d\Gamma = &\mathbf{F}^*(x_i, -1) \cdot \left(-\hat{y}\right) \ell_j(-1) w_i^{(x)} + \mathbf{F}^*(1, y_j) \cdot \hat{x} \ell_i(1) w_j^{(y)} \\ &+ \mathbf{F}^*(x_i, 1) \cdot \hat{y} \ell_j(1) w_i^{(x)} + \mathbf{F}^*(-1, y_j) \cdot \left(-\hat{x}\right) \ell_i(-1) w_j^{(y)}.\end{aligned} \tag{5.151}$$

We find the final semi-discrete approximation to (5.143) after we divide by $w_i^{(x)} w_j^{(y)}$ and rearrange

$$\begin{aligned}&\frac{d\mathbf{Q}_{i,j}}{dt} + \left\{ \left[\mathbf{F}^*(-1, y_j) \cdot \left(-\hat{x}\right) \frac{\ell_i(-1)}{w_i^{(x)}} + \mathbf{F}^*(1, y_j) \cdot \hat{x} \frac{\ell_i(1)}{w_i^{(x)}} \right] + \sum_{k=0}^{N} \mathbf{F}_{k,j} \hat{D}_{ik}^{(x)} \right\} \\ &\quad + \left\{ \left[\mathbf{F}^*(x_i, -1) \cdot \left(-\hat{y}\right) \frac{\ell_j(-1)}{w_j^{(y)}} + \mathbf{F}^*(x_i, 1) \cdot \hat{y} \frac{\ell_j(1)}{w_j^{(y)}} \right] + \sum_{k=0}^{N} \mathbf{G}_{i,k} \hat{D}_{jk}^{(y)} \right\} \\ &\quad = 0, \quad i, j = 0, 1, \ldots, N,\end{aligned} \tag{5.152}$$

where, again

$$\hat{D}_{jn} = -\frac{D_{nj} w_n}{w_j}, \tag{5.153}$$

and $D_{nj} = \ell'_j(x_n)$ is the transpose of the standard derivative matrix, computed with Algorithm 37 (PolynomialDerivativeMatrix).

Notice that the two terms in the braces are nothing more than the one-dimensional discontinuous Galerkin derivative that already appears in the braces in (4.138). Its implementation is Algorithm 60 (NodalDiscontinuousGalerkin:DGDerivative) for the scalar problem. Therefore, the computation of the time derivative for the system in two dimensions will proceed just like the computation for the scalar, one dimensional problem. The only thing we have left is to determine how to compute the boundary fluxes, $\mathbf{F}^* \cdot \hat{n}$.

5.4.1.1 The Boundary Flux

In the one dimensional problem of Sect. 4.7, we set the boundary condition on the *upwind* side. Which is the upwind side is determined by the sign of the wave speed. Positive wavespeeds (with respect to the x direction) mean that the boundary condition is set on the left. Negative wavespeeds require the solution to be set on the right. At the downwind side, we evaluated the solution from the interpolant.

By extension, the two dimensional problem requires that we compute a value of the flux,

$$\mathscr{F} \cdot \hat{n} = \mathbf{f} n_x + \mathbf{g} n_y = \left(B n_x + C n_y\right) \mathbf{q} = A\mathbf{q} \tag{5.154}$$

at the boundary so that boundary values are set on the upwind side and computed from the interior on the downwind side. Unfortunately, the system that describes the wave equation couples three wavespeeds, positive, negative and zero, with respect to the direction vector $\alpha\hat{x} + \beta\hat{y} = n_x\hat{x} + n_y\hat{y}$, so it is not immediately clear what the upwind value is. To determine the upwind directions, we must decouple the wave components that make up the solution vector. In terms of our discussion at the beginning of this section, we decouple using the eigenvectors of the coefficient matrix, A.

Since the matrix A has a full set of eigenvectors, it can be diagonalized. If we create a matrix,

$$S = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{\alpha}{2c} & -\frac{\alpha}{2c} & \beta \\ \frac{\beta}{2c} & -\frac{\beta}{2c} & -\alpha \end{bmatrix}, \tag{5.155}$$

whose columns are right eigenvectors of A, then

$$AS = S \begin{bmatrix} +c & 0 & 0 \\ 0 & -c & 0 \\ 0 & 0 & 0 \end{bmatrix} = S\Lambda. \tag{5.156}$$

When we premultiply by S^{-1}, we see that $S^{-1}AS = \Lambda$, or equivalently, $A = S\Lambda S^{-1}$. The matrix S^{-1} is nothing more than the matrix whose rows are left eigen-

vectors of the matrix A,

$$S^{-1} = \begin{bmatrix} 1 & \alpha c & \beta c \\ 1 & -\alpha c & -\beta c \\ 0 & \beta & -\alpha \end{bmatrix}, \tag{5.157}$$

since the left and right eigenvectors of the matrix are orthogonal.

The ability to diagonalize $A = S\Lambda S^{-1}$ allows us to separate the system into left going, right going and stationary wave components. Let

$$\begin{aligned} \Lambda &= \begin{bmatrix} +c & 0 & 0 \\ 0 & -c & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} +c & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & -c & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ &= \Lambda^+ + \Lambda^- + \Lambda^0 \end{aligned} \tag{5.158}$$

be the splitting of the three wave components. Then

$$A = S\Lambda^+ S^{-1} + S\Lambda^- S^{-1} + S\Lambda^0 S^{-1} = A^+ + A^- + A^0 \tag{5.159}$$

splits the matrix A into components that have right going, left going and stationary waves with respect to the direction $\alpha\hat{x} + \beta\hat{y}$. We see, therefore, that we can decompose the normal flux into its wave components using (5.154)

$$\mathbf{F} \cdot \hat{n} = A^+\mathbf{q} + A^-\mathbf{q} + A^0\mathbf{q}. \tag{5.160}$$

We are finally in the position to decide how to apply a boundary condition that is upwind in each of its three components. Suppose that we have two states $\mathbf{q}^{ext}$ and $\mathbf{Q}^{int}$ that describe the solution external to the domain and internal to the domain (Fig. 5.8). We specify the external state as a boundary condition. We compute the internal state from the polynomial approximation of the solution via (5.140). The boundary flux will be a function of these two states evaluated at the boundary, $\mathbf{F}^*(\mathbf{Q}^L, \mathbf{Q}^R; \hat{n})$, that separates the waves into those that originate from outside and those that originate from the interior.

From (5.160), we see that we should compute the boundary flux from the internal and external states (with the designation determined relative to the normal at the boundary) as

$$\mathbf{F}^*\left(\mathbf{Q}^L, \mathbf{Q}^R; \hat{n}\right) = A^+\mathbf{Q}^L + A^-\mathbf{Q}^R. \tag{5.161}$$

In this way, outgoing waves are approximated with interior solution values (upwind) and incoming waves are specified from the external state (also upwind). The derivation of the boundary flux, (5.161), also known in some contexts as the *numerical flux*, is an example of the construction of the solution of what is known as the *Riemann problem*, and the algorithm used to solve it the *Riemann solver*. The construction of Riemann solvers for different physical systems has been important in computational mathematics, particularly in fluid mechanics [24].

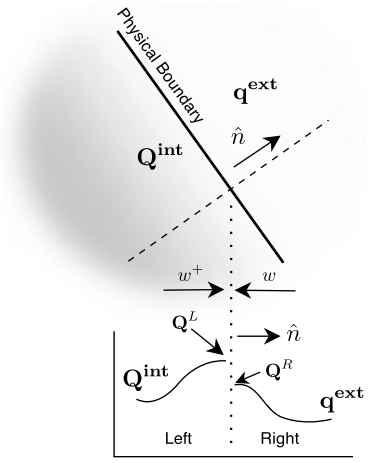

Fig. 5.8 Interior and exterior states at a boundary viewed along the normal direction

To write an algorithm for the Riemann solver, let us explicitly do the algebra that leads to the numerical flux. Let's look first at the quantities $A^{\pm}\mathbf{Q} = S\Lambda^{\pm}S^{-1}\mathbf{Q}$. First,

$$S^{-1}\mathbf{q} = \begin{bmatrix} 1 & \alpha c & \beta c \\ 1 & -\alpha c & -\beta c \\ 0 & \beta & -\alpha \end{bmatrix} \begin{bmatrix} p \\ u \\ v \end{bmatrix} = \begin{bmatrix} p + c\,(\alpha u + \beta v) \\ p - c\,(\alpha u + \beta v) \\ \beta u - \alpha v \end{bmatrix} \equiv \begin{bmatrix} w^+ \\ w^- \\ w^0 \end{bmatrix}. \tag{5.162}$$

The w's are the wave quantities associated with the three eigenvalues since

$$\begin{aligned} \Lambda^+ S^{-1}\mathbf{q} &= \begin{bmatrix} +c & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} w^+ \\ w^- \\ w^0 \end{bmatrix} = c \begin{bmatrix} w^+ \\ 0 \\ 0 \end{bmatrix}, \\ \Lambda^- S^{-1}\mathbf{q} &= \begin{bmatrix} 0 & 0 & 0 \\ 0 & -c & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} w^+ \\ w^- \\ w^0 \end{bmatrix} = -c \begin{bmatrix} 0 \\ w^- \\ 0 \end{bmatrix}. \end{aligned} \tag{5.163}$$

Notice that when we take the direction vector $\alpha\hat{x} + \beta\hat{y}$ to be the boundary normal vector, the quantity $\alpha u + \beta v$ corresponds to the normal velocity of the wave. Under the same condition, the last component, w^0 is the tangential velocity. Finally, we see that to compute the numerical boundary flux, w^+ must be computed from the left state, $\mathbf{Q}^L$, since it represents waves coming from the left and moving to the right relative to the outward normal. Similarly w^- must be computed from the right state, $\mathbf{Q}^R$.

When we multiply from the left by the matrix S, we get an explicit representation of the numerical flux

$$\mathbf{F}^*\big(\mathbf{Q}^L, \mathbf{Q}^R; \hat{n}\big) = \begin{bmatrix} \frac{c}{2}(w^{+,L} - w^{-,R}) \\ \frac{n_x}{2}(w^{+,L} + w^{-,R}) \\ \frac{n_y}{2}(w^{+,L} + w^{-,R}) \end{bmatrix}, \tag{5.164}$$

Algorithm 88: *RiemannSolver:* The Numerical Flux for the Wave Equation

Procedure RiemannSolver
Input: $\{Q_n^L\}_{n=1}^{nEqn}, \{Q_n^R\}_{n=1}^{nEqn}, \hat{n}$
$p^L \leftarrow Q_1^L; u^L \leftarrow Q_2^L; v^L \leftarrow Q_3^L$
$p^R \leftarrow Q_1^R; u^R \leftarrow Q_2^R; v^R \leftarrow Q_3^R$
$w^{+,L} \leftarrow p^L + c * (n_x * u^L + n_y * v^L)$
$w^{-,R} \leftarrow p^R - c * (n_x * u^R + n_y * v^R)$
$F_1^* \leftarrow c * (w^{+,L} - w^{-,R})/2$
$F_2^* \leftarrow n_x * (w^{+,L} + w^{-,R})/2$
$F_3^* \leftarrow n_y * (w^{+,L} + w^{-,R})/2$
return $\{F_n^*\}_{n=1}^{nEqn}$
End Procedure RiemannSolver

where we have used the fact that we want to take the vector $\alpha\hat{x} + \beta\hat{y}$ to be the normal, $n_x\hat{x} + n_y\hat{y}$, at the boundary. For a consistency check, verify that $\mathbf{F}^*(\mathbf{q}, \mathbf{q}; \hat{n}) = \mathbf{F}(\mathbf{q}) \cdot \hat{n}$. If the states on both sides of the boundary are identical, then the numerical flux must equal the flux for that state.

We show a procedure to compute the numerical flux in Algorithm 88 (RiemannSolver). It takes two states, one on the left and one on the right, plus the normal, and computes the two wave components $w^{\pm}$ defined in (5.162) from their proper side. It then reconstructs the flux from those components using (5.164).

To incorporate boundary conditions into the discontinuous Galerkin approximation, (5.152), we need only to provide the left and right state vectors and the normal to the Riemann solver. If we know an analytical representation of the external state, $\mathbf{q}^{ext}(\mathbf{x}, t)$, we use its value as the second argument in (5.164). For instance, the semi-discrete approximation (5.152) requires the numerical flux $\mathbf{F}^*(1, y_j) \cdot \hat{x}$ along the right boundary. We compute it with the Riemann solver as $\mathbf{F}^*(1, y_j) \cdot \hat{x} = \mathbf{F}^*(\mathbf{Q}(1, y_j), \mathbf{q}^{ext}(1, y_j); \hat{x})$.

We represent wall (reflection) boundaries when we choose the external state to be the mirror image of the internal state. Reflection means that the w^- wave is created by reflecting the w^+ wave at the boundary, i.e. $w^- = w^+$. The reflection condition implies that the normal velocity is zero (which makes physical sense as a representation of a solid wall boundary), for

$$p + c\,(\alpha u + \beta v) = p - c\,(\alpha u + \beta v)\,, \tag{5.165}$$

where $\alpha\hat{x} + \beta\hat{y} = n_x\hat{x} + n_y\hat{y}$ implies

$$n_x u + n_y v = 0 \quad (\textit{reflection boundary})\,. \tag{5.166}$$

We enforce the reflection condition if we set the external state to have a normal velocity that is equal in magnitude and opposite in direction to the normal velocity in the interior. We simply reflect the quantities p and the tangential velocity component across the boundary, so $p^{ext} = p^{int}$ and $\beta u^{ext} - \alpha v^{ext} = \beta u^{int} - \alpha v^{int}$. We have two

equations, then, that define the external values u^{ext} and v^{ext}

$$\begin{aligned} \alpha u^{ext} + \beta v^{ext} &= -\left(\alpha u^{int} + \beta v^{int}\right), \\ \beta u^{ext} - \alpha v^{ext} &= \beta u^{int} - \alpha v^{int}, \end{aligned} \tag{5.167}$$

which we solve to find the external state for a wall/reflection boundary

$$\mathbf{q}^{ext}_{refl} = \begin{bmatrix} p^{int} \\ (\beta^2 - \alpha^2)u^{int} - 2\alpha\beta v^{int} \\ -2\alpha\beta u^{int} + (\alpha^2 - \beta^2)v^{int} \end{bmatrix}. \tag{5.168}$$

To approximate a wall boundary, we supply this external state, along with the internal state, to the Riemann solver to compute the boundary flux.

5.4.2 How to Implement the Nodal Discontinuous Galerkin Approximation

We now develop the algorithms that we will use to compute the nodal discontinuous Galerkin approximation to the wave equation in two space dimensions, or for that matter, any other system of conservation laws for which we have a flux function and have derived a Riemann solver. As in Sect. 4.7, where we developed the approximation of a scalar equation in one space dimension, we need algorithms to evaluate the spatial derivatives and the time derivative. The algorithm to integrate in time and a driver to manage the integration will just be modifications of Algorithms 51 (LegendreCollocation) and 62 (DGStepByRK3) that we developed in Sects. 4.4 and 4.7 for one dimensional problems, so we will not discuss them further.

We define classes to store the arrays associated with the spatial approximation. To start, we extend the Nodal2DStorage structure (Algorithm 63) to include the vectors that the procedure uses to interpolate the solutions to the boundaries, which creates Algorithm 89 (NodalDG2DStorage). We then define a class, NodalDG2DClass, that has the new nodal storage class and the solution array as members. We present the new class in Algorithm 90 (NodalDG2DClass) and its constructor in Algorithm 91 (NodalDG2D:Construct).

Algorithm 89: *NodalDG2DStorage:* Data Storage for a Nodal Spectral Method

Structure NodalDG2DStorage **Extends** Nodal2DStorage
Uses Algorithms:
Algorithm 63 (Nodal2DStorage)
Data:
$\left\{\ell_i^{(\xi)}(-1)\right\}_{i=0}^{N}, \left\{\ell_i^{(\xi)}(1)\right\}_{i=0}^{N}, \left\{\ell_j^{(\eta)}(-1)\right\}_{j=0}^{M}, \left\{\ell_j^{(\eta)}(1)\right\}_{j=0}^{M}$
End Structure NodalDG2DStorage

Algorithm 90: *NodalDG2DClass:* A Discontinuous Galerkin Class Definition

Class NodalDG2DClass
Uses Algorithms:
Algorithm 89 (NodalDG2DStorage)
Data:
nEqn
spA ; `// Of type NodalDG2DStorage`
$\left\{Q_{i,j,n}\right\}_{i=0;j=0;n=1}^{N;M;nEqn}$
Procedures:
Construct$(nEqn, N, M)$; `// Algorithm 91`
DG2DTimeDerivative(t); `// Algorithm 92`
End Class NodalDG2DClass

Algorithm 91: *NodalDG2D:Construct:* Constructor for the Discontinuous Galerkin Class

Procedure Construct
Input: $nEqn, N, M$
Uses Algorithms:
Algorithm 23 (LegendreGaussNodesAndWeights)
Algorithm 37 (PolynomialDerivativeMatrix)
Algorithm 34 (LagrangeInterpolatingPolynomials)
Algorithm 30 (BarycentricWeights)

$this.nEqn \leftarrow nEqn$
$this.spA.N \leftarrow N$
$\left\{this.spA.\{\xi_i\}_{i=0}^{N}, this.spA.\left\{w_i^{(\xi)}\right\}_{i=0}^{N}\right\} \leftarrow LegendreGaussNodesAndWeights(N)$
$\left\{w_i^B\right\}_{i=0}^{N} \leftarrow BarycentricWeights(N, this.spA.\{\xi_i\}_{i=0}^{N})$
$this.spA.\left\{\ell_i^{(\xi)}(-1)\right\}_{i=0}^{N} \leftarrow$
$LagrangeInterpolatingPolynomials\left(-1.0, N, this.spA.\{\xi_i\}_{i=0}^{N}, \left\{w_i^B\right\}_{i=0}^{N}\right)$
$this.spA.\left\{\ell_i^{(\xi)}(1)\right\}_{i=0}^{N} \leftarrow$
$LagrangeInterpolatingPolynomials\left(1.0, N, this.spA.\{\xi_i\}_{i=0}^{N}, \left\{w_i^B\right\}_{i=0}^{N}\right)$
$\left\{D_{ij}\right\}_{i,j=0}^{N} \leftarrow PolynomialDerivativeMatrix\left(N, this.spA.\left\{\xi_j\right\}_{i=0}^{N}\right)$
for $j = 0$ **to** N **do**
  for $i = 0$ **to** N **do**
    $this.spA.D_{i,j}^{(\xi)} \leftarrow -D_{j,i} * this.spA.w_j^{(\xi)} / this.spA.w_i^{(\xi)}$
  end
end
Repeat for η direction...
End Procedure Construct

We already know how to use Algorithm 60 (NodalDiscontinuousGalerkin: DGDerivative) compute the spatial derivative approximation given the interior point and two boundary solutions. We modify that algorithm now to compute the terms in

Algorithm 92: *SystemDGDerivative:* Compute the First Derivative via the Discontinuous Galerkin Approximation

Procedure SystemDGDerivative
Input: $\left\{F_n^L\right\}_{n=1}^{nEqn}, \left\{F_n^R\right\}_{n=1}^{nEqn}, \left\{F_{j,n}\right\}_{j=0;n=1}^{N;nEqn}$
Input: $\left\{D_{i,j}\right\}_{i,j=0}^{N}, \{\ell_i(-1)\}_{i=0}^{N}, \{\ell_i(1)\}_{i=0}^{N}, \{w_i\}_{i=0}^{N}$
Uses Algorithms:
Algorithm 19 (MxVDerivative)
for $n = 1$ **to** $nEqn$ **do**
  $\left\{F'_{j,n}\right\}_{j=0}^{N} \leftarrow MxVDerivative\left(\left\{D_{i,j}\right\}_{i,j=0}^{N}, \left\{F_{j,n}\right\}_{j=0}^{N}\right)$
end
for $j = 0$ **to** N **do**
  for $n = 1$ **to** $nEqn$ **do**
    $F'_{j,n} \leftarrow F'_{j,n} + (F_n^R * \ell_j(1) + F_n^L * \ell_j(-1))/w_j$
  end
end
return $\left\{F'_{j,n}\right\}_{j=0;n=1}^{N;nEqn}$
End Procedure SystemDGDerivative

braces in (5.152), which are now vector quantities and require the outward normals at the boundaries. We implement this modification in Algorithm 92 (SystemDGDerivative).

We show how to implement the time derivatives, defined in (5.152), in Algorithm 93 (DGSystemTimeDerivative). The computation of the time derivatives consists of two parts, courtesy of the tensor product approximation of the solution. The first part computes the derivatives for the flux, $\mathbf{F}$, in the x direction for each value of y. Each component of the solution is interpolated to the left and right boundaries by the procedure *InterpolateToBoundary* in Algorithm 61 (DGTimeDerivative). Next, Algorithm 93 computes the external state at those same points using an external procedure that we will have to supply for that purpose. To be able to set a known exterior state, we pass the position and the time to the external state procedure. To allow reflection conditions of the type (5.168), we also pass the interpolated value of the internal state. From the external and internal states, the Riemann solver (Algorithm 88) is called to compute the boundary flux for both the left and right boundaries. Since the spatial derivatives in (5.152) are taken on the flux, the procedure computes the horizontal flux at the internal grid points from the approximate solution at those points using the procedure *xFlux* in Algorithm 94 (WaveEquationFluxes). The second stage follows the same steps to compute the derivatives of the vertical flux, $\mathbf{G}$.

Algorithms 92 (SystemDGDerivative) and 93 (DGSystemTimeDerivative) form the core of the discontinuous Galerkin approximation of a system of conservation laws. To change the system of equations, we only need to change the flux functions and the Riemann solver. To change the boundary conditions, we only need to change the external state procedure.

Algorithm 93: *NodalDG2D:DG2DTimeDerivative:* Time Derivative in 2D for the Discontinuous Galerkin Approximation

Procedure DG2DTimeDerivative
Input: t
Uses Algorithms:
Algorithm 92 (SystemDGDerivative), Algorithm 61 (InterpolateToBoundary)
Algorithm 88 (RiemannSolver), Algorithm 94 (WaveEquationFluxes)

$N \leftarrow this.spA.N$; $M \leftarrow this.spA.M$; $nEqn \leftarrow this.nEqn$
for $j = 0$ **to** M **do**
  $y \leftarrow this.spA.\eta_j$
  for $n = 1$ **to** $nEqn$ **do**
    $Q_n^{L,int} \leftarrow InterpolateToBoundary\big(this.\{Q_{i,j,n}\}_{i=0}^{N}, this.spA.\{\ell_i^{(\xi)}(-1)\}_{i=0}^{N}\big)$
    $Q_n^{R,int} \leftarrow InterpolateToBoundary\big(this.\{Q_{i,j,n}\}_{i=0}^{N}, this.spA.\{\ell_i^{(\xi)}(1)\}_{i=0}^{N}\big)$
  end
  $\{Q_n^{L,ext}\}_{n=1}^{nEqn} \leftarrow ExternalState\big(\{Q_n^{L,int}\}_{n=1}^{nEqn} - 1, y, t, LEFT\big)$
  $\{Q_n^{R,ext}\}_{n=1}^{nEqn} \leftarrow ExternalState\big(\{Q_n^{R,int}\}_{n=1}^{nEqn} 1, y, t, RIGHT\big)$
  $\{F_n^{*,L}\}_{n=1}^{nEqn} \leftarrow RiemannSolver\big(\{Q_n^{L,int}\}_{n=1}^{nEqn}, \{Q_n^{L,ext}\}_{n=1}^{nEqn}, -\hat{x}\big)$
  $\{F_n^{*,R}\}_{n=1}^{nEqn} \leftarrow RiemannSolver\big(\{Q_n^{R,int}\}_{n=1}^{nEqn}, \{Q_n^{R,ext}\}_{n=1}^{nEqn}, \hat{x}\big)$
  for $i = 0$ **to** N **do**
    $\{F_{i,n}\}_{n=1}^{nEqn} \leftarrow xFlux\big(this.\{Q_{i,j,n}\}_{n=1}^{nEqn}\big)$
  end
  $\{F'_{i,n}\}_{i=0;n=1}^{N;nEqn} \leftarrow SystemDGDerivative\big(\{F_n^{*,L}\}_{n=1}^{nEqn}, \{F_n^{*,R}\}_{n=1}^{nEqn}, \{F_{i,n}\}_{i=0;m=1}^{N;nEqn},$
  $this.spA.\{D^{\xi}\}_{i,j}^{N}, \{\ell_i^{(\xi)}(-1)\}_{i=0}^{N}, \{\ell_i^{(\xi)}(1)\}_{i=0}^{N}, \{w_i^{(\xi)}\}_{i=0}^{N}\big)$
  for $i = 0$ **to** N **do**
    for $n = 1$ **to** $nEqn$ **do**
      $\dot{Q}_{i,j,n} \leftarrow -F'_{i,n}$
    end
  end
end
for $i = 0$ **to** N **do**
  $x \leftarrow this.spA.\xi_i$
  for $n = 1$ **to** $nEqn$ **do**
    $Q_n^{L,int} \leftarrow InterpolateToBoundary\big(this.\{Q_{i,j,n}\}_{j=0}^{M}, this.spA.\{\ell_j^{(\eta)}(-1)\}_{j=0}^{M}\big)$
    $Q_n^{R,int} \leftarrow InterpolateToBoundary\big(this.\{Q_{i,j,n}\}_{j=0}^{M}, this.spA.\{\ell_j^{(\eta)}(1)\}_{j=0}^{M}\big)$
  end
  $\{Q_n^{L,ext}\}_{n=1}^{nEqn} \leftarrow ExternalState\big(\{Q_n^{L,int}\}_{n=1}^{nEqn}, x, -1, t, BOTTOM\big)$
  $\{Q_n^{R,ext}\}_{n=1}^{nEqn} \leftarrow ExternalState\big(\{Q_n^{R,int}\}_{n=1}^{nEqn}, x, 1, t, TOP\big)$
  $\{G_n^{*,L}\}_{n=1}^{nEqn} \leftarrow RiemannSolver\big(\{Q_n^{L,int}\}_{n=1}^{nEqn}, \{Q_n^{L,ext}\}_{n=1}^{nEqn}, -\hat{y}\big)$
  $\{G_n^{*,R}\}_{n=1}^{nEqn} \leftarrow RiemannSolver\big(\{Q_n^{R,int}\}_{n=1}^{nEqn}, \{Q_n^{R,ext}\}_{n=1}^{nEqn}, \hat{y}\big)$
  for $j = 0$ **to** M **do**
    $\{G_{j,n}\}_{n=1}^{nEqn} \leftarrow yFlux\big(this.\{Q_{i,j,n}\}_{n=1}^{nEqn}\big)$
  end
  $\{G'_{j,n}\}_{j=0;n=1}^{M;nEqn} \leftarrow SystemDGDerivative\big(\{G_n^{*,L}\}_{n=1}^{nEqn}, \{G_n^{*,R}\}_{n=1}^{nEqn}, \{G_{i,n}\}_{i=0;m=1}^{N;nEqn},$
  $this.spA.\{D^{\eta}\}_{i,j}^{N}, \{\ell_i^{(\eta)}(-1)\}_{i=0}^{N}, \{\ell_i^{(\eta)}(1)\}_{i=0}^{N}, \{w_i^{(\eta)}\}_{i=0}^{N}\big)$
  for $j = 0$ **to** M **do**
    for $n = 1$ **to** $nEqn$ **do**
      $\dot{Q}_{i,j,n} \leftarrow \dot{Q}_{i,j,n} - G'_{j,n}$
    end
  end
end
return $\{\dot{Q}_{i,j,n}\}_{i=0;j=0;n=1}^{N,M,nEqn}$
End Procedure DGSystemTimeDerivative

Algorithm 94: *WaveEquationFluxes:* Flux Vectors for the Two Dimensional Wave Equation

Procedure xFlux
Input: $\{Q_n\}_{n=1}^{nEqn}$
$F_1 \leftarrow c^2 Q_2$; $F_2 \leftarrow Q_1$; $F_3 \leftarrow 0$
return $\{F_n\}_{n=1}^{nEqn}$
End Procedure xFlux

Procedure yFlux
Input: $\{Q_n\}_{n=1}^{nEqn}$
$G_1 \leftarrow c^2 Q_3$; $G_2 \leftarrow 0$; $G_3 \leftarrow Q_1$
return $\{G_n\}_{n=1}^{nEqn}$
End Procedure yFlux

5.4.3 Benchmark Solution: Plane Wave Propagation

We present two simple examples to benchmark the ability of the nodal discontinuous Galerkin method to propagate and reflect plane waves. The first example is the propagation of a single plane Gaussian wave through the grid. The second adds a reflecting wall boundary.

To propagate a plane wave across the domain we only need to create a procedure that defines the wave in space and time. We use that procedure to generate the initial condition and the external state. For the first benchmark solution, we define the plane wave by

$$\begin{bmatrix} p \\ u \\ v \end{bmatrix} = \begin{bmatrix} 1 \\ \frac{k_x}{c} \\ \frac{k_y}{c} \end{bmatrix} e^{-\frac{(k_x(x-x_0)+k_y(y-y_0)-ct)^2}{d^2}} \tag{5.169}$$

with the wavevector $\mathbf{k}$ normalized to satisfy $k_x^2 + k_y^2 = 1$. This is a wave with Gaussian shape where we compute the parameter d from the full width at half maximum, $w = 0.2$, by $d = w/2\sqrt{\ln 2}$. The other parameters are $c = 1$ and $x_0 = y_0 = -0.8$.

We show contour plots of the pressure at three times in Fig. 5.9. For that calculation, we chose the wavevector to be $\mathbf{k} = (\sqrt{2}/2, \sqrt{2}/2)$, $N = M = 40$ and $\Delta t = 2.6 \times 10^{-3}$. To get smooth contours, we interpolated the computed solution with Algorithm 35 (2DCoarseToFineInterpolation) to a fine grid before plotting. Next, we compare the computed solutions to the exact along the straight line between $(-1, -1)$ and $(1, 1)$ at time $t = 2$ for $N = 20$ and $N = 30$ in Fig. 5.10. Note that the $N = 20$ solution shows significant dispersion errors. When we increase the number of points by only 50% in each direction, those errors are no longer visible.

To illustrate the use of reflection boundary conditions, (5.168), we present Fig. 5.11, which shows the reflection of the same plane wave off a reflecting wall

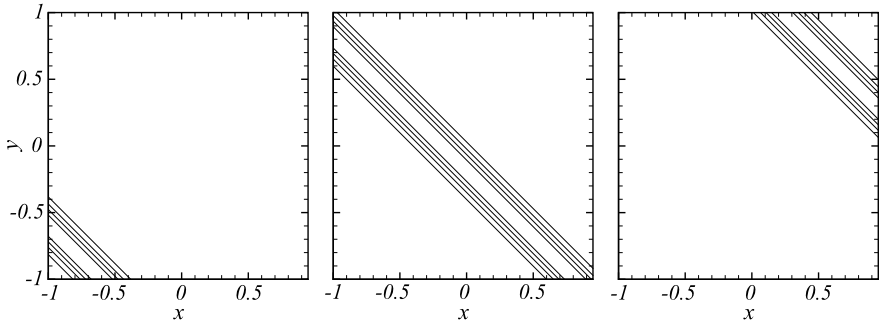

Fig. 5.9 Propagation of a plane Gaussian wave using the nodal discontinuous Galerkin approximation with $N = 40$ for $k_x = k_y = \sqrt{2}/2$ shown at times $t = 0.0$ (*left*), $t = 1.0$ (*center*), and $t = 2.0$ (*right*). Contour levels are 0.2, 0.4, 0.6, 0.8

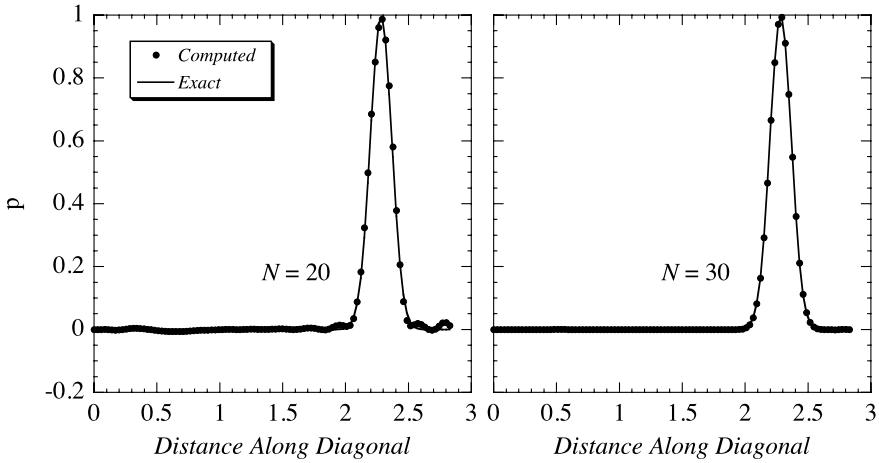

Fig. 5.10 Computed and exact values of the pressure interpolated at 100 points along the line $y = x$ for the plane wave shown in Fig. 5.9 at time $t = 2$

boundary on the right. We used the external state to be the exact solution (derived with the method of images) except along the right boundary. At the right boundary we used the external state specified by (5.168).

5.4.4 Benchmark Solution: Propagation of a Circular Sound Wave

A more challenging problem in two space dimensions for many numerical approximations to the wave equation is to propagate a circular sound wave. Anisotropy in the numerical wave speeds usually distorts the wave badly as it propagates. The circular wave problem is an excellent one with which to see the effects of anisotropy.

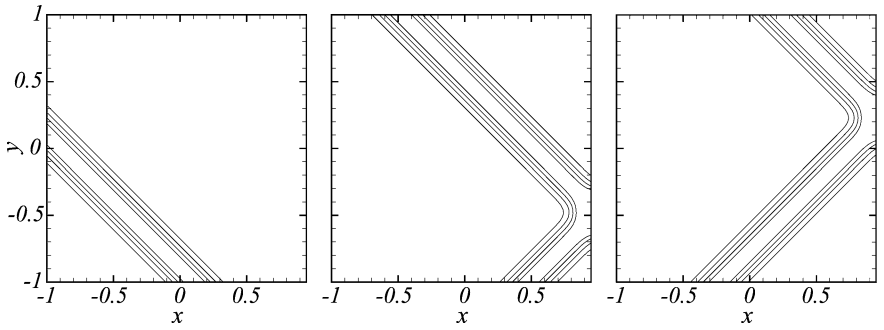

Fig. 5.11 Reflection of a plane Gaussian wave off the right wall boundary using the nodal discontinuous Galerkin approximation with $N = 40$ and $k_x = k_y = \sqrt{2}/2$. Contours are shown at times $t = 0.5$ (*left*), $t = 1.5$ (*center*), and $t = 2.0$ (*right*). Contour levels are 0.2, 0.4, 0.6, 0.8

The benchmark solution that we present now is to solve the wave equation with the initial condition

$$\begin{aligned} p(x, y, 0) &= \exp\left[-\ln 2\left(\frac{x^2 + y^2}{0.06^2}\right)\right], \\ u(x, y, 0) &= v(x, y, 0) = 0. \end{aligned} \tag{5.170}$$

The exact solution to the wave equation with this initial condition is found in polar coordinates by separation of variables. The pressure as a function of distance from the origin, $r = \sqrt{x^2 + y^2}$, and time is

$$p(x, y, t) = -\int_0^\infty \frac{e^{-\omega^2/4b}}{2b} \omega J_0(r\omega) \cos(\omega t) d\omega, \tag{5.171}$$

where J_0 is the Bessel function of the first kind of order zero, and $b = \ln 2/w^2$ with $w = 0.06$.

We present solutions for the propagating circular wave at time $t = 0.7$ in Figs. 5.12 and 5.13. The solutions were computed with 70 points in each direction and a time step of $\Delta t = 8.75 \times 10^{-4}$. With this number of points, the initial Gaussian for the pressure is resolved by about eight points in each direction. We interpolated the solutions to 100 points in each direction for the plots by Algorithm 35 (2DCoarseToFineInterpolation). Figure 5.12 shows contours of the pressure, which illustrates that the circular shape of the wave is retained. Figure 5.13 allows the comparison of the exact and computed solutions along the line $y = x$.

Exercises

5.1 Derive a collocation approximation for

$$\varphi_{xx} + \varphi_{xy} + \varphi_{yy} = f.$$

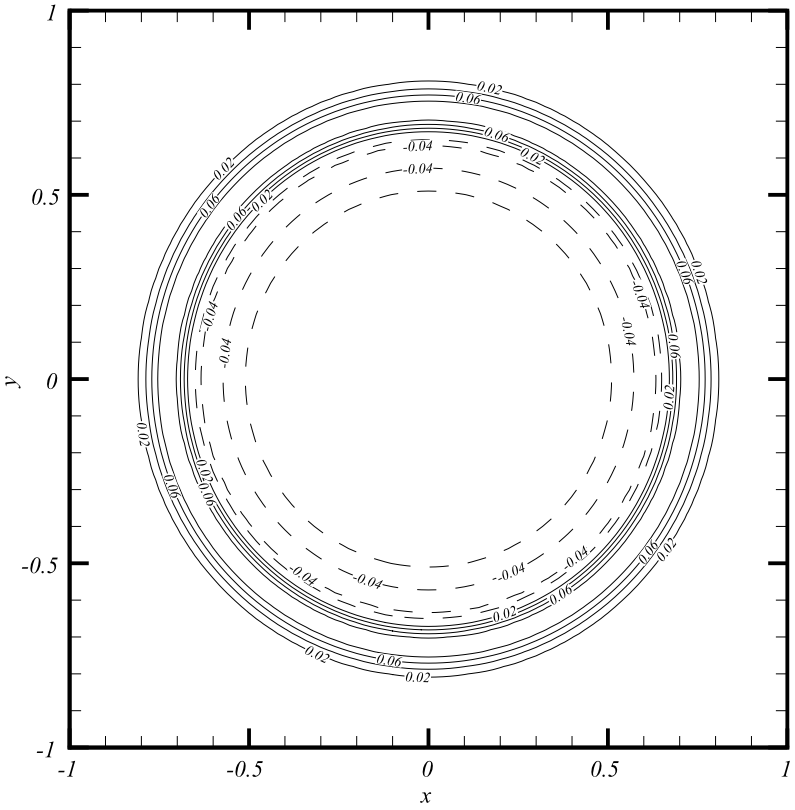

Fig. 5.12 Computed pressure contours at time $t = 0.7$ for a propagating circular wave for $N = M = 70$ and $\Delta t = 8.75 \times 10^{-4}$ interpolated to 100 uniformly spaced points in each direction

Cross derivatives appear when considering problems in curvilinear coordinates.

5.2 Repeat Problem 5.1 for the nodal Galerkin approximation.

5.3 The steady solution of the advection-diffusion equation can be computed by letting time go to infinity for the time dependent equation or by solving the steady-state equation

$$\mathbf{q} \cdot \nabla \varphi - \nabla^2 \varphi = f$$

directly. Derive the collocation approximation to the steady-state advection-diffusion equation for Dirichlet boundary conditions. What iterative method should be used to solve the system of equations? (Unfortunately, preconditioning for the advective term is problematic. See [7].)

5.4 Repeat Problem 5.3 for the nodal Galerkin approximation.

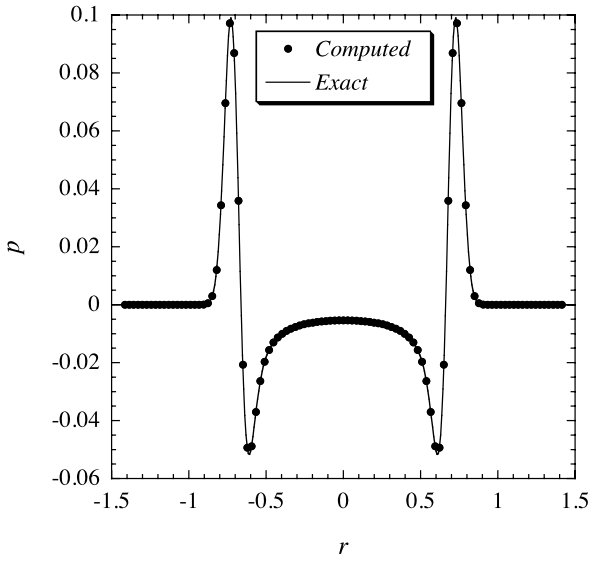

Fig. 5.13 Comparison of the computed circular wave pressure interpolated to 100 points with the exact solution along the line $y = x$ at $t = 0.7$

5.5 Modify Algorithms 63 (Nodal2DStorage) and 65 (Construct) to compute the Chebyshev collocation approximation using Algorithm 40 (FastChebyshevDerivative) instead of matrix multiplication to compute the spatial derivatives.

5.6 Extend the NodalPotentialClass to solve variable diffusion coefficient problems, approximated by (5.27), for $\nu = \nu(x, y, \varphi)$.

5.7 Show that the coefficient matrix for the Laplace operator, (5.32), is not symmetric.

5.8 A thin, rectangular plate shown in Fig. 5.14 is kept at a fixed temperature along its edges and is allowed to radiate through its surface. When suitably scaled, the steady temperature distribution satisfies the equation

$$\nabla^2 \varphi = \gamma^2 (\varphi - \varphi_0),$$

where γ^2 is a constant that is inversely proportional to the thermal resistance of the material.

1. Derive the collocation approximation for the problem.
2. Compute the solution of the collocation approximation, plot its contours, and compare to the exact solution

$$\varphi = \frac{2}{a} \sum_{n=1}^{\infty} \frac{\sin(\frac{n\pi x}{a}) \sinh[(b-y)(\gamma^2 + \frac{n^2\pi^2}{a^2})^{1/2}]}{\sinh[b(\gamma^2 + \frac{n^2\pi^2}{a^2})^{1/2}]} \int_0^a f(x) \sin\left(\frac{n\pi x}{a}\right) dx$$

for $f(x) = e^{-(x-a/2)^2}$.

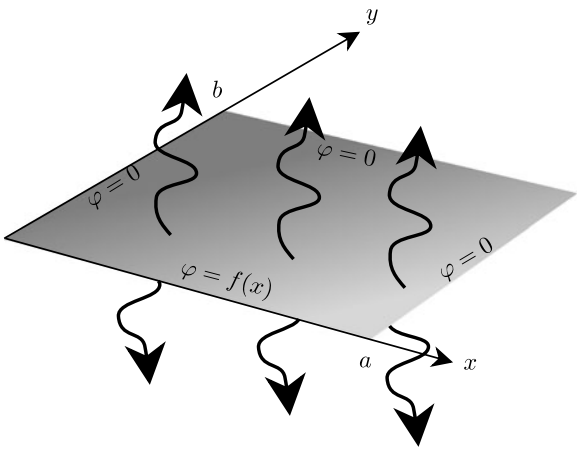

Fig. 5.14 Geometry and boundary conditions for steady temperatures on a thin plate with radiation

3. Compare the contours for several values of γ^2 and to those of a fully insulated plate.

5.9 Redo Problem 5.8 with the nodal Galerkin method.

5.10 For wave reflection at a straight boundary, the angle of incidence is equal to the angle of reflection. This appears to be true in Fig. 5.11. Compute the benchmark problem of Sect. 5.4.3 for various angles of incidence and find the range of angles over which the angle of reflection is accurate.

5.11 Typical rules of thumb for the number of points per wavelength needed to propagate sinusoidal waves accurately with finite difference approximations are 32 points per wavelength for second order methods and eight points per wavelength for fourth order methods. Multiply the exponential factor in (5.169) by a sinusoidal factor $\sin(\omega(k_x(x-x_0)+k_y(y-y_0)-ct))$ and choose the frequency ω so that there is at least one wavelength fully represented across the Gaussian envelope. Experiment with the discontinuous Galerkin method to find the number of points per wavelength needed to propagate the pulse accurately. In practice, polynomial spectral methods need only an average of 4–5 average points per wavelength.

5.12 If a wall boundary is placed along one of the boundaries in the benchmark problem of Sect. 5.4.4, the method of images can be used to create the exact solution from (5.171). Compute the solution with a single wall to study how well a circular wave is reflected from a straight wall.

Chapter 6
Transformation Methods from Square to Non-Square Geometries

The first step to extend spectral methods to complex geometries is to derive methods for quadrilateral domains with curved sides. The approach most commonly used is to create a transformation from the domain on which we want the solution to the square. In common terminology, the original domain is called the *physical domain*. The square is the *computational domain*. After we map the domain and transform the equations to incorporate that mapping, we apply techniques that we developed in the previous chapter to the equations on the square.

6.1 Mappings and Coordinate Transformations

To use the spectral methods that we developed in the previous chapter, we create an algebraic map $\mathbf{x} = (x, y) = \mathbf{X}(\xi, \eta)$ between the *physical space coordinates* (x, y) and *computational space coordinates* (ξ, η). In practice, the mapping will take a point in the *reference square* $[-1, 1] \times [-1, 1]$ and transform it to a point in the physical domain, as we sketch in Fig. 6.1.

Except in special cases, we will not have an explicit formula for what seems to be the most natural transformation, namely, from the physical domain to the reference square. An example where we do is the orthogonal mapping between the reference square and the quarter of the annulus $(r, \theta) \in [1, 3] \times [0, \pi/2]$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \mathbf{X}(\xi, \eta) = \begin{bmatrix} (2 + \xi) \cos(\pi (\eta + 1)/4) \\ (2 + \xi) \sin(\pi (\eta + 1)/4) \end{bmatrix}. \tag{6.1}$$

Orthogonal analytic mappings are known for some other special geometries, e.g. giving cylindrical or elliptical coordinate systems. In other situations, we could derive conformal mappings. But in general, for simply connected domains like quadrilateral domains without holes, we derive an algebraic transformation that at least guarantees that the boundaries of the reference square are mapped onto the boundaries of the physical domain.

Algebraic techniques are commonly used with spectral methods to transform from a square to a general quadrilateral domain. The techniques assume that the boundaries of the quadrilateral domain are smooth, except at the four corners. It is possible to generate smooth mappings when the boundaries are not smooth through the solution of an elliptic system of partial differential equations, but we will not cover those methods here. The topic of mapping domains is an important one in grid generation. For further study, the books [15] and [11] can provide starting points.

D.A. Kopriva, *Implementing Spectral Methods for Partial Differential Equations,* Scientific Computation,

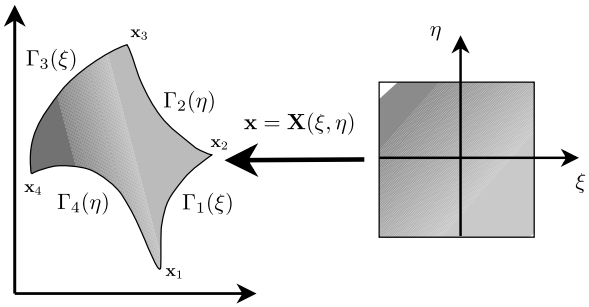

Fig. 6.1 Mapping of a general quadrilateral to the reference square

6.1.1 Mapping a Straight Sided Quadrilateral

To motivate the algebraic approach to mapping, we map a square to a straight-sided quadrilateral whose four corners are $\{\mathbf{x}_j\}_{j=1}^4$ numbered counter clockwise as shown in Fig. 6.1. Although such a simple physical domain does not seem to be a particularly big extension from a square domain, it will prove useful for spectral element approximations that we derive in Chap. 8, since most mesh generation programs will generate straight sided elements.

The creation of the mapping from the unit square to a straight sided quadrilateral is nothing but an interpolation problem. The problem is to find a function that is linear in ξ and η and takes on the values of the corners of the physical domain at the corners of the reference square. In fact, this is exactly the bilinear interpolation problem that we used in Sect. 5.2.2.3, to develop a bilinear finite element preconditioner. This time, instead of interpolating the solution that takes on particular values at the corners, we interpolate the location bilinearly between the four corners of the quadrilateral. Since the mapping we used previously started from the unit square, we will start here with a mapping from the unit square with coordinate $(\tilde{\xi}, \tilde{\eta})$, and transform the result afterwards to the reference square with coordinates (ξ, η).

Using the notation of Sect. 5.2.2.3, the bi-linear interpolation between the unit square and the quadrilateral is

$$\mathbf{X}(\tilde{\xi}, \tilde{\eta}) = \mathbf{x}_1\psi_{0,0} + \mathbf{x}_2\psi_{1,0} + \mathbf{x}_3\psi_{1,1} + \mathbf{x}_4\psi_{0,1}. \tag{6.2}$$

If we write the basis functions explicitly,

$$\mathbf{X}(\tilde{\xi}, \tilde{\eta}) = \mathbf{x}_1(1-\tilde{\xi})(1-\tilde{\eta}) + \mathbf{x}_2\tilde{\xi}(1-\tilde{\eta}) + \mathbf{x}_3\tilde{\xi}\tilde{\eta} + \mathbf{x}_4(1-\tilde{\xi})\tilde{\eta}. \tag{6.3}$$

Clearly, this mapping takes on the four corner values of the physical domain at the four corners of the unit square, that is, $\mathbf{X}(0, 0) = \mathbf{x}_1$, etc., and is linear in each of the coordinate directions.

We write the transform from the reference square to the quadrilateral through another transformation. To map from the reference square to the unit square, we

Algorithm 95: *QuadMap:* Mapping of the Reference Square to a Straight Sided Quadrilateral

Procedure *QuadMap*
Input: $\{(x_j, y_j)\}_{j=1}^4, \xi, \eta$
$x \leftarrow \frac{1}{4}\{x_1(1-\xi)(1-\eta) + x_2(\xi+1)(1-\eta) + x_3(\xi+1)(1+\eta) + x_4(1-\xi)(\eta+1)\}$
$y \leftarrow \frac{1}{4}\{y_1(1-\xi)(1-\eta) + y_2(\xi+1)(1-\eta) + y_3(\xi+1)(1+\eta) + y_4(1-\xi)(\eta+1)\}$
return (x, y)
End Procedure *QuadMap*

introduce the affine map

$$\tilde{\xi} = \frac{\xi+1}{2}, \qquad \tilde{\eta} = \frac{\eta+1}{2}. \tag{6.4}$$

When we substitute this transformation, we get the map from the reference square to the quadrilateral

$$\begin{aligned}\mathbf{X}(\xi,\eta) = \frac{1}{4}\{&\mathbf{x}_1(1-\xi)(1-\eta) + \mathbf{x}_2(\xi+1)(1-\eta) \\ &+ \mathbf{x}_3(\xi+1)(1+\eta) + \mathbf{x}_4(1-\xi)(\eta+1)\}.\end{aligned} \tag{6.5}$$

Implementation of the mapping from the unit square to a straight sided quadrilateral is straightforward. Algorithm 95 (QuadMap) takes the four corner points and returns the location (x, y) in physical space for any point (ξ, η) in the reference square.

6.1.2 How to Approximate Curved Boundaries

The more general situation occurs when the physical domain is bounded by four curves, $\mathbf{\Gamma}_j$, $j = 1, 2, 3, 4$ as shown in Fig. 6.1. These curves might be represented by an analytical formula or by an interpolation. As a general rule, we will represent the curves as a polynomial interpolant of the same order as the spectral solution approximations,

$$I_N \Gamma(s) = \sum_{k=0}^{N} \Gamma(s_k)\, \ell_k(s) \tag{6.6}$$

known as an *isoparametric* approximation. A polynomial representation is convenient since we will need derivatives of the boundary curves after we transform the equations later in Sect. 6.2. The polynomial approximation of the boundaries enables a general representation of the transformation of the equations; The isoparametric representation means that the boundary is approximated to the same polynomial order as the solution is.

Algorithm 96: *CurveInterpolant:* A Curve Interpolant Class Definition

Class CurveInterpolant
Data:
$N, \{nodes_j\}_{j=0}^{N}, \{(x_j, y_j)\}_{j=0}^{N}, \{w_j\}_{j=0}^{N}$
Procedures:
Construct$(N, \{nodes_j,\}_{j=0}^{N}, \{(x_j, y_j)\}_{j=0}^{N})$; `// Algorithm 97`
EvaluateAt(s) ; `// Algorithm 97`
DerivativeAt(s) ; `// Algorithm 97`
End Class CurveInterpolant

From the implementation point of view, it is useful to represent a curve as an instance of a class that represents an interpolant, such as we show Algorithm 96 (CurveInterpolant). The minimum data that we need for such a class are the nodes and the values at those nodes. The methods in this class are the constructor and a function that returns the location of the curve for a given value of the parameter. Later we will want the derivative of the curve with respect to the parameter, so we include it now in the class definition. The array w stores the barycentric weights that we defined in Sect. 3.4.

We show the methods to be implemented by the class in Algorithm 97 (CurveInterpolantProcedures). For convenience, we have the constructor for the CurveInterpolant make a copy of the input data. It then computes the barycentric weights for use later to evaluate the interpolant and its derivatives. The methods *EvaluateAt* and *DerivativeAt* use the Lagrange interpolation methods that we developed in Sects. 3.4 and 3.5.1.

An issue is how to parametrize the boundary curves. As a general rule, it is best to parametrize them according to arc length, or a reasonable approximation thereto, along the curve. We illustrate the difference the choice of parametrization can make in Fig. 6.2, which shows how well equally spaced points in the parameter map along a circular arc. Clearly the parametrization in x does not adequately represent the arc, while the parametrization in angle (proportional to arc length in this case) does.

Parametrization according to arc length has been a subject of research in the Computer Aided Design (CAD) field for many years, where it is important to draw accurate representations of curves quickly. Here we will outline a procedure that is not optimized for speed, and hence is best used as part of a pre-processing stage of a computation.

To explain how to (re-)parametrize a curve, let's assume that it is originally defined by some function of a parameter, $\mathbf{x}(t)$. The goal is to find the relationship between the parameter t and a new parameter s that varies linearly with the arc length. Since we assume that the boundaries will be parametrized by an argument that varies between negative one and one, we will take s to be the arc length divided by the total arc length, L, mapped to $[-1, 1]$. Once we have the relationship between s and t, we can compute the interpolant through values $\mathbf{x}(t(s))$ for points along the curve, typically the Chebyshev Gauss-Lobatto points, (1.130).

Algorithm 97: *CurveInterpolantProcedures*:

Procedure Construct
Input: $N, \{nodes_j, \}_{j=0}^{N}, \{(x_j, y_j)\}_{j=0}^{N}$,
Uses Algorithms:
Algorithm 30 (BarycentricWeights)

$this.N \leftarrow N$
for $j = 0$ **to** N **do**
  $this.nodes_j \leftarrow nodes_j$
  $this.x_j \leftarrow x_j$
  $this.y_j \leftarrow y_j$
end
$\{this.w_j\}_{j=0}^{N} \leftarrow BarycentricWeights(\{nodes_j\}_{j=0}^{N})$
End Procedure Construct

Procedure EvaluateAt
Input: s
Uses Algorithms:
Algorithm 31 (LagrangeInterpolation)
$x \leftarrow LagrangeInterpolation(s, this.\{nodes_j\}_{j=0}^{N}, this.\{x_j\}_{j=0}^{N}, this.\{w_j\}_{j=0}^{N})$
$y \leftarrow LagrangeInterpolation(s, this.\{nodes_j\}_{j=0}^{N}, this.\{y_j\}_{j=0}^{N}, this.\{w_j\}_{j=0}^{N})$
return (x, y)
End Procedure EvaluateAt

Procedure DerivativeAt
Input: s
Uses Algorithms:
Algorithm 36 (LagrangeInterpolantDerivative)
$x' \leftarrow LagrangeInterpolantDerivative(s, this.\{nodes_j\}_{j=0}^{N}, this.\{x_j\}_{j=0}^{N}, this.\{w_j\}_{j=0}^{N})$
$y' \leftarrow LagrangeInterpolantDerivative(s, this.\{nodes_j\}_{j=0}^{N}, this.\{y_j\}_{j=0}^{N}, this.\{w_j\}_{j=0}^{N})$
return (x', y')
End Procedure DerivativeAt

Given the parametric representation of a curve, the arc length from a location t_0 to a location t is

$$s - s_0 = \int_{t_0}^{t} \sqrt{(x'(z))^2 + (y'(z))^2} dz. \tag{6.7}$$

If the analytical derivatives of the curve are not easily found, we could use a finite difference approximation of them. The total arc length of the curve is just

$$L = \int_{t_0}^{t_f} \sqrt{(x'(z))^2 + (y'(z))^2} dz. \tag{6.8}$$

We could compute the total arc length with a standard Newton-Cotes composite quadrature rule, such as the Trapezoidal or Simpson Rules.

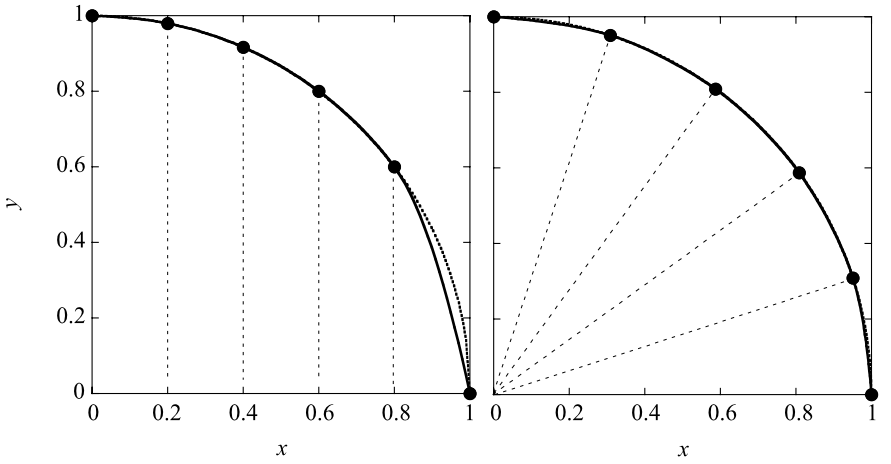

Fig. 6.2 Two parameterizations of a circle. *On the left* is uniform parametrization in x. *On the right* is uniform parametrization in arc length. *Circles* are interpolation nodes. *The solid line* is the fifth order interpolation through the nodes and *the dotted line* is the exact circle

The problem of finding the relationship between the arc length then reduces to one of finding t such that

$$s = \frac{1}{L}\int_{t_0}^{t} \sqrt{(x'(z))^2 + (y'(z))^2}\,dz. \tag{6.9}$$

Specifically, given a set of interpolation nodes $s_j,\ j = 0, 1, \ldots, N$, with $s_0 = -1$ and $s_N = 1$, we seek t_j such that

$$s_j = \frac{1}{L}\int_{t_0}^{t_j} \sqrt{(x'(z))^2 + (y'(z))^2}\,dz - 1. \tag{6.10}$$

Since we know t_0 for s_0 as the starting point of the curve, we can re-write the problem as a rootfinding problem

$$s_j - s_{j-1} - \frac{1}{L}\int_{t_{j-1}}^{t_j} \sqrt{(x'(z))^2 + (y'(z))^2}\,dz = 0 \tag{6.11}$$

for t_j, $j = 1, 2, \ldots, N-1$. Therefore, we can compute the arclength parametrization by computing the integral by a quadrature rule, with the derivatives approximated by finite differences if the exact values are not available, and Newton's method to solve the rootfinding problem.

6.1.3 How to Map the Reference Square to a Curved-Sided Quadrilateral

The commonly used method to create a transformation between the quadrilateral bounded by the four curves and the reference square is called *transfinite interpolation*. Specifically, *transfinite interpolation with linear blending* is most often used with spectral methods. We create this transformation by the linear interpolation between the four curves that represent the physical boundary. We will again develop the map from the unit square to the physical domain first, and then incorporate the affine transformation (6.4) to the reference square.

To derive the mapping, we first create a linear interpolation between two opposing curves, say Γ_2 and Γ_4 in Fig. 6.1

$$\mathbf{X}_{42}\left(\tilde{\xi},\tilde{\eta}\right)=\left(1-\tilde{\xi}\right)\Gamma_4\left(\tilde{\eta}\right)+\tilde{\xi}\,\Gamma_2\left(\tilde{\eta}\right). \tag{6.12}$$

This interpolation matches the two entire boundaries $\mathbf{\Gamma}_4$ and $\mathbf{\Gamma}_2$ for $\tilde{\xi}=0$ and $\tilde{\xi}=1$. If the other two sides of the domain are straight, we're done. If not, we need to incorporate those. The linear interpolation between the other two sides is,

$$\mathbf{X}_{13}\left(\tilde{\xi},\tilde{\eta}\right)=\left(1-\tilde{\eta}\right)\mathbf{\Gamma}_1\left(\tilde{\xi}\right)+\tilde{\eta}\mathbf{\Gamma}_3\left(\tilde{\xi}\right). \tag{6.13}$$

The final result will be a combination of the two interpolations, starting with the sum

$$\mathbf{\Sigma}\left(\tilde{\xi},\tilde{\eta}\right)=\left(1-\tilde{\xi}\right)\mathbf{\Gamma}_4\left(\tilde{\eta}\right)+\tilde{\xi}\mathbf{\Gamma}_2\left(\tilde{\eta}\right)+\left(1-\tilde{\eta}\right)\mathbf{\Gamma}_1\left(\tilde{\xi}\right)+\tilde{\eta}\mathbf{\Gamma}_3\left(\tilde{\xi}\right). \tag{6.14}$$

The combination no longer matches the boundaries, in general. For instance,

$$\begin{aligned}\mathbf{\Sigma}(0,\tilde{\eta})&=\mathbf{\Gamma}_4(\tilde{\eta})+\{(1-\tilde{\eta})\mathbf{\Gamma}_1(0)+\tilde{\eta}\mathbf{\Gamma}_3(0)\},\\ \mathbf{\Sigma}(1,\tilde{\eta})&=\mathbf{\Gamma}_2(\tilde{\eta})+\{(1-\tilde{\eta})\mathbf{\Gamma}_1(1)+\tilde{\eta}\mathbf{\Gamma}_3(1)\}.\end{aligned} \tag{6.15}$$

To match the boundaries, we need to subtract the linear interpolant of the additional terms that appear in the braces of (6.15),

$$\left(1-\tilde{\xi}\right)\{(1-\tilde{\eta})\mathbf{\Gamma}_1(0)+\tilde{\eta}\mathbf{\Gamma}_3(0)\}+\tilde{\xi}\,\{(1-\tilde{\eta})\mathbf{\Gamma}_1(1)+\tilde{\eta}\mathbf{\Gamma}_3(1)\}. \tag{6.16}$$

Subtraction of the correction term gives the transfinite map between the unit square and the physical domain

$$\begin{aligned}\mathbf{X}\left(\tilde{\xi},\tilde{\eta}\right)=&\left(1-\tilde{\xi}\right)\Gamma_4(\tilde{\eta})+\tilde{\xi}\,\Gamma_2(\tilde{\eta})+\left(1-\tilde{\eta}\right)\Gamma_1\left(\tilde{\xi}\right)+\tilde{\eta}\Gamma_3\left(\tilde{\xi}\right)\\ &-\left(1-\tilde{\xi}\right)\{(1-\tilde{\eta})\,\Gamma_1(0)+\tilde{\eta}\Gamma_3(0)\}\\ &-\tilde{\xi}\,\{(1-\tilde{\eta})\,\Gamma_1(1)+\tilde{\eta}\Gamma_3(1)\}.\end{aligned} \tag{6.17}$$

Algorithm 98: *TransfiniteQuadMap:* Mapping of the Reference Square to a Curve-Bounded Quadrilateral

Procedure *TransfiniteQuadMap*
Input: ξ, η
Input: $\{\Gamma_j\}_{j=1}^{4}$; `// Of type CurveInterpolant`
Uses Algorithms:
Algorithm 96 (CurveInterpolant)
Algorithm 97 (CurveInterpolantProcedures)

$(x_1, y_1) \leftarrow \Gamma_1.EvaluateAt(-1)$
$(x_2, y_2) \leftarrow \Gamma_1.EvaluateAt(1)$
$(x_3, y_3) \leftarrow \Gamma_3.EvaluateAt(1)$
$(x_4, y_4) \leftarrow \Gamma_3.EvaluateAt(-1)$
$(X_1, Y_1) \leftarrow \Gamma_1.EvaluateAt(\xi)$
$(X_2, Y_2) \leftarrow \Gamma_2.EvaluateAt(\eta)$
$(X_3, Y_3) \leftarrow \Gamma_3.EvaluateAt(\xi)$
$(X_4, Y_4) \leftarrow \Gamma_4.EvaluateAt(\eta)$

$$x = \frac{1}{2}\left[(1-\xi)\,X_4 + (1+\xi)\,X_2 + (1-\eta)\,X_1 + (1+\eta)\,X_3\right]$$
$$- \frac{1}{4}\left[(1-\xi)\,\{(1-\eta)\,x_1 + (1+\eta)\,x_4\} + (1+\xi)\,\{(1-\eta)\,x_2 + (1+\eta)\,x_3\}\right]$$
$$y = \frac{1}{2}\left[(1-\xi)\,Y_4 + (1+\xi)\,Y_2 + (1-\eta)\,Y_1 + (1+\eta)\,Y_3\right]$$
$$- \frac{1}{4}\left[(1-\xi)\,\{(1-\eta)\,y_1 + (1+\eta)\,y_4\} + (1+\xi)\,\{(1-\eta)\,y_2 + (1+\eta)\,y_3\}\right]$$

return (x, y)
End Procedure *TransfiniteQuadMap*

We get the final mapping function when we apply the affine transformation from the unit to reference square (6.4)

$$\mathbf{X}(\xi,\eta) = \frac{1}{2}\left[(1-\xi)\,\Gamma_4(\eta) + (1+\xi)\,\Gamma_2(\eta) + (1-\eta)\,\Gamma_1(\xi) + (1+\eta)\,\Gamma_3(\xi)\right]$$
$$- \frac{1}{4}\left[(1-\xi)\,\{(1-\eta)\,\Gamma_1(-1) + (1+\eta)\,\Gamma_3(-1)\}\right.$$
$$\left. + (1+\xi)\,\{(1-\eta)\,\Gamma_1(1) + (1+\eta)\,\Gamma_3(1)\}\right]. \tag{6.18}$$

We easily construct the transfinite mapping with the CurveInterpolants that we introduced in the previous section. Algorithm 98 (TransfiniteQuadMap) takes the four curves as input and the location in computational space. It returns the corresponding location in physical space. The algorithm takes into account the fact that the transfinite map (6.18) needs the four curves to be evaluated at the four corners and at four locations along the curves.

The transformations from the reference square to the physical domain can be orthogonal or non-orthogonal. To illustrate, we show in Fig. 6.3 the results of two

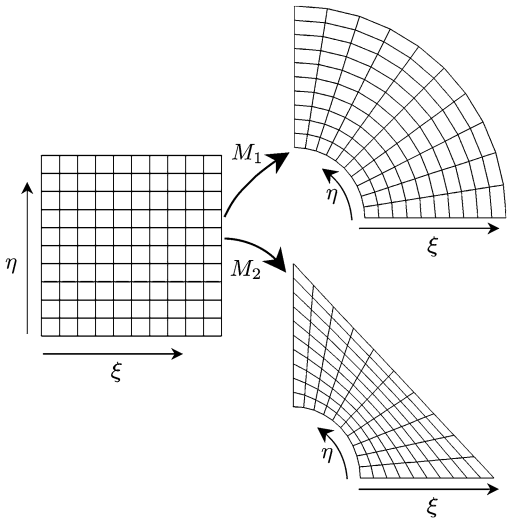

Fig. 6.3 Two mappings from the reference square

transformations, M_1 and M_2 given by

$$M_1 \quad \begin{cases} \mathbf{\Gamma}_1(\xi) = (2+\xi)\,\hat{x}, \\ \mathbf{\Gamma}_2(\eta) = 3\cos(\pi(\eta+1)/4)\,\hat{x} + 3\sin(\pi(\eta+1)/4)\,\hat{y}, \\ \mathbf{\Gamma}_3(\xi) = (2+\xi)\,\hat{y}, \\ \mathbf{\Gamma}_4(\eta) = \cos(\pi(\eta+1)/4)\,\hat{x} + \sin(\pi(\eta+1)/4)\,\hat{y} \end{cases} \tag{6.19}$$

and

$$M_2 \quad \begin{cases} \mathbf{\Gamma}_1(\xi) = (2+\xi)\,\hat{x}, \\ \mathbf{\Gamma}_2(\eta) = 3(1-\eta)/2\hat{x} + 3(1+\eta)/2\hat{y}, \\ \mathbf{\Gamma}_3(\xi) = (2+\xi)\,\hat{y}, \\ \mathbf{\Gamma}_4(\eta) = \cos(\pi(\eta+1)/4)\,\hat{x} + \sin(\pi(\eta+1)/4)\,\hat{y}. \end{cases} \tag{6.20}$$

We see that M_1 produces an orthogonal grid on the mapped domain, but that M_2 does not.

6.2 Transformation of Equations under Mappings

We introduced a mapping from the square computational space to physical space because we can already use spectral methods on the square. Under mappings the equations themselves are transformed, essentially by result of the chain rule. In this section, we describe a general approach to transform equations under mappings.

Formally, we find the effect of a mapping by use of the chain rule. For instance, given a map $\mathbf{x} = \mathbf{X}(\xi, \eta)$, derivatives of a function $u(x, y)$ transform as

$$\begin{aligned} \frac{\partial u}{\partial x} &= \frac{\partial u}{\partial \xi}\frac{\partial \xi}{\partial x} + \frac{\partial u}{\partial \eta}\frac{\partial \eta}{\partial x}, \\ \frac{\partial u}{\partial y} &= \frac{\partial u}{\partial \xi}\frac{\partial \xi}{\partial y} + \frac{\partial u}{\partial \eta}\frac{\partial \eta}{\partial y}. \end{aligned} \tag{6.21}$$

Unfortunately, we typically know the transformation from the reference square to physical space, $\mathbf{x} = \mathbf{X}(\xi, \eta)$, like those we derived in the previous section. The derivatives ξ_x, ξ_y, etc. require the inverse of the transformation, which we don't know in general, nor is it often practical to find it by inverting the original mapping function. We usually compute the values by recognizing that we can write the transformation of the derivatives in matrix-vector form,

$$\begin{bmatrix} \dfrac{\partial u}{\partial x} \\ \dfrac{\partial u}{\partial y} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial \xi}{\partial x} & \dfrac{\partial \eta}{\partial x} \\ \dfrac{\partial \xi}{\partial y} & \dfrac{\partial \eta}{\partial y} \end{bmatrix} \begin{bmatrix} \dfrac{\partial u}{\partial \xi} \\ \dfrac{\partial u}{\partial \eta} \end{bmatrix}. \tag{6.22}$$

Similarly, the chain rule says

$$\begin{bmatrix} \dfrac{\partial u}{\partial \xi} \\ \dfrac{\partial u}{\partial \eta} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial x}{\partial \xi} & \dfrac{\partial y}{\partial \xi} \\ \dfrac{\partial x}{\partial \eta} & \dfrac{\partial y}{\partial \eta} \end{bmatrix} \begin{bmatrix} \dfrac{\partial u}{\partial x} \\ \dfrac{\partial u}{\partial y} \end{bmatrix}. \tag{6.23}$$

It follows that

$$\begin{bmatrix} \dfrac{\partial \xi}{\partial x} & \dfrac{\partial \eta}{\partial x} \\ \dfrac{\partial \xi}{\partial y} & \dfrac{\partial \eta}{\partial y} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial x}{\partial \xi} & \dfrac{\partial y}{\partial \xi} \\ \dfrac{\partial x}{\partial \eta} & \dfrac{\partial y}{\partial \eta} \end{bmatrix}^{-1}. \tag{6.24}$$

So we can compute the matrix, which is nothing but the Jacobian of the transformation, indirectly from the mapping by way of the matrix inverse.

Although the tack of using the Jacobian matrices is familiar, we will take a differential geometry approach to find how equations transform under mappings that extends well to three dimensions. In fact, it is most easily derived in three dimensions and then later simplified to two. The differential geometry approach will also make concepts like normals along curved boundaries straightforward to understand.

To begin the general study of the transformation of derivatives under a mapping, we introduce the spatial coordinates in physical space $\mathbf{x} = (x, y, z) = (x_1, x_2, x_3)$ and in computational space $\boldsymbol{\xi} = (\xi, \eta, \zeta) = (\xi^1, \xi^2, \xi^3)$. Both define curvilinear coordinate systems. Which is physical and which is computational is simply a matter of convention. We will assume, however, that we have a mapping $\mathbf{x} = \mathbf{X}(\xi, \eta, \zeta)$ that transforms the computational space to the physical space.

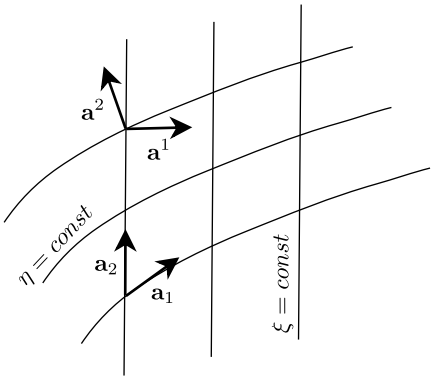

Fig. 6.4 Covariant and contravariant coordinate vectors in relation to the coordinate lines

We next introduce two (useful) types of basis vectors to represent directions in physical space. The first is the *covariant* basis $\mathbf{a}_i$, for $i = 1, 2, 3$, which varies along a coordinate line. The second is the *contravariant* basis, $\mathbf{a}^i$, which points normally to a coordinate line. We compare these two bases in two space dimensions in Fig. 6.4.

The covariant basis is defined to be tangent to a coordinate line, so it is the limit (see Fig. 6.4)

$$\mathbf{a}_i = \lim_{\Delta\xi^i \to 0} \frac{\Delta \mathbf{x}}{\Delta \xi^i} = \frac{\partial \mathbf{x}}{\partial \xi^i}, \quad i = 1, 2, 3. \tag{6.25}$$

The covariant basis vectors are the vectors that we can actually compute from the mapping function.

The contravariant basis vectors are normal to the coordinate lines and are defined by the gradients

$$\mathbf{a}^i = \nabla \xi^i, \quad i = 1, 2, 3. \tag{6.26}$$

Formally, it appears that we need the inverse of the transformation, $\boldsymbol{\xi} = X^{-1}(\mathbf{x})$ to compute the contravariant basis vectors.

The first step before we transform equations is to write how derivative operators transform under a mapping. First, we see that we can write a differential element in terms of the covariant basis vectors

$$d\mathbf{x} = \frac{\partial \mathbf{x}}{\partial \xi} d\xi + \frac{\partial \mathbf{x}}{\partial \eta} d\eta + \frac{\partial \mathbf{x}}{\partial \zeta} d\zeta = \sum_{i=1}^{3} \mathbf{a}_i d\xi^i. \tag{6.27}$$

The arc length differential element is the magnitude of this vector

$$(ds)^2 = |d\mathbf{x}|^2 = \sum_{i=1}^{3}\sum_{j=1}^{3} \mathbf{a}_i \cdot \mathbf{a}_j d\xi^i d\xi^j = \sum_{i=1}^{3}\sum_{j=1}^{3} g_{ij} d\xi^i d\xi^j, \tag{6.28}$$

where we have used the definition of the *covariant metric tensor* $g_{ij} \equiv \mathbf{a}_i \cdot \mathbf{a}_j = g_{ji}$ to simplify the sums.

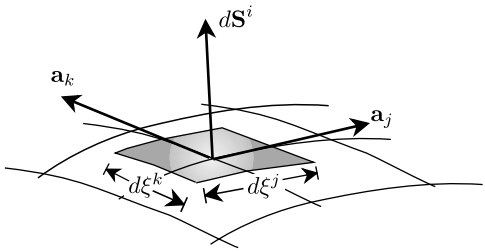

Fig. 6.5 Surface element vector computed from covariant basis vectors

Next, we define the surface area element, which we compute from the cross product (Fig. 6.5)

$$d\mathbf{S}^i = \mathbf{a}_j d\xi^j \times \mathbf{a}_k d\xi^k = (\mathbf{a}_j \times \mathbf{a}_k) d\xi^j d\xi^k. \tag{6.29}$$

Here, i, j, k are considered to be *cyclic*, that is, the ordering for (i, j, k) is $(1, 2, 3)$, $(2, 3, 1)$, etc. For instance, a surface element in Cartesian space with size Δx in the $\hat{x}$ direction and Δy in the $\hat{y}$ direction has the surface area element $dS^{(z)} = \Delta x \Delta y \hat{z}$ according to this relation.

Finally, the volume element extends the surface element in the normal direction

$$dV = \mathbf{a}_i \cdot (\mathbf{a}_j \times \mathbf{a}_k) d\xi^i d\xi^j d\xi^k. \tag{6.30}$$

In terms of the covariant metric tensor, a bit of vector algebra shows that the coefficient is

$$\mathbf{a}_i \cdot (\mathbf{a}_j \times \mathbf{a}_k) = \sqrt{\det(g)} = J, \tag{6.31}$$

giving the well-known result from calculus

$$dV = J d\xi^i d\xi^j d\xi^k. \tag{6.32}$$

We can now derive how different derivative operators transform under a mapping. First, recall that the divergence of a flux, $\mathbf{F}$, is defined as

$$\nabla \cdot \mathbf{F} = \lim_{\Delta V \to 0} \frac{1}{\Delta V} \int_{\partial \Delta V} \mathbf{F} \cdot d\mathbf{S}. \tag{6.33}$$

If we take the volume to be a differential pillbox, Fig. 6.6, the surface integrals are ((i, j, k) are always cyclic)

$$\sum_{i=1}^{3} \left\{ \mathbf{F} \cdot (\mathbf{a}_j \times \mathbf{a}_k) \Delta\xi^j \Delta\xi^k \big|^+ - \mathbf{F} \cdot (\mathbf{a}_j \times \mathbf{a}_k) \Delta\xi^j \Delta\xi^k \big|^- \right\}. \tag{6.34}$$

With $\Delta V = J \Delta\xi^i \Delta\xi^j \Delta\xi^k$,

$$\frac{1}{\Delta V} \int_{\partial V} \mathbf{F} \cdot d\mathbf{S} = \frac{1}{J} \sum_{i=1}^{3} \left\{ \frac{\mathbf{F} \cdot (\mathbf{a}_j \times \mathbf{a}_k) \Delta\xi^j \Delta\xi^k|^+ - \mathbf{F} \cdot (\mathbf{a}_j \times \mathbf{a}_k) \Delta\xi^j \Delta\xi^k|^-}{\Delta\xi^i} \right\}. \tag{6.35}$$

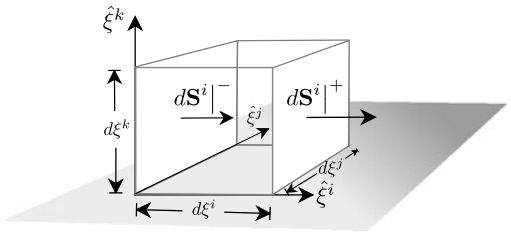

Fig. 6.6 Differential volume element for divergence

The limit as $\Delta V \to 0$ gives the divergence,

$$\nabla \cdot \mathbf{F} = \frac{1}{J} \sum_{i=1}^{3} \frac{\partial}{\partial \xi^i} \big(\mathbf{F} \cdot (\mathbf{a}_j \times \mathbf{a}_k) \big). \tag{6.36}$$

Equation (6.36) is known as the *conservative form* of the divergence because of its relation to the differential form of a conservation law.

We derive and alternative, called *nonconservative*, form by noting that the divergence is invariant under the transformation. Suppose that the flux is an arbitrary constant, $\mathbf{F} = \mathbf{c}$, then

$$\nabla \cdot \mathbf{F} = \frac{1}{J} \sum_{i=1}^{3} \frac{\partial}{\partial \xi^i} \big(\mathbf{c} \cdot (\mathbf{a}_j \times \mathbf{a}_k) \big) = 0. \tag{6.37}$$

Since the vector $\mathbf{c}$ is arbitrary, each coordinate contribution must individually satisfy the relations

$$\sum_{i=1}^{3} \frac{\partial}{\partial \xi^i} (\mathbf{a}_j \times \mathbf{a}_k) = 0 \tag{6.38}$$

known as *metric identities*. Notice that (6.38) says that the divergence in computational space of specific products of the covariant bases must vanish.

With the help of the metric identities, we rewrite the divergence in the nonconservative form

$$\nabla \cdot \mathbf{F} = \frac{1}{J} \sum_{i=1}^{3} (\mathbf{a}_j \times \mathbf{a}_k) \frac{\partial \mathbf{F}}{\partial \xi^i}. \tag{6.39}$$

We derive the representation of the gradient operator in computational space from the divergence operator. If we write out the terms in the nonconservative form of the divergence,

$$\frac{\partial F_1}{\partial x} + \frac{\partial F_2}{\partial y} + \frac{\partial F_3}{\partial z} = \nabla \cdot \mathbf{F} = \frac{1}{J} \sum_{i=1}^{3} (\mathbf{a}_j \times \mathbf{a}_k) \cdot \left(\frac{\partial F_1}{\partial \xi^i} \hat{x} + \frac{\partial F_2}{\partial \xi^i} \hat{y} + \frac{\partial F_3}{\partial \xi^i} \hat{z} \right), \tag{6.40}$$

we can match the terms on either side of the equation to get the nonconservative form of the gradient of a scalar, f,

$$\nabla f = \frac{1}{J}\sum_{i=1}^{3}(\mathbf{a}_j \times \mathbf{a}_k)\frac{\partial f}{\partial \xi^i}. \tag{6.41}$$

We use the metric identities, (6.38) again to derive the conservative form

$$\nabla f = \frac{1}{J}\sum_{i=1}^{3}\frac{\partial}{\partial \xi^i}\left[(\mathbf{a}_j \times \mathbf{a}_k) f\right]. \tag{6.42}$$

Note that all of the transformations of the differential operators are in terms of quantities that we can compute directly from the mapping, namely the covariant basis vectors, $\mathbf{a}_i$.

We use the gradient relations to find the contravariant basis vectors. To relate the contravariant and covariant basis vectors, we set $f = \xi^i$. Then the gradient of ξ^i is

$$\nabla \xi^i = \frac{1}{J}\sum_{l=1}^{3}(\mathbf{a}_j \times \mathbf{a}_k)\frac{\partial \xi^i}{\partial \xi^l}, \tag{6.43}$$

with (l, j, k) cyclic. Since $\partial \xi^i / \partial \xi^l = \delta_{i,l}$,

$$\nabla \xi^i = \mathbf{a}^i = \frac{1}{J}(\mathbf{a}_j \times \mathbf{a}_k), \tag{6.44}$$

or

$$J\mathbf{a}^i = \mathbf{a}_j \times \mathbf{a}_k. \tag{6.45}$$

Because we construct the transformations from the computational to the physical domains so that physical boundaries correspond to either a $\xi = const$ or $\eta = const$ line, we can compute boundary normals from the contravariant basis vectors. We need boundary normals to set Neumann or normal flux boundary conditions. Since the contravariant basis vectors are normal to a coordinate line, a normal in the positive direction of the ith coordinate variable is in the direction of $\mathbf{a}^i$. Therefore we use (6.43) to derive the normal in the direction of increasing ξ^i as

$$\hat{n}^i = \frac{|J|}{J}\frac{\mathbf{a}_j \times \mathbf{a}_k}{\|\mathbf{a}_j \times \mathbf{a}_k\|}, \tag{6.46}$$

where $\|\cdot\|$ denotes the Euclidean norm of the vector.

The relationship between the covariant and contravariant basis vectors allows us to rewrite the conservative forms of the differential operators that we have derived so far. For instance, we can write the divergence in the more compact form

$$\nabla \cdot \mathbf{F} = \frac{1}{J}\sum_{i=1}^{3}\frac{\partial}{\partial \xi^i}\left(J\mathbf{a}^i \cdot \mathbf{F}\right). \tag{6.47}$$

Similarly, the gradient becomes

$$\nabla f = \frac{1}{J}\sum_{i=1}^{3} J\mathbf{a}^i \frac{\partial f}{\partial \xi^i}. \tag{6.48}$$

Finally, the metric identities are equivalent to

$$\sum_{i=1}^{3} \frac{\partial J\mathbf{a}^i}{\partial \xi^i} = 0. \tag{6.49}$$

We derive the curl, div-grad and Laplace operators using the same sorts of arguments that we have used to derive the gradient operator, so we merely list them here. They are

$$\nabla \times \mathbf{F} = \frac{1}{J}\sum_{i=1}^{3} \frac{\partial}{\partial \xi^i}\left[J\mathbf{a}^i \times \mathbf{F}\right] = \mathbf{a}^i \times \frac{\partial \mathbf{F}}{\partial \xi^i}, \tag{6.50}$$

$$\nabla \cdot (\nu \nabla f) = \frac{1}{J}\sum_{i=1}^{3} \frac{\partial}{\partial \xi^i}\left(\nu \mathbf{a}^i \cdot \sum_{j=1}^{3} J\mathbf{a}^j \frac{\partial f}{\partial \xi^j}\right), \tag{6.51}$$

$$\nabla^2 f = \frac{1}{J}\sum_{i=1}^{3} \frac{\partial}{\partial \xi^i}\left(\mathbf{a}^i \cdot \sum_{j=1}^{3} J\mathbf{a}^j \frac{\partial f}{\partial \xi^j}\right). \tag{6.52}$$

Both the covariant and contravariant vectors form bases for three dimensional space, so we can express vectors as expansions in either basis. The *covariant components* of a vector, $\mathbf{F}$, are the projections onto the covariant basis,

$$F_i = \mathbf{a}_i \cdot \mathbf{F}, \tag{6.53}$$

while the *contravariant components* of the vector are the projections onto the contravariant basis vectors

$$F^i = \mathbf{a}^i \cdot \mathbf{F}. \tag{6.54}$$

We can then express the vector as an expansion in either basis

$$\mathbf{F} = \sum_{i=1}^{3} F_i \mathbf{a}_i = \sum_{i=1}^{3} F^i \mathbf{a}^i. \tag{6.55}$$

We conclude this section with the definition of the *contravariant metric tensor* $g^{ij} = \mathbf{a}^i \cdot \mathbf{a}^j = g^{ji}$. Since (6.45) implies that $\mathbf{a}_i \cdot \mathbf{a}^j = \delta_{i,j}$, the contravariant and covariant metric tensors are inverses of each other, i.e., $g^{ij} = (g_{ij})^{-1}$. We can write operators like the Laplacian compactly in terms of the contravariant metric tensor,

for

$$\nabla^2 f = \frac{1}{J}\sum_{i=1}^{3}\frac{\partial}{\partial \xi^i}\left(\mathbf{a}^i \cdot \sum_{j=1}^{3} J\mathbf{a}^j \frac{\partial f}{\partial \xi^j}\right)$$
$$= \frac{1}{J}\sum_{i=1}^{3}\frac{\partial}{\partial \xi^i}\left(\sum_{j=1}^{3} J\mathbf{a}^i \cdot \mathbf{a}^j \frac{\partial f}{\partial \xi^j}\right) = \frac{1}{J}\sum_{i=1}^{3}\frac{\partial}{\partial \xi^i}\left(\sum_{j=1}^{3} J g^{ij}\frac{\partial f}{\partial \xi^j}\right). \quad (6.56)$$

6.2.1 Two-Dimensional Forms

Since we are only really considering two-dimensional problems here, we should specialize the common differential operators to two dimensions. In two dimensions, the covariant basis vectors for a mapping $\mathbf{x} = \mathbf{X}(\xi, \eta) = X\hat{x} + Y\hat{y}$ are

$$\begin{aligned}\mathbf{a}_1 &= \frac{\partial \mathbf{X}}{\partial \xi} = X_\xi \hat{x} + Y_\xi \hat{y},\\ \mathbf{a}_2 &= \frac{\partial \mathbf{X}}{\partial \eta} = X_\eta \hat{x} + Y_\eta \hat{y},\\ \mathbf{a}_3 &= \hat{z}.\end{aligned} \quad (6.57)$$

We derive the contravariant bases from these by the relation (6.45) for $i = 1, 2$ and the Jacobian

$$J = \mathbf{a}_1 \cdot (\mathbf{a}_2 \times \mathbf{a}_3) = X_\xi Y_\eta - Y_\xi X_\eta. \quad (6.58)$$

When we evaluate the cross product directly we get

$$\begin{aligned}J\mathbf{a}^1 &= \mathbf{a}_2 \times \mathbf{a}_3 = Y_\eta \hat{x} - X_\eta \hat{y},\\ J\mathbf{a}^2 &= \mathbf{a}_3 \times \mathbf{a}_1 = -Y_\xi \hat{x} + X_\xi \hat{y}.\end{aligned} \quad (6.59)$$

We then compute boundary normals from (6.46)

$$\begin{aligned}\hat{n}^1 &= \frac{|J|}{J}\frac{Y_\eta \hat{x} - X_\eta \hat{y}}{\sqrt{Y_\eta^2 + X_\eta^2}},\\ \hat{n}^2 &= \frac{|J|}{J}\frac{-Y_\xi \hat{x} + X_\xi \hat{y}}{\sqrt{Y_\xi^2 + X_\xi^2}}.\end{aligned} \quad (6.60)$$

We write the conservative and nonconservative forms of the divergence using the contravariant basis vectors. The conservative form is

$$\nabla \cdot \mathbf{F} = \frac{1}{J}\left\{\frac{\partial}{\partial \xi}\left(F^1\right) + \frac{\partial}{\partial \eta}\left(F^2\right)\right\} \quad (6.61)$$

where

$$\begin{aligned} F^1 &= Y_\eta F_1 - X_\eta F_2, \\ F^2 &= -Y_\xi F_1 + X_\xi F_2. \end{aligned} \tag{6.62}$$

The nonconservative form is

$$\nabla \cdot \mathbf{F} = \frac{1}{J}\left\{ Y_\eta \left(F_1\right)_\xi - X_\eta \left(F_2\right)_\xi + X_\xi \left(F_2\right)_\eta - Y_\xi \left(F_1\right)_\eta \right\}. \tag{6.63}$$

The gradient reduces to

$$\begin{aligned} \nabla f &= \frac{1}{J}\left\{ \left(Y_\eta \hat{x} - X_\eta \hat{y}\right)\frac{\partial f}{\partial \xi} + \left(-Y_\xi \hat{x} + X_\xi \hat{y}\right)\frac{\partial f}{\partial \eta} \right\} \\ &= \frac{1}{J}\left\{ \left(Y_\eta \frac{\partial f}{\partial \xi} - Y_\xi \frac{\partial f}{\partial \eta} \right)\hat{x} + \left(-X_\eta \frac{\partial f}{\partial \xi} + X_\xi \frac{\partial f}{\partial \eta} \right)\hat{y} \right\} \end{aligned} \tag{6.64}$$

in two space dimensions.

We rewrite $\nabla \cdot (\nu \nabla f)$ using (6.61) and (6.64). From (6.64),

$$\begin{aligned} F_1 &= \frac{1}{J}\left(Y_\eta \frac{\partial f}{\partial \xi} - Y_\xi \frac{\partial f}{\partial \eta} \right), \\ F_2 &= \frac{1}{J}\left(-X_\eta \frac{\partial f}{\partial \xi} + X_\xi \frac{\partial f}{\partial \eta} \right). \end{aligned} \tag{6.65}$$

The contravariant fluxes are therefore

$$\begin{aligned} F^1 &= \frac{\nu Y_\eta}{J}\left(Y_\eta \frac{\partial f}{\partial \xi} - Y_\xi \frac{\partial f}{\partial \eta} \right) - \frac{\nu X_\eta}{J}\left(-X_\eta \frac{\partial f}{\partial \xi} + X_\xi \frac{\partial f}{\partial \eta} \right), \\ F^2 &= \frac{-\nu Y_\xi}{J}\left(Y_\eta \frac{\partial f}{\partial \xi} - Y_\xi \frac{\partial f}{\partial \eta} \right) + \frac{\nu X_\xi}{J}\left(-X_\eta \frac{\partial f}{\partial \xi} + X_\xi \frac{\partial f}{\partial \eta} \right). \end{aligned} \tag{6.66}$$

or, when we gather together the derivatives of f,

$$\begin{aligned} F^1 &= \nu \frac{Y_\eta^2 + X_\eta^2}{J} f_\xi - \nu \frac{Y_\xi Y_\eta + X_\xi X_\eta}{J} f_\eta, \\ F^2 &= -\nu \frac{Y_\xi Y_\eta + X_\xi X_\eta}{J} f_\xi + \nu \frac{Y_\xi^2 + X_\xi^2}{J} f_\eta. \end{aligned} \tag{6.67}$$

We get the Laplacian when $\nu = 1$.

As an example of mapping an equation in two space dimensions, let us write the Laplace operator in cylindrical coordinates. We know the transformation between (x, y) and $(r, \theta) \equiv (\xi, \eta)$ exactly as

$$\begin{bmatrix} x \\ y \end{bmatrix} = \mathbf{X} = \begin{bmatrix} \xi \cos(\eta) \\ \xi \sin(\eta) \end{bmatrix}. \tag{6.68}$$

The metric (transformation) derivatives of this mapping are

$$\begin{aligned} X_\xi &= \cos(\eta), \qquad & X_\eta &= -\xi \sin(\eta), \\ Y_\xi &= \sin(\eta), \qquad & Y_\eta &= \xi \cos(\eta), \end{aligned} \tag{6.69}$$

so the Jacobian of the transformation is

$$J = X_\xi Y_\eta - X_\eta Y_\xi = \xi. \tag{6.70}$$

When we make the substitutions for the metric derivatives, we find that the cross derivative terms in (6.67) exactly cancel

$$\begin{aligned} -\frac{Y_\eta}{J} Y_\xi \frac{\partial f}{\partial \eta} - \frac{X_\eta}{J} X_\xi \frac{\partial f}{\partial \eta} = 0, \\ -\frac{Y_\xi}{J} Y_\eta \frac{\partial f}{\partial \xi} - \frac{X_\xi}{J} X_\eta \frac{\partial f}{\partial \xi} = 0, \end{aligned} \tag{6.71}$$

as we should expect since both coordinate systems are orthogonal. When we combine the remaining terms we get the recognizable formula

$$\nabla^2 f = \frac{1}{\xi} \frac{\partial}{\partial \xi} \left(\xi \frac{\partial f}{\partial \xi} \right) + \frac{1}{\xi^2} \frac{\partial^2 f}{\partial \eta^2}. \tag{6.72}$$

6.3 How to Approximate the Metric Terms

To approximate the metric terms, $\partial X_i / \partial \xi^j$, $i, j = 1, 2$, let us assume that $\mathbf{X}(\xi, \eta)$ is defined by the transfinite mapping shown in (6.18) and evaluated using the procedure in Algorithm 98 (TransfiniteQuadMap). The derivatives of the mapping are

$$\begin{aligned} \frac{\partial \mathbf{X}}{\partial \xi} &= \frac{1}{2} \left\{ \boldsymbol{\Gamma}_2(\eta) - \boldsymbol{\Gamma}_4(\eta) + (1-\eta) \boldsymbol{\Gamma}_1'(\xi) + (1+\eta) \boldsymbol{\Gamma}_3'(\xi) \right\} \\ &\quad - \frac{1}{4} \left\{ (1-\eta) \left[\boldsymbol{\Gamma}_1(1) - \boldsymbol{\Gamma}_1(-1) \right] + (1+\eta) \left[\boldsymbol{\Gamma}_3(1) - \boldsymbol{\Gamma}_3(-1) \right] \right\}, \\ \frac{\partial \mathbf{X}}{\partial \eta} &= \frac{1}{2} \left\{ (1-\xi) \boldsymbol{\Gamma}_4'(\eta) + (1+\xi) \boldsymbol{\Gamma}_2'(\eta) + \boldsymbol{\Gamma}_3(\xi) - \boldsymbol{\Gamma}_1(\xi) \right\} \\ &\quad - \frac{1}{4} \left\{ (1-\xi) \left[\boldsymbol{\Gamma}_3(-1) - \boldsymbol{\Gamma}_1(-1) \right] + (1+\xi) \left[\boldsymbol{\Gamma}_3(1) - \boldsymbol{\Gamma}_1(1) \right] \right\}. \end{aligned} \tag{6.73}$$

Since we know how to compute the derivative of the boundary curves exactly using the procedure *DerivativeAt* in Algorithm 97 (CurveInterpolantProcedures), the computation of the metric terms is straightforward.

No matter how the metric terms are computed, it is important to verify that the metric identities, (6.38), which are satisfied by the exact metric terms, are also satisfied by their approximations. Satisfaction of the metric identities can be viewed as

a consistency condition. It is especially important for the integration of wave propagation problems in curvilinear coordinates when using the conservative form of the equations. If the metric identities are not satisfied, spurious waves can be generated solely by the grid.

To help appreciate how important it is to satisfy the metric identities, let's approximate the one-way wave equation

$$\begin{aligned} \varphi_t + \varphi_x + \varphi_y = \varphi_t + \nabla \cdot \mathbf{F} = 0, \\ \mathbf{F} = \varphi \hat{x} + \varphi \hat{y} \end{aligned} \tag{6.74}$$

by a collocation approximation on a mapped domain. Under the transformation, (6.61) gives the conservative form of the equation

$$\varphi_t + \frac{1}{J}\left\{ \frac{\partial}{\partial \xi}\left[\left(Y_\eta - X_\eta \right) \varphi \right] + \frac{\partial}{\partial \eta}\left[\left(-Y_\xi + X_\xi \right) \varphi \right] \right\} = 0. \tag{6.75}$$

In the original equation (6.74), it is clear that a constant solution remains constant for all time. The same must be true analytically for the equation in the computational coordinates. If φ is a constant in space, then

$$\varphi_t = -\frac{\varphi}{J}\left\{ \frac{\partial}{\partial \xi}\left[Y_\eta - X_\eta \right] + \frac{\partial}{\partial \eta}\left[-Y_\xi + X_\xi \right] \right\} = 0, \tag{6.76}$$

provided that the transformation is smooth and the cross derivatives are equal, which allows the spatial derivative terms to cancel. Thus, $\varphi_t = 0$ for all points in space.

If we approximate (6.76) by polynomial collocation, the solution will not necessarily remain constant unless certain conditions are met. To get the collocation method, we approximate the solution by a polynomial

$$\varphi\left(x\left(\xi, \eta \right), y\left(\xi, \eta \right) \right) \approx \Phi\left(\xi, \eta \right) = \sum_{i,j=0}^{N} \Phi_{i,j} \ell_i\left(\xi \right) \ell_j\left(\eta \right). \tag{6.77}$$

We also approximate the fluxes by a polynomial

$$\begin{aligned} \left(Y_\eta - X_\eta \right) \varphi \approx I_N\left[\left(Y_\eta - X_\eta \right) \Phi \right] = \sum_{i,j=0}^{N} \left\{ \left(Y_\eta \right)_{i,j} - \left(X_\eta \right)_{i,j} \right\} \Phi_{i,j} \ell_i\left(\xi \right) \ell_j\left(\eta \right), \\ \left(-Y_\xi + X_\xi \right) \varphi \approx I_N\left[\left(-Y_\xi + X_\xi \right) \Phi \right] = \sum_{i,j=0}^{N} \left\{ \left(-Y_\xi \right)_{i,j} + \left(X_\xi \right)_{i,j} \right\} \Phi_{i,j} \ell_i\left(\xi \right) \ell_j\left(\eta \right). \end{aligned} \tag{6.78}$$

The collocation approximation enforces the equation at the collocation points, so

$$\dot{\Phi}_{i,j} = \frac{1}{J}\left\{ \frac{\partial}{\partial \xi} I_N\left[\left(Y_\eta - X_\eta \right) \Phi \right] + \frac{\partial}{\partial \eta} I_N\left[\left(-Y_\xi + X_\xi \right) \Phi \right] \right\}_{i,j}. \tag{6.79}$$

We'll complete the approximation later when we will substitute for Φ from (6.77) and write the result in terms of the derivative matrix, D. At this point however, it suffices to see that the right hand side of (6.79) vanishes when $\Phi = const$ if $\frac{\partial}{\partial \xi} I_N Y_\eta = \frac{\partial}{\partial \eta} I_N (Y_\xi)$ and $\frac{\partial}{\partial \xi} I_N X_\eta = \frac{\partial}{\partial \eta} I_N X_\xi$. Unfortunately, in general they are not, for recall that differentiation and interpolation do not always commute. Since the right hand side of (6.79) does not necessarily vanish even when Φ is constant, the solution can change in time, causing unphysical waves to be generated and to propagate through the grid.

To ensure the metric identities are satisfied for the collocation approximation, we see that a sufficient condition is

$$\begin{aligned} \frac{\partial}{\partial \xi} I_N Y_\eta &= \frac{\partial}{\partial \eta} I_N Y_\xi, \\ \frac{\partial}{\partial \xi} I_N X_\eta &= \frac{\partial}{\partial \eta} I_N X_\xi, \end{aligned} \tag{6.80}$$

which means that differentiation and interpolation need to commute. Since the interpolation and differentiation commute only if there are no errors, the metric identities are satisfied if we represent the mapping $\mathbf{X}$ as a polynomial of degree N or less.

To require that the mapping $\mathbf{X}(\xi, \eta)$ be a polynomial of degree N or less is perhaps a surprising result. It means that we should *not* use an analytical mapping such as (6.1) and differentiate it analytically to get the metric derivatives. Using the exact derivatives can generate errors. Instead, it is better to approximate the mapping by a polynomial (most likely by interpolation, although trucation is of course possible) and then compute the metric terms.

Armed with the principle that the mapping should be represented by a polynomial of degree N or less, we see that the isoparametric transfinite interpolation of (6.18) and Algorithm 98 (TransfiniteQuadMap), i.e, where the boundaries are approximated by polynomials of degree N, nicely fits the bill. This also means that the metric terms computed via (6.73) will satisfy the metric identities.

6.4 How to Compute the Metric Terms

We implement the computation of (6.73) in Algorithm 99 (TransfiniteQuadMetrics), which takes the four curves, each represented by a polynomial interpolant, plus the desired location in the computational domain, and returns the metric coefficients at that location. The boundary curves may not necessarily be represented at the same points as are needed by the spatial approximation. For instance, it is possible to represent the boundaries by Chebyshev interpolants, while the spatial approximation is Legendre. To be flexible, we compute the boundary derivatives using the general form of the barycentric interpolant derivative, (3.45) and (3.46). If we evaluate and store the metric coefficients at the beginning of a computation, how we compute them is not the most significant cost factor. To be complete, we also include a procedure to compute the metric terms when the sides of the domain are straight lines in Algorithm 100 (QuadMapMetrics).

Algorithm 99: *TransfiniteQuadMetrics:* Computation of the Metric Terms on a Curve-Bounded Quadrilateral

Procedure *TransfiniteQuadMetrics*
Input: ξ, η
Input: $\left\{\mathbf{\Gamma}_j\right\}_{j=1}^{4}$; `// Of type CurveInterpolant`
Uses Algorithms:
Algorithm 96 (CurveInterpolant)
Algorithm 97 (CurveInterpolantProcedures)

$(x_1, y_1) \leftarrow \Gamma_1.EvaluateAt(-1)$
$(x_2, y_2) \leftarrow \Gamma_1.EvaluateAt(1)$
$(x_3, y_3) \leftarrow \Gamma_3.EvaluateAt(1)$
$(x_4, y_4) \leftarrow \Gamma_3.EvaluateAt(-1)$
$(X_1, Y_1) \leftarrow \Gamma_1.EvaluateAt(\xi)$
$(X_2, Y_2) \leftarrow \Gamma_2.EvaluateAt(\eta)$
$(X_3, Y_3) \leftarrow \Gamma_3.EvaluateAt(\xi)$
$(X_4, Y_4) \leftarrow \Gamma_4.EvaluateAt(\eta)$
$(X_1', Y_1') \leftarrow \Gamma_1.DerivativeAt(\xi)$
$(X_2', Y_2') \leftarrow \Gamma_2.DerivativeAt(\eta)$
$(X_3', Y_3') \leftarrow \Gamma_3.DerivativeAt(\xi)$
$(X_4', Y_4') \leftarrow \Gamma_4.DerivativeAt(\eta)$

$$X_\xi \leftarrow \frac{1}{2}\left\{X_2 - X_4 + (1-\eta)X_1' + (1+\eta)\,X_3'\right\} - \frac{1}{4}\left\{(1-\eta)\,(x_2 - x_1) + (1+\eta)\,(x_3 - x_4)\right\}$$

$$Y_\xi \leftarrow \frac{1}{2}\left\{Y_2 - Y_4 + (1-\eta)Y_1' + (1+\eta)\,Y_3'\right\} - \frac{1}{4}\left\{(1-\eta)\,(y_2 - y_1) + (1+\eta)\,(y_3 - y_4)\right\}$$

$$X_\eta \leftarrow \frac{1}{2}\left\{(1-\xi)X_4' + (1+\xi)X_2' + X_3 - X_1\right\} - \frac{1}{4}\left\{(1-\xi)\,(x_4 - x_1) + (1+\xi)\,(x_3 - x_2)\right\}$$

$$Y_\eta \leftarrow \frac{1}{2}\left\{(1-\xi)Y_4' + (1+\xi)Y_2' + Y_3 - Y_1\right\} - \frac{1}{4}\left\{(1-\xi)\,(y_4 - y_1) + (1+\xi)\,(y_3 - y_2)\right\}$$

return $\left(X_\xi, X_\eta, Y_\xi, Y_\eta\right)$
End Procedure *TransfiniteQuadMetrics*

Algorithm 100: *QuadMapMetrics:* Computation of the Metric Terms on a Straight Sided Quadrilateral

Procedure *QuadMapMetrics*
Input: $\left\{(x_j, y_j)\right\}_{j=1}^{4}, \xi, \eta$

$$X_\xi \leftarrow \frac{1}{4}\left\{(1-\eta)(x_2 - x_1) + (1+\eta)(x_3 - x_4)\right\}$$

$$Y_\xi \leftarrow \frac{1}{4}\left\{(1-\eta)(y_2 - y_1) + (1+\eta)(y_3 - y_4)\right\}$$

$$X_\eta \leftarrow \frac{1}{4}\left\{(1-\xi)(x_4 - x_1) + (1+\xi)(x_3 - x_2)\right\}$$

$$Y_\eta \leftarrow \frac{1}{4}\left\{(1-\xi)(y_4 - y_1) + (1+\xi)(y_3 - y_2)\right\}$$

return $\left(X_\xi, X_\eta, Y_\xi, Y_\eta\right)$
End Procedure *QuadMapMetrics*

Algorithm 101: *MappedGeometryClass:* Manage Geometry and Metric Terms for Quadrilateral Domains

Uses Algorithms:
Algorithm 96 (CurveInterpolant)
Algorithm 63 (Nodal2DStorage)

Class MappedGeometry
Data:
N, M
$\{x_{i,j}\}_{i,j=0}^{N,M}, \{y_{i,j}\}_{i,j=0}^{N,M}$; // Node locations
$\{x_i^s\}_{i=0;s=1}^{\max(N,M);4}, \{y_i^s\}_{i=0;s=1}^{\max(N,M);4}$; // Boundary Node locations
$\{\partial X/\partial\xi_{i,j}\}_{i,j=0}^{N,M}, \{\partial X/\partial\eta_{i,j}\}_{i,j=0}^{N,M}$; // Metric Terms
$\{\partial Y/\partial\xi_{i,j}\}_{i,j=0}^{N,M}, \{\partial Y/\partial\eta_{i,j}\}_{i,j=0}^{N,M}$; // Metric Terms
$\{J_{i,j}\}_{i,j=0}^{N,M}$; // Jacobian
$\{\hat{n}_i^s\}_{i=0;s=1}^{\max(N,M);4}$; // Boundary normals for each side
$\{scal_i^s\}_{i=0;s=1}^{\max(N,M);4}$; // Scaling factor for each side
Procedures:
Construct($spA, \{\mathbf{\Gamma}_j\}_{j=1}^4$) ; // Algorithm 102
End Class MappedGeometry

Finally, let us wrap the storage and the computation of the metric and geometry terms into a class that we can use later in our approximations of PDEs on general quadrilateral domains. This class, which we define in Algorithm 101 (MappedGeometryClass), stores the physical space locations of the nodes both on the grid and along the boundaries. For some approximations, like collocation, the boundary points coincide with grid points. For others like the nodal discontinuous Galerkin approximation, they don't, so we will store them separately. We have the class store the metric terms, which will be used by all nodal approximations. We store the boundary normals to be able to approximate Neumann boundary conditions and to use the discontinuous Galerkin approximation. Finally, we store the normalization of the normal vectors, $|J\mathbf{a}^i|$, along the boundaries as the variable *scal*, which will be useful later when we implement the discontinuous Galerkin approximation. To construct an instance of this class, we need the parameters of the spatial approximation, which are stored in an instance of the *Nodal2DStorage* class, and the four bounding curves. With these we use Algorithm 102 (MappedGeometry:Construct) to construct the geometry.

Exercises

6.1 Derive (6.39).

6.2 Derive equations (6.47)–(6.49).

Algorithm 102: *MappedGeometry:Construct:* Constructor for Geometry and Metric Terms for Quadrilateral Domains

Procedure Construct
Input $\{\mathbf{\Gamma}_j\}_{j=1}^4$; `// CurveInterpolant`
Input *spA* ; `// Nodal2DStorage`

Uses Algorithms:
Algorithm 99 (TransfiniteQuadMetrics)
Algorithm 98 (TransfiniteQuadMap)
Algorithm 63 (Nodal2DStorage)

$this.N \leftarrow spA.N$; $this.M \leftarrow spA.M$
for $j = 0$ **to** $this.M$ **do**
  for $i = 0$ **to** $this.N$ **do**
    $\{this.x_{i,j}, this.y_{i,j}\} \leftarrow TransfiniteQuadMap(\{\mathbf{\Gamma}_j\}_{j=1}^4, spA.\xi_i, spA.\eta_j)$
    $\{this.X_\xi, this.X_\eta, this.Y_\xi, this.Y_\eta\}_{i,j} \leftarrow$
    $TransfiniteQuadMetrics(\{\mathbf{\Gamma}_j\}_{j=1}^4, spA.\xi_i, spA.\eta_j)$
    $this.J_{i,j} \leftarrow (this.X_\xi * this.Y_\eta - this.X_\eta * this.Y_\xi)_{i,j}$
  end
end
for $j = 0$ **to** $this.M$ **do**
  $\{this.x_j^2, this.y_j^2\} \leftarrow TransfiniteQuadMap(\{\mathbf{\Gamma}_j\}_{j=1}^4, 1, spA.\eta_j)$
  $\{X_\xi, X_\eta, Y_\xi, Y_\eta\} \leftarrow TransfiniteQuadMetrics(\{\mathbf{\Gamma}_j\}_{j=1}^4, 1, spA.\eta_j)$
  $J \leftarrow X_\xi * Y_\eta - X_\eta * Y_\xi$
  $this.scal_j^2 \leftarrow \sqrt{Y_\eta^2 + X_\eta^2}$
  $this.\hat{n}_j^2 \leftarrow SIGN(J) * (Y_\eta \hat{x} - X_\eta \hat{y}) / this.scal_j^2$
  $\{this.x_j^4, this.y_j^4\} \leftarrow TransfiniteQuadMap(\{\mathbf{\Gamma}_j\}_{j=1}^4, -1, this.\eta_j)$
  $\{X_\xi, X_\eta, Y_\xi, Y_\eta\} \leftarrow TransfiniteQuadMetrics(\{\mathbf{\Gamma}_j\}_{j=1}^4, -1, spA.\eta_j)$
  $J \leftarrow X_\xi * Y_\eta - X_\eta * Y_\xi$
  $this.scal_j^4 \leftarrow \sqrt{Y_\eta^2 + X_\eta^2}$
  $this.\hat{n}_j^4 \leftarrow -SIGN(J) * (Y_\eta \hat{x} - X_\eta \hat{y}) / this.scal_j^4$; `// outward normal`
end
for $i = 0$ **to** $this.N$ **do**
  $\{this.x_i^1, this.y_i^1\} \leftarrow TransfiniteQuadMap(\{\mathbf{\Gamma}_j\}_{j=1}^4, spA.\xi_i, -1)$
  $\{X_\xi, X_\eta, Y_\xi, Y_\eta\} \leftarrow TransfiniteQuadMetrics(\{\mathbf{\Gamma}_j\}_{j=1}^4, spA.\xi_i, -1)$
  $J \leftarrow X_\xi * Y_\eta - X_\eta * Y_\xi$
  $this.scal_i^1 \leftarrow \sqrt{Y_\xi^2 + X_\xi^2}$
  $this.\hat{n}_i^1 \leftarrow -SIGN(J) * (-Y_\xi \hat{x} + X_\xi \hat{y}) / this.scal_i^1$; `// outward normal`
  $\{this.x_i^3, this.y_i^3\} \leftarrow TransfiniteQuadMap(\{\mathbf{\Gamma}_j\}_{j=1}^4, spA.\xi_i, 1)$
  $\{X_\xi, X_\eta, Y_\xi, Y_\eta\} \leftarrow TransfiniteQuadMetrics(\{\mathbf{\Gamma}_j\}_{j=1}^4, spA.\xi_i, 1)$
  $J \leftarrow X_\xi * Y_\eta - X_\eta * Y_\xi$
  $this.scal_i^3 \leftarrow \sqrt{Y_\xi^2 + X_\xi^2}$
  $this.\hat{n}_i^3 \leftarrow SIGN(J) * (-Y_\xi \hat{x} + X_\xi \hat{y}) / this.scal_i^3$
end
End Procedure Construct

6.3 Generate the two meshes shown in Fig. 6.3 using an isoparametric mapping with the uniform spacing on the reference square shown there, and with the Chebyshev Gauss-Lobatto points that would be used by a spectral collocation method.

6.4 For the first mapping M_1 defined in (6.19) and shown in Fig. 6.3, compute and plot the maximum error of the boundary normals for each of the four sides as a function of N using the Chebyshev Gauss-Lobatto points. Discuss your results.

6.5 Map the equation for non-isotropic diffusion,

$$\frac{\partial}{\partial x}\left(\nu^{(x)}\frac{\partial \varphi}{\partial x}\right)+\frac{\partial}{\partial y}\left(\nu^{(y)}\frac{\partial \varphi}{\partial y}\right).$$

(Non-isotropic diffusion will be the subject of an example in Chap. 7.)

6.6 Develop and implement an algorithm to use Gauss quadrature to approximate integrals of the type

$$I=\iint_{\Omega} f(x,y)dxdy,$$

where Ω is a quadrilateral domain. Use your implementation to compute the area of the isoparametric Chebyshev Gauss-Lobatto mapping of the domain M_1 of (6.19). Plot the logarithm of the error as a function of N and comment on your results. (Your procedures will also be used later in Chaps. 7 and 8.)

6.7 Suppose that a mapping is made from cylindrical coordinates to Cartesian and that the metric terms are computed analytically. Show that the conservative form of ∇f does not vanish when f is a constant.

Chapter 7
Spectral Methods in Non-Square Geometries

With the ability to map the reference square to non-square domains and to modify the equations to reflect those mappings, we can use spectral methods to compute solutions to PDEs in geometries more complex than the square. In this chapter, we retrace our steps in Chap. 5 to develop spectral methods for non-square geometries.

7.1 Steady Potentials in a Quadrilateral Domain

The most generally applicable approximations for the solution of potential problems on a quadrilateral domain are the collocation or nodal Galerkin methods. The potential equation for a quadrilateral domain will generally have non-constant coefficients. The variable coefficients will limit the exact Galerkin approximation to special cases, such as cylindrical coordinates, where the integrals can be evaluated analytically. The two nodal approximations have the advantage of being generally applicable, at the expense of some spectrally small aliasing and quadrature errors.

7.1.1 The Collocation Approximation

To derive a collocation approximation of the potential equation

$$\nabla^2 \varphi = s \tag{7.1}$$

we follow the derivation of the approximation of the variable coefficient equation (5.25). This is necessary because we showed in Sect. 6.2.1 that under the mapping from the reference square to the physical, quadrilateral domain, the potential equation is transformed to the variable coefficient problem

$$\nabla^2 \varphi = \nabla \cdot \mathbf{F} = \frac{1}{J}\left\{\frac{\partial}{\partial \xi}\left(F^1\right) + \frac{\partial}{\partial \eta}\left(F^2\right)\right\} \tag{7.2}$$

where the contravariant fluxes are

$$\begin{aligned} F^1 &= \frac{Y_\eta}{J}\left(Y_\eta \frac{\partial \varphi}{\partial \xi} - Y_\xi \frac{\partial \varphi}{\partial \eta}\right) - \frac{X_\eta}{J}\left(-X_\eta \frac{\partial \varphi}{\partial \xi} + X_\xi \frac{\partial \varphi}{\partial \eta}\right), \\ F^2 &= \frac{-Y_\xi}{J}\left(Y_\eta \frac{\partial \varphi}{\partial \xi} - Y_\xi \frac{\partial \varphi}{\partial \eta}\right) + \frac{X_\xi}{J}\left(-X_\eta \frac{\partial \varphi}{\partial \xi} + X_\xi \frac{\partial \varphi}{\partial \eta}\right). \end{aligned} \tag{7.3}$$

D.A. Kopriva, *Implementing Spectral Methods for Partial Differential Equations,* Scientific Computation,

We approximate the solution of the potential equation (7.2) on the reference square by the polynomial

$$\varphi(\mathbf{x}) \approx \Phi(\xi, \eta) = \sum_{i,j=0}^{N,M} \Phi_{i,j} \ell_i(\xi) \ell_j(\eta). \tag{7.4}$$

We also approximate the contravariant fluxes (7.3) by their own polynomials. If we call $f \equiv F^1$ and $g \equiv F^2$, approximate them by polynomials F and G, and approximate their derivatives in the usual collocation manner, i.e., by the analytical derivative of the interpolant, the nodal values of the fluxes are

$$\begin{aligned} F_{i,j} &= \left\{ \frac{Y_\eta^2 + X_\eta^2}{J} \right\}_{i,j} \sum_{k=0}^{N} D_{ik}^{(\xi)} \Phi_{k,j} - \left\{ \frac{Y_\eta Y_\xi + X_\eta X_\xi}{J} \right\}_{i,j} \sum_{k=0}^{M} D_{jk}^{(\eta)} \Phi_{i,k}, \\ G_{i,j} &= \left\{ \frac{Y_\xi^2 + X_\xi^2}{J} \right\}_{i,j} \sum_{k=0}^{M} D_{jk}^{(\eta)} \Phi_{i,k} - \left\{ \frac{Y_\eta Y_\xi + X_\eta X_\xi}{J} \right\}_{i,j} \sum_{k=0}^{N} D_{ik}^{(\xi)} \Phi_{k,j}. \end{aligned} \tag{7.5}$$

The final stage of the discretization is to compute the divergence of the fluxes using (7.2)

$$\frac{1}{J_{i,j}} \left(\sum_{k=0}^{N} D_{ik}^{(\xi)} F_{k,j} + \sum_{k=0}^{M} D_{jk}^{(\eta)} G_{i,k} \right) = S_{i,j}, \quad \begin{array}{l} i = 1, 2, \ldots, N-1; \\ j = 1, 2, \ldots, M-1, \end{array} \tag{7.6}$$

which defines the linear system that we must solve for the nodal values of the solution, $\Phi_{i,j}$.

Since the transformation that maps the physical domain onto the reference square maps boundaries onto boundaries, boundary conditions are still easy to apply. We apply Dirichlet conditions just as on the square. We apply Neumann conditions, which specify the normal derivative of the solution, through the contravariant fluxes.

To illustrate how to impose Dirichlet conditions along a boundary, let us suppose that $\varphi(x, y, t) = b_1(\mathbf{x}, t)$ along boundary $\mathbf{\Gamma}_1(\xi)$. That boundary corresponds to points mapped by $(x, y) = \mathbf{x} = \mathbf{X}(\xi, 0)$. Therefore, the approximate solution is set to the boundary values by

$$\Phi_{i,0} = b_1(\mathbf{X}(\xi_i, 0), t), \quad i = 0, 1, \ldots, N. \tag{7.7}$$

Other Dirichlet boundary conditions are set similarly.

To generate collocation approximations to Neumann boundary conditions,

$$\frac{\partial \varphi}{\partial n} = \nabla \varphi \cdot \hat{n} = b'(x, y, t), \tag{7.8}$$

where $\hat{n}$ is the outward normal to the boundary, we first use results from Sect. 6.2.1 to show that the contravariant fluxes F and G are proportional to the normal fluxes.

The gradient in the computational domain is (6.64)

$$\nabla\varphi = \frac{1}{J}\left\{\left(Y_\eta\hat{x} - X_\eta\hat{y}\right)\frac{\partial\varphi}{\partial\xi} + \left(-Y_\xi\hat{x} + X_\xi\hat{y}\right)\frac{\partial\varphi}{\partial\eta}\right\}. \tag{7.9}$$

When we take the dot product of the gradient with the unit vector $\hat{n}^1$ (6.60), which corresponds to the outward normal along $\mathbf{\Gamma}_2$,

$$\nabla\varphi\cdot\hat{n}^1 = \frac{1}{|J|\sqrt{Y_\eta^2 + X_\eta^2}}\left\{\left(Y_\eta^2 + X_\eta^2\right)\frac{\partial\varphi}{\partial\xi} - \left(Y_\eta Y_\xi + X_\eta X_\xi\right)\frac{\partial\varphi}{\partial\eta}\right\}. \tag{7.10}$$

We then match terms with F^1 in (7.3) to see that

$$\nabla\varphi\cdot\hat{n}^1 = \frac{|J|}{J}\sqrt{Y_\eta^2 + X_\eta^2}F^1. \tag{7.11}$$

We can make a similar correspondence between the normal derivative in the $\hat{n}^2$ direction and the flux $g = F^2$. Therefore, to set the Neumann boundary condition (7.8) along boundary $\mathbf{\Gamma}_2$, we need only to set the approximate value of the contravariant flux along the boundary to be

$$F_{N,j} = \left(\frac{|J|}{J}\sqrt{Y_\eta^2 + X_\eta^2}\right)_{N,j} b'\left(\mathbf{X}(1,\eta_j),t\right), \quad i = 0,1,\ldots,N \tag{7.12}$$

and update the solution values $\Phi_{N,j}$ by solving (7.6) for $i = 1,2,\ldots,N$. The other boundaries are treated similarly. The coefficient of b' in (7.12) is computed and stored as $scal_j^2$ by Algorithm 101 (MappedGeometryClass).

7.1.1.1 How to Implement the Collocation Approximation

The implementation of the collocation approximation of the potential problem on a mapped domain is a straightforward extension of the implementation for the square. To be explicit, we create a new class in Algorithm 103 (MappedNodalPotentialClass) that simply extends Algorithm 64 (NodalPotentialClass). We see that we only need to add the geometry information using the MappedGeometryClass of Algorithm 101 and replace the computation of the Laplace operator approximation and MatrixAction routines.

The changes the we need to make to the constructor, which we show in Algorithm 104 (MappedNodalPotentialClass:Construct), are minimal. We must now supply the boundary curve information, of course, and use the constructor for the MappedGeometry class to compute the physical space locations of the nodes and the metric terms. Since the Laplacian in the mapped domain (7.6) uses a succession of first derivative evaluations, we store first derivative instead of second derivative matrices. Of course, to change to a Legendre approximation, we only need to change the procedure used to compute the nodes and weights; we would use Algorithm 25

Algorithm 103: *MappedNodalPotentialClass:* A Class for the Potential Problem in a Mapped Domain

Class MappedNodalPotentialClass **Extends** NodalPotentialClass
Uses Algorithms:
Algorithm 63 (Nodal2DStorage)
Algorithm 64 (NodalPotentialClass)
Algorithm 101 (MappedGeometryClass)

Data:
geom ; `// Of type MappedGeometry`

Procedures:
Construct$(N, M, \{\Gamma_k\}_{k=1}^4)$; `// Algorithm 104`
MappedLaplacian$(\{U_{ij}\}_{i,j=0}^{M}, geom)$; `// Algorithm 105`
MatrixAction$(\{U_{ij}\}_{i,j=0}^{M})$; `// Algorithm 68, Modified`
End Class MappedNodalPotentialClass

Algorithm 104: *MappedNodalPotentialClass:Construct:* Constructor for Collocation Potential Solution on a Mapped Domain

Procedure Construct
Input: N, M
Input: $\{\mathbf{\Gamma}_j\}_{j=1}^4$; `// Of type CurveInterpolant`
Uses Algorithms:
Algorithm 27 (ChebyshevGaussLobattoNodesAndWeights)
Algorithm 102 (MappedGeometry:Construct)

$this.spA.N \leftarrow N$; $this.spA.M \leftarrow M$
$\{this.spA.\{\xi_i\}_{i=0}^N, this.spA.\{w_i^{(\xi)}\}_{i=0}^N\} \leftarrow ChebyshevGaussLobattoNodesAndWeights(N)$
$\{this.spA.\{\eta_j\}_{j=0}^M, this.spA.\{w_j^{(\eta)}\}_{j=0}^M\} \leftarrow ChebyshevGaussLobattoNodesAndWeights(M)$
[For Legendre collocation or nodal Galerkin, use LegendreGaussLobattoNodesAndWeights, Algorithm 25]
$this.geom.Construct(N, M, \{\mathbf{\Gamma}_j\}_{j=1}^4, this.spA)$
$this.spA.\{D_{ij}^{(\xi)}\}_{i,j=0}^N \leftarrow PolynomialDerivativeMatrix(N, this.spA.\{\xi_i\}_{i=0}^N)$
$this.spA.\{D_{ij}^{(\eta)}\}_{i,j=0}^M \leftarrow PolynomialDerivativeMatrix(M, this.spA.\{\eta_j\}_{j=0}^M)$
End Procedure Construct

(LegendreGaussLobattoNodesAndWeights) instead of Algorithm 27 (ChebyshevGaussLobattoNodesAndWeights).

The approximation to the Laplace operator for the mapped domain, (7.6), differs from that on the square, so we must replace the procedure that computes the Laplacian. We show the new procedure in Algorithm 105 (MappedNodalPotentialClass:MappedLaplacian). The algorithm makes it clear that the approximation of the Laplace operator for the mapped quadrilateral is at least twice as expensive to evaluate as on the square.

Algorithm 105: *MappedNodalPotentialClass:MappedLaplacian:* The Collocation Approximation to the Laplace Operator on a Mapped Domain

Procedure MappedLaplacian
Input: $\{U_{i,j}\}_{i,j=1}^{N,M}$
Input: *geom* ; `// Of type MappedGeometry`
Uses Algorithms:
Algorithm 19 (MxVDerivative)

$N \leftarrow this.spA.N$; $M \leftarrow this.spA.M$
for $j = 0$ **to** M **do**
$\left\{ \left. \frac{\partial U}{\partial \xi} \right|_{i,j} \right\}_{i=0}^{N} \leftarrow MxVDerivative\left(this.spA.\left\{D_{n,m}^{(\xi)}\right\}_{n,m=0}^{N}, \left\{U_{i,j}\right\}_{i=0}^{N}\right)$
end
for $i = 0$ **to** N **do**
$\left\{ \left. \frac{\partial U}{\partial \eta} \right|_{i,j} \right\}_{j=0}^{M} \leftarrow MxVDerivative\left(this.spA.\left\{D_{n,m}^{(\eta)}\right\}_{n,m=0}^{M}, \left\{U_{i,j}\right\}_{j=0}^{M}\right)$
end
for $j = 0$ **to** M **do**
for $i = 0$ **to** N **do**
$A \leftarrow \left[\frac{(geom.Y_\eta)^2 + (geom.X_\eta)^2}{geom.J} \right]_{i,j}$
$B \leftarrow \left[\frac{geom.Y_\eta * geom.Y_\xi + geom.X_\eta * geom.X_\xi}{geom.J} \right]_{i,j}$
$C \leftarrow \left[\frac{(geom.Y_\xi)^2 + (geom.X_\xi)^2}{geom.J} \right]_{i,j}$
$F_{i,j} = A * \left. \frac{\partial U}{\partial \xi} \right|_{i,j} - B * \left. \frac{\partial U}{\partial \eta} \right|_{i,j}$
$G_{i,j} = C * \left. \frac{\partial U}{\partial \eta} \right|_{i,j} - B * \left. \frac{\partial U}{\partial \xi} \right|_{i,j}$
end
end
for $j = 0$ **to** M **do**
$\left\{ \left. \frac{\partial F}{\partial \xi} \right|_{i,j} \right\}_{i=0}^{N} \leftarrow MxVDerivative\left(this.spA.\left\{D_{n,m}^{(\xi)}\right\}_{n,m=0}^{N}, \left\{F_{i,j}\right\}_{i=0}^{N}\right)$
end
for $i = 0$ **to** N **do**
$\left\{ \left. \frac{\partial G}{\partial \eta} \right|_{i,j} \right\}_{j=0}^{M} \leftarrow MxVDerivative\left(this.spA.\left\{D_{n,m}^{(\eta)}\right\}_{n,m=0}^{M}, \left\{G_{i,j}\right\}_{j=0}^{M}\right)$
end
for $j = 0$ **to** M **do**
for $i = 0$ **to** N **do**
$\nabla_N^2 U_{i,j} \leftarrow \left(\left. \frac{\partial F}{\partial \xi} \right|_{i,j} + \left. \frac{\partial G}{\partial \eta} \right|_{i,j} \right) / geom.J_{i,j}$
end
end
return $\left\{\nabla_N^2 U_{i,j}\right\}_{i,j=0}^{N,M}$
End Procedure MappedLaplacian

7.1.2 The Nodal Galerkin Approximation

To derive the Galerkin approximation, we convert the strong form of the equation, (7.1) on the physical domain to a weak form on the reference square. To start, we multiply (7.1) by a smooth function ϕ that satisfies the Dirichlet boundary conditions and integrate over the physical domain

$$\int \phi \nabla^2 \varphi dx dy = \int \phi s dx dy. \tag{7.13}$$

We convert to integrals over the reference square by using the transformation of the Laplacian

$$\nabla^2 \varphi = \frac{1}{J}\left\{\frac{\partial F^1}{\partial \xi} + \frac{\partial F^2}{\partial \eta}\right\}, \tag{7.14}$$

and the volume element, $dxdy = Jd\xi d\eta$. Then on the reference square

$$\int_{-1}^{1}\int_{-1}^{1}\left\{\frac{\partial F^1}{\partial \xi} + \frac{\partial F^2}{\partial \eta}\right\}\phi(\xi,\eta)\, d\xi d\eta = \int_{-1}^{1}\int_{-1}^{1} sJ\phi(\xi,\eta)\, d\xi d\eta. \tag{7.15}$$

Next, we integrate the integral of the Laplace operator by parts

$$\begin{aligned}&\int_{-1}^{1}\left[F^1\phi\right]\Big|_{\xi=-1}^{1} d\eta + \int_{-1}^{1}\left[F^2\phi\right]\Big|_{\eta=-1}^{1} d\xi - \int_{-1}^{1}\int_{-1}^{1}\left\{F^1\phi_\xi + F^2\phi_\eta\right\} d\xi d\eta \\ &\quad = \int_{-1}^{1}\int_{-1}^{1} sJ\phi d\xi d\eta.\end{aligned} \tag{7.16}$$

As before, ϕ vanishes along the boundary for Dirichlet boundary conditions, so the boundary integrals vanish to leave

$$-\int_{-1}^{1}\int_{-1}^{1}\left\{F^1\phi_\xi + F^2\phi_\eta\right\} d\xi d\eta = \int_{-1}^{1}\int_{-1}^{1} sJ\phi d\xi d\eta. \tag{7.17}$$

Now that we have the weak form of the equation, we approximate the solution φ by a polynomial Φ and the contravariant fluxes F^1 and F^2 by the polynomials F and G as shown in (7.5). The derivation of the approximation then follows the steps that we took in Sect. 5.2.2 to generate the approximation on the square. We take ϕ to be the polynomial (5.66) and replace the integrals by quadrature. The independence of the nodal values $\phi_{i,j}$ leads to the equations

$$\begin{aligned}&-\sum_{n,m=0}^{N,M}\left\{F_{n,m}\ell_i'(\xi_n)\ell_j(\eta_m) + G_{n,m}\ell_i(\xi_n)\ell_j'(\eta_m)\right\} w_n w_m \\ &\quad = \sum_{n,m=0}^{N,M} s_{n,m} J_{n,m}\ell_i(\xi_n)\ell_i(\eta_m) w_n w_m, \quad i = 1,2,\ldots,N;\ \ j = 1,2,\ldots,M.\end{aligned} \tag{7.18}$$

Again, most of the terms vanish leaving

$$-\sum_{n=0}^{N} F_{n,j}\ell_i'(\xi_n)\,w_n w_j + \sum_{m=0}^{M} G_{i,m}\ell_j'(\eta_m)\,w_i w_m = s_{i,j}J_{i,j}w_i w_j. \tag{7.19}$$

Next, we recognize that $\ell_i'(\xi_n) = D_{in}^{(\xi)T}$, to give us the final approximation

$$\left(\nabla^2\Phi, \ell_i\ell_j\right)_N = s_{i,j}J_{i,j}w_i w_j \tag{7.20}$$

where

$$\left(\nabla^2\Phi, \ell_i\ell_j\right)_N = -\left\{\sum_{n=0}^{N} D_{in}^{(\xi)T}F_{n,j}w_n w_j + \sum_{m=0}^{M} D_{im}^{(\eta)T}G_{i,m}w_i w_m\right\}. \tag{7.21}$$

To impose Neumann boundary conditions, we retain the boundary integrals in (7.16). When we substitute the approximating polynomials and replace the integrals by quadrature,

$$\begin{aligned}&\left[F\left(1,\eta_j\right)\phi\left(1,\eta_j\right) - F\left(-1,\eta_j\right)\phi\left(-1,\eta_j\right)\right]w_j\\ &\quad+\left[G\left(\xi_i,1\right)\phi\left(\xi_i,1\right) - G\left(\xi_i,-1\right)\phi\left(\xi_i,-1\right)\right]w_i\end{aligned} \tag{7.22}$$

approximates the boundary terms. For Dirichlet boundaries, the appropriate value of ϕ is set to zero; if all are Dirichlet, all the terms vanish as above. Along Neumann boundaries there is a flux contribution to add to (7.21). For instance, if boundary $\mathbf{\Gamma}_2$ is a Neumann boundary and the others are Dirichlet, then (7.21) becomes

$$\begin{aligned}\left(\nabla^2\Phi, \ell_i\ell_j\right) &= F\left(1,\eta_j\right)\ell_i(1)w_j\\ &\quad-\left\{\sum_{n=0}^{N} D_{in}^{(\xi)T}F_{n,j}w_n w_j + \sum_{m=0}^{M} D_{im}^{(\xi)T}G_{i,m}w_i w_m\right\}.\end{aligned} \tag{7.23}$$

We specify the boundary value of the contravariant flux as we did for the collocation approximation in the previous section with (7.12). Note that $\ell_i(1) = \delta_{iN}$ vanishes at all the interior grid points. Therefore we only need to add the flux term to the equation along the Neumann boundaries and solve for $\Phi_{i,j}$ along the boundary as well as the interior. Cf. (4.125).

7.1.2.1 How to Implement the Nodal Galerkin Method

Once again, we see that although the derivation of the nodal Galerkin method differs significantly from the collocation method, we end with an approximation of a similar form. We can therefore reuse most of what we have already developed. In

Algorithm 106: *TransposeMatrixMultiply:* Matrix Transpose-Vector Multiplication Algorithm

```
Procedure TransposeMatrixMultiply
Input: {D_{i,j}}_{i,j=s}^{e}, {f_j}_{j=s}^{e}
for i = s to e do
    t = 0
    for j = s to e do
        t ← t + D_{j,i} * f_j
    end
    (I_N f)'_i ← t
end
return {(I_N f)'_i}_{i=s}^{e}
End Procedure TransposeMatrixMultiply
```

particular, we do not have to change the data of the MappedNodalPotentialClass, Algorithm 103, and the constructor is identical to the constructor for Legendre collocation.

The only significant change that we must make is to the approximation of the Laplacian operator. Unlike the collocation approximation, the Galerkin approximation (7.21) requires the matrix vector multiplication of the transpose of the derivative matrix D times the slice of the flux vector multiplied by the appropriate quadrature weights. To compute that, we create a modification of the matrix-vector multiplication algorithm, Algorithm 19 (MxVDerivative), that swaps the subscripts i and j in the inner loop that computes the intermediate variable, t, rather than transpose and store the derivative matrix. With the modified matrix-vector product, which we show in Algorithm 106 (TransposeMatrixMultiply), we can replace the flux derivative evaluations in Algorithm 105 (MappedLaplacian) to get the nodal Galerkin version, Algorithm 107 (MappedLaplacian).

7.1.3 Solution of the Linear Systems

Both the collocation and nodal Galerkin approximations require a full matrix linear system to be solved, just as on the square. It is still possible to use a direct solver as we discussed in Sect. 5.2.1.3, particularly for small N. For larger systems, iterative methods such as the BiCGStab for collocation and Conjugate Gradient for Galerkin, are still more efficient. As before, preconditioning of the systems remains critical.

The iterative solution of the linear system drives a norm of the iteration residual to within a specified tolerance of zero. The iteration residual for the collocation approximation remains that given in (5.34). The iteration residual for the nodal Galerkin approximation (5.78) needs to be modified to account for the presence of the Jacobian of the transformation. When the physical domain is mapped, we now

Algorithm 107: *MappedNodalPotentialClass:MappedLaplacian:* Nodal Galerkin Approximation to the Laplace Operator on a Mapped Domain

Procedure MappedLaplacian
Input: $\left\{U_{i,j}\right\}_{i,j=1}^{N,M}$
Input: *geom* ; `// Of type MappedGeometry`
Uses Algorithms:
Algorithm 19 (MxVDerivative)
Algorithm 106 (TransposeMatrixMultiply)

$N \leftarrow this.spA.N$; $M \leftarrow this.spA.M$
for $j = 0$ **to** M **do**
$\left\{ \left.\frac{\partial U}{\partial \xi}\right|_{i,j} \right\}_{i=0}^{N} \leftarrow MxVDerivative\left(this.spA.\left\{D_{n,m}^{(\xi)}\right\}_{n,m=0}^{N}, \left\{U_{i,j}\right\}_{i=0}^{N}\right)$
end
for $i = 0$ **to** N **do**
$\left\{ \left.\frac{\partial U}{\partial \eta}\right|_{i,j} \right\}_{j=0}^{M} \leftarrow MxVDerivative\left(this.spA.\left\{D_{n,m}^{(\eta)}\right\}_{n,m=0}^{M}, \left\{U_{i,j}\right\}_{j=0}^{M}\right)$
end
for $j = 0$ **to** M **do**
for $i = 0$ **to** N **do**

$$A \leftarrow \left[\frac{(geom.Y_\eta)^2 + (geom.X_\eta)^2}{geom.J}\right]_{i,j}$$

$$B \leftarrow \left[\frac{geom.Y_\eta * geom.Y_\xi + geom.X_\eta * geom.X_\xi}{geom.J}\right]_{i,j}$$

$$C \leftarrow \left[\frac{(geom.Y_\xi)^2 + (geom.X_\xi)^2}{geom.J}\right]_{i,j}$$

$$F_{i,j} = this.spA.w_i^{(\xi)} * \left[A * \left.\frac{\partial U}{\partial \xi}\right|_{i,j} - B * \left.\frac{\partial U}{\partial \eta}\right|_{i,j}\right]$$

$$G_{i,j} = this.spA.w_j^{(\eta)} * \left[C * \left.\frac{\partial U}{\partial \eta}\right|_{i,j} - B * \left.\frac{\partial U}{\partial \xi}\right|_{i,j}\right]$$

end
end
for $j = 0$ **to** M **do**
$\{D^{(\xi)T} F|_{i,j}\}_{i=0}^{N} \leftarrow TransposeMatrixMultiply\left(this.spA.\left\{D_{n,m}^{(\xi)}\right\}_{n,m=0}^{N}, \left\{F_{i,j}\right\}_{i=0}^{N}\right)$
end
for $i = 0$ **to** N **do**
$\{D^{(\eta)T} G|_{i,j}\}_{j=0}^{M} \leftarrow TransposeMatrixMultiply\left(this.spA.\left\{D_{n,m}^{(\eta)}\right\}_{n,m=0}^{M}, \left\{G_{i,j}\right\}_{j=0}^{M}\right)$
end
for $j = 1$ **to** $M - 1$ **do**
for $i = 1$ **to** $N - 1$ **do**
$\left(\nabla^2 U, \ell_i \ell_j\right) \leftarrow -\left(this.spA.w_j^{(\eta)} * D^{(\xi)T} F\big|_{i,j} + this.spA.w_i^{(\xi)} * D^{(\eta)T} G\big|_{i,j}\right)$
end
end
return $\left\{\left(\nabla^2 U, \ell_i \ell_j\right)\right\}_{i,j=0}^{N,M}$
End Procedure MappedLaplacian

have

$$r_{i,j} = S_{i,j} J_{i,j} w_i^{(\xi)} w_j^{(\eta)} + \left\{ \sum_{n=0}^{N} D_{in}^{(\xi)T} F_{n,j} w_n^{(\xi)} w_j^{(\eta)} + \sum_{m=0}^{M} D_{jm}^{(\xi)T} G_{i,m} w_i^{(\xi)} w_m^{(\eta)} \right\} \tag{7.24}$$

for the residual of the nodal Galerkin approximation.

With the introduction of variable coefficients via the mapping, it is worth considering the diagonal preconditioner. The diagonal preconditioner is very easy to derive and to implement, so it takes relatively little effort to increase the efficiency of the iterative methods when it does work. To derive a diagonal preconditioner, we must identify the coefficient of the solution at a given location (i, j) in the grid. We will derive the diagonal for the collocation approximation first, followed by the Galerkin.

To find the matrix diagonal entry for the collocation approximation to the Laplace operator we replace the fluxes in (7.6) by their explicit representation (7.5). For convenience, we replace the metric derivatives by the notationally more compact forms provided by the contravariant metric tensor. The full approximation is

$$\begin{aligned} \nabla^2 \Phi\big|_{i,j} = \frac{1}{J_{i,j}} \Bigg\{ & \sum_{n=0}^{N} D_{in}^{(\xi)} \left[\left(J g^{11} \right)_{n,j} \sum_{k=0}^{N} D_{nk}^{(\xi)} \Phi_{k,j} + \left(J g^{12} \right)_{n,j} \sum_{k=0}^{M} D_{jk}^{(\eta)} \Phi_{n,k} \right] \\ & + \sum_{m=0}^{M} D_{jm}^{(\eta)} \left[\left(J g^{22} \right)_{i,m} \sum_{k=0}^{M} D_{mk}^{(\eta)} \Phi_{i,k} + \left(J g^{21} \right)_{i,m} \sum_{k=0}^{N} D_{ik}^{(\xi)} \Phi_{k,m} \right] \Bigg\}. \end{aligned} \tag{7.25}$$

If we look at the first term,

$$\sum_{n=0}^{N} D_{in}^{(\xi)} \left[\left(J g^{11} \right)_{n,j} \sum_{k=0}^{N} D_{nk}^{(\xi)} \Phi_{k,j} \right] = \sum_{n} \sum_{k} D_{in}^{(\xi)} D_{nk}^{(\xi)} \left(J g^{11} \right)_{n,j} \Phi_{k,j} \tag{7.26}$$

we see that the coefficient of $\Phi_{i,j}$ has $k = i$. Therefore, the contribution to the diagonal from the first term is

$$\sum_{n=0}^{N} D_{in}^{(\xi)} D_{ni}^{(\xi)} \left(J g^{11} \right)_{n,j}. \tag{7.27}$$

If we continue in the same manner with the other terms in (7.26), we find that the diagonal of the system for the collocation approximation of the Laplace operator is

$$\begin{aligned} d_{i,j} = \frac{1}{J_{i,j}} \Bigg\{ & \sum_{n=0}^{N} D_{in}^{(\xi)} D_{ni}^{(\xi)} \left(J g^{11} \right)_{n,j} + 2 D_{ii}^{(\xi)} D_{jj}^{(\eta)} \left(J g^{12} \right)_{i,j} \\ & + \sum_{m=0}^{N} D_{jm}^{(\eta)} D_{mj}^{(\eta)} \left(J g^{22} \right)_{i,m} \Bigg\}. \end{aligned} \tag{7.28}$$

A similar set of steps leads to the diagonal coefficient of the nodal Galerkin approximation

$$d_{i,j} = -\frac{1}{J_{i,j}} \left\{ w_j^{(\eta)} \sum_{n=0}^{N} D_{in}^{(\xi)T} w_n^{(\xi)} D_{ni}^{(\xi)} \left(Jg^{11}\right)_{n,j} + 2 w_i^{(\xi)} w_j^{(\eta)} D_{ii}^{(\xi)} D_{jj}^{(\eta)} \left(Jg^{12}\right)_{i,j} + w_i^{(\xi)} \sum_{m=0}^{N} D_{jm}^{(\eta)T} w_m^{(\eta)} D_{mj}^{(\eta)} \left(Jg^{22}\right)_{i,m} \right\}. \tag{7.29}$$

For either the collocation or the nodal Galerkin methods, the implementation of the diagonal preconditioner $z = H^{-1} r$ simply divides the residual by the diagonal entry,

$$z_{i,j} = r_{i,j}/d_{i,j}, \quad \begin{cases} i = 1, 2, \ldots, N-1, \\ j = 1, 2, \ldots, M-1. \end{cases} \tag{7.30}$$

With the variable coefficients that arise from the mapping, the finite difference and finite element preconditioners become more complicated. However, we can easily extend the approximate finite element preconditioner that we derived in Sect. 5.2.2.3 to non-orthogonal grids, with virtually no extra effort in the derivation or implementation. It also allows us to reuse algorithms that we have already developed. We can use it to precondition both the collocation and the nodal Galerkin approximations. One difference to recognize between the approximate finite element preconditioner on the square, where the coefficients are constant, and its extension to general geometries is that the stencil will increase from five points on the square to nine in general. Therefore, the ILU solver of Algorithm 74 (FDPreconditioner:Solve) would have to be extended. Rather than do that, we will just use the SSOR iterative solver, Algorithm 79 (SSORSweep), which is equally applicable to a five or nine point stencil. Our experience with the potential equation on the square shows that the SSOR solver is almost as effective as the ILU solver as long we make a reasonable selection for the overrelaxation parameter.

The starting point to derive the approximate finite element preconditioner for non-orthogonal grids is to recognize that the problem to approximate has changed from the inner product $(\nabla\Phi, \nabla\phi_p)$ to the inner product $(\frac{1}{J}\mathbf{F}, \nabla\phi_p)$ where $\mathbf{F} = F\hat{\xi} + G\hat{\eta}$. Thus, all we need to do in the computational space is to change the function g_{kl}^{nm} defined in equation (5.91) to account for the new integrand and the fact that in the computational space (x, y) has been replaced by (ξ, η). If we now define the affine mapping between an element in the computational space and the unit square in terms of a new pair of variables, (v, w), we get the new integrand

$$\begin{aligned} g_{kl}^{nm}(v, w) = \Bigg\{ & A\left(\xi_i + v\Delta\xi_i, \eta_j + w\Delta\eta_j\right) \frac{\partial\varphi_{kl}}{\partial v} \frac{\partial\varphi_{nm}}{\partial v} \\ & + B\left(\xi_i + v\Delta\xi_i, \eta_j + w\Delta\eta_j\right) \left[\frac{\partial\varphi_{kl}}{\partial w} \frac{\partial\varphi_{nm}}{\partial v} + \frac{\partial\varphi_{kl}}{\partial v} \frac{\partial\varphi_{nm}}{\partial w} \right] \\ & + C\left(\xi_i + v\Delta\xi_i, \eta_j + w\Delta\eta_j\right) \frac{\partial\varphi_{kl}}{\partial w} \frac{\partial\varphi_{nm}}{\partial w} \Bigg\}, \end{aligned} \tag{7.31}$$

where

$$A = \frac{X_\eta^2 + Y_\eta^2}{J\,\Delta\xi_i^2}, \qquad B = -\frac{Y_\eta Y_\xi + X_\xi X_\eta}{J\,\Delta\xi_i\,\Delta\eta_j}, \qquad C = \frac{X_\xi^2 + Y_\xi^2}{J\,\Delta\eta_j^2}. \tag{7.32}$$

Implementation of the approximate finite element preconditioner to the mapped problem requires only simple modifications to the procedure *LocalStiffnessMatrix* in Algorithm 78 (ApproximateFEMStencil) to account for the variable coefficients that come from the mapping. The input Δx and Δy will now correspond to the computational space variables $\Delta\xi$ and $\Delta\eta$. The metric terms and the Jacobian for the four corners of the element and the position (i, j) in the grid must be added as input variables. Finally, we replace the line that defines the intermediate variable t,

$$\begin{aligned} t \leftarrow t &+ PhiXi(k,l,s) * PhiXi(n,m,s)/\Delta x^2 \\ &+ PhiEta(k,l,r) * PhiEta(n,m,r)/\Delta y^2, \end{aligned} \tag{7.33}$$

which is marked by the comment R1 in the procedure *LocalStiffnessMatrix*, with the quantities defined in (7.31) to become

$$\begin{aligned} A &\leftarrow \frac{(geom.X_\eta)^2_{i+r,j+s} + (geom.Y_\eta)^2_{i+r,j+s}}{geom.J_{i+r,j+s}\,\Delta\xi_i^2} \\ B &\leftarrow -\frac{(geom.Y_\eta)_{i+r,j+s}(geom.Y_\xi)_{i+r,j+s} + (geom.X_\xi)_{i+r,j+s}(X_\eta)_{i+r,j+s}}{geom.J_{i+r,j+s}\,\Delta\xi_i\,\Delta\eta_j} \\ C &\leftarrow \frac{(geom.X_\xi)^2_{i+r,j+s} + (geom.Y_\xi)^2_{i+r,j+s}}{geom.J_{i+r,j+s}\,\Delta\eta_j^2} \\ t &\leftarrow t + A * Psi\_Xi(k,l,s) * Psi\_Xi(n,m,s) \\ &\quad + B * [Psi\_Eta(k,l,r) * Psi\_Xi(n,m,s) + Psi\_Eta(n,m,r) * Psi\_Xi(k,l,s)] \\ &\quad + C * Psi\_Eta(k,l,r) * Psi\_Eta(n,m,r) \end{aligned}$$

To solve potential problems on mapped domains, we need to extend the driver in Algorithm 76 (ChebyshevCollocationDriver) to create the four boundary curves and the mapping to the interior before it constructs and solves the spatial approximation. As an example, we'll show how to construct a driver for the Chebyshev collocation approximation. The equivalent driver for the nodal Galerkin approximation is similar.

Algorithm 108 (MappedCollocationDriver) shows how to compute the solutions to the potential equation on a mapped domain. The procedure constructs the four curve interpolants for the prescribed boundary, here provided by a curve function to be supplied. These curves are then used to construct an instance of the MappedNodalPotentialClass. After the approximate finite element preconditioner has been constructed (using procedures modified for curved boundaries, as discussed in this section), the solver is used to compute the solution.

Algorithm 108: *MappedCollocationDriver:* Driver for the Collocation Approximation to Steady Potential in a Non-Square Geometry

Procedure Main
Input: N, M, N_{it}, TOL
Uses Algorithms:
Algorithm 27 (ChebyshevGaussLobattoNodesAndWeights)
Algorithm 96 (CurveInterpolant)
Algorithm 97 (CurveInterpolantProcedures)
Algorithm 103 (MappedNodalPotentialClass)
Algorithm 104 (MappedNodalPotentialClass:Construct)
Algorithm 72 (Class: FDPreconditioner—Modified)
Algorithm 73 (FDPreconditioner:Construct—Modified)
Algorithm 75 (BiCGStabSolve)
Derived Types: CurveInterpolant: $\left\{\mathbf{\Gamma}_j\right\}_{j=1}^{4}$, MappedNodalPotentialClass: *mnpc* , AFEMPreconditioner: H
$\left\{\left\{s_j\right\}_{j=0}^{N}, \left\{w_j\right\}_{j=0}^{N}\right\} \leftarrow ChebyshevGaussLobattoNodesAndWeights(N)$
$\left\{(x_j, y_j)\right\}_{j=0}^{N} \leftarrow EvaluateCurve1\left(N, \left\{s_j, \right\}_{j=0}^{N}\right)$; `// To be supplied`
$\mathbf{\Gamma}_1.Construct\left(N, \left\{s_j, \right\}_{j=0}^{N}, \left\{(x_j, y_j)\right\}_{j=0}^{N}\right)$
$\vdots$
$mnpc.Construct\left(N, M, \left\{\mathbf{\Gamma}_j\right\}_{j=1}^{4}\right)$
for $j = 0$ **to** M **do**
  for $i = 0$ **to** N **do**
    $mnpc.s_{i,j} \leftarrow SourceValue(mnpc.spA.\xi_i, mnpc.spA.\eta_j)$
  end
end
$mnpc.\left\{mask_{ij}\right\}_{k=1}^{4} \leftarrow \{true, true, true, true\}$
$mnpc.\left\{\Phi_{ij}\right\}_{i,j=0}^{N,M} \leftarrow SetBoundaryValues\left(mnpc.\left\{\Phi_{ij}\right\}_{i,j=0}^{N,M}\right)$
$H.Construct(mnpc)$
$mnpc \leftarrow BiCGStabSolve(N_{it}, TOL, mnpc, H)$
End Procedure Main

7.1.4 Benchmark Solution: Potential in Non-Square Domains

To understand how to choose between the two approximations and the various solver options, let's compare the performance of the collocation and nodal Galerkin methods on the Dirichlet problem for

$$\nabla^2\varphi = -\frac{16\ln(r)}{r^2}\sin(4\theta), \tag{7.34}$$

where $r = \sqrt{x^2 + y^2}$ and $\theta = \tan^{-1}(y/x)$. This problem has the exact solution

$$\varphi = \ln(r)\sin(4\theta). \tag{7.35}$$

We will solve the problem on the two domains drawn in Fig. 7.1. The domains are shown with their grids generated by the transfinite mapping for $N = M = 10$.

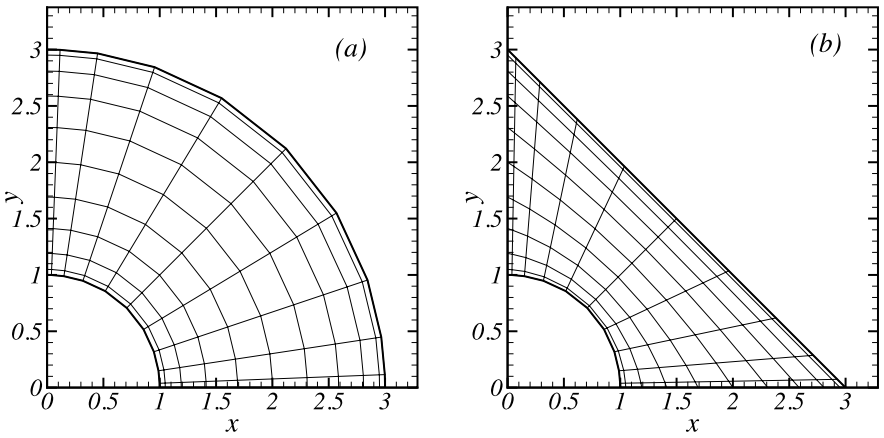

Fig. 7.1 Two examples of mapped quadrilateral domains. (**a**) Orthogonal grid. (**b**) Non-Orthogonal grid

The domain on the left is a quarter of an annulus, whose boundaries are defined by polynomial interpolants of the boundary curves of mapping (6.19). The transfinite mapping for the geometry in Fig. 7.1a is orthogonal, i.e. $g^{12} = g^{21} = 0$. The boundary curves for the domain in Fig. 7.1b are given by the mapping (6.20). The transfinite mapping for this domain generates a non-orthogonal grid, $g^{12} = g^{21} \neq 0$.

We have already seen from Algorithms 105 (MappedCollocationLaplacian) and 107 (MappedNodalGalerkinLaplacian) that the two spatial approximations are virtually the same to implement. We also know from Algorithms 75 (BiCGSSTAB-Solve) and 80 (PreconditionedConjugateGradientSolve) that the effort needed to implement the iterative schemes appropriate for the two spatial approximations (but not the computational work per iteration) is virtually the same. Of course, we have already compared the performance of the methods on the square in Sect. 5.2.1.

On each domain, we find that the Chebyshev collocation and nodal Galerkin methods have similar accuracy. Table 7.1 shows that both are spectrally accurate, but that the Galerkin approximation is more accurate than collocation, the amount depending on the grid.

Since the difference in accuracy is not significant, and the implementations are similar, the speed which we can compute the solutions becomes important. Table 7.2 presents the number of iterations and CPU seconds to converge the residual to 10^{-14} for $N = M = 72$ for the Chebyshev collocation approximation. Table 7.3 presents the same results for the nodal Galerkin approximation.

We make three conclusions from Tables 7.2 and 7.3. First, there is a significant difference in the performance of the iterative schemes when the grid is orthogonal or not. Second, the diagonal preconditioner performs better with the BiCGStab algorithm than with the preconditioned Conjugate Gradient solver. Finally, we get the best overall performance with the Galerkin approximation plus the approximate finite element preconditioner plus the SSOR solver. This combination was up to ten times faster than the collocation runs.

Table 7.1 Maximum errors for steady potentials on two domains

N	Orthogonal		Non-orthogonal	
	Collocation	Galerkin	Collocation	Galerkin
8	1.0E−4	9.0E−6	2.1E−3	1.5E−3
12	3.0E−8	1.2E−9	4.75E−5	3.5E−5
16	1.0E−11	1.2E−12	8.9E−7	7.3E−7
20	3.4E−14	3.6E−14	1.8E−8	1.2E−8
24			5.0E−10	1.7E−11
28			2.5E−11	1.7E−11
32			1.2E−12	7.9E−13
36			5.0E−14	6.9E−14

Table 7.2 Efficiency of solvers for the Chebyshev collocation approximation solution for $N = 72$ on two grids

Preconditioner	Orthogonal		Non-orthogonal	
	Iterations	CPU (s)	Iterations	CPU (s)
None	443	1.77	473	1.88
AFEM ILU	271	1.03	–	–
AFEM SSOR	435	1.76	523	2.12
Diagonal	267	0.96	348	1.25

Table 7.3 Efficiency of solvers for the nodal Galerkin approximation solution for $N = 72$ on two grids

Preconditioner	Orthogonal		Non-orthogonal	
	Iterations	CPU (s)	Iterations	CPU (s)
None	938	1.71	1148	2.1
AFEM ILU	87	0.17	–	–
AFEM SSOR	81	0.17	135	0.28
Diagonal	755	1.42	669	1.26

7.1.5 Benchmark Solution: Incompressible Flow over a Circular Obstacle

A more interesting example is to compute the inviscid, incompressible fluid flow over a cylindrical obstacle on the ground, such as we sketch in Fig. 7.2. The flow has an exact solution with which we will test the spectral convergence of the approximation.

The steady incompressible, irrotational flow over an obstacle is governed by the potential equation. If the flow speed of a gas is low enough and the density is approximately constant, the fluid can be considered to be incompressible. If the flow upstream of the obstacle is uniform, then its curl is zero, i.e., it is irrotational. That flow will stay irrotational throughout the domain if there is no viscosity.

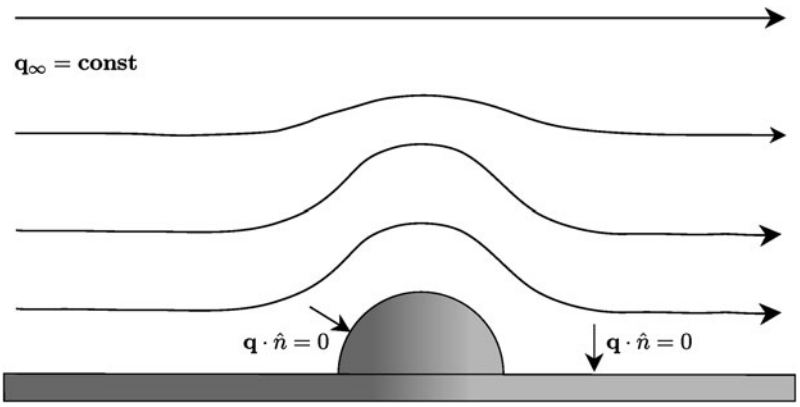

Fig. 7.2 Flow over an obstacle

We derive the equation to solve starting from the law of conservation of mass, which is the conservation law (see Sect. 5.4)

$$\rho_t + \nabla \cdot (\rho \mathbf{q}) = 0. \tag{7.36}$$

Here, ρ is the density, $\mathbf{q}$ is the fluid velocity and $\rho\mathbf{q}$ is the mass flux. For steady flow,

$$\nabla \cdot (\rho \mathbf{q}) = 0. \tag{7.37}$$

If we assume that the density is constant, i.e., the flow is incompressible, then the velocity field is divergence free

$$\nabla \cdot \mathbf{q} = 0. \tag{7.38}$$

Furthermore, if the flow is irrotational i.e. $\nabla \times \mathbf{q} = 0$, then we can write the velocity as the gradient of a potential, $\mathbf{q} = \nabla\varphi$. Therefore, the potential satisfies the equation

$$\nabla \cdot \nabla\varphi = \nabla^2\varphi = 0. \tag{7.39}$$

We must set boundary conditions along the obstacle, the ground, and far away in the free stream. Along the obstacle and along the ground, the normal velocity must vanish. Far from the obstacle in the free stream, the effect of the obstacle must vanish and it is natural to specify a uniform, horizontal velocity. Since the velocity is the gradient of the potential, these conditions imply that the gradient of the potential is a constant in the free stream. Therefore the potential is linear in x and y. At the body and along the ground, $\nabla\varphi \cdot \hat{n} = 0$. Therefore, the boundary conditions we need to specify are mixed Dirichlet/Neumann

$$\begin{aligned} &\varphi = V_\infty x, \qquad x^2 + y^2 \to \infty, \\ &\frac{\partial\varphi}{\partial n} = \nabla\varphi \cdot \hat{n} = 0, \qquad \textit{ground} + \textit{obstacle}, \end{aligned} \tag{7.40}$$

where $V_\infty = |\mathbf{q}_\infty|$.

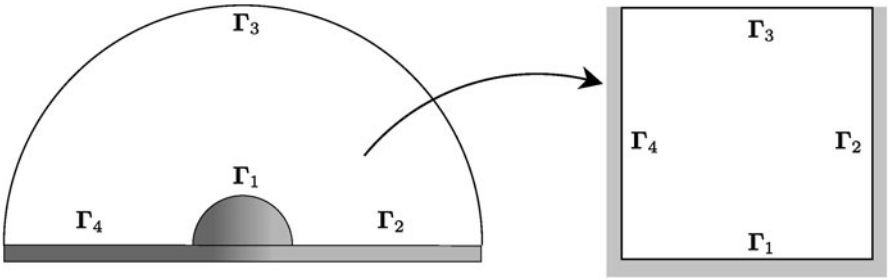

Fig. 7.3 Mapping the exterior of a cylindrical obstacle onto the reference square

Finally, conservation of energy leads to Bernoulli's law, from which we can compute the pressure of the gas in the flow. Bernoulli's law is

$$p + \frac{1}{2}\rho \left|\mathbf{q}\right|^2 = p + \frac{1}{2}\rho \left|\nabla\varphi\right|^2 = K, \tag{7.41}$$

where p is the pressure and K is a constant. Clearly as the fluid speed increases, the pressure decreases. Stagnation points, where the fluid speed goes to zero, are the points with highest pressure. Once we compute the potential, we can compute the velocity and pressure fields from its gradient.

The problem of irrotational, inviscid, irrotational flow around an obstacle has an exact solution that can be derived using conformal mapping. If the obstacle is a cylinder, for example, the exact potential is

$$\varphi = V_\infty \left(r + \frac{r_0}{r}\right)\cos\left(\theta\right). \tag{7.42}$$

Here, r is the radial distance from the center of the cylinder, r_0 is the cylinder's radius, and θ is the usual polar angle measured from the aft (downstream) side.

To compute the flow over a cylindrical obstacle, we map the cylindrical region shown in Fig. 7.3 to the reference square by the transfinite mapping

$$M \quad \begin{cases} \mathbf{\Gamma}_1\left(\xi\right) = r_0 \cos\left(\pi\left(\xi+1\right)/2\right)\hat{x} + r_0 \sin\left(\pi\left(\xi+1\right)/2\right)\hat{y}, \\ \mathbf{\Gamma}_2\left(\eta\right) = r_0 + (r_\infty - r_0)(\eta+1)/2\hat{x} + 0\hat{y}, \\ \mathbf{\Gamma}_3\left(\xi\right) = r_\infty \cos\left(\pi\left(\xi+1\right)/2\right)\hat{x} + r_\infty \sin\left(\pi\left(\xi+1\right)/2\right)\hat{y}, \\ \mathbf{\Gamma}_4\left(\eta\right) = -r_0 - (r_\infty - r_0)(\eta+1)/2\hat{x} + 0\hat{y}, \end{cases} \tag{7.43}$$

where $r_0 = 0.5$ and $r_\infty = 10$ corresponds to a distance far from the obstacle. The mapping produces the grid shown in Fig. 7.4 when $N = M = 30$.

There are three Neumann boundaries on the reference square, which we show shaded on Fig. 7.3, and one Dirichlet boundary, which is not shaded. We can easily set the Neumann conditions for the Galerkin approximation. We only need to set the *mask* variable to be *false* for the Neumann boundaries. To impose the Neumann conditions for the collocation approximation, we need to set the normal fluxes to be zero. One way to implement this is to create two more mask arrays, one each

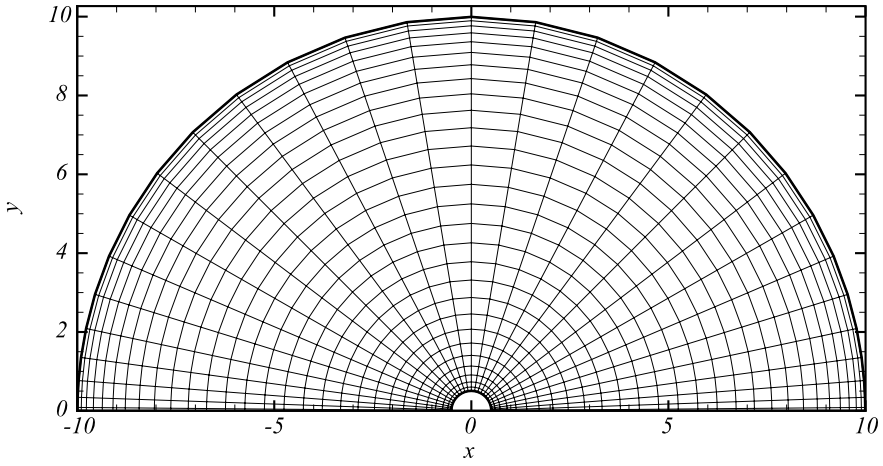

Fig. 7.4 Chebyshev collocation grid around a cylindrical obstacle

for the fluxes F and G. For Neumann boundaries the new mask will be *true*. Then we add two calls to the *MaskSides* function of Algorithm 67, with the appropriate mask arrays, to Algorithm 105 (MappedLaplacian) immediately after the fluxes are computed, one for F and one for G.

To illustrate the flow solution, we show the pressure and streamlines computed using the Chebyshev collocation approximation and $N = M = 30$ in Fig. 7.5 for $V_\infty = 0.5$, $\rho_\infty = 1$ and $p_\infty = 1$. To make the plots, we computed the velocity from the Chebyshev approximation of the gradient of the potential, $\mathbf{q} = \nabla\varphi$ using (7.9), and interpolated the solutions to 100 points in each direction using Algorithm 35 (2DCoarseToFineInterpolation) to draw the contours smoothly. We see that the flow stagnates at the leading (upstream) corner of the cylinder and the ground. It is here and at the trailing (aft) corner that the largest pressures appear. The flow accelerates as it goes over the obstacle and so the pressure decreases to a minimum at the top of the cylinder.

Convergence of the Chebyshev collocation approximation is exponential. Figure 7.6 shows the logarithm of the maximum error in the potential as a function of $N = M$. Note that doubling the polynomial order in each direction decreases the error by more than a factor of one hundred.

7.2 Steady Potentials in an Annulus

Potential problems with at least one periodic direction, such as annular regions, are perfect for mixed basis approximations that use Fourier basis functions in the periodic direction(s). We will see that the use of the Fourier basis easily enables the efficient solution of the linear systems by way of the FFT.

In this section, we will derive algorithms to solve for the potential in an annulus shown in Fig. 7.7. We can view this geometry, for instance, as a section of a long,

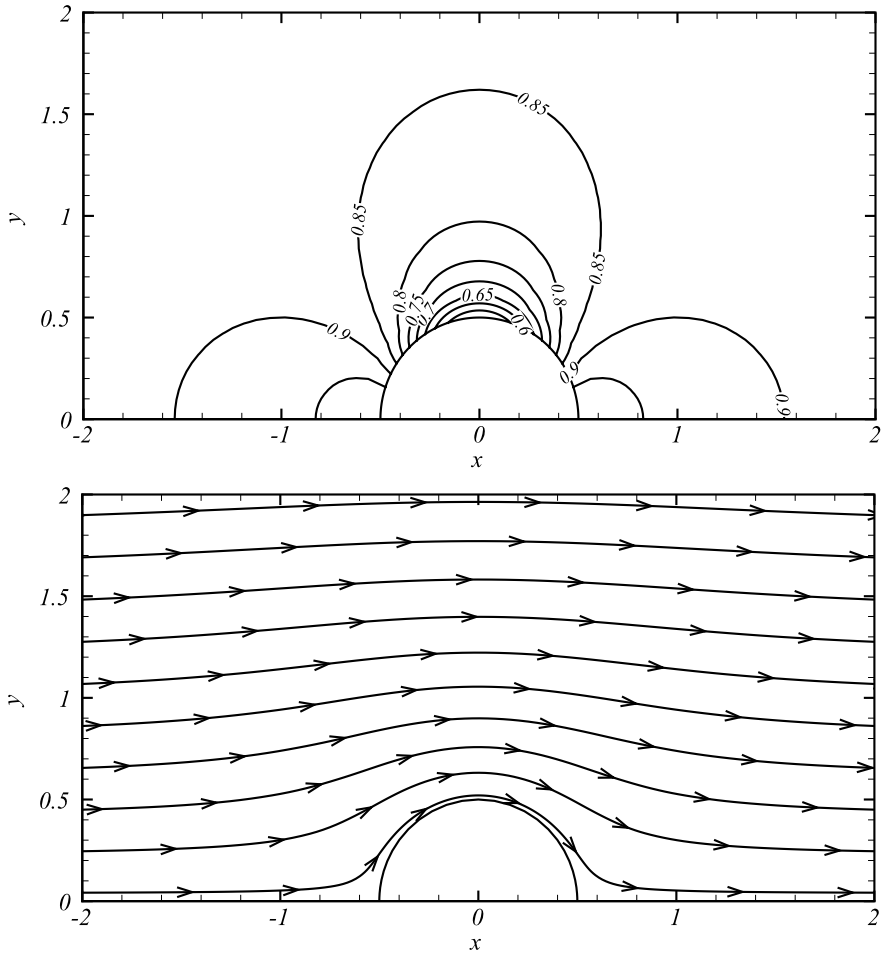

Fig. 7.5 Computed pressure (*top*) and streamlines (*bottom*) for the Chebyshev collocation approximation of steady, inviscid incompressible near a cylindrical obstacle

cylindrical capacitor or thermal insulation. A simple mathematical model for the potential in this annular region is

$$\nabla^2\varphi = \frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial\varphi}{\partial r}\right) + \frac{1}{r^2}\frac{\partial^2\varphi}{\partial\theta^2} = s \tag{7.44}$$

with boundary conditions that represent an applied voltage or temperature

$$\begin{aligned} \varphi\left(r_I,\theta\right) &= \varphi_I\left(\theta\right), \\ \varphi\left(r_O,\theta\right) &= \varphi_O\left(\theta\right). \end{aligned} \tag{7.45}$$

To keep things general, we will leave the source term, s, in the equation.

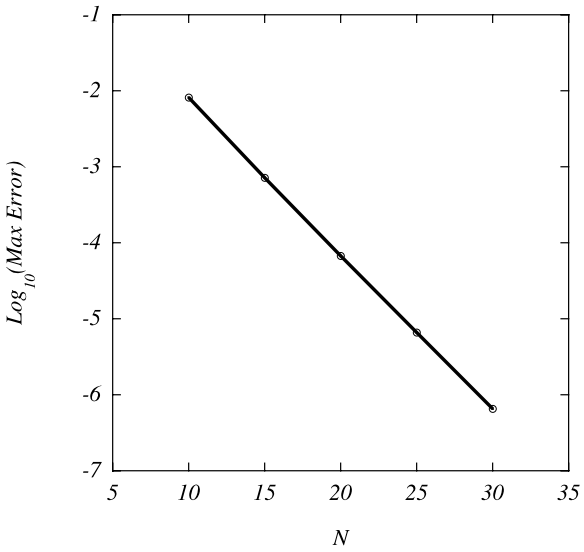

Fig. 7.6 Convergence of the maximum error in the velocity potential for the Chebyshev collocation approximation

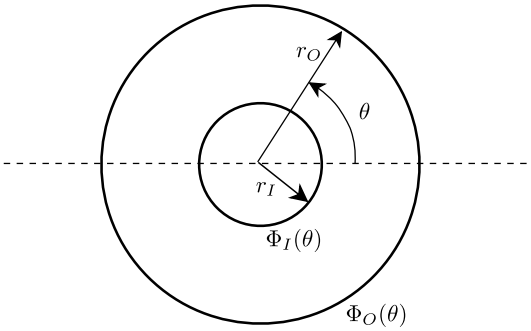

Fig. 7.7 Geometry and boundary conditions for steady potential in an annulus

The first stage to approximate the potential is to map the annular region to the reference domain $[-1, 1] \times [0, 2\pi]$. Since the coordinate system is orthogonal, the mapping is straightforward, namely

$$\begin{aligned} r &= r_I + \frac{\xi + 1}{2} (r_O - r_I), \\ \theta &= \eta. \end{aligned} \tag{7.46}$$

Under this transformation, the PDE becomes

$$\frac{4}{r(\xi)(r_O - r_I)^2} \frac{\partial}{\partial \xi} \left(r(\xi) \frac{\partial \varphi}{\partial \xi} \right) + \frac{1}{r(\xi)^2} \frac{\partial^2 \varphi}{\partial \eta^2} = s. \tag{7.47}$$

In what follows, we will describe the collocation approximation to (7.47). The nodal Galerkin approximation will lead to a similar system of equations to solve.

We approximate the potential in the annulus by a mixed basis polynomial

$$\Phi(\xi,\eta)=\sum_{l=0}^{N}\sum_{j=0}^{M-1}\Phi_{l,j}\ell_l(\xi)h_j(\eta)=\sum_{l=0}^{N}\sum_{k=-M/2}^{M/2}\frac{\tilde{\Phi}_{lk}}{\bar{c}_k}\ell_l(\xi)e^{i\eta}, \tag{7.48}$$

where the collocation grid corresponds to the Gauss-Lobatto points (either Chebyshev or Legendre) in the ξ direction and the uniformly spaced points $\eta_j = 2\pi/M$ in the azimuthal direction.

We now derive the collocation approximation. After we substitute the polynomial approximation into the differential equation, require the residual to vanish at the collocation points, and simplify using the facts that $\ell_n(\xi_l)=\delta_{n,l}$ and $h_m(\eta_j)=\delta_{m,j}$, we get the system of equations that must be solved for the potential at the collocation points

$$\begin{aligned}&\frac{4}{r_l(r_O-r_I)^2}\sum_{n=0}^{N}\Phi_{n,m}\frac{\partial}{\partial\xi}\left(r\ell_n'\right)\Big|_{\xi_l}+\frac{1}{r_l^2}\sum_{m=0}^{M-1}\Phi_{l,m}h_m''(\eta_j)\\ &\quad=S_{l,j},\quad\begin{array}{l}l=1,2,\ldots,N-1,\\ j=0,1,\ldots,M-1,\end{array}\end{aligned} \tag{7.49}$$

where $r_l = r(\xi_l)$. To complete the approximation, we need to approximate the product term by a polynomial of degree N, which in Lagrange form is

$$r(\xi)\ell_n'(\xi)\approx\sum_{\mu=0}^{N}r\left(\xi_\mu\right)\ell_n'\left(\xi_\mu\right)\ell_\mu(\xi). \tag{7.50}$$

So we approximate

$$\frac{\partial}{\partial\xi}\left(r(\eta)\ell_n'(\xi)\right)\Big|_{\xi_l}\approx\sum_{\mu=0}^{N}r\left(\xi_\mu\right)\ell_n'\left(\xi_\mu\right)\ell_\mu'(\xi_l)=\sum_{\mu=0}^{N}r\left(\xi_\mu\right)D_{\mu n}D_{l\mu}. \tag{7.51}$$

Therefore, in terms of the collocation derivative matrices,

$$\sum_{n=0}^{N}B_{ln}\Phi_{n,j}+\frac{1}{r_l^2}\sum_{m=0}^{M-1}\Phi_{l,m}h_m''(\eta_j)=S_{l,j},\quad\begin{array}{l}l=1,2,\ldots,N-1,\\ j=0,1,\ldots,M-1,\end{array} \tag{7.52}$$

where

$$B_{ln}=\frac{4}{r_l(r_O-r_I)^2}\sum_{\mu=0}^{N}r\left(\xi_\mu\right)D_{\mu n}D_{l\mu}. \tag{7.53}$$

We have left the derivative of the Fourier Lagrange interpolating polynomial, h_m'', alone for the moment because the solution procedure that we will introduce next eliminates the need for it.

We now turn to the solution of the linear system (7.52). We could solve it directly or by one of the iterative solvers that we have used in previous sections. However, the Fourier basis gives us the opportunity to develop a fast solver that uses the FFT. What we will describe next is an example of a *matrix diagonalization technique*, in this case made simple by the FFT, which will diagonalize the matrix $h''_m(\eta_j)$ to simplify the solution of the system.

To (partially) diagonalize the system (7.52) we take the discrete Fourier transform of both sides in the η direction. The first term on the left, which has only ξ dependence in its coefficients, becomes

$$\frac{4}{r_l\,(r_O - r_I)^2}\sum_{n=0}^{N} B_{ln}\tilde{\Phi}_{nk}. \tag{7.54}$$

The Fourier transform of the second term diagonalizes the matrix $h''_m(\eta_j)$. First,

$$\begin{aligned}&\frac{1}{M}\sum_{j=0}^{M-1}\left[\sum_{m=0}^{M-1}\Phi_{im}h''(\eta_j)\right]e^{-2\pi ijk/M}\\ &\quad=\frac{1}{M}\sum_{m=0}^{M-1}\Phi_{im}\left[\sum_{j=0}^{M-1}h''(\eta_j)e^{-2\pi ijk/M}\right],\quad k=-M/2,\ldots,M/2.\end{aligned} \tag{7.55}$$

We then use the exactness of the quadrature that allows us to replace summation by integration

$$\frac{1}{M}\sum_{j=0}^{M-1}h''(\eta_j)e^{-2\pi ijk/M}=\frac{1}{2\pi}\int_0^{2\pi}h''(\eta)\,e^{-ik\eta}d\eta. \tag{7.56}$$

We next integrate the right side by parts twice. The fact that the integrand is periodic leads to

$$\frac{1}{M}\sum_{j=0}^{M-1}h''(\eta_j)e^{-2\pi ijk/M}=-k^2e^{-2\pi ijk/M}. \tag{7.57}$$

The idea to convert a summation to an integral and back when the quadrature is exact is powerful and commonly used in spectral methods.

We now replace the factor in square brackets on the right hand side of (7.55)

$$\frac{1}{M}\sum_{j=0}^{M-1}\left[\sum_{m=0}^{M-1}\Phi_{l,m}h''(\eta_j)\right]e^{-2\pi ijk/N}=-\frac{k^2}{M}\sum_{m=0}^{M-1}\Phi_{l,m}e^{-2\pi ijk/N}=-k^2\tilde{\Phi}_{lm}. \tag{7.58}$$

This leads to the system of equations

$$\sum_{n=0}^{N}\left(B_{ln}-\frac{k^2}{r_l^2}\delta_{l,n}\right)\tilde{\Phi}_{nk}=\tilde{S}_{lk},\quad\begin{array}{l}l=1,2,\ldots,N-1,\\ k=-M/2,\ldots M/2-1.\end{array} \tag{7.59}$$

Since $\tilde{\Phi}_{n,M/2} = \tilde{\Phi}_{n,-M/2}$, we don't have to compute the last mode. We know the boundary values from the Fourier transform of the boundary conditions. If we put those on the right hand side, we have

$$\sum_{n=1}^{N-1} \left(B_{ln} - \frac{k^2}{r_l^2} \delta_{l,n} \right) \tilde{\Phi}_{nk} = \tilde{S}_{lk} - \left(B_{l0} \tilde{\Phi}_{0k} + B_{lN} \tilde{\Phi}_{Nk} \right), \quad \begin{array}{l} l = 1, 2, \ldots, N-1, \\ k = -M/2, \ldots M/2 - 1. \end{array} \tag{7.60}$$

Equation (7.60) represents M independent $N \times N$ systems of equations for the Fourier coefficients. Let $A^{(k)}$ be the matrix whose elements are

$$A_{ln}^{(k)} = \left(B_{ln} - \frac{k^2}{r_l^2} \delta_{l,n} \right), \tag{7.61}$$

$$RHS_l^{(k)} = \tilde{S}_{lk} - \left(B_{l0} \tilde{\Phi}_{0k} + B_{lN} \tilde{\Phi}_{Nk} \right). \tag{7.62}$$

Then there are M systems to be solved of the form

$$A^{(k)} \mathbf{v}^{(k)} = RHS^{(k)}. \tag{7.63}$$

We are now able to develop an algorithm to compute the collocation approximation to the problem of steady potentials in an annulus. To compute the right hand side vectors, we need the discrete Fourier coefficients of the boundary functions and the source term, if one is present. We can compute these transforms efficiently using the FFT. Next, for each wavenumber, k, we form the matrices $A^{(k)}$ and the right hand side vectors. We solve the systems as we form them. There is no need to store all of them at once. The matrices are relatively small—the Fourier transform changes the original $(N-1)M \times (N-1)M$ system to M systems of size $(N-1) \times (N-1)$—so direct solution of the systems is fast and does not require a large amount of storage. In a parallel computing environment, we could solve the completely independent M systems simultaneously. Finally, the matrices are real, and the solutions are complex. This means we can solve for the real and imaginary parts independently. More importantly, we can solve for them with the same LU decomposition.

We use Algorithm 109 (*PotentialOnAnnulus*) to solve for the potential on an annulus. The procedure first computes the collocation points and the derivative matrix for the radial direction. It then sets up the matrix B, which is independent of the wavenumber. Then the boundary and source terms are computed from externally supplied functions. These boundary and source arrays are transformed by the FFT. Since the input arrays are real, we use the FFT that implements the even-odd decomposition. Once the boundary and source arrays are transformed, the M systems are solved in the k' loop. Recall that the FFT orders the wavenumbers differently from the way we need to use them, so the transformation from k' to k is made to account for that. After the matrix A is computed for the current k and factorized, the right hand side of the system is constructed. To take advantage of the *LUSolve*

Algorithm 109: *PotentialOnAnnulus:* Use of the FFT to Compute Potentials with One Periodic Direction

Procedure PotentialOnAnnulus
Input: N, M, r_I, r_O, $Source$, $\Phi_I(\theta)$, $\Phi_O(\theta)$
Uses Algorithms:
Algorithm 37 (PolynomialDerivativeMatrix)
Algorithm 27 (ChebyshevGaussLobattoNodesAndWeights)
Algorithm 7 (InitializeFFT)
Algorithm 11 (FFTEO), Algorithm 12 (BFFTEO), Algorithm 142 (LUFactorization)

$\{\{\xi_l\}_{l=0}^{N}, \{w_l\}_{l=0}^{N}\} \leftarrow ChebyshevGaussLobattoNodesAndWeights(N)$
$\{D_{lj}\}_{l,j=0}^{N} \leftarrow PolynomialDerivativeMatrix(N, \{\xi_l\}_{l=0}^{N})$
for $l = 0$ **to** N **do**
  $r_l \leftarrow r_I + (\xi_l + 1) * (r_O - r_I)$
  for $n = 0$ **to** N **do**
    $s = 0$
    for $m = 0$ **to** N **do**
      $s \leftarrow s + r_m * D_{nm} * D_{ml}$
    end
    $B_{ln} \leftarrow 4 * s/(r_l(r_O - r_I)^2)$
  end
end
$\Delta\theta \leftarrow 2\pi/M$
for $j = 0$ **to** $M - 1$ **do**
  $\theta \leftarrow j\Delta\theta$; $\Phi_j^I \leftarrow \Phi_I(\theta)$; $\Phi_j^O \leftarrow \Phi_O(\theta)$
  for $l = 0$ **to** N **do**
    $S_{lj} \leftarrow Source(r_l, theta)$
  end
end
$\{w_j^+\}_{j=0}^{M-1} = InitializeFFT(M, FORWARD)$
$\{\tilde{\Phi}_k^I\}_{k=0}^{M-1} \leftarrow FFTEO(M, \{\Phi_j^I\}_{j=0}^{M-1}, \{w_j^+\}_{j=0}^{M-1})$; $\{\tilde{\Phi}_k^O\}_{k=0}^{M-1} \leftarrow FFTEO(M, \{\Phi_j^O\}_{j=0}^{M-1}, \{w_j^+\}_{j=0}^{M-1})$
for $l = 1$ **to** $N - 1$ **do**
  $\{\tilde{S}_{lk}\}_{k=0}^{M-1} \leftarrow FFTEO(M, \{S_{lj}\}_{j=0}^{M-1}, \{w_j^+\}_{j=0}^{M-1})$
end
for $k' = 0$ **to** $M - 1$ **do**
  if $k' < M/2$ **then** $k \leftarrow k'$ **else** $k \leftarrow k' - M$
  for $l = 1$ **to** $N - 1$ **do**
    for $n = 1$ **to** $N - 1$ **do**
      $A_{ln} \leftarrow B_{ln}$
    end
    $A_{ll} \leftarrow A_{ll} - (k/r_l)^2$
  end
  $\{\{A_{i,j}\}_{i,j=1}^{N}, \{p_j\}_{j=1}^{N}\} \leftarrow Factorize(\{A_{i,j}\}_{i,j=1}^{N})$
  for $l = 1$ **to** $N - 1$ **do**
    $RHS_{l1} \leftarrow \mathrm{Re}(\tilde{S}_{lk} - (B_{l0}\,\mathrm{Re}(\tilde{\Phi}_k^I) + B_{lN}\,\mathrm{Re}(\tilde{\Phi}_k^O))$; $RHS_{l2} \leftarrow \mathrm{Im}(\tilde{S}_{lk} - (B_{l0}\,\mathrm{Im}(\tilde{\Phi}_k^I) + B_{lN}\,\mathrm{Im}(\tilde{\Phi}_k^O))$
  end
  $\{RHS_{nm}\}_{n=1,m=1}^{N-1,2} \leftarrow LUSolve(\{A_{n,m}\}_{n,m=1}^{N-1}, \{p_j\}_{j=1}^{N}, \{RHS_{n,m}\}_{n=1,m=1}^{N-1,2})$
  for $l = 1$ **to** $N - 1$ **do**
    $\tilde{\Phi}_{lk} \leftarrow RHS_{l1} + i * RHS_{l2}$
  end
end
for $j = 0$ **to** $M - 1$ **do**
  $\Phi_{0j} \leftarrow \Phi_j^I$; $\Phi_{Nj} \leftarrow \Phi_j^O$
end
$\{w_j^-\}_{j=0}^{M-1} = InitializeFFT(M, BACKWARD)$
for $l = 1$ **to** $N - 1$ **do**
  $\{\Phi_{lj}\}_{j=0}^{M-1} \leftarrow BFFTEO(M, \{\tilde{\Phi}_{ik}\}_{k=0}^{M-1}, \{w_k^-\}_{k=0}^{M-1})$
end
return $\{\Phi_{l,j}\}_{l,=0,j=0}^{N,M-1}$
End Procedure PotentialOnAnnulus

routine, which allows multiple right hand sides to be solved together, we create an $(N-1) \times 2$ array for the right hand sides from the real and imaginary parts. We use Algorithm 142 (LUFactorization) to solve the system, which returns the result in the original right hand side. We then construct the complex potential from the two parts. Once the discrete Fourier coefficients of the solution are computed for each k, they are used to synthesize the collocation point values of the solution through calls to the backward FFT.

The use of the FFT for the diagonalization procedure represents a very powerful and efficient solver for problems with constant coefficients. It is also possible to apply such diagonalization procedures for polynomial approximations, with the added complexity of computing the eigenfunctions for the matrix operators. See [7] for examples.

7.2.1 Benchmark Solution: Potential in an Annulus with a Source

As a benchmark problem, we solve the potential equation (7.44) with a source term

$$s(r,\theta) = \left[\left(1 - \frac{1}{r}\right) + \frac{4}{r^2}\left(\sin^2(2\theta) - \cos(2\theta)\right)\right] e^{-r} e^{-\cos(2\theta)}$$

and boundary conditions so that the exact solution is

$$\varphi = e^{-r} e^{\cos(2\theta)}.$$

Figure 7.8 shows the computed and exact contours for $r_I = 1$, $r_O = 4$, and $N = M = 16$. To draw the plots, we interpolated the solution to a uniform 40×80 mesh. With these parameters, the maximum error on the mesh is 2.6×10^{-4} so the approximate solution is indistinguishable from the exact.

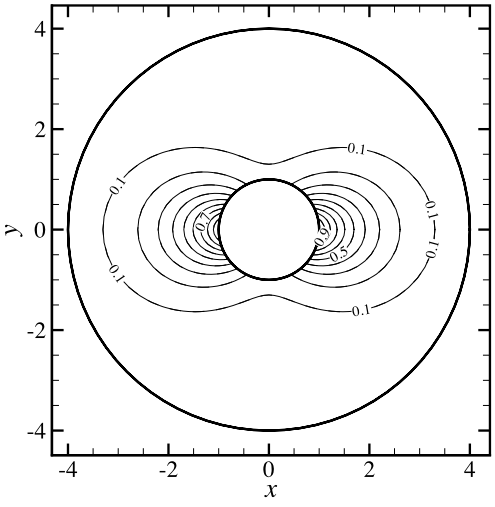

Fig. 7.8 Computed (*solid*) and exact (*dashed*) contours for the mixed Chebysehev-Fourier collocation approximation of the potential in an annulus. The contours are uniformly spaced with spacing 0.1

7.3 Advection and Diffusion in Quadrilateral Domains

In Sect. 5.3 we developed both collocation and nodal Galerkin approximations to the advection-diffusion equation on the square

$$\begin{aligned} &\frac{\partial \varphi}{\partial t} + \mathbf{q} \cdot \nabla \varphi = \eta \nabla^2 \varphi, \quad (x, y) \in (-1, 1) \times (-1, 1), \\ &\varphi(x, y, 0) = \varphi_0(x, y), \quad (x, y) \in [-1, 1] \times [-1, 1]. \end{aligned} \tag{7.64}$$

We integrated the systems of ODEs that we generated from the spatial approximations with a semi-implicit method. We integrated the advection terms explicitly and the diffusion terms implicitly. The implicit approximation of the diffusion terms led to a system of equations that we solved at each time step by an iterative method that we developed in Sects. 5.2.1.4 or 5.2.2.3.

The derivations of approximations to the advection-diffusion equation for non-square geometries mirror those on the square, as do the algorithms. As before, we will present both collocation and nodal Galerkin approximations. We will integrate the discrete systems with the same semi-implicit BDF/extrapolation method that we used on the square. At each time step we solve the algebraic system of equations with the techniques of Sect. 7.1 that we developed for the potential equation in non-square geometries.

7.3.1 Transformation of the Advection-Diffusion Equation

Under the mapping from the physical to the computational domain, the advection-diffusion equation, (7.64), will no longer be constant coefficient. For that reason, and to be able to use the transformation relations that we derived in Sect. 6.2.1, we re-write the advection-diffusion equation as

$$\varphi_t + \mathbf{q} \cdot \nabla \varphi = \nabla \cdot \mathscr{F}, \tag{7.65}$$

where

$$\mathscr{F} = \nu \nabla \varphi. \tag{7.66}$$

We can then apply the transformations for the gradient, (6.64), and the divergence, (6.61), to map the advection and diffusion terms.

From (6.64), the gradient on the reference square is

$$\nabla \varphi = \frac{1}{J} \left\{ \left(Y_\eta \hat{x} - X_\eta \hat{y} \right) \frac{\partial \varphi}{\partial \xi} + \left(-Y_\xi \hat{x} + X_\xi \hat{y} \right) \frac{\partial \varphi}{\partial \eta} \right\}. \tag{7.67}$$

Also, using results from Sect. 6.2.1, we transform the diffusion term to

$$\nabla \cdot \mathscr{F} = \frac{1}{J} \left\{ \frac{\partial f}{\partial \xi} + \frac{\partial g}{\partial \eta} \right\}, \tag{7.68}$$

where

$$
\begin{aligned}
f &= \nu \left\{ \frac{Y_\eta^2 + X_\eta^2}{J} \frac{\partial \varphi}{\partial \xi} - \frac{Y_\eta Y_\xi + X_\eta X_\xi}{J} \frac{\partial \varphi}{\partial \eta} \right\}, \\
g &= \nu \left\{ -\frac{Y_\eta Y_\xi + X_\eta X_\xi}{J} \frac{\partial \varphi}{\partial \xi} + \frac{Y_\xi^2 + X_\xi^2}{J} \frac{\partial \varphi}{\partial \eta} \right\}.
\end{aligned}
\tag{7.69}
$$

Therefore, the advection-diffusion equation is

$$
\varphi_t + \frac{1}{J} \left\{ \left(u Y_\eta - v X_\eta \right) \frac{\partial \varphi}{\partial \xi} + \left(-u Y_\xi + v X_\xi \right) \frac{\partial \varphi}{\partial \eta} \right\} = \frac{1}{J} \left\{ \frac{\partial f}{\partial \xi} + \frac{\partial g}{\partial \eta} \right\} \tag{7.70}
$$

on the reference square.

7.3.2 The Collocation Approximation

The collocation approximation to (7.70) is derived as usual. We approximate the solution and fluxes by polynomials in Lagrange form, and evaluate the derivatives of those polynomials at the collocation points. When we follow the usual procedure, we get the semi-discrete collocation approximation

$$
\frac{d \Phi_{i,j}}{dt} = \nabla \cdot \mathbf{F}_{i,j} - \mathbf{q} \cdot \nabla \Phi_{i,j}, \quad i, j = 1, 2, \ldots, N-1. \tag{7.71}
$$

The approximation of the advection term is

$$
\begin{aligned}
\mathbf{q} \cdot \nabla \Phi_{i,j} = \frac{1}{J_{i,j}} \Bigg\{ & \left(u Y_\eta - v X_\eta \right)_{i,j} \sum_n^N D_{in}^{(\xi)} \Phi_{n,j} \\
& + \left(-u Y_\xi + v X_\xi \right)_{i,j} \sum_m^N D_{jm}^{(\eta)} \Phi_{i,m} \Bigg\}.
\end{aligned}
\tag{7.72}
$$

We showed how to derive the approximation of the diffusion term in Sect. 7.1.1. It is

$$
\nabla \cdot \mathbf{F}_{i,j} = \frac{1}{J_{i,j}} \left(\sum_{k=0}^N D_{ik}^{(\xi)} F_{k,j} + \sum_{k=0}^M D_{jk}^{(\eta)} G_{i,k} \right), \tag{7.73}
$$

where

$$
\begin{aligned}
F_{i,j} &= \nu \left\{ \left[\frac{Y_\eta^2 + X_\eta^2}{J} \right]_{i,j} \sum_{k=0}^N D_{ik}^{(\xi)} \Phi_{k,j} - \left[\frac{Y_\eta Y_\xi + X_\eta X_\xi}{J} \right]_{i,j} \sum_{k=0}^M D_{jk}^{(\eta)} \Phi_{i,k} \right\}, \\
G_{i,j} &= \nu \left\{ \left[\frac{Y_\xi^2 + X_\xi^2}{J} \right]_{i,j} \sum_{k=0}^M D_{jk}^{(\eta)} \Phi_{i,k} - \left[\frac{Y_\eta Y_\xi + X_\eta X_\xi}{J} \right]_{i,j} \sum_{k=0}^N D_{ik}^{(\xi)} \Phi_{k,j} \right\}.
\end{aligned}
\tag{7.74}
$$

7.3.3 The Nodal Galerkin Approximation

We follow the procedure of Sect. 5.3 to derive the nodal Galerkin approximation. Again, we approximate the solution by a polynomial written in Lagrange form and substitute that approximation into (7.70). After we use Gauss-Lobatto quadratures to approximate the integrals of the weak form of the equations, the time derivative term for a mapped domain becomes

$$\int_{-1}^{1}\int_{-1}^{1}\frac{d\Phi}{dt}\phi_{ij}J\,d\xi\,d\eta \approx w_i^{(\xi)}w_j^{(\eta)}J_{i,j}\frac{d\Phi_{i,j}}{dt}. \tag{7.75}$$

To approximate the advection terms, we replace the integrand $d\Phi/dt$ in (7.75) with $\mathbf{q}\cdot\nabla\Phi$. It follows that the nodal Galerkin approximation of the advection term is (7.72) multiplied by $w_i^{(\xi)}w_j^{(\eta)}J_{i,j}$. Finally, we have already derived the nodal Galerkin approximation of the diffusion terms written as (7.68) in Sect. 7.1.2 . The only difference now is that the fluxes, (7.69), include the metric terms as in (7.72).

When we collect the approximations for the nodal Galerkin approximation, we have the system of ordinary differential equations

$$w_i^{(\xi)}w_j^{(\eta)}J_{i,j}\frac{d\Phi_{i,j}}{dt} = \left(\nabla\cdot\mathbf{F},\phi_{ij}\right)_N - \left(\mathbf{q}\cdot\nabla\Phi,\phi_{ij}\right)_N \tag{7.76}$$

to integrate in time where

$$\begin{aligned}\left(\mathbf{q}\cdot\nabla\Phi,\phi_{ij}\right)_N = w_i^{(\xi)}w_j^{(\eta)}\Bigg\{&\left(uY_\eta - vX_\eta\right)_{i,j}\sum_n^N D_{in}^{(\xi)}\Phi_{n,j}\\ &+\left(-uY_\xi + vX_\xi\right)_{i,j}\sum_m^N D_{jm}^{(\eta)}\Phi_{i,m}\Bigg\}.\end{aligned} \tag{7.77}$$

We take the diffusion term from (7.21)

$$\left(\nabla\cdot\mathbf{F},\phi_{i,j}\right)_N = -\left\{\sum_{n=0}^{N}D_{in}^{(\xi)T}F_{n,j}w_nw_j + \sum_{m=0}^{M}D_{im}^{(\eta)T}G_{i,m}w_iw_m\right\}, \tag{7.78}$$

where the fluxes are computed by (7.74).

Since the transformation that maps to the physical domain from the reference square maps boundaries onto boundaries, boundary conditions are still easy to apply. We apply Dirichlet conditions for either the collocation or nodal Galerkin approximations just as on the square. We set Neumann conditions, which specify the normal derivative of the solution, through the contravariant fluxes as we described in Sect. 5.2.2.

We have the same issues with the time integration of the collocation system (7.71)–(7.74) and the nodal Galerkin system (7.76)–(7.78) as we did on the square in Sect. 5.3. The approximations to the diffusion terms are stiff compared to the approximations of the advection terms. For that reason, it is still convenient to use the implicit/explicit integration that we presented there.

7.3.4 How to Implement the Approximations

Few modifications need to be made to the implementations that we presented in Sect. 5.3 to implement either the collocation or nodal Galerkin approximations in non-square geometries. We can quickly convert the implementation presented there to handle more general geometries. The additions and changes that we need to make are:

- *Add mapping and metric terms.* We must add mapping and metric term information in the form of a MappedGeometry instance to the NodalAdvDiffClass, Algorithm 81. We saw how to add the mapping in Sect. 7.1.1 for the MappedNodalPotentialClass, Algorithm 103 and its constructor, Algorithm 104 (MappedNodalPotentialClass:Construct). The constructor no longer needs to compute the second derivative matrix, so we can remove the calls to *mthOrderPolynomialDerivativeMatrix*.
- *Modify the transport computation, Algorithm 83* (*Transport*). To evaluate the transport term for a quadrilateral domain, we need to modify Algorithm 83 by replacing the line

$$this.transport^k_{i,j} \leftarrow this.u * \Phi x_{ij} + this.v * \Phi y_{ij} \tag{7.79}$$

with

$$\begin{aligned} this.transport^k_{i,j} \leftarrow \frac{1}{this.geom.J_{i,j}} \big\{ &\big(this.u * (this.geom.Y_\eta)_{i,j} \\ &- this.v * (this.geom.X_\eta)_{i,j}\big)\Phi x_{i,j} \\ &+ \big(-this.u(this.geom.Y_\xi)_{i,j} \\ &+ this.v(this.geom.X_\xi)_{i,j}\big)\Phi y_{i,j}\big\}. \end{aligned} \tag{7.80}$$

To make the procedure compute the nodal Galerkin approximation, (7.77), we follow this with the line

$$\begin{aligned} this.transport^k_{i,j} \leftarrow &this.geom.J_{i,j} * this.spA.w^{(\xi)}_i \\ &* this.spA.w^{(\eta)}_j * this.transport^k_{i,j}. \end{aligned} \tag{7.81}$$

- *Use the mapped version of the Laplacian approximation in Algorithm 85* (*AdvDiffImplicitResidual*). We now need to use the mapped approximation of the action of the Laplace operator, Algorithm 105 (MappedLaplacian), for collocation approximations, or the equivalent Algorithm 107 for the nodal Galerkin approximation in place of the procedure LaplacianOnTheSquare of Algorithms 66 and 77.

Otherwise, the Algorithms 84 (ExplicitRHS)–87 (MultistepIntegration) will remain the same. Just be sure to use the appropriate solvers and preconditioners for the implicit systems. We described the changes needed in Sect. 7.1.

7.3.5 Benchmark Solution: Advection and Diffusion in a Non-Square Geometry

The first benchmark problem is to compute the constant coefficient advection-diffusion problem with boundary and initial conditions to match the exact solution (5.123) in a non square domain. The domain we choose is bounded by the four curves

$$M \quad \begin{cases} \mathbf{\Gamma}_1(\xi) = \frac{3\xi}{2}\hat{x} - (0.3 + 0.35(\tanh(2\xi) + 1))\,\hat{y}, \\ \mathbf{\Gamma}_2(\eta) = \frac{3}{2}\hat{x} + (\frac{1}{2} + 0.35(\tanh(3) + 1))\eta\hat{y}, \\ \mathbf{\Gamma}_3(\xi) = \frac{3\xi}{2}\hat{x} + (0.3 + 0.35(\tanh(2\xi) + 1))\,\xi\,\hat{y}, \\ \mathbf{\Gamma}_4(\eta) = -\frac{3}{2}\hat{x} + (0.3 + 0.35(\tanh(3) + 1))\,\eta\hat{y}. \end{cases} \tag{7.82}$$

When we apply the transfinite interpolation, we generate a grid like that shown in Fig. 7.9.

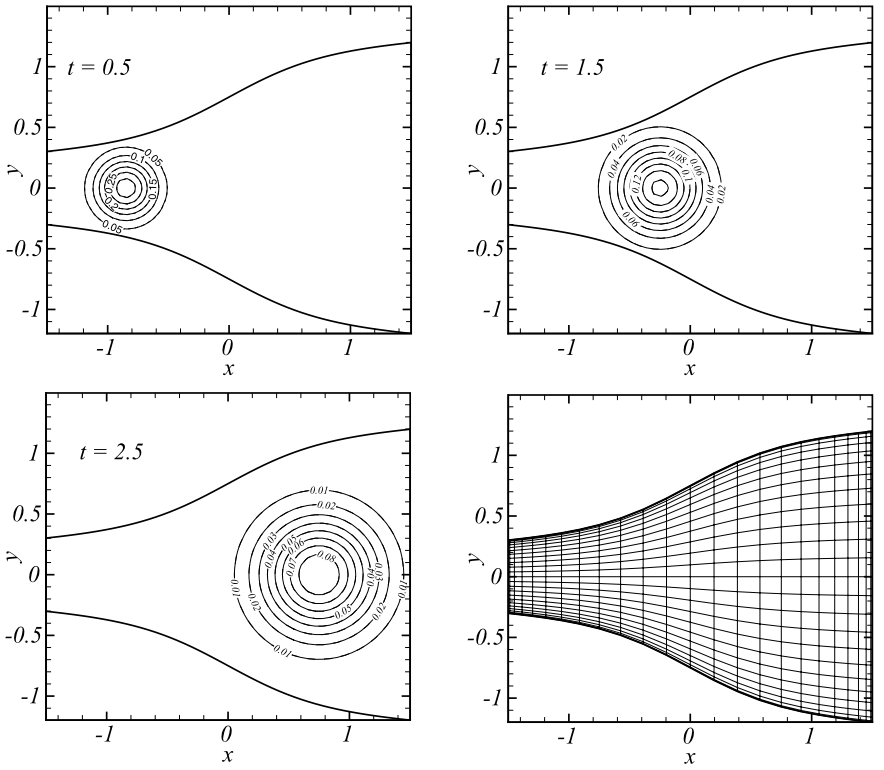

Fig. 7.9 Solutions and grid for the constant coefficient advection-diffusion equation. Exact solutions are contoured by *dashed lines*, computed by *solid lines*. Solutions are interpolated from the original 24×24 grid to a 50×50 grid to plot

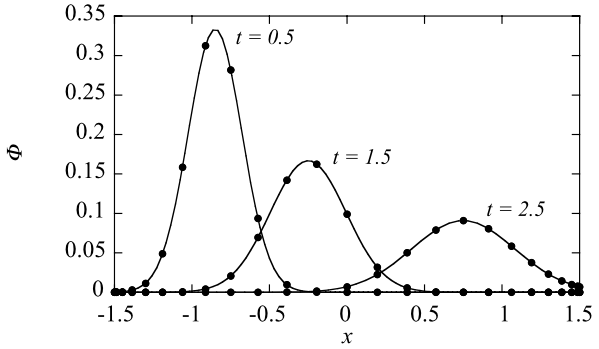

Fig. 7.10 Exact (*line*) and computed (*symbol*) solutions along $y = 0$ for the constant coefficient advection-diffusion equation

We show contour plots of the exact and computed solutions for three times in Fig. 7.9. The parameters for the solutions were $u = 0.8$, $v = 0$, $x_0 = -1.25$, $y_0 = 0$, and $\nu = 0.2$. We used a 24×24 node grid and $\Delta t = 5 \times 10^{-3}$ to compute the solutions. In Fig. 7.10 we show the exact and computed solutions along the centerline, $y = 0$ at three times.

7.3.6 Benchmark Solution: Advection and Diffusion of a Pollutant in a Curved Channel

A more complex problem is to compute the transport and non-isotropic diffusion of a pollutant in a curved channel. That problem is modeled by the equation

$$\varphi_t + \mathbf{q}(x,t) \cdot \nabla\varphi = \frac{\partial}{\partial x}\left(\nu^{(x)}\frac{\partial\varphi}{\partial x}\right) + \frac{\partial}{\partial y}\left(\nu^{(y)}\frac{\partial\varphi}{\partial y}\right). \tag{7.83}$$

If we choose the model parameters to be the variable coefficients

$$\begin{aligned} u &= u_0 x, \qquad v = -u_0 y, \\ \nu^{(x)} &= \nu_0 u_0^2 x^2, \qquad \nu^{(y)} = \nu_0 u_0^2 y^2, \end{aligned} \tag{7.84}$$

there is an analytical solution

$$\begin{aligned} \varphi(x,y,t) = {} & \frac{1}{2\pi\nu_0 u_0^2 (t+\delta)\sqrt{xyx_0y_0}} \\ & \times \left\{ \left(\frac{xy_0}{x_0 y}\right)^{\frac{1}{2u_0\nu_0}} \exp\left(\frac{-\rho_1^2 - 2(1+\nu_0^2 u_0^2)(t+\delta)^2}{4\nu_0(t+\delta)}\right) \right. \\ & \left. - \left(\frac{xy_1}{x_1 y}\right)^{\frac{1}{2u_0\nu_0}} \exp\left(\frac{-\rho_2^2 - 2(1+\nu_0^2 u_0^2)(t+\delta)^2}{4\nu_0(t+\delta)}\right) \right\}. \end{aligned} \tag{7.85}$$

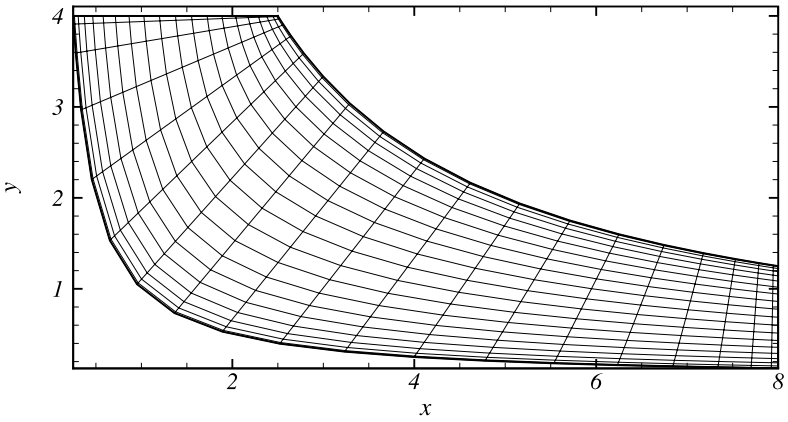

Fig. 7.11 Grid for nonisotropic advection-diffusion in a curved channel

The point (x_0, y_0) corresponds to the initial location of pollutant, a^2 is a parameter that defines the streamline along which the concentration vanishes,

$$\begin{aligned} \rho_1 &= \frac{1}{u_0}\sqrt{\ln^2\left(\frac{x}{x_0}\right) + \ln^2\left(\frac{y}{y_0}\right)}, \\ \rho_2 &= \frac{1}{u_0}\sqrt{\ln^2\left(\frac{x}{x_1}\right) + \ln^2\left(\frac{y}{y_1}\right)} \end{aligned} \tag{7.86}$$

and

$$x_1 = \frac{a^2}{y_0}, \qquad y_1 = \frac{a^2}{x_0}. \tag{7.87}$$

The streamlines for the flow are the lines $xy = const$ so if we choose two of the boundaries to correspond to these streamlines, we can simulate the flow in a curved channel. For this benchmark, we choose the four boundaries to be

$$\begin{cases} \mathbf{\Gamma}_1(\xi) = (8 + 7.75\cos(\pi + \xi\pi/2))\,\hat{x} + (8 + 7.75\cos(\pi + \xi\pi/2))^{-1}\,\hat{y}, \\ \mathbf{\Gamma}_2(\eta) = 8\hat{x} + (0.125 + 1.125\eta)\,\hat{y}, \\ \mathbf{\Gamma}_3(\xi) = (8 + 5.5\cos(\pi + \xi\pi/2))\,\hat{x} + 10\,(8 + 7.75\cos(\pi + \xi\pi/2))^{-1}\,\hat{y}, \\ \mathbf{\Gamma}_4(\eta) = (0.25 + 2.25\eta)\,\hat{x} + 4\hat{y}. \end{cases} \tag{7.88}$$

The lower boundary $\mathbf{\Gamma}_1$ is the streamline for which $a^2 = 1$. The linear transfinite map of the four curves produces the geometry and grid for $N = 20$ shown in Fig. 7.11.

We only need to make two changes to the algorithms we developed earlier to solve the channel problem. First, we must modify the diffusive fluxes (7.74) to incorporate the variable and non-isotropic diffusion. How to do that is the subject

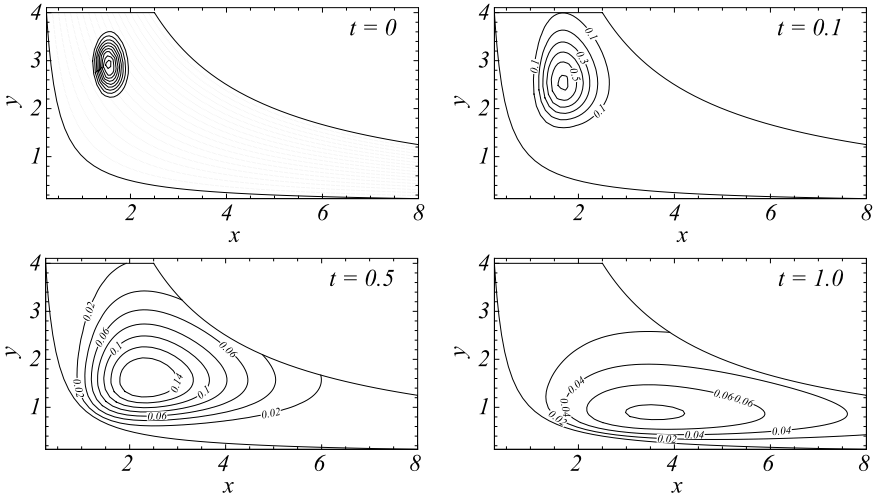

Fig. 7.12 Exact and computed solutions of nonisotropic advection-diffusion in a curved channel. Exact contours are shown as *dashed lines*, computed as *solid lines*. At time $t = 0$ the streamlines are plotted

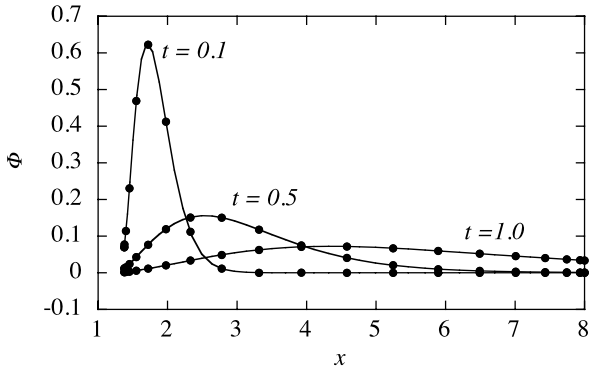

Fig. 7.13 Exact (*line*) and computed (*symbol*) solutions of nonisotropic advection-diffusion in a curved channel along the center grid line

of Problem 7.8. Second, the transport terms, (7.72) and (5.110) must evaluate the velocities u and v at the nodes.

We show benchmark solutions for $N = M = 20$ at three times in Figs. 7.12 and 7.13.

7.4 Conservation Laws in Quadrilateral Domains

In Sect. 5.4 we showed how to approximate a system of conservation laws

$$\mathbf{q}_t + \mathbf{f}_x + \mathbf{g}_y = 0 \tag{7.89}$$

on the square by a nodal discontinuous Galerkin method. In this section, we go beyond the square and we show how to use the transformations that we developed in Chap. 6 to use the method to solve conservation laws in quadrilateral domains.

7.4.1 The Nodal Discontinuous Galerkin Approximation

Under a mapping from the physical domain to the reference square, we use (6.61) to transform the conservation law (7.89) to

$$\mathbf{q}_t + \frac{1}{J}\left\{\frac{\partial \tilde{\mathbf{f}}}{\partial \xi} + \frac{\partial \tilde{\mathbf{g}}}{\partial \eta}\right\} = 0, \tag{7.90}$$

where $\tilde{\mathbf{f}}$ and $\tilde{\mathbf{g}}$ are the contravariant fluxes

$$\begin{aligned} \tilde{\mathbf{f}} &= Y_\eta \mathbf{f} - X_\eta \mathbf{g}, \\ \tilde{\mathbf{g}} &= -Y_\xi \mathbf{f} + X_\xi \mathbf{g}. \end{aligned} \tag{7.91}$$

To approximate the solution of (7.90), we need only to extend the derivation of Sect. 5.4. As usual, we approximate the solution and the contravariant fluxes by polynomials, $\mathbf{Q} \approx \mathbf{q}$, $\tilde{\mathbf{F}} \approx \tilde{\mathbf{f}}$ and $\tilde{\mathbf{G}} \approx \tilde{\mathbf{g}}$ where now

$$\begin{aligned} \tilde{\mathbf{F}}_{i,j} &= \left(Y_\eta\right)_{i,j} \mathbf{f}\left(\mathbf{Q}_{i,j}\right) - \left(X_\eta\right)_{i,j} \mathbf{g}\left(\mathbf{Q}_{i,j}\right), \\ \tilde{\mathbf{G}}_{i,j} &= -\left(Y_\xi\right)_{i,j} \mathbf{f}\left(\mathbf{Q}_{i,j}\right) + \left(X_\xi\right)_{i,j} \mathbf{g}\left(\mathbf{Q}_{i,j}\right). \end{aligned} \tag{7.92}$$

The nodal values of the solution are found by way of the Galerkin projection on the reference square

$$\left(\mathbf{Q}_t, \phi_{ij}\right) + \left(\frac{1}{J}\left\{\frac{\partial \tilde{\mathbf{F}}}{\partial \xi} + \frac{\partial \tilde{\mathbf{G}}}{\partial \eta}\right\}, \phi_{ij}\right) = 0. \tag{7.93}$$

The inner products are the integrals over the reference square. Therefore

$$\left(\mathbf{Q}_t, \phi_{ij}\right) = \int_{-1}^{1}\int_{-1}^{1} \mathbf{Q}_t \phi_{ij} J d\xi d\eta. \tag{7.94}$$

When we choose the Lagrange basis that interpolates the Legendre Gauss points, replace the integrals by Legendre Gauss quadrature, the inner product of the time derivative with the basis functions leads to the approximation

$$\left(\mathbf{Q}_t, \phi_{ij}\right) \approx \left(\mathbf{Q}_t, \ell_i^{(\xi)} \ell_j^{(\eta)}\right)_N = \dot{\mathbf{Q}}_{i,j} J_{i,j} w_i^{(\xi)} w_j^{(\eta)}. \tag{7.95}$$

Before we approximate the second inner product in (7.93) we integrate by parts, which moves the derivative onto the basis functions and generates boundary integrals. That is, we convert

$$\left(\frac{1}{J}\left\{\frac{\partial\tilde{\mathbf{F}}}{\partial\xi}+\frac{\partial\tilde{\mathbf{G}}}{\partial\eta}\right\},\phi_{ij}\right)=\int_{-1}^{1}\int_{-1}^{1}\frac{1}{J}\left\{\frac{\partial\tilde{\mathbf{F}}}{\partial\xi}+\frac{\partial\tilde{\mathbf{G}}}{\partial\eta}\right\}\phi_{ij}J\,d\xi\,d\eta$$
$$=\int_{-1}^{1}\int_{-1}^{1}\left\{\frac{\partial\tilde{\mathbf{F}}}{\partial\xi}+\frac{\partial\tilde{\mathbf{G}}}{\partial\eta}\right\}\phi_{ij}\,d\xi\,d\eta \tag{7.96}$$

to

$$\left(\frac{1}{J}\left\{\frac{\partial\tilde{\mathbf{F}}}{\partial\xi}+\frac{\partial\tilde{\mathbf{G}}}{\partial\eta}\right\},\phi_{ij}\right)$$
$$=\int_{-1}^{1}\left\{\phi_{ij}\tilde{\mathbf{F}}\Big|_{\xi=-1}^{1}-\int_{-1}^{1}\tilde{\mathbf{F}}\frac{\partial\phi_{ij}}{\partial\xi}d\xi\right\}d\eta$$
$$+\int_{-1}^{1}\left\{\phi_{ij}\tilde{\mathbf{G}}\Big|_{\eta=-1}^{1}-\int_{-1}^{1}\tilde{\mathbf{G}}\frac{\partial\phi_{ij}}{\partial\eta}d\eta\right\}d\xi. \tag{7.97}$$

When we gather terms, we have

$$\left(\frac{1}{J}\left\{\frac{\partial\tilde{\mathbf{F}}}{\partial\xi}+\frac{\partial\tilde{\mathbf{G}}}{\partial\eta}\right\},\phi_{ij}\right)=\int_{-1}^{1}\phi_{ij}\tilde{\mathbf{F}}\Big|_{\xi=-1}^{1}d\eta+\int_{-1}^{1}\phi_{ij}\tilde{\mathbf{G}}\Big|_{\eta=-1}^{1}d\xi$$
$$-\int_{-1}^{1}\int_{-1}^{1}\left\{\tilde{\mathbf{F}}\frac{\partial\phi_{ij}}{\partial\xi}+\tilde{\mathbf{G}}\frac{\partial\phi_{ij}}{\partial\eta}\right\}d\xi\,d\eta, \tag{7.98}$$

which should be carefully compared to (5.142).

Now that we have integrated by parts, we approximate the integrals by Legendre Gauss quadrature and use the Lagrange basis that interpolates the associated quadrature points. When we follow the arguments that lead to (5.147), (5.151) and (5.152) we get the nodal approximation for the conservation law on a quadrilateral domain

$$\frac{d\mathbf{Q}_{i,j}}{dt}+\frac{1}{J_{i,j}}\left\{\left[\tilde{\mathbf{F}}^{*}(1,\eta_j)\frac{\ell_i(1)}{w_i^{(\xi)}}-\tilde{\mathbf{F}}^{*}(-1,\eta_j)\frac{\ell_i(-1)}{w_i^{(\xi)}}\right]+\sum_{k=0}^{N}\tilde{\mathbf{F}}_{k,j}\hat{D}_{ik}^{(\xi)}\right\}$$
$$+\frac{1}{J_{i,j}}\left\{\left[\tilde{\mathbf{G}}^{*}(\xi_i,1)\frac{\ell_j(1)}{w_j^{(\eta)}}-\tilde{\mathbf{G}}^{*}(\xi_i,-1)\frac{\ell_j(-1)}{w_j^{(\eta)}}\right]+\sum_{k=0}^{N}\tilde{\mathbf{G}}_{i,k}\hat{D}_{jk}^{(\eta)}\right\}=0. \tag{7.99}$$

When we compare (7.99) to its equivalent on the square, (5.152), we see that they differ only in the fact that the approximation on a general quadrilateral domain uses the contravariant rather than the covariant fluxes. Along the boundaries, we must use the Riemann solver to compute the contravariant fluxes rather than the normal fluxes. Fortunately, there is a simple relationship between the two.

To understand how to use the Riemann solver to find the boundary contravariant fluxes, we go back to the argument that led to the relation (7.12) between the boundary and normal fluxes for the Neumann problem for the advection-diffusion equation. The polynomial approximation of the contravariant flux in the $\hat{\xi}$ direction is

$$\tilde{\mathbf{F}} = Y_\eta \mathbf{F} - X_\eta \mathbf{G} = \left(\mathbf{F}\hat{x} + \mathbf{G}\hat{y}\right) \cdot J\mathbf{a}^1. \tag{7.100}$$

But boundary normals are proportional to the vectors $J\mathbf{a}^1$ and $J\mathbf{a}^2$ (cf. (6.60)), precisely, $\hat{n}^1 = J\mathbf{a}^1/|J\mathbf{a}^1|$ and $\hat{n}^2 = J\mathbf{a}^2/|J\mathbf{a}^2|$. Therefore,

$$\tilde{\mathbf{F}} = \left|J\mathbf{a}^1\right| \left(\mathbf{F}\hat{x} + \mathbf{G}\hat{y}\right) \cdot \hat{n}^1. \tag{7.101}$$

To compute the boundary fluxes with the Riemann solver, then, we need only to compute the normal Riemann flux and then scale by $|J\mathbf{a}^1|$. For example along the boundary $\mathbf{\Gamma}_2$, we compute the boundary flux $\tilde{\mathbf{F}}^*(1, \eta_j)$ by

$$\begin{aligned} \tilde{\mathbf{F}}^*(1, \eta_j) &= \left|J\mathbf{a}^1\right| \mathbf{F}^* \left(\mathbf{Q}\left(1, \eta_j\right), \mathbf{Q}^{ext}; \hat{n}\right) \\ &= \frac{|J|\sqrt{Y_\eta^2 + X_\eta^2}}{J} \mathbf{F}^* \left(\mathbf{Q}\left(1, \eta_j\right), \mathbf{Q}^{ext}; \hat{n}\right) \end{aligned} \tag{7.102}$$

using (6.60).

7.4.2 How to Implement the Nodal Discontinuous Galerkin Approximation

From the discussion above, we see that we can implement the nodal discontinuous Galerkin approximation by additions to and modifications of the algorithms for the square developed in Sect. 5.4.

7.4.2.1 Data Storage

For general quadrilateral domains, we need to add metric and boundary information, as we did to extend the NodalPotentialClass to get the MappedNodalPotentialClass, Algorithm 103. To implement the time derivative for the mapped equations, we could simply modify the algorithm (and code) for the time derivative on the square. Physical space locations that are needed by the external boundary functions are now

Algorithm 110: *DGSolutionStorage:* Storage of Interior and Boundary Solutions

Structure DGSolutionStorage
Data:
nEqn
$\{Q_{i,j,n}\}_{i=0;j=0;n=1}^{N;M;nEqn}$; // interior point solution values
$\{\dot{Q}_{i,j,n}\}_{i=0;j=0;n=1}^{N;M;nEqn}$; // interior point solution time derivative
$\{Qb_{i,n,k}\}_{i=0;n=1;k=1}^{Max(N,M);nEqn;4}$; // boundary solution values
$\{Fb^*_{i,n,k}\}_{i=0;n=1;k=1}^{Max(N,M);nEqn;4}$; // boundary flux fluxes
End Structure DGSolutionStorage

located in the x and y arrays of the MappedGeometry structure. The normals at the boundaries are no longer the $\pm\hat{x}$ and $\pm\hat{y}$ directions, but are computed from the mapping function and are also stored in the MappedGeometry structure. The physical flux routines *xFlux* and *yFlux* remain the same. However, we now have to compute the divergence of the contravariant flux rather than the covariant flux.

We will not make the straightforward extension of the discontinuous Galerkin algorithms on the square to the mapped domain. Instead, we will refactor the data storage and the procedures in anticipation of reusing them in a spectral element framework later in Chap. 8. The new organization will create a new class in which to store the data, and new procedures that will break the operation of computing the time derivative into its component parts.

To refactor the data, we will create the new solution storage structure shown in Algorithm 110 (DGSolutionStorage). The nodal discontinuous Galerkin method requires the solution in the interior of the domain plus boundary values that are interpolated from the interior. We will now store the boundary values along the four sides in an array so that they can be used outside of the procedure in which they are computed. Since the size of the arrays can differ in the two directions, a simple array is not the best storage model (an array of arrays would be better), but it will suit our purpose here. We will store the numerical boundary fluxes in the same kind of array. The flux arrays are not absolutely necessary, since it turns out that we can reuse the storage used by the boundary solutions. However, having the flux arrays make the algorithms more clear. For convenience and clarity, we also store the time derivative of the solution.

7.4.2.2 The MappedNodalDGClass

Now that we have the data model for the solution, we compose the data model for the nodal discontinuous Galerkin method from the usual nodal storage model, the geometry model and the solution model. We show this storage model in Algorithm 111 (MappedNodalDG2DClass). In that class we also show two new procedures. The

Algorithm 111: *MappedNodalDG2DClass:* A Discontinuous Galerkin Class Definition

Class MappedNodalDG2DClass
Uses Algorithms:
Algorithm 89 (NodalDG2DStorage)
Algorithm 110 (DGSolutionStorage)
Algorithm 101 (MappedGeometryClass)
Data:
spA ; `// Of type NodalDG2DStorage`
dGS ; `// Of type DGSolutionStorage`
geom ; `// Of type MappedGeometry`
Procedures:
Construct$(nEqn, N, M, \{\Gamma_k\}_{k=1}^4)$; `// Algorithm 91, extended`
GlobalTimeDerivative(t); `// Algorithm 115`
End Class MappedNodalDG2DClass

constructor has changed from the one we used on the square. We have changed the constructor from

$$Construct(nEqn, N, M)$$

to

$$Construct\big(nEqn, N, M, \{\Gamma_k\}_{k=1}^4\big),$$

where the Γ_k are curve interpolants defined in Algorithm 96 (CurveInterpolant). To the constructor we then add

$$this.geom.Construct\big(N, M, \{\mathbf{\Gamma}_j\}_{j=1}^4, this.spA\big)$$

to create the metric terms, normals, etc. that are needed by the approximation.

7.4.2.3 The Time Derivative

The second new procedure that we need to write computes the time derivative. If we look back at Algorithm 93 (DG2DTimeDerivative), we see that it performs three fundamental operations. It interpolates the solutions to the boundaries and computes the numerical fluxes from the interpolated values and the external state using the Riemann solver. Finally it computes the time derivative from the spatial derivatives of the fluxes. When we refactor the algorithm into these three parts, we get Algorithms 112 (DG2DProlongToFaces), 113 (MappedDG2DBoundaryFluxes), and 114 (MappedDG2DTimeDerivative). Put them together as we do in Algorithm 115 (DG2DTimeDerivative), and we have extended and replaced the equivalent time derivative routine for the square.

Algorithm 112: *DG2DProlongToFaces:* Interpolate the Solution from Gauss Points to the Boundaries

Procedure DG2DProlongToFaces
Input: *spA* ; `// Of type NodalDG2DStorage`
Input: *geom* ; `// Of type MappedGeometryClass`
Input: *dGS* ; `// Of type DGSolutionStorage`
Uses Algorithms:
Algorithm 89 (NodalDG2DStorage)
Algorithm 110 (DGSolutionStorage)
Algorithm 101 (MappedGeometryClass)

$nEqn \leftarrow dGS.nEqn$; $N \leftarrow spA.N$; $M \leftarrow spA.M$
for $j = 0$ **to** M **do**
  for $n = 1$ **to** $nEqn$ **do**
    $dGS.Qb_{j,n,4} \leftarrow InterpolateToBoundary\left(dGS.\left\{Q_{i,j,n}\right\}_{i=0}^{N}, spA.\left\{\ell_i^{(\xi)}(-1)\right\}_{i=0}^{N}\right)$
    $dGS.Qb_{j,n,2} \leftarrow InterpolateToBoundary\left(dGS.\left\{Q_{i,j,n}\right\}_{i=0}^{N}, spA.\left\{\ell_i^{(\xi)}(1)\right\}_{i=0}^{N}\right)$
  end
end
for $i = 0$ **to** N **do**
  for $n = 1$ **to** $nEqn$ **do**
    $dGS.Qb_{i,n,1} \leftarrow InterpolateToBoundary\left(dGS.\left\{Q_{i,j,n}\right\}_{j=0}^{M}, spA.\left\{\ell_j^{(\eta)}(-1)\right\}_{j=0}^{M}\right)$
    $dGS.Qb_{i,n,3} \leftarrow InterpolateToBoundary\left(dGS.\left\{Q_{i,j,n}\right\}_{j=0}^{M}, spA.\left\{\ell_j^{(\eta)}(1)\right\}_{j=0}^{M}\right)$
  end
end
return *dGS*
End Procedure DG2DProlongToFaces

7.4.3 Benchmark Solution: Acoustic Scattering off a Cylinder

The noise inside an aircraft due to a propeller or other engine noise source comes from the acoustic pressure loading on the surface of the fuselage. To model such loading, we can represent the aircraft by a circular cylinder and the engine by a source placed nearby, as sketched in Fig. 7.14. We have seen in Fig. 5.4.3 how well the spectral discontinuous Galerkin approximation approximates reflection from a straight boundary. With this model we can test how well it approximates scattering off a curved boundary.

The model problem and the mapping to the reference square is shown in Fig. 7.14. The initial velocities are $u = v = 0$. The sound source will be an initial pressure perturbation

$$p(x, y, 0) = e^{-\ln(2)\left(\frac{(x-x_s)^2+y^2}{w^2}\right)}, \tag{7.103}$$

where x_s is the location of the initial disturbance along the x axis. Since the cylinder and the source are symmetric about the line $y = 0$, we only need to solve in the upper half plane. Since the solution reflects across the symmetry boundary, we implement it as a wall, i.e. reflection boundary. Therefore, we set wall reflection

Algorithm 113: *MappedDG2DBoundaryFluxes:* Boundary Fluxes in 2D for the Discontinuous Galerkin Approximation

Procedure MappedDG2DBoundaryFluxes
Input: t
Input: spA ; `// Of type DGNodal2DStorage`
Input: $geom$; `// Of type MappedGeometryClass`
Input: dGS ; `// Of type DGSolutionStorage`
Uses Algorithms:
Algorithm 88 (RiemannSolver)

$nEqn \leftarrow dGS.nEqn$; $N \leftarrow spA.N$; $M \leftarrow spA.M$
for $j = 0$ **to** M **do**
  $\left\{Q_n^{L,ext}\right\}_{n=1}^{nEqn} \leftarrow$
  $ExternalState\left(dGS.\left\{Qb_{j,n,4}\right\}_{n=1}^{nEqn}, geom.\hat{n}_j^4, geom.x_j^4, geom.y_j^4, t, LEFT\right)$
  $dGS.\left\{F_{j,n,4}^*\right\}_{n=1}^{nEqn} \leftarrow$
  $geom.scal_j^4 * RiemannSolver\left(dGS.\left\{Qb_{j,n,4}\right\}_{n=1}^{nEqn}, \left\{Q_n^{L,ext}\right\}_{n=1}^{nEqn}, geom.\hat{n}_j^4\right)$
  $\left\{Q_n^{R,ext}\right\}_{n=1}^{nEqn} \leftarrow$
  $ExternalState\left(dGS.\left\{Qb_{j,n,2}\right\}_{n=1}^{nEqn}, geom.\hat{n}_j^2, geom.x_j^2, geom.y_j^2, t, RIGHT\right)$
  $dGS.\left\{F_{j,n,2}^*\right\}_{n=1}^{nEqn} \leftarrow$
  $geom.scal_j^2 * RiemannSolver\left(dGS.\left\{Qb_{j,n,2}\right\}_{n=1}^{nEqn}, \left\{Q_n^{R,ext}\right\}_{n=1}^{nEqn}, geom.\hat{n}_j^2\right)$
end
for $i = 0$ **to** N **do**
  $\left\{Q_n^{B,ext}\right\}_{n=1}^{nEqn} \leftarrow$
  $ExternalState\left(dGS.\left\{Qb_{i,n,1}\right\}_{n=1}^{nEqn}, geom.\hat{n}_i^1, geom.x_i^1, geom.y_i^1, t, BOTTOM\right)$
  $dGS.\left\{F_{i,n,1}^*\right\}_{n=1}^{nEqn} \leftarrow$
  $geom.scal_i^1 * RiemannSolver\left(dGS.\left\{Qb_{i,n,1}\right\}_{n=1}^{nEqn}, \left\{Q_n^{B,ext}\right\}_{n=1}^{nEqn}, geom.\hat{n}_i^1\right)$
  $\left\{Q_n^{T,ext}\right\}_{n=1}^{nEqn} \leftarrow$
  $ExternalState\left(dGS.\left\{Qb_{i,n,3}\right\}_{n=1}^{nEqn}, geom.\hat{n}_i^3, geom.x_i^3, geom.y_i^3, t, TOP\right)$
  $dGS.\left\{F_{i,n,3}^*\right\}_{n=1}^{nEqn} \leftarrow$
  $geom.scal_i^3 * RiemannSolver\left(dGS.\left\{Qb_{i,n,3}\right\}_{n=1}^{nEqn}, \left\{Q_n^{T,ext}\right\}_{n=1}^{nEqn}, geom.\hat{n}_i^3\right)$
end
return dGS
End Procedure MappedDG2DBoundaryFluxes

conditions along boundaries Γ_1 and Γ_3. The surface of the cylinder, which is Γ_4, is also a wall boundary. The outer boundary, Γ_2 should be a transparent or radiation boundary. We will approximate the radiation boundary by setting the external state to be zero everywhere. Since the physical model implied by this approximation is that there are no incoming plane waves, and since the expanding waves will be circular, there will be an artificial reflection when waves hit the outer boundary. In our benchmark solution we choose the outer boundary far enough away so that the sound waves have not yet reached it at the final time. A better approach would be to use Perfectly Matched Layer (PML) approximations at the radiation boundary, an

Algorithm 114: *MappedDG2DTimeDerivative:* Time derivative in 2D for the Discontinuous Galerkin Approximation

Procedure MappedDG2DTimeDerivative
Input: *spA* ; `// Of type DGNodal2DStorage`
Input: *geom* ; `// Of type MappedGeometryClass`
Input: *dGS* ; `// Of type DGSolutionStorage`
Uses Algorithms:
Algorithm 92 (SystemDGDerivative)
Algorithm 94 (WaveEquationFluxes)

$nEqn \leftarrow dGS.nEqn$; $N \leftarrow spA.N$; $M \leftarrow spA.M$
for $j = 0$ **to** M **do**
  for $i = 0$ **to** N **do**
    $\left\{F_{i,n}\right\}_{n=1}^{nEqn} \leftarrow$
    $(geom.Y_\eta)_{i,j} * xFlux\left(this.\left\{Q_{i,j,n}\right\}_{n=1}^{nEqn}\right) - (geom.X_\eta)_{i,j} * yFlux\left(this.\left\{Q_{i,j,n}\right\}_{n=1}^{nEqn}\right)$
  end
  $\left\{F'_{i,n}\right\}_{i=0;n=1}^{N;nEqn} \leftarrow$
  $SystemDGDerivative\left(dGS.\left\{F^*_{j,n,4}\right\}_{n=1}^{nEqn}, dGS.\left\{F^*_{j,n,2}\right\}_{n=1}^{nEqn}, \left\{F_{i,n}\right\}_{i=0;n=1}^{N;nEqn}, spA.\left\{D^\xi\right\}_{i,j=0}^{N},\right.$
  $\left.spA.\left\{\ell_i^{(\xi)}(-1)\right\}_{i=0}^{N}, spA.\left\{\ell_i^{(\xi)}(1)\right\}_{i=0}^{N}, spA.\left\{w_i^{(\xi)}\right\}_{i=0}^{N}\right)$
  for $i = 0$ **to** N **do**
    for $n = 1$ **to** $nEqn$ **do**
      $dGS.\dot{Q}_{i,j,n} \leftarrow -F'_{i,n}$
    end
  end
end
for $i = 0$ **to** N **do**
  for $j = 0$ **to** M **do**
    $\left\{G_{j,n}\right\}_{n=1}^{nEqn} \leftarrow -(geom.Y_\xi)_{i,j} * xFlux\left(this.\left\{Q_{i,j,n}\right\}_{n=1}^{nEqn}\right) + (geom.X_\xi)_{i,j} *$
    $yFlux\left(this.\left\{Q_{i,j,n}\right\}_{n=1}^{nEqn}\right)$
  end
  $\left\{G'_{j,n}\right\}_{j=0;n=1}^{M;nEqn} \leftarrow$
  $SystemDGDerivative\left(dGS.\left\{F^*_{i,n,1}\right\}_{n=1}^{nEqn}, dGS.\left\{F^*_{i,n,3}\right\}_{n=1}^{nEqn}, \left\{G_{j,n}\right\}_{j=0;n=1}^{N;nEqn}, spA.\left\{D^\eta\right\}_{i,j=0}^{M},\right.$
  $\left.spA.\left\{\ell_i^{(\eta)}(-1)\right\}_{i=0}^{M}, spA.\left\{\ell_i^{(\eta)}(1)\right\}_{i=0}^{M}, spA.\left\{w_i^{(\eta)}\right\}_{i=0}^{M}\right)$
  for $j = 0$ **to** M **do**
    for $n = 1$ **to** $nEqn$ **do**
      $dGS.\dot{Q}_{i,j,n} \leftarrow (dGS.\dot{Q}_{i,j,n} - G'_{j,n})/geom.J_{i,j}$
    end
  end
end
return *dGS*
End Procedure MappedDG2DTimeDerivative

advanced method applicable to many types of approximations that we will not cover here.

We present benchmark solutions for the scattering problem when the half width is $w = 0.125$, the cylinder radius is $r_0 = 0.5$ and $x_s = 1.5$ for time up to $t = 3$. We

Algorithm 115: *GlobalTimeDerivative:* Full Time Derivative in 2D for the Discontinuous Galerkin Approximation

Procedure GlobalTimeDerivative
Input: t
Uses Algorithms:
Algorithm 112 (DG2DProlongToFaces); Algorithm 113 (MappedDG2DBoundaryFluxes); Algorithm 114 (MappedDGSystemTimeDerivative);

this.dGS ← *DG2DProlongToFaces*(*this.spA*, *this.geom*, *this.dGS*)
this.dGS ← *MappedDG2DBoundaryFluxes*(*t*, *this.spA*, *this.geom*, *this.dGS*)
this.dGS ← *MappedDG2DTimeDerivative*(*this.spA*, *this.geom*, *this.dGS*)
End Procedure DG2DTimeDerivative

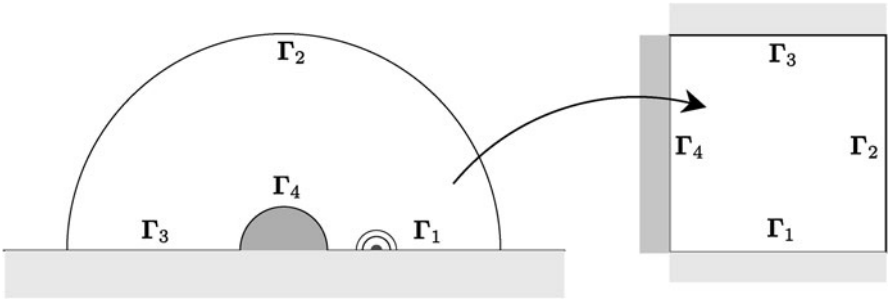

Fig. 7.14 Model for scattering of a sound wave off a circular cylinder

choose the benchmark domain to be the half circle with inner radius r_0 and outer radius $r_\infty = 5$, which is mapped onto the reference square by the transfinite mapping of the boundary curves

$$M \quad \begin{cases} \mathbf{\Gamma}_1(\xi) = r_0 + (r_\infty - r_0)(\xi + 1)/2\hat{x} + 0\hat{y}, \\ \mathbf{\Gamma}_2(\eta) = r_\infty \cos(\pi(\eta + 1)/2)\hat{x} + r_\infty \sin(\pi(\eta + 1)/2)\hat{y}, \\ \mathbf{\Gamma}_3(\xi) = -r_0 - (r_\infty - r_0)(\xi + 1)/2\hat{x} + 0\hat{y}, \\ \mathbf{\Gamma}_4(\eta) = r_0 \cos(\pi(\eta + 1)/2)\hat{x} + r_0 \sin(\pi(\eta + 1)/2)\hat{y}. \end{cases} \tag{7.104}$$

Under this mapping, the grid looks similar to that shown in Fig. 7.4, which we used to compute the incompressible flow over a circular obstacle. We compute the solutions with $N = M = 74$ and $\Delta t = 5 \times 10^{-5}$.

We show the computed pressure contours for $t = 3$ in Fig. 7.15. The front farthest from the cylinder has its center at $x = 1.5$ and was created by the initial pressure disturbance. The next closest front to the cylinder has its center at $x = 0$ and is the wave reflected by the cylinder. Also visible is a weak sound wave created when the incident wave front merges to the left of the cylinder. To produce the contour plot, we used Algorithm 35 (2DCoarseToFineInterpolation) to interpolate the solution and grid onto 100 uniformly spaced points in each direction.

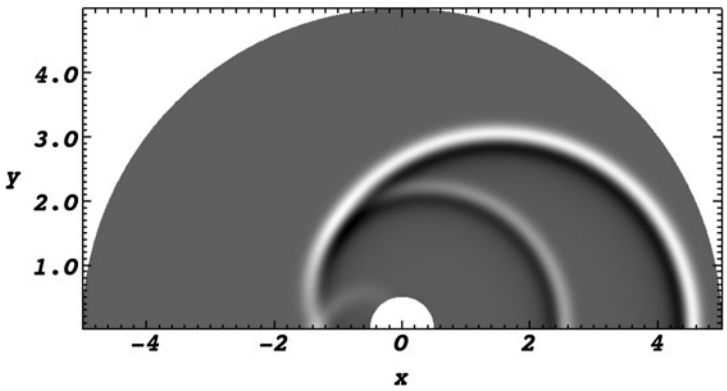

Fig. 7.15 Pressure contours at $t = 3$ for the model acoustic scattering of an initial pressure disturbance off a circular cylinder

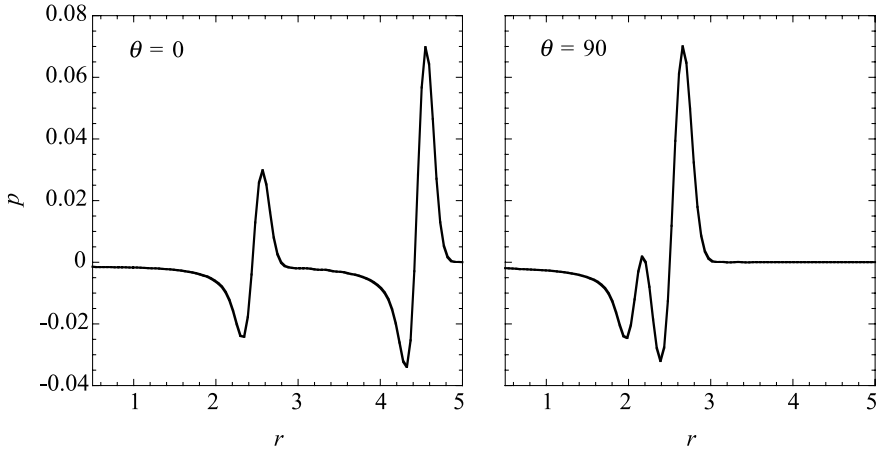

Fig. 7.16 Incident and scattered pressure values at $t = 3$ at two angles from the origin

We present profiles of the pressure at two angles about the cylinder, interpolated to the fine grid, in Fig. 7.16. At angle $\theta = 0°$, along $y = 0$ to the right of the cylinder, the incident and reflected waves are well separated. The waves are still interacting along the line $x = 0$, which corresponds to $\theta = 90°$.

Exercises

7.1 Derive equations (7.5) and (7.6).

7.2 Newton's law of cooling says that the normal heat flux at a boundary is proportional to the difference between the surface and ambient temperatures,

$$\nabla\varphi \cdot \hat{n} = -\gamma\,(\varphi - \varphi_0)\,.$$

Write this boundary condition for a curved boundary of a quadrilateral domain.

7.3 Derive the nodal Galerkin approximation to approximate the potential in the annulus of Fig. 7.7 with potentials specified along the boundaries.

7.4 Derive the nodal Galerkin and collocation approximations to approximate the steady temperature in the annulus of Fig. 7.7 for a specified temperature distribution along the inner cylinder and an insulated outer cylinder, i.e. $\nabla\varphi \cdot \hat{n} = 0$.

7.5 Derive the collocation approximation (7.71)–(7.74) to the advection-diffusion equation.

7.6 Explain why the nodal Galerkin approximation of the diffusion terms of the advection-diffusion equation and (7.95) are only approximate on a mapped domain after quadrature, in contrast to the approximation on the square.

7.7 Derive equations (7.76)–(7.78) for the nodal Galerkin approximation to the advection-diffusion equation.

7.8 Derive the collocation and nodal Galerkin approximations to the non-isotropic advection-diffusion equation (7.83) and show how to modify the algorithms to implement the methods.

7.9 The conservation form of the advection term is $\nabla \cdot (\mathbf{q}\varphi)$. Show how to approximate this in a mapped coordinate system.

7.10 Implement the spectral collocation method to compute the inviscid, incompressible flow over a cylinder. Use the computed potential to compute and plot spectrally accurate approximations to the velocity.

7.11 Implement the nodal Galerkin method to compute the inviscid, incompressible over a cylinder. Compare the accuracy to that of the Chebyshev collocation method.

7.12 *Fluid flow in a concentric pipe.* An important class of viscous fluid flows for which exact solutions exist are those in pipes of arbitrary yet constant shape. To force fluid down the pipe, an inlet pressure higher than the outlet pressure is applied. In the steady state, far away from the inlet and outlet, we can assume that there is no variation of the velocity in the axial, $\hat{z}$, direction and that the pressure varies linearly from the inlet to the outlet. The fact that the pipe has a constant cross section means that we can assume that the radial and azimuthal velocity components are zero. The viscosity in the flow forces the axial velocity to vanish at the pipe walls. Therefore,

the only unknown is the axial velocity, $u(x, y)$. Under these conditions, conservation of momentum requires that the axial velocity satisfies

$$\nabla^2 u(x, y) = \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial x^2} \right) = \frac{1}{\nu} \frac{\partial p}{\partial z} = \gamma = const < 0.$$

Since the pipe has constant cross section, the axial velocity is really only a function of the radial distance.

The flow in the concentric cylinder shown in Fig. 7.7 has an exact solution, with the axial velocity given by

$$u(r) = -\frac{\gamma}{4} \left\{ r_O^2 - r^2 + \left(r_O^2 - r_I^2 \right) \frac{\ln(r_O/r)}{\ln(r_I/r_O)} \right\}.$$

1. Use a mixed basis spectral collocation method to compute the flow in the concentric cylinder and compare the computed solutions to the exact for velocities scaled so that $\gamma = -1$.
2. Compute to spectral accuracy the *volume flow rate* Q defined by

$$Q = \int_{disk} u dA$$

and compare to the exact analytical value

$$Q = -\frac{\gamma \pi}{8} \left\{ r_O^2 - r_I^2 + \frac{(r_O^2 - r_I^2)^2}{\ln(r_I/r_O)} \right\}$$

7.13 *Fluid flow in an eccentric pipe.* Referring to the flow in a concentric pipe described in Problem 7.12, derive and implement a spectral collocation approximation to solve for the flow rate in the eccentric annular pipe shown in Fig. 7.17. Explain

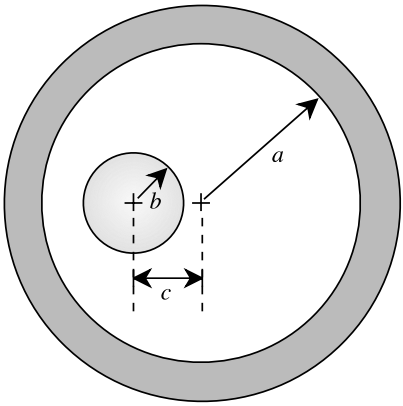

Fig. 7.17 A pipe described by an eccentric annulus

your choice of solver. Compute the flow rate to spectral accuracy and calculate how it varies with relative eccentricity

$$\varepsilon = \frac{c}{a-b}.$$

Finally, compare your results with the analytical solution

$$Q = \frac{-\gamma\pi}{8}\left\{a^4 - b^4 - \frac{4c^2M^2}{\beta-\alpha} - 8c^2M^2\sum_{n=1}^{\infty}\frac{ne^{-n(\alpha+\beta)}}{\sinh(n\beta - n\alpha)}\right\},$$

where

$$M = \sqrt{F^2 - a^2}, \qquad F = \frac{a^2 - b^2 + c^2}{2c},$$
$$\alpha = \frac{1}{2}\ln\frac{F+M}{F-M}, \qquad \beta = \frac{1}{2}\ln\frac{F-c+M}{F-c-M}.$$

7.14 In Sect. 6.3, we noted that if we are not careful when computing the metric terms for a mapped domain that spurious waves can be generated. With the geometry of the benchmark problem of Sect. 7.4.3, and the initial condition $p = const$, $u = v = 0$, compute the nodal discontinuous Galerkin approximation to the wave equation when the boundaries are represented by polynomials of degree $N + 5$ for several low values of N, where N is the polynomial order of the approximation of the solution. Plot the pressure at various times and observe. Compare the results to what you get when the boundaries are approximated by polynomials of order N.

Chapter 8
Spectral Element Methods

The main feature of nodal polynomial spectral methods that give them their power, namely the global polynomial approximation with grid points at the nodes of a Gauss quadrature rule, also limits them to a fairly small class of problems. The global nature of the approximation makes it difficult for us to apply the methods to complex geometries or to problems with discontinuities. The fixed node placement makes it difficult to refine the grid locally as may be needed. The cost to compute the derivatives gets large in large problems that need high order polynomials to resolve all the features in a problem. An example of that is the benchmark problem of Sect. 7.4.3. Coupled with the small time steps to which explicit methods are limited or the large condition numbers that adversely affect the solution of the linear systems that arise from implicit methods, the spectral methods that we have presented so far can become expensive, although very accurate.

To compute problems in geometries that are more complex than those we have presented so far, we introduce multidomain methods. In multidomain spectral methods, we divide the domain of interest into smaller subdomains that we can map individually onto the square. We can then apply a spectral method like one of those discussed in Chap. 7 to each of the subdomains. Multidomain spectral methods have become so useful and so common that the methods that we have derived so far in this book are now often called "single domain" or "mono-domain" methods.

We can extend any of the nodal spectral methods that we have presented so far to a multidomain version. We only need to develop methods to couple the subdomains together. However, in this book we will only discuss a subclass of multidomain methods that starts from the weak form of the equations, that is, the nodal Galerkin methods. The Galerkin based nodal methods have natural coupling that follows from the weak form. We call this subclass "Spectral Element Methods" because of its similarity to finite element methods. We note, however, that some authors refer to spectral multidomain methods in general as spectral element methods. For a taxonomy of multidomain methods and presentations of other methods that use the strong form of the equations, see the book by Canuto et al. [8].

Multidomain methods, and spectral element methods in particular, have many advantages over their single domain counterparts. Let us quickly review situations where we should or must use a multidomain method. They include:

- *Problems in complex geometries.* We can use multiple domain decompositions to solve problems in domains that are difficult to map or cannot be mapped onto a single square. Figure 8.1 shows a decomposition of such a domain into four elements. A more convincing geometry might be what we'd need to compute flow over a three element airfoil like that shown in Fig. 8.2.

D.A. Kopriva, *Implementing Spectral Methods for Partial Differential Equations,* Scientific Computation,

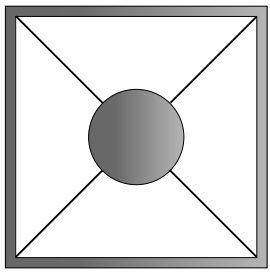

Fig. 8.1 A decomposition of an interior domain by four subdomains

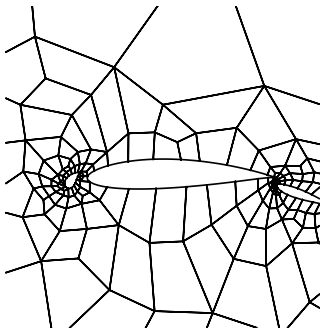

Fig. 8.2 A portion of a decomposition of the exterior of a three element airfoil into multiple subdomains

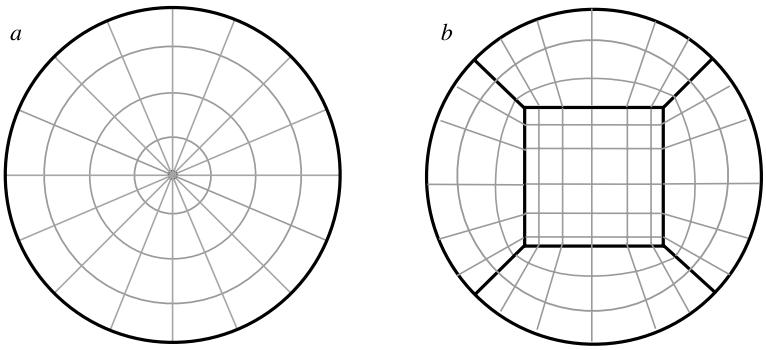

Fig. 8.3 Single (**a**) and multidomain (**b**) decompositions of a disk

- *Problems with coordinate singularities.* Even if the domain is simple, the coordinate mapping onto the square may have singularities. We can use a single domain to approximate an equation on the disk using a cylindrical coordinate system, as seen in Fig. 8.3a. We can eliminate the singularity at the origin by subdividing the disk as in Fig. 8.3b. We can use subdivision to eliminate pole singularities on the sphere as well, as in Fig. 8.4.
- *Problems with discontinuous coefficients or solutions.* Since the convergence rate of spectral methods depends on the smoothness of the solutions, we would not expect accurate solutions when the coefficients in an equation or the solutions are discontinuous. A simple problem on which we should not use a single domain

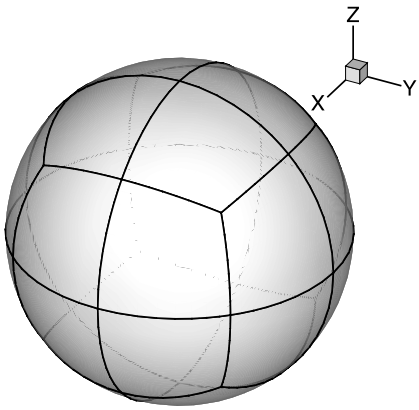

Fig. 8.4 A multidomain decomposition of the surface of a sphere

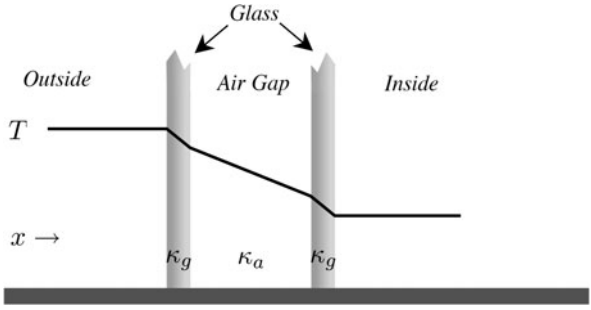

Fig. 8.5 A double glazed window model as an example of a problem with discontinuous media

spectral method is to compute the heat flux through a double glazed window as we sketch in Fig. 8.5. The temperature satisfies the equation

$$\frac{\partial}{\partial x}\left(\kappa \frac{\partial T}{\partial x}\right) = 0. \tag{8.1}$$

The steady temperature, also shown in the figure, is continuous even though the diffusion coefficient, κ, is discontinuous at the material boundaries. If we use a single domain spectral method to solve the problem, the convergence rate of temperature will be slow. On the other hand, if we break the domain into subintervals whose boundaries are at the material interfaces, we could represent the solution exactly by piecewise linear polynomials.

Other problems where the presence of material boundaries would lead us to spectral element methods include electromagnetic wave propagation through different media, electromagnetic scattering off optically coated surfaces, or ultrasound detection of tumors where waves must propagate through muscle, fat and bone.

- *Solution and efficiency driven considerations.* Finally, even if neither the geometry nor the solution dictate that we should use multiple domains, efficiency con-

siderations may instead. To find a technical discussion of the issues we suggest the book by Canuto et al. [8]. We will only give a rough idea here how work and accuracy are related.

The work required to compute spectral solutions depends strongly on the polynomial order. If fast transforms are not available, the cost to compute the derivatives increases as N^2. If we use an explicit time integration method, the time step is limited by the size of the eigenvalues of the problem. The fact that the eigenvalues grow as N^2 for first order and N^4 for second order derivatives means that the number of time steps, which is inversely proportional to Δt, grows at these rates. Overall, the work to compute a time dependent solution with explicit time integration goes as

$$W \sim \begin{cases} N^4 & \textit{Advection dominated}, \\ N^6 & \textit{Diffusion dominated}. \end{cases} \tag{8.2}$$

On the other hand, if we subdivide into K subdomains, with only N/K order polynomials in each, then the work grows more slowly as

$$W \sim \begin{cases} K(\frac{N}{K})^4 & \textit{Advection dominated}, \\ K(\frac{N}{K})^6 & \textit{Diffusion dominated}. \end{cases} \tag{8.3}$$

For instance for a diffusion dominated problem, splitting a domain into only two subdomains reduces the work by a significant factor of $2/2^6 = 1/32$.

We need to balance the savings in work with the fact that the error will likely grow if the total number of points is fixed and the number of subdomains is increased. Roughly speaking, the error will vary with N and K as

$$E \sim \frac{e^{-\alpha N}}{K^{N+1}}. \tag{8.4}$$

So for smooth solutions where the error decays exponentially with polynomial order, the convergence rate is faster if we increase the polynomial order instead of the number of subdomains. We therefore need to balance accuracy and work to decide on the optimal strategy.

The solution itself may also direct us to use multidomain methods. When the solutions have sharp fonts, boundary layers, or other regions where local refinement is needed, along with other regions where the solution has little variation, we can localize the effort needed to compute the solution. We can use large regions with low order polynomials where the solution varies little, and place small subdomains with high order where the solution varies significantly.

8.1 Spectral Element Methods in One Space Dimension

To introduce the additional complexity of a multidomain method, we begin with a presentation of the spectral element method in one space dimension. In one dimension, we subdivide the interval into multiple non-overlapping subintervals (Fig. 8.6).

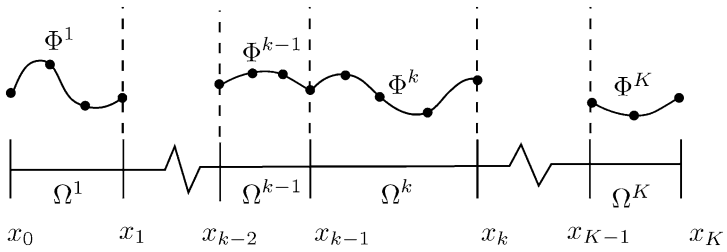

Fig. 8.6 Subdivision of the interval $[0, L] = [x_0, x_K]$ into K elements. The nodal Galerkin approximation is continuous and solutions are defined on Gauss-Lobatto points

Each interval will become an element. We couple an element to its neighbors at a single point, which we will call the element interface, or end point, in one space dimension. We will derive and implement two spectral element approximations. The first uses the continuous Galerkin approximation. The second uses the discontinuous version.

8.1.1 The Continuous Galerkin Spectral Element Method

As our first illustration of a spectral element method, we re-visit the approximation of the diffusion problem

$$\begin{aligned} \varphi_t &= \varphi_{xx} + f(x), \quad 0 < x < L, \\ \varphi(0,t) &= \varphi(L,t) = 0, \quad t > 0, \\ \varphi(x,0) &= \varphi_0(x), \quad 0 \le x \le L, \end{aligned} \tag{8.5}$$

that we solved with a single domain method on $[-1, 1]$ in Sect. 4.6.

As we did with the single domain approximation, we derive the continuous Galerkin spectral element method from a weak form of the equation. To get the weak form, we multiply the equation by a sufficiently smooth test function, ϕ, that satisfies the boundary conditions $\phi(0,t) = \phi(L,t) = 0$, is continuous, and has square integrable first and second derivatives. We then integrate over $[0, L]$ to get

$$\int_0^L \varphi_t \phi dx = \int_0^L \varphi_{xx} \phi dx + \int_0^L f \phi dx. \tag{8.6}$$

When we integrate the first integral on the right of (8.6) by parts and apply the boundary conditions,

$$\int_0^L \varphi_t \phi dx = -\int_0^L \varphi_x \phi_x dx + \int_0^L f \phi dx. \tag{8.7}$$

At this point, we would approximate the solution by a polynomial and replace the integrals by quadrature to derive a single domain spectral method, as we did in Sect. 4.6.

To derive the spectral element approximation, we subdivide the interval $[0, L]$ into K subintervals $\Omega^k = [x_{k-1}, x_k]$ (Fig. 8.6). These subintervals will be our spectral elements, e^k. We do not need to place any restrictions on the relative sizes of the elements.

Once we have subdivided the interval, we can break the integrals into sums of integrals over the elements

$$\sum_{k=1}^{K} \int_{x_{k-1}}^{x_k} \varphi_t \phi dx = -\sum_{k=1}^{K} \int_{x_{k-1}}^{x_k} \varphi_x \phi_x dx + \sum_{k=1}^{K} \int_{x_{k-1}}^{x_k} f \phi dx, \tag{8.8}$$

so that on each element the solution satisfies

$$\int_{x_{k-1}}^{x_k} \varphi_t \phi dx = -\int_{x_{k-1}}^{x_k} \varphi_x \phi_x dx + \int_{x_{k-1}}^{x_k} f \phi dx. \tag{8.9}$$

To use a Legendre polynomial approximation on each element, we map each element onto the reference interval $\xi \in [-1, 1]$, which will serve as our reference element, by the affine mapping

$$x = X^k(\xi) = x_{k-1} + \frac{\xi + 1}{2} \Delta x_k, \tag{8.10}$$

where $\Delta x_k = x_k - x_{k-1}$. Then

$$dx = \frac{\Delta x_k}{2} d\xi, \qquad \frac{\partial}{\partial x} = \frac{2}{\Delta x_k} \frac{\partial}{\partial \xi} \tag{8.11}$$

so that on the reference element the weak form of the PDE becomes

$$\frac{\Delta x_k}{2} \int_{-1}^{1} \varphi_t \phi d\xi = -\frac{2}{\Delta x_k} \int_{-1}^{1} \varphi_\xi \phi_\xi d\xi + \frac{\Delta x_k}{2} \int_{-1}^{1} f \phi d\xi. \tag{8.12}$$

To derive the continuous Galerkin spectral element method we approximate the solution by a piecewise continuous polynomial (Fig. 8.6). On each element, e^k, we write the polynomial in Lagrange form with nodes at the Legendre Gauss-Lobatto points,

$$\Phi^k(\xi, t) = \sum_{j=0}^{N} \Phi_j^k(t) \ell_j(\xi). \tag{8.13}$$

To keep the notation as simple as possible, we will keep the polynomial order N to be the same across all elements, but this is not necessary in practice. With N constant, the Lagrange interpolating polynomials are the same in each element. For

the approximation to be continuous and the first derivatives to be square integrable, we require that

$$\Phi_0^k = \Phi_N^{k-1}. \tag{8.14}$$

Similarly, we require that

$$\phi^k(\xi) = \sum_{j=0}^{N} \phi_j^k \ell_j(\xi), \qquad \phi_0^k = \phi_N^{k-1}. \tag{8.15}$$

Other than the continuity constraint, the values of ϕ_j^k are arbitrary since the weak form holds for any sufficiently smooth function ϕ.

We now replace each integral in (8.12) by Legendre Gauss-Lobatto quadrature. On the left,

$$\frac{\Delta x_k}{2} \int_{-1}^{1} \varphi_l \phi d\xi \approx \frac{\Delta x_k}{2} \sum_{j=0}^{N} w_j \dot{\Phi}_j^k \phi_j^k. \tag{8.16}$$

Since we are assuming that the same polynomial order is used in each element, the quadrature weights do not change with k. We approximate the last integral on the right of (8.12) in a similar manner. Finally, we follow the derivation of (4.117) to write the diffusion term in the compact form

$$\frac{2}{\Delta x_k} \int_{-1}^{1} \varphi_\xi \phi_x d\xi \approx \frac{2}{\Delta x_k} \sum_{j=0}^{N} \left\{ \sum_{m=0}^{N} \Phi_m^k G_{jm} \right\} \phi_j^k, \tag{8.17}$$

where

$$G_{jm} = G_{mj} = \sum_{l=0}^{N} w_l \ell_m'(\xi_l) \ell_j'(\xi_l) \tag{8.18}$$

is a symmetric matrix. When we gather the terms, we see that the solution satisfies the single domain approximation (4.118) element by element

$$\sum_{j=0}^{N} \left\{ w_j \frac{\Delta x_k}{2} \dot{\Phi}_j^k + \frac{2}{\Delta x_k} \sum_{m=0}^{N} \Phi_m^k G_{jm} - w_j \frac{\Delta x_k}{2} F_j^k \right\} \phi_j^k = 0. \tag{8.19}$$

By (8.8), we get the complete spectral element approximation when we sum over all of the elements

$$\sum_{k=1}^{K} \sum_{j=0}^{N} \left\{ w_j \frac{\Delta x_k}{2} \dot{\Phi}_j^k + \frac{2}{\Delta x_k} \sum_{m=0}^{N} \Phi_m^k G_{jm} - w_j \frac{\Delta x_k}{2} F_j^k \right\} \phi_j^k = 0. \tag{8.20}$$

We use (8.20) to derive the equations that the approximate values of the solution satisfy. Since the values of ϕ_j^k are independent except at the element boundaries, the

equations for interior node solution values depend only on others within their own element. If we take the test function ϕ_j^k to equal one at the single interior node ξ_j in element k and zero at all the others, we find that the interior node solutions satisfy

$$w_j \frac{\Delta x_k}{2} \dot{\Phi}_j^k = -\frac{2}{\Delta x_k} \sum_{m=0}^{N} \Phi_m^k G_{jm} + w_j \frac{\Delta x_k}{2} F_j^k,$$
$$j = 1, 2, \ldots, N-1; \; k = 1, 2, \ldots, K. \tag{8.21}$$

Elements couple because the solution and the test functions are continuous at the points shared by two elements. When we choose ϕ_j^k to equal one at the point shared by the kth and $k-1$st elements and zero elsewhere, then

$$w_N \frac{\Delta x_{k-1}}{2} \dot{\Phi}_N^{k-1} + w_0 \frac{\Delta x_k}{2} \dot{\Phi}_0^k = -\frac{2}{\Delta x_{k-1}} \sum_{m=0}^{N} \Phi_m^{k-1} G_{Nm} + w_N \frac{\Delta x_{k-1}}{2} F_N^{k-1}$$
$$- \frac{2}{\Delta x_k} \sum_{m=0}^{N} \Phi_m^k G_{0m} + w_0 \frac{\Delta x_k}{2} F_0^k. \tag{8.22}$$

When we impose the continuity constraint (8.14), $\Phi_0^k = \Phi_N^{k-1} \equiv \Phi_*^k$, we get the element boundary equations

$$\left[w_N \frac{\Delta x_{k-1}}{2} + w_0 \frac{\Delta x_k}{2} \right] \dot{\Phi}_*^k = -\frac{2}{\Delta x_{k-1}} \sum_{m=0}^{N} \Phi_m^{k-1} G_{Nm} + w_N \frac{\Delta x_{k-1}}{2} F_N^{k-1}$$
$$- \frac{2}{\Delta x_k} \sum_{m=0}^{N} \Phi_m^k G_{0m} + w_0 \frac{\Delta x_k}{2} F_0^k \tag{8.23}$$

for $k = 2, 3, \ldots, K-1$. Finally, we add the two boundary conditions $\Phi_0^1 = \Phi_N^K = 0$ to complete the system of equations that we integrate in time.

When we compare (8.21) to the single domain approximation (4.122), we see that we have the same equations to compute the interior and boundary points, except that now we have not yet divided by the quadrature weights and the element length can vary. The spectral element approximation differs only because we add the interior element boundary equations (8.23) that couple the elements. Notice that (8.22) is simply the sum of two equations (8.21), one from each element that shares the point. We will use this observation when we implement the approximations.

To integrate (8.21) and (8.23) in time, let us now use the second order implicit trapezoidal rule. Let's define

$$\left(G\Phi^k\right)_j \equiv \sum_{m=0}^{N} \Phi_m^k G_{jm}, \tag{8.24}$$

$$D_j^k \equiv -\frac{2}{\Delta x_k} \left(G\Phi^k\right)_j, \tag{8.25}$$

and let $\Phi_j^{k,n}$ denote the solution at point j in element k at time t_n. Then at interior points the solutions satisfy

$$w_j \frac{\Delta x_k}{2} \Phi_j^{k,n+1} - \frac{\Delta t}{2} D_j^{k,n+1} = w_j \frac{\Delta x_k}{2} \Phi_j^{k,n} + \frac{\Delta t}{2} D_j^{k,n} + \Delta t w_j \frac{\Delta x_k}{2} F_j^k \equiv RHS_j^k. \tag{8.26}$$

To get the equations for the element interface points, we sum the contributions from neighboring elements, as we are instructed to do by (8.22) and (8.23).

We need to compute the residual to use Algorithm 80 (PreconditionedConjugateGradientSolve) to solve the system of equations at each time step. Let us define the local element residual at each node, including the endpoints, by

$$\tilde{r}_j^k = RHS_j^k - w_j \frac{\Delta x_k}{2} \Phi_j^k + \frac{\Delta t}{2} D_j^k, \quad j = 0, 1, \ldots, N;\ k = 1, 2, \ldots, K. \tag{8.27}$$

Since the global residuals at the interior element boundaries are just the sum of the local residuals at those points, we can compute the global residuals after we compute all the local residuals by

$$r_j^k = \begin{cases} \tilde{r}_j^k, & j = 1, 2, \ldots, N;\ k = 1, 2, \ldots, K, \\ \tilde{r}_N^k + \tilde{r}_0^{k+1}, & j = 0, N;\ k = 1, 2, \ldots, K, \\ 0, & k = 0,\ j = 0;\ k = K,\ j = N. \end{cases} \tag{8.28}$$

The boundary values on the last line are, of course, for Dirichlet boundary conditions. Algorithm 80 needs to be modified slightly because the limits on k start from one rather than zero, however we could accommodate this by a change of index in the data.

8.1.2 How to Implement the Continuous Galerkin Spectral Element Method

The first decision that we must make to evolve the nodal Galerkin approximation from a single domain to the spectral element approximation is to choose the data structures. We can choose between two extremes. One is to store all data locally. In this scheme, the solution, element size, Δx, etc. are stored by element within a structure or class. The other structure is "flat", where data is stored globally. The advantage of the local scheme is that the operations on the data look like the single domain approximations that we have developed in earlier chapters. The global scheme is useful if we want to use BLAS routines effectively. With the global implementation, we can use the Conjugate Gradient solver, Algorithm 80 (PreconditionedConjugateGradientSolve), at each time step to solve the symmetric system of equations that (8.26) and the boundary equations generate.

8.1.2.1 The Spectral Element Class

We will take a global view of the data with some organization to simplify local computations. In Sect. 8.1.4, where we will use explicit integration in time, we will show how to use local data structures. Much of the effort here about organization of the data in one space dimension is overkill, but it will prepare us for multidimensional problems where the effort is justified.

To use the iterative solver efficiently, we will store the solution of the diffusion problem in an array. Since we have assumed the same polynomial degree in each element and since Algorithm 80 has already been designed for two dimensional arrays, we will store the solution for each point on each element in a two dimensional array, $\{\Phi_{j,k}\}_{j=0;k=1}^{N;K}$. This array will be a member of a spectral element class, along with the data that defines the mesh, as we show in Algorithm 116 (SEM1DClass).

The array storage duplicates the solution at the interior element boundaries. It does, however, enable us to perform operations such as (8.24) locally on an element

Algorithm 116: *SEM1DClass:* Data Storage for the One-Dimensional Spectral Element Method

Class SEM1D
Data:
N, K
$\{\xi_j\}_{j=0}^{N}$; // Node locations
$\{w_j\}_{j=0}^{N}$; // Legendre Gauss-Lobatto quadrature weights
$\{x_k\}_{k=1}^{K}$; // Element boundary locations
$\{\Delta x_k\}_{k=1}^{K}$; // Element sizes
$\{G_{ij}\}_{i,j=0}^{N}$; // Derivative Matrix
$\{p_k\}_{k=1}^{K}$; // Shared node pointers
$\{\Phi_{j,k}\}_{j=0;k=1}^{N;K}$; // Solution
$\{RHS_{j,k}\}_{j=0;k=1}^{N;K}$; // For implicit integration
Procedures:
Construct$(N, K, \{x_k\}_{k=1}^{K})$
Mask$(\{a_{j,k}\}_{j=0;k=1}^{N,K})$; // Algorithm 117
UnMask$(\{a_{j,k}\}_{j=0;k=1}^{N,K})$; // Algorithm 117
GlobalSum$(\{a_{j,k}\}_{j=0;k=1}^{N,K})$; // Algorithm 117
LaplaceOperator; // Algorithm 118
MatrixAction$(s, \Delta t)$; // Algorithm 118
End Class SEM1D

Structure sharedNodePtrs
eLeft ; // index or pointer of element to the left
eRight ; // index or pointer of element to the right
nodeLeft ; // index of node to the left
nodeRight ; // index of node to the right
End Structure sharedNodePtrs

as if we were still doing a single domain computation. The fact that the element boundary equations are sums of the local element contributions from each side allows us to go back and "clean up" globally afterwards. We will store RHS_j^k and the residual r_j^k in the same kind of array as the solution.

We need a data structure to connect the elements. We assumed the simplest relationship between elements in Fig. 8.6, where elements are ordered from left to right. With that ordering we could assume that the continuity of the shared point is given by (8.14). In two dimensional problems we cannot assume such a simple ordering of elements. (See Fig. E.2 in Appendix E for a look ahead.) To allow for more general relationships, we create a structure that stores the values of k and j for the elements on the left and on the right that contribute to the solution at an interface (Algorithm 116, sharedNodePtrs).

We will store the sharedNodePtrs structures in an array, $\{p^k\}_{k=1}^{K-1}$, to mark the element interface points. When we need to do operations at interface points, we can access these points indirectly through the elements of this array. For the mesh that we show in Fig. 8.6, where elements are ordered left to right,

$$\begin{aligned} p^k.eleft &= k, \\ p^k.eRight &= k+1, \\ p^k.nodeLeft &= N, \\ p^k.nodeRight &= 0. \end{aligned} \tag{8.29}$$

With flat storage, we store all the data for the spectral element approximation, including the solutions, element relationships, element sizes, etc., in a single class or structure as we show in Algorithm 116 (SEM1DClass). We will not present a detailed algorithm for the constructor of this class since it looks much like constructors that we have presented earlier.

8.1.2.2 Global Operations

Before we can solve the system with the Conjugate Gradient solver, we must account for the fact that we have duplicated the solution and residuals at the element boundaries to perform the local computations as if on a single domain. One way to delete the duplicates is to copy the arrays to a new array that does not include them. We will use a mask instead. After we compute the residuals, we will simply set one of the duplicate solutions and the corresponding residuals to zero so that they have no contribution to the inner products or direction vectors in the Conjugate Gradient solver. Then to perform the local computations, we remove the mask by setting the solution value that has been zeroed to its duplicate. The number of masked points, $K-1$, is very small compared to the total number of degrees of freedom, $K \times N$, so the extra work to mask, unmask and evaluate masked quantities in the Conjugate Gradient algorithm is negligible. What we gain in return is an algorithm that is straightforward to implement.

We perform the global operations with the utility procedures *Mask*, *UnMask* and *GlobalSum*. It doesn't matter which duplicate we mask, so we will arbitrarily choose

Algorithm 117: *SEMGlobalProcedures1D:* Global Operations for the One-Dimensional Spectral Element Method

Uses Algorithms:
Algorithm 116 (SEM1DClass)
Procedure Mask
Input: $\left\{a_{j,k}\right\}_{j=0;k=1}^{N;K}$

for $k = 1$ **to** $K - 1$ **do**
  $jR \leftarrow this.p^k.nodeRight$; $eR \leftarrow this.p^k.eRight$
  $a_{jR,eR} \leftarrow 0$
end
return $\left\{a_{j,k}\right\}_{j=0;k=1}^{N;K}$
End Procedure Mask

Procedure UnMask
Input: $\left\{a_{j,k}\right\}_{j=0;k=1}^{N;K}$

for $k = 1$ **to** $K - 1$ **do**
  $jR \leftarrow this.p^k.nodeRight$; $eR \leftarrow this.p^k.eRight$
  $jL \leftarrow this.p^k.nodeLeft$; $eL \leftarrow this.p^k.eLeft$
  $a_{jR,eR} \leftarrow a_{jL,eL}$
end
return $\left\{a_{j,k}\right\}_{j=0;k=1}^{N;K}$
End Procedure UnMask

Procedure GlobalSum
Input: $\left\{a_{j,k}\right\}_{j=0;k=1}^{N;K}$

for $k = 1$ **to** $K - 1$ **do**
  $jR \leftarrow this.p^k.nodeRight$; $eR \leftarrow this.p^k.eRight$
  $jL \leftarrow this.p^k.nodeLeft$; $eL \leftarrow this.p^k.eLeft$
  $tmp \leftarrow a_{jR,eR} + a_{jL,eL}$
  $a_{jR,eR} \leftarrow tmp$
  $a_{jL,eL} \leftarrow tmp$
end
return $\left\{a_{j,k}\right\}_{j=0;k=1}^{N;K}$
End Procedure GlobalSum

the value on the right. We show the global operation procedures in Algorithm 117 (SEMGlobalProcedures1D). Each procedure uses the shared node array to perform its operation globally on an array of mesh values, which may be the solution or residual, for example.

8.1.2.3 The Diffusion Approximation

Both the right and left sides of the system (8.26) require us to compute a matrix-vector multiply to evaluate the contribution from the diffusion term of the equation.

We use Algorithm 57 (CGDerivativeMatrix) to compute the matrix G. We show the implementation of the diffusion term in the procedure *LaplaceOperator* in Algorithm 118 (SEM1DProcedures).

8.1.2.4 Side Operators and Residual Procedures

When we use the trapezoidal rule to integrate in time, the only difference between the matrices whose actions are given in (8.26) is the sign in front of the diffusion operator. For that reason, we will implement only one procedure for the matrix action in Algorithm 118 (SEM1DProcedures) and pass a sign variable with values ± 1 as we did with Fourier transforms. We set the residual to zero at the physical boundaries to implement Dirichlet conditions in the procedure *Residual* of Algorithm 118.

8.1.2.5 Iterative Solver

We need only make minor modifications to Algorithm 80 (PreconditionedConjugateGradientSolve). The first we have already mentioned, namely that the arrays now have different extents. The second is that the residual computation section will be replaced by the procedure *Residual* in Algorithm 118. We could still use a finite element preconditioner or not precondition at all.

8.1.2.6 The Time Integration Procedure

The trapezoidal rule integrator that we show in Algorithm 119 (TrapezoidalRuleIntegration) is straightforward to implement. At each time step we compute the *RHS* array and use Algorithm 80 to compute the new values. Since the solution values are no longer needed after *RHS* is evaluated, we need only one time level of storage for the solution, unlike if we were to use Algorithm 87 (MultistepIntegration). Note that as in Algorithm 87, we have hidden the boundary condition implementation in a procedure *SetBoundaryConditions* that needs to be provided. It will do nothing more than set the solution values at the boundaries.

8.1.3 Benchmark Solution: Cooling of a Temperature Spot

We present one benchmark example of the solution of (8.5) with the spectral element method. We have chosen the initial and boundary conditions so that the exact solution as a function of time describes a cooling Gaussian spot,

$$\varphi(x,t) = \frac{e^{-x^2/(4t+1)}}{\sqrt{4t+1}}. \tag{8.30}$$

Algorithm 118: *SEM1DProcedures:* Spatial Approximations for the One-Dimensional Spectral Element Method

Uses Algorithms:
Algorithm 116 (SEM1DClass)
Algorithm 19 (MxVDerivative)
Algorithm 117 (SEMGlobalProcedures1D)

Procedure LaplaceOperator
Input: $\{U_{j,k}\}_{j=0;k=1}^{N;K}$
$N \leftarrow this.N$; $K \leftarrow this.K$
for $k = 1$ **to** K **do**
  $\{D_j^k\}_{j=0}^{N} \leftarrow MxVDerivative(this.\{G_{i,j}\}_{i=0;j=0}^{N;N}, \{U_{j,k}\}_{j=0}^{N})$
  for $j = 0$ **to** N **do**
    $D_j^k \leftarrow -2 * D_j^k / \Delta x_k$
  end
end
return $\{D_j^k\}_{j=0;k=1}^{N;K}$
End Procedure LaplaceOperator

Procedure MatrixAction
Input: s, Δt, $\{U_{j,k}\}_{j=0;k=1}^{N;K}$
$N \leftarrow this.N$; $K \leftarrow this.K$
$\{U_{j,k}\}_{j=0;k=1}^{N;K} \leftarrow this.UnMask(\{U_{j,k}\}_{j=0;k=1}^{N;K})$
$\{AU_{j,k}\}_{j=0;k=1}^{N;K} \leftarrow this.LaplaceOperator(\{U_{j,k}\}_{j=0;k=1}^{N;K})$
for $k = 1$ **to** $K - 1$ **do**
  for $j = 0$ **to** N **do**
    $AU_{j,k} \leftarrow this.w_j/2 * this.\Delta x_k * U_{j,k} + s * \Delta t/2 * AU_{j,k}$
  end
end
$\{AU_{j,k}\}_{j=0;k=1}^{N;K} \leftarrow this.GlobalSum(\{AU_{j,k}\}_{j=0;k=1}^{N;K})$
$\{U_{j,k}\}_{j=0;k=1}^{N;K} \leftarrow this.Mask(\{U_{j,k}\}_{j=0;k=1}^{N;K})$
return $\{AU_{j,k}\}_{j=0;k=1}^{N;K}$
End Procedure MatrixAction

Procedure Residual
Input: *sem* ; `// Of type SEM1D`
$\{r_{j,k}\}_{j=0;k=1}^{N;K} \leftarrow this.MatrixAction(+1, \Delta t, sem.\{\Phi_{j,k}\}_{j=0;k=1}^{N;K})$
for $k = 1$ **to** K **do**
  for $j = 0$ **to** N **do**
    $r_{j,k} \leftarrow sem.RHS_{j,k} - r_{j,k}$
  end
end
$\{r_{j,k}\}_{j=0;k=1}^{N;K} \leftarrow this.Mask(\{r_{j,k}\}_{j=0;k=1}^{N;K})$
$r_{0,1} \leftarrow 0$
$r_{N,K} \leftarrow 0$
return $\{r_{j,k}\}_{j=0;k=1}^{N;K}$
End Procedure Residual

Algorithm 119: *TrapezoidalRuleIntegration:* Integration of the One-Dimensional Spectral Element Method in Time

Procedure TrapezoidalRuleIntegration
Input: N_T, N_{it}, TOL
Input: *sem* ; `// Of type SEM1D class`
Uses Algorithms:
Algorithm 116 (SEM1DClass)
Algorithm 117 (SEMGlobalProcedures1D)
Algorithm 118 (SEM1DProcedures)
Algorithm 80 (PreconditionedConjugateGradientSolve) //Modified

for $n = 0$ **to** $N_T - 1$ **do**
  $t \leftarrow n * \Delta t$
  $sem.\{RHS_{j,k}\}_{j=0;k=1}^{N;K} \leftarrow sem.MatrixAction(-1, \Delta t, sem.\{\Phi_{j,k}\}_{j=0;k=1}^{N;K})$
  $sem.\{\Phi_{j,k}\}_{j=0;k=1}^{N;K} \leftarrow SetBoundaryConditions(sem, t + \Delta t)$
  $sem \leftarrow PreconditionedConjugateGradientSolve(N_{it}, TOL, sem)$
end
$sem.\{\Phi_{j,k}\}_{j=0;k=1}^{N;K} \leftarrow sem.UnMask(sem.\{\Phi_{j,k}\}_{j=0;k=1}^{N;K})$
$t \leftarrow t + \Delta t$
return *sem*
End Procedure TrapezoidalRuleIntegration

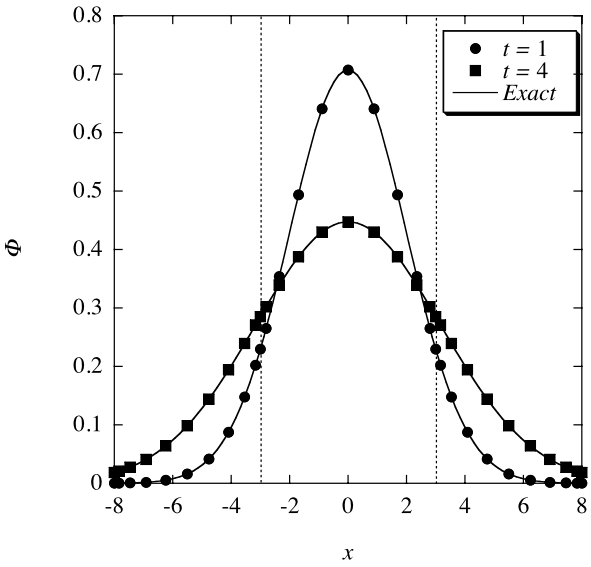

Fig. 8.7 Three element spectral element solution of the diffusion equation at two times. Element boundaries are marked with vertical dotted lines

We solve the equation on the interval $[-8, 8]$ with three elements $\Omega^1 = [-8, -3]$, $\Omega^2 = [-3, 3]$, and $\Omega^3 = [3, -8]$, $N = 10$, and $\Delta t = 0.05$. Figure 8.7 compares the computed and exact solutions at times $t = 1$ and $t = 4$. We could also make comparisons of multiple element approximations to the results of Sect. 4.6.

8.1.4 The Discontinuous Galerkin Spectral Element Method

We will motivate the development and implementation of the discontinuous Galerkin spectral element method with the approximation of the one-dimensional conservation law

$$\mathbf{q}_t + \mathbf{f}_x = 0, \quad x \in (0, L)\,. \tag{8.31}$$

As we discussed in Sect. 5.4, we can write the wave equation and others in conservation form.

Like the continuous spectral element approximation, we get the discontinuous approximation by dividing the interval into segments or elements (Fig. 8.8). Now, however, we will choose the nodes to be the Legendre Gauss points inside each element. More importantly, we will not assume that the solution is continuous at the element boundaries.

The spectral element approximation starts from the weak form of (8.31). After we multiply by a suitable test function, integrate, and subdivide into elements as we did in the previous section, the weak form is

$$\sum_{k=1}^{K} \left[\int_{x_{k-1}}^{x_k} (\mathbf{q}_t + \mathbf{f}_x)\, \phi dx \right] = 0. \tag{8.32}$$

On each element then,

$$\int_{x_{k-1}}^{x_k} (\mathbf{q}_t + \mathbf{f}_x)\, \phi dx = 0. \tag{8.33}$$

When we map (8.33) onto the reference element by the affine map (8.10),

$$\frac{\Delta x_k}{2} \int_{-1}^{1} \mathbf{q}_t \phi d\xi + \int_{-1}^{1} \mathbf{f}_\xi \phi d\xi = 0. \tag{8.34}$$

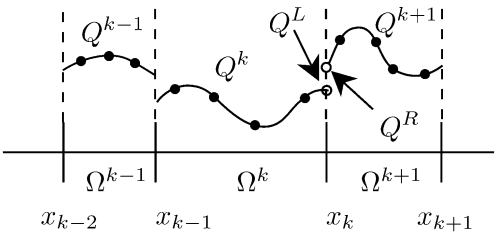

Fig. 8.8 Subdivision of the interval $[0, L] = [x_0, x_K]$ into K elements. The nodal discontinuous Galerkin approximation is not continuous at element boundaries and is defined on Gauss points

As usual for a nodal method, we approximate the solution and fluxes by polynomials in Lagrange form

$$\begin{aligned}\mathbf{q}\big(X^k(\xi)\big) &\approx \mathbf{Q}^k(\xi) = \sum_{j=0}^{N} \mathbf{Q}_j^k \ell_j(\xi),\\ \mathbf{f}\big(X^k(\xi)\big) &\approx \mathbf{F}^k(\xi) = \sum_{j=0}^{N} \mathbf{f}\big(\mathbf{Q}_j^k\big) \ell_j(\xi).\end{aligned} \tag{8.35}$$

Since the discontinuous Galerkin approximation does not enforce continuity at the element boundaries, we can take advantage of the higher precision of the Gauss rules and place the nodes at the Legendre Gauss points on the reference element. When we substitute the polynomial approximations into (8.12), the approximate solution satisfies

$$\frac{\Delta x_k}{2} \int_{-1}^{1} \mathbf{Q}_t \phi d\xi + \int_{-1}^{1} \mathbf{F}_\xi \phi d\xi = 0. \tag{8.36}$$

We write the test function ϕ also as a polynomial,

$$\phi = \sum_{j=0}^{N} \phi_j \ell_j(\xi). \tag{8.37}$$

The discontinuous Galerkin approximation does not require the test functions to be continuous at element boundaries either. Therefore to say that (8.33) holds for any piecewise continuous function ϕ is equivalent to saying that the ϕ_j's are independent of each other, without any constraints.

The approximation and test functions are continuous within any element. Therefore, we can integrate the second integral in (8.36) by parts to get

$$\frac{\Delta x_k}{2} \int_{-1}^{1} \mathbf{Q}_t \phi d\xi + \mathbf{F}\phi|_{-1}^{1} - \int_{-1}^{1} \mathbf{F}\phi_\xi d\xi = 0. \tag{8.38}$$

We've done the next two steps in the derivation of a nodal spectral approximation several times before. We replace the integrals by Gauss quadrature and use the fact that the ϕ_j's are independent. These steps lead us to

$$\frac{\Delta x_k}{2} \dot{\mathbf{Q}}_j + \mathbf{F} \frac{\ell_j}{w_j}\bigg|_{-1}^{1} - \sum_{n=0}^{N} \mathbf{F}_n \frac{w_n \ell_j'(\xi_n)}{w_j} = 0 \tag{8.39}$$

or, using the definition (4.139),

$$\frac{\Delta x_k}{2} \dot{\mathbf{Q}}_j + \left\{ \mathbf{F} \frac{\ell_j}{w_j}\bigg|_{-1}^{1} + \sum_{n=0}^{N} \mathbf{F}_n \hat{D}_{jn} \right\} = 0. \tag{8.40}$$

In the final step, we couple the elements. This step is peculiar to the discontinuous Galerkin approximation. We replace the fluxes at the element boundaries by the numerical flux $\mathbf{F}^* \cdot \hat{n}$ as we did in Sect. 5.4.1.1 at physical boundaries. Recall that the numerical flux is a function of two states, one to the left and one to the right, where we define the direction according to the normal at the boundary. At element boundaries, the left and right states are what we get when we evaluate the solution polynomial from the neighboring elements, as we show in Fig. 8.8. For the element ordering in Fig. 8.8, the numerical fluxes at the two element boundaries are

$$\begin{aligned} \mathbf{F}(-1) &\leftarrow \mathbf{F}^* \left(Q^{k-1}(1), Q^k(-1), -\hat{x} \right), \\ \mathbf{F}(+1) &\leftarrow \mathbf{F}^* \left(Q^k(1), Q^{k+1}(-1), +\hat{x} \right). \end{aligned} \tag{8.41}$$

At a physical boundary, we use the external state, just as in a single domain approximation.

When we replace the fluxes at the element boundaries by the numerical fluxes, we get the final version of the discontinuous Galerkin spectral element approximation for the kth element

$$\begin{aligned} \dot{\mathbf{Q}}_j^k + \frac{2}{\Delta x_k} \Bigg\{ & \mathbf{F}^* \left(Q^k(1), Q^{k+1}(-1), +\hat{x} \right) \frac{\ell_j(1)}{w_j} \\ & + \mathbf{F}^* \left(Q^{k-1}(1), Q^k(-1), -\hat{x} \right) \frac{\ell_j(-1)}{w_j} + \sum_{n=0}^{N} \mathbf{F}_n \hat{D}_{jn} \Bigg\} = 0. \end{aligned} \tag{8.42}$$

We make two observations about (8.42). First, except for the element size factor, $2/\Delta x_k$, the approximation is identical to the single domain approximation. Therefore we still compute the derivative approximation with Algorithm 92 (SystemDGDerivative). Second, the only coupling between elements occurs when we compute the boundary flux values. Therefore, we need only to supply the boundary values from the immediate neighbor to compute the time derivative of the solution.

8.1.5 How to Implement the Discontinuous Galerkin Spectral Element Method

The fact that the discontinuous Galerkin spectral element approximation (8.42) is virtually the same as the single domain approximation except for information to be set from the nearest neighbor elements suggests that we use a two level view for the implementation. The lowest level view will be that of an element on which the local operations such as the spatial derivative approximation are performed. The higher level view will be that of the mesh, which will keep track of the elements and perform global operations such as the computation of the numerical fluxes. To implement the two level view, we create the two classes *Element* and *Mesh* shown in

Algorithm 120: *DGSEM1DClasses:* Element and Mesh Definitions for the One-Dimensional Discontinuous Galerkin Spectral Element Method

Class Element
Uses Algorithms:
Algorithm 58 (NodalDiscontinuousGalerkin);

Data:
$\Delta x, x_L, x_R$; // Size and left and right boundaries of this element
dG ; // Of type NodalDiscontinuousGalerkin
$nEqn$; // # of equations in the physical system
$\{Q_{j,n}\}_{j=0;n=1}^{N;nEqn}$ $\{\dot{Q}_{j,n}\}_{j=0;n=1}^{N;nEqn}$; // Solution and time derivative
$\{G_{j,n}\}_{j=0;n=1}^{N;nEqn}$; // For low storage Runge-Kutta
$\left\{Q_n^L\right\}_{n=1}^{nEqn}, \left\{Q_n^R\right\}_{n=1}^{nEqn}$; // Solution on left/right element boundary
$\left\{F_n^{*,L}\right\}_{n=1}^{nEqn}, \left\{F_n^{*,R}\right\}_{n=1}^{nEqn}$; // Flux on left/right element boundary
Procedures:
Construct$(dG, nEqn, x_L, x_R)$; // See text.
InterpolateToBoundaries(); // Algorithm 121
LocalTimeDerivative(); // Algorithm 121
AffineMap(ξ); // Equation (8.10)
End Class Element

Class Mesh1D
Data:
K ; // # Elements
$\{e_k\}_{k=1}^{K}$; // Elements
$\{p^k\}_{k=0}^{K}$; // sharedNodePointers from Algorithm 116
Procedures:
Construct$(K, N, \{x_k\}_{n=0}^{K})$; // Algorithm 122
GlobalTimeDerivative(); // Algorithm 122
End Class Mesh1D

Algorithm 120 (DGSEM1DClasses). Again, the organization that we present here is overkill for one-dimensional problems, but it is helpful to see it on a simpler problem before moving to multidimensional problems.

8.1.5.1 The Elements

The element class (Algorithm 120) stores the element's geometry data and solution. The geometry data for the one dimensional elements are the left and right boundary locations and the length. To compute the spatial derivative in (8.42), the element needs access to the data stored in the *NodalDiscontinuousGalerkin* class (Algorithm 58). For simplicity, we show the Element class storing an instance of that class, though to do so clearly wastes storage if N is the same in each element. Even if N varies, there would likely only be a few unique values. In a large implementation, we would store the *NodalDiscontinuousGalerkin* instances in a collection (e.g.

a linked list discussed in Appendix E.1) and have the element only store a pointer to a member of that collection.

Other data includes the numerical fluxes, $F^{*,L/R}$, on the left and the right of the element and the interpolated values of the solution, $Q^{L/R}$, from which to compute the numerical flux. In practice, once we compute the numerical fluxes, we no longer need the interpolated values of the solution. The boundary solutions and fluxes could use the same storage. For clarity, however, we will store them separately. We also have the element store the number of equations rather than write a separate "physics" class. Finally, we store the time derivative with each element. If we were to modify the time integrator appropriately, we could reuse the storage of only one array.

The basic methods that the element needs to implement are also shown in Algorithm 120 (SEM1DClasses). We won't show the constructor explicitly since it only needs to set the local data. The interpolation and time derivative methods are implemented in Algorithm 121 (LocalDSEMProcedures). We have seen the use of the interpolation routines before in Algorithms 61 (InterpolateToBoundary) and 114

Algorithm 121: *LocalDSEMProcedures:* Local Procedures for the Discontinuous Galerkin Spectral Element Method

Procedure InterpolateToBoundaries
Uses Algorithms:
Algorithm 61 (InterpolateToBoundary);

$N \leftarrow this.dG.N$; $nEqn \leftarrow this.nEqn$
for $n = 1$ **to** $nEqn$ **do**
  $this.Q_n^L \leftarrow InterpolateToBoundary(this.\{Q_{j,n}\}_{j=0}^N, this.dG.\{\ell_j(-1)\}_{j=0}^N)$
  $this.Q_n^R \leftarrow InterpolateToBoundary(this.\{Q_{j,n}\}_{j=0}^N, this.dG.\{\ell_j(+1)\}_{j=0}^N)$
end
End Procedure InterpolateToBoundaries

Procedure LocalTimeDerivative
Uses Algorithms:
Algorithm 92 (SystemDGDerivative);

$N \leftarrow this.N$; $nEqn \leftarrow this.nEqn$
for $j = 0$ **to** N **do**
  $\{F_{j,n}\}_{n=1}^{nEqn} \leftarrow Flux(this.\{Q_{j,n}\}_{n=1}^{nEqn})$
end
$\left\{F'_{j,n}\right\}_{j=0;n=1}^{N;nEqn} \leftarrow$
$SystemDGDerivative\big(this.\left\{F_n^{*,L}\right\}_{n=1}^{nEqn}, this.\left\{F_n^{*,R}\right\}_{n=1}^{nEqn}, \{F_{j,n}\}_{j=0;n=1}^{N;nEqn}, this.dG.\{D_{i,j}\}_{i,j=0}^N,$
$this.dG.\{\ell_i(-1)\}_{i=0}^N, this.dG.\{\ell_i(1)\}_{i=0}^N, this.dG.\{w_i\}_{i=0}^N\big)$
for $j = 0$ **to** N **do**
  for $n = 1$ **to** $nEqn$ **do**
    $this.\dot{Q}_{j,n} \leftarrow -2F'_{j,n}/this.\Delta x$
  end
end
End Procedure LocalTimeDerivative

(MappedDGSystemTimeDerivative). The local time derivative procedure is of the form we have already seen in those two algorithms. We do not show an implementation of the *Flux* function, but it is similar to what we used in Algorithm 94 (WaveEquationFluxes). Finally, the *AffineMap* procedure merely implements (8.10) so we don't provide an implementation for it, either.

8.1.5.2 The Mesh

We manage global data at the mesh level. The mesh, also described in Algorithm 120 (DGSEM1DClasses), stores the number of elements, the elements themselves, and the connections between the elements. As in Sect. 8.1.1, the *sharedNodePointers* store pointers to the elements on the left and the right of an interface. To simplify the presentation, we will assume here that $x_R > x_L$ so that the Q^L and Q^R arrays are on the left and right of the elements. That way we do not have to store information in the *sharedNodePointers* to distinguish between which corresponds to the left and which to the right. When we go to two dimensional problems later, we will have to be more general.

We use the constructor for the mesh class to create the elements and connections. The constructor will take the number of elements and the location of the element boundaries as input. It constructs an instance of the *NodalDiscontinuousGalerkin* class and uses that and the element boundary information to construct the elements. The element connections are constructed next. Since there is essentially no difference between a physical boundary and an element boundary, the limits on the p^k array include the endpoints. At the physical boundaries we set the neighboring element to a defined constant *NONE*. Later, we can test for being on a boundary by checking to see if one of the elements equals *NONE*.

The global time derivative procedure in Algorithm 122 (GlobalMeshProcedures) performs four basic operations. First, it interpolates the solutions to the boundaries on each of the elements. It then computes the physical boundary values by way of a procedure *ExternalState* whose implementation is problem dependent. We pass a parameter with defined values of *LEFT* or *RIGHT* to the procedure so that different boundary conditions can be applied at the left and right boundaries. We also pass the boundary value of the solution to allow reflection boundary conditions like (5.168) to be implemented. Then the numerical fluxes are computed for each element boundary point and sent to the appropriate element. Again, we have assumed that the elements are laid out left to right. Finally, each element computes its local time derivative values.

8.1.5.3 Time Integration

We can easily modify the third order explicit Runge-Kutta algorithm Algorithm 62 (DGStepByRK3) to accommodate the spectral element approximation. We first change the inputs to the procedure to be an instance of the *Mesh* class, the time

Algorithm 122: *GlobalMeshProcedures:* Mesh Global Procedures for the Discontinuous Galerkin Spectral Element Approximation

Procedure Construct
Input: $K, N, \{x_k\}_{n=0}^{K}$
Uses Algorithms:
Algorithm 120 (SEM1DClasses)
Algorithm 58 (NodalDiscontinuousGalerkin)

$this.K \leftarrow K$
$dG.Construct(N)$; `// Of type NodalDiscontinuousGalerkin`
for $k = 1$ **to** $this.K$ **do**
  $this.e^k.Construct(dG, nEqn, x_{k-1}, x_k)$
end
for $k = 1$ **to** $K - 1$ **do**
  $this.p^k.eLeft \leftarrow k$
  $this.p^k.eRight \leftarrow k + 1$
end
$this.p^0.eLeft \leftarrow NONE$
$this.p^0.eRight \leftarrow 1$
$this.p^K.eLeft \leftarrow K$
$this.p^K.eRight \leftarrow NONE$
End Procedure Construct

Procedure GlobalTimeDerivative
Input: t
Uses Algorithms:
Algorithm 88 (RiemannSolver)

for $k = 1$ **to** $this.K$ **do**
  $this.e^k.InterpolateToBoundaries()$
end
$k \leftarrow this.p^0.eRight$
$\{Q_n^{ext,L}\}_{n=1}^{nEqn} \leftarrow ExternalState(this.e^k.\{Q_n^L\}_{n=1}^{nEqn}, this.e^k.AffineMap(-1), LEFT)$
$k \leftarrow this.p^K.eLeft$
$\{Q_n^{ext,R}\}_{n=1}^{nEqn} \leftarrow ExternalState(this.e^k.\{Q_n^R\}_{n=1}^{nEqn}, this.e^k.AffineMap(+1), RIGHT)$
for $k = 0$ **to** $this.K$ **do**
  $idL \leftarrow this.p^k.eLeft$; $idR \leftarrow this.p^k.eRight$
  if $idL = NONE$ **then**
    $this.e^{idR}.\{F_n^L\}_{n=1}^{nEqn} \leftarrow RiemannSolver(\{Q_n^{ext,L}\}_{n=1}^{nEqn}, this.e^{idR}.\{Q_n^L\}_{n=1}^{nEqn}, -1)$
  else if $idR = NONE$ **then**
    $this.e^{idL}.\{F_n^R\}_{n=1}^{nEqn} \leftarrow RiemannSolver(this.e^{idL}.\{Q_n^R\}_{n=1}^{nEqn}, \{Q_n^{ext,R}\}_{n=1}^{nEqn}, +1)$
  else
    $\{F_n^L\}_{n=1}^{nEqn} \leftarrow RiemannSolver(this.e^{idL}.\{Q_n^R\}_{n=1}^{nEqn}, this.e^{idR}.\{Q_n^L\}_{n=1}^{nEqn}, +1)$
    $this.e^{idR}.\{F_n^L\}_{n=1}^{nEqn} \leftarrow -\{F_n^L\}_{n=1}^{nEqn}$
    $this.e^{idL}.\{F_n^R\}_{n=1}^{nEqn} \leftarrow \{F_n^L\}_{n=1}^{nEqn}$
  end
end
for $k = 1$ **to** $this.K$ **do**
  $this.e^k.LocalTimeDerivative()$
end
End Procedure GlobalTimeDerivative

and the time step. Next, the time derivative computation is performed by the global mesh procedure *GlobalTimeDerivative* in Algorithm 122 (GlobalMeshProcedures). Finally, we update the solution element by element by replacing

$$\begin{array}{l}\textbf{for } j = 0 \textbf{ to } N \textbf{ do}\\ \quad G_j \leftarrow a_m G_j + \dot{\Phi}_j^n\\ \quad \Phi_j^{n+1} \leftarrow \Phi_j^{n+1} + g_m \Delta t G_j\\ \textbf{end}\end{array} \tag{8.43}$$

with

$$\begin{array}{l}\textbf{for } k = 1 \textbf{ to } K \textbf{ do}\\ \quad \textbf{for } j = 0 \textbf{ to } N \textbf{ do}\\ \quad\quad \textbf{for } n = 1 \textbf{ to } nEqn \textbf{ do}\\ \quad\quad\quad mesh.e^k.G_{j,n} \leftarrow a_m * mesh.e^k.G_{j,n} + mesh.e^k.\dot{Q}_{j,n}\\ \quad\quad\quad mesh.e^k.Q_{j,n} \leftarrow mesh.e^k.Q_{j,n} + g_m * \Delta t * mesh.e^k.G_{j,n}\\ \quad\quad \textbf{end}\\ \quad \textbf{end}\\ \textbf{end}\end{array} \tag{8.44}$$

where *mesh* is the mesh class instance.

8.1.6 Benchmark Solution: Wave Propagation and Reflection

We now solve the one-dimensional wave equation in system form

$$\begin{bmatrix} u \\ v \end{bmatrix}_t + \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}_x = \begin{bmatrix} u \\ v \end{bmatrix}_t + \begin{bmatrix} v \\ u \end{bmatrix}_x = 0, \quad x \in [0, L] \tag{8.45}$$

with a reflection boundary at $x = 0$ and a nonreflecting boundary at $x = L$ as the benchmark problem to illustrate the use of the discontinuous Galerkin spectral element approximation on a conservation law. We can use the transformation matrix

$$S = \frac{1}{2} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{8.46}$$

and its inverse to convert the system to the decoupled equation

$$\begin{bmatrix} w^+ \\ w^- \end{bmatrix}_t + \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} w^+ \\ w^- \end{bmatrix}_x = 0 \tag{8.47}$$

to see that the system has a leftgoing and a rightgoing wave, each of which moves with unit speed. The wave components are

$$\begin{aligned} w^+ &= u + v, \\ w^- &= u - v. \end{aligned} \tag{8.48}$$

In terms of the wave variables, the reflection boundary condition at the left is $w^+ = w^-$ and the non-reflection boundary condition at the right is $w^- = 0$.

To implement the discontinuous Galerkin spectral element method, we need to provide the numerical flux, which for this problem is

$$F^*\left(Q^L, Q^R; \hat{n}\right) = \frac{1}{2}\begin{bmatrix} u^L - u^R + v^L + v^R \\ u^L + u^R + v^L - v^R \end{bmatrix} \hat{n}. \tag{8.49}$$

We implement the boundary condition on the left by specifying an external state with $u^L = u^R$ and $v^L = -v^R$. We use the exact solution as the external state on the right.

The initial condition to the benchmark problem is

$$\begin{aligned} u &= 2^{-(x-1)^2/b^2}, \\ v &= 0 \end{aligned} \tag{8.50}$$

with $b = 0.15$. We derive the exact solution using the methods of characteristics and images.

Figure 8.9 shows the computed and exact solutions for five elements of equal size on the interval $[0, 5]$. For benchmark purposes, the other parameters for the computation were $N = 14$ and $\Delta t = 4 \times 10^{-2}$.

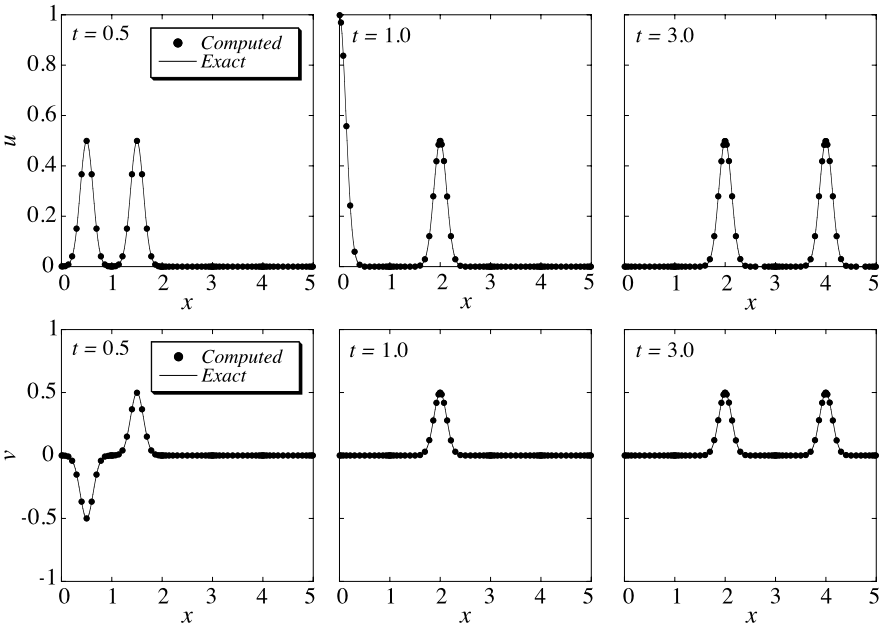

Fig. 8.9 Discontinuous Galerkin spectral element solution of the wave equation showing propagation and reflection of a wave with a five element approximation. A Gaussian spot with initial maximum at $x = 1$ splits into a leftgoing and rightgoing wave. The leftgoing wave reflects at the left boundary and then propagates to the right

8.2 The Two-Dimensional Mesh and Its Specification

In two space dimensions, we divide the physical domain into non-overlapping quadrilaterals like we have sketched in Fig. 8.10. These quadrilaterals will be our elements and the collection of elements is the mesh. The elements can have straight or curved sides, just like a single domain. Within each element we will place an $N \times N$ set of nodes—Gauss-Lobatto or Gauss, depending on the spatial approximation—at which we will approximate the solution. To avoid confusion, we explicitly refer to the set of nodes within an element as a *grid* and to the set of elements as the *mesh*.

The elements do not have to be tiled in any regular pattern or numbered in any particular order. In other words, the mesh can be *unstructured*. The only absolute restriction that we must place on the elements is that they each must have a shape that we can map onto the reference square like we did in Chap. 6. There are, however, other issues of *mesh quality* that affect the accuracy of solutions. Mesh quality measures are not as developed yet for spectral methods as they are for low order finite element methods. For a discussion of mesh quality, we defer to more technical books like [20]. As a rule of thumb, we try to keep the angles in the mesh to be as close to 90 degrees as possible.

To make the methods easier to implement, we will make one restriction on how the elements tile a domain. We will require that neighboring elements share either an entire side or a corner point. Such a mesh is called *geometrically conforming*. Figures 8.2, 8.3b, and 8.4 show examples of geometrically conforming meshes. We contrast a conforming mesh to nonconforming meshes in Fig. 8.11. The geometrically nonconforming mesh of Fig. 8.11b has elements that share a partial side. The mesh of Fig. 8.11c is geometrically conforming, but has different order polynomials on each side so that the nodes do not match across the element boundaries. Such meshes are *functionally nonconforming*. Of course, a mesh could be both geometrically and functionally nonconforming. Nonconforming meshes are sometimes easier to generate, particularly if we want to refine the mesh locally, but they will lead

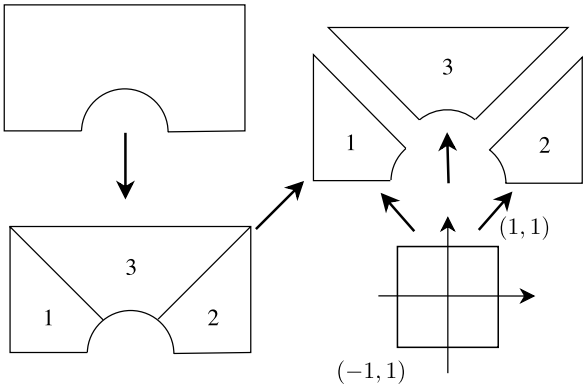

Fig. 8.10 Subdivision of a domain into quadrilateral elements, which are mapped individually from the reference square

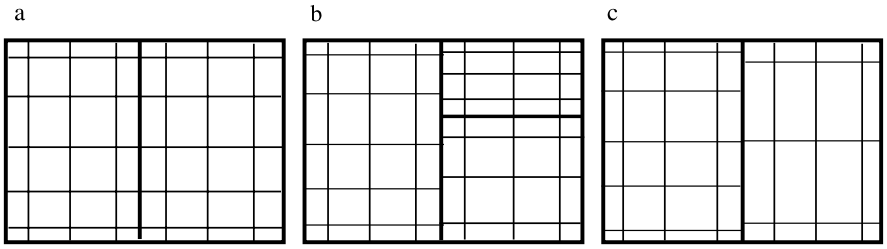

Fig. 8.11 A comparison of a conforming mesh (**a**) to a geometrically nonconforming mesh (**b**) and a functionally nonconforming mesh (**c**). Within each element of the meshes, we show grids of Gauss-Lobatto nodes at which a spectral element approximation might be approximated

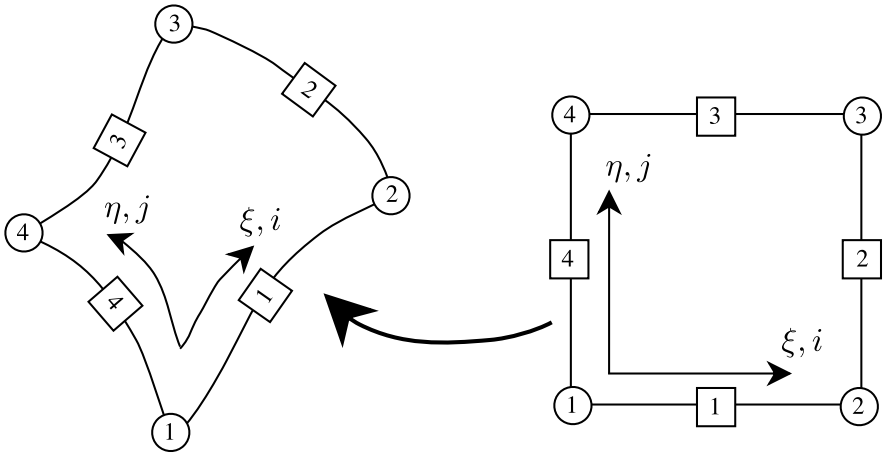

Fig. 8.12 Location of boundary curves and corner nodes that define an element shown in physical space and on the reference square

to more complex spatial approximations. Spectral approximations can be derived for both geometrically and functionally nonconforming meshes. For a discussion of how, see [8].

Let us start our specification of the mesh with the local definition of an element. An element in two dimensions is nothing but a quadrilateral domain that we used in Chap. 7. Therefore, we need four bounding curves to specify the geometry of an element. If a side is straight, then it is sufficient to specify only the two endpoints of the curve, which will be two *corner nodes* of the element. In finite element applications, the boundary curves are called *edges* and the corner nodes are called *nodes*. Since we will also have nodes associated with the Gauss or Gauss-Lobatto grids within each element, we make a distinction here between the two types. To fully define an element, we will now specify the four boundary curves and the four corner nodes, numbered counter clockwise, as shown in Fig. 8.12. We will use the corner nodes to determine how the elements are coupled.

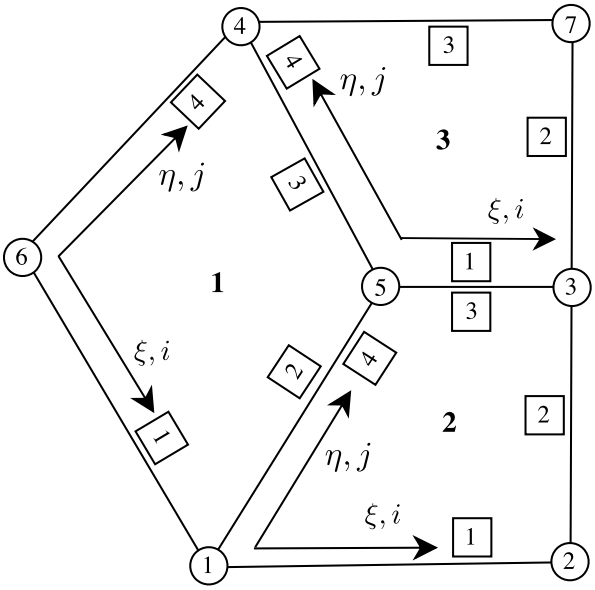

Fig. 8.13 A three element mesh that shows the local element structure constructed from seven globally numbered corner nodes (*circles*). Element sides and grid orientation are located as in Fig. 8.12

We need to know how elements are coupled if the mesh has more than one. Since we have restricted ourselves to conforming meshes, an element is coupled to a neighbor either along a boundary curve or at a corner node. What we need, then, is a list of which elements share a common side and a list of which elements share a common corner node. We can generate both of these lists (see the examples in Appendix E) if we are given a globally numbered list of corner nodes and a list of which corner nodes are used to construct each element. The list of nodes and the list of element connectivity, as the latter is known, are standard output of finite element mesh generators.

To see how to build a mesh, let us practice with the three elements shown in Fig. 8.13. The mesh has seven globally numbered corner nodes from which we construct the three elements. (If the sides were not straight, we would also have to include boundary curve information.) The elements are defined by four corners numbered counterclockwise. The choice of the first node is arbitrary, but once we choose it, we have specified the element topology (Fig. 8.12). For the mesh in Fig. 8.13, we have chosen the element connectivity as shown in Table 8.1.

The elements couple through their corner nodes and their sides. At the corners, we will need to know the elements that contribute and the location in the local nodal grid within these elements. This information is the equivalent to what was stored in the *SharedNodePointer* that we introduced in the previous section. We will use the information similarly to mask, unmask and do global sums on the solution. The

Table 8.1 Element definitions for Fig. 8.13

Element	Node 1	Node 2	Node 3	Node 4
1	6	1	5	4
2	1	2	3	5
3	5	3	7	4

Table 8.2 Corner node connectivity and local grid index

Node	Element	i	j
1	2	0	0
	1	N	0
2	2	N	0
3	2	N	N
	3	N	0
4	1	0	N
	3	0	N
5	1	N	N
	2	0	N
	3	0	0
6	1	0	0
7	3	N	N

corner connectivity is expressed by Table 8.2. Note each corner node can have a different number of adjacent elements.

The elements are also connected along their sides. We index the sides by an ordered pair of two corner nodes, which we will call an *edge*, even if the sides are not straight. When we look at Fig. 8.13, we see what information we should extract to be able to perform mask, unmask and global sum operations. First, we should find which elements border a side. If only one does, then we conclude that edge is along a boundary. Next, we need to know which sides of the neighboring elements are adjacent so that we know which part of the local element grids are connected. The last piece of information concerns the direction in which the local grid index varies along the side. For instance, along the edge indexed by Nodes 4 and 5, Side 3 of Element 1 and Side 4 of Element 3 are the neighbors. The order in which we list the two neighbors is arbitrary, but we have to choose something. Let us say that Element 1/Side 3 is the first (primary) and Element 3/Side 4 is the second (secondary). Then the nodes of Element 1 that lay along the edge are those with index (i, N) for $i = 1, 2, \ldots, N-1$ in order. We don't include the $i = 0, N$ nodes because they are corner nodes of the element. The nodes of Element 3 are those with index (N, j) with j running in reverse order, $j = N-1, N-2 \ldots, 1$. We need to flag this situation. The approach that we will take here is to set the sign of the secondary element side to be negative if the index runs in the direction opposite of the primary side. If we remember the starting index ($N-1$ or 1) and

Table 8.3 Edge information for the mesh in Fig. 8.13

Edge	Node 1	Node 2	Element 1	Side 1	Element 2	Side 2	Start	Inc.
1	6	1	1	1				
2	1	5	1	2	2	4	1	1
3	5	4	1	3	3	−4	$N-1$	−1
4	4	6	1	4				
5	1	2	2	1				
6	2	3	2	2				
7	3	5	2	3	3	1	1	1
8	3	7	3	2				
9	7	4	3	3				

the increment (± 1), we can simplify the logic when we perform computations along element edges. The edge data of the mesh in Fig. 8.13 is shown in Table 8.3. Note that the ordering of the edges is arbitrary.

8.2.1 How to Construct a Two-Dimensional Mesh

Although we can construct the node, edge, and element information by hand for a simple three element mesh just as we've done for the mesh in Fig. 8.13, it gets tedious and error prone very fast as the number of elements grows. It is better to automate the process as much as possible. How much we have to automate depends on what mesh generators we have available. For the purposes of this book, we have decided to choose the lowest common denominator so that simple mesh files can be created by hand. The minimum information needed is a list of corner node locations and a list of elements, with each element defined by its four corner nodes. To allow sides to be curves, we will include boundary curves that are constructed using Algorithm 96 (CurveInterpolant). We would normally read node, element, and boundary curve information from a file.

8.2.1.1 Nodes

The first data structure that we need to define is a corner node. At a minimum, a corner node stores its (x, y) values. To define the element connectivity so that we can perform mask, unmask and sum operations at nodes, we also need a list of what elements share the node. Since different numbers of elements can share a common node in an unstructured mesh, it is better to store this information in a linked list than in an array of fixed size. (See Appendix E.) Most mesh generators will try to keep this number, called the *valence*, low, typically less than or equal to six. So we could reasonably choose to store the adjacent element information in a

Algorithm 123: *CornerNodeClass:* Corner Node for Two-Dimensional Spectral Element Methods

Class CornerNode
Uses Algorithms:
Algorithm 144 (LinkedList)
Data:
type ; // Kind of node—INTERIOR or BOUNDARY
x, y ; // location
nodeConnectivity ; // Linked list of type CornerConnectivity
Procedures:
Construct(x, y)
End Class CornerNode

Procedure Construct
Input: x, y
$this.x \leftarrow x$; $this.y \leftarrow y$
$this.type \leftarrow INTERIOR$
nodeConnectivity.Construct()
End Procedure Construct

Structure CornerConnectivity
id, i, j
End Structure CornerConnectivity

fixed array of length six instead of a linked list. However, since we will not have to search for a particular element in this list, we will use the linked list. Finally, for convenience, we will also store the type of node, either boundary or interior, to help when we set boundary conditions. We show a corner node definition in Algorithm 123 (CornerNode).

A mesh file will typically contain a sequence of (x, y) locations that correspond to corner nodes. As each is read, we construct the node with the procedure *Construct* in Algorithm 123, which simply stores the location of the node and a default value of the type of node. We number the nodes with a node number/identifier, *id*, according to their order in the file, and will access them by their location in an array of nodes stored by the mesh. We will construct the list of adjacent elements, which is the data we collected for Table 8.2, after we construct the elements and edges.

8.2.1.2 Elements

An element is usually defined in a mesh file by an array of the *id*'s of its four corner nodes, ordered counter-clockwise. This is the data that we gathered in Table 8.1 for our three element example. For straight sided elements, the location of the four corner nodes is enough information to compute the element's geometry using Algorithm 95 (QuadMap) and metric terms using Algorithm 100 (QuadMapMetrics). For curved sides, we need additional information to define the curves.

Algorithm 124: *QuadElementClass:* Quadrilateral Element Definition for Two-Dimensional Spectral Element Methods

Class QuadElement
Uses Algorithms:
Algorithm 101 (MappedGeometry)
Algorithm 63 (Nodal2DStorage)
Data:
$\{nodeIds_j\}_{j=1}^4$; `// Corner node id's in node array.`
geom ; `// MappedGeometry to store metrics, etc.`
Procedures:
Construct(*spA*, $\{nodeIds_j\}_{j=1}^4$, $\{\mathbf{\Gamma}_j\}_{j=1}^4$)
End Class QuadElement

Procedure Construct
Input: $\{nodeIds_j\}_{j=1}^4$
Input: $\{\mathbf{\Gamma}_j\}_{j=1}^4$; `// CurveInterpolant`
Input: *spA* ; `// Nodal2DStorage`
this.$\{nodeIds_j\}_{j=1}^4 \leftarrow \{nodeIds_j\}_{j=1}^4$
this.geom.Construct(*spA*, $\{\mathbf{\Gamma}_j\}_{j=1}^4$)
End Procedure Construct

This is information not typically provided by finite element mesh generators. We usually specify Chebyshev polynomial interpolants only for those sides that are curved. From the four curves, we will store the grid and metric arrays in our standard *MappedGeometry* structure of Algorithm 101 (MappedGeometryClass). We show the storage that we need to define an element in Algorithm 124 (QuadElementClass). The constructor takes the array that lists the four corner nodes and, as we did to define a quadrilateral single domain, an array of the four boundary curves.

8.2.1.3 Edges

We will use an edge class to store the information along a row in Table 8.3. Our implementation of an edge is Algorithm 125 (EdgeClass). The constructor is simple. It takes the *id*'s of two nodes, an element of which the edge is a side, and the number of the side. We will find the identity of any additional elements that may share that side later as we construct the mesh.

8.2.1.4 The Mesh

Lastly, we need to define our *Mesh* data structure. The mesh will store the nodes, the elements and the edges. Since we are going to assume that the mesh is conforming, we will also assume that the polynomial order in all elements will be

Algorithm 125: *EdgeClass:* Edge Definition for Two-Dimensional Spectral Element Methods

Class Edge
Data:
$type$; // Kind of edge—interior or boundary
$\{nodes_k\}_{k=1}^2$; // start and end node id's
$\{elementIDs_k\}_{k=1}^2$; // Elements that share this edge
$\{elementSides_k\}_{k=1}^2$; // Sides of Elements that share this edge
$start, inc$; // Loop start and increment for the secondary side
Procedures:
$Construct(\{nodes_k\}_{k=1}^2, elementID, side)$
End Class Edge

Procedure Construct
Input: $\{nodes_k\}_{k=1}^2, elementID, side$
$this.\{nodeIds_j\}_{j=1}^2 \leftarrow \{nodeIds_j\}_{j=1}^2$
$this.type \leftarrow INTERIOR$
$this.elementIDs_1 \leftarrow elementID$
$this.elementIDs_2 \leftarrow NONE$
$this.elementSides_1 \leftarrow side$
$this.elementSides_2 \leftarrow NONE$
End Procedure Construct

the same. Therefore, we need to store only one instance of a *Nodal2DStorage* object to hold the quadrature nodes, weights, and the derivative matrices in the mesh structure. Since the number of elements and the number of corner nodes are usually listed in, or can be determined from, the mesh file, we store an array of CornerNodes and an array of Elements in the mesh structure. Finally we will store three convenience arrays that we will describe below to help navigate local data structures. We show the structure for the *Mesh* class in Algorithm 126 (QuadMesh).

We do not know the number of edges beforehand. We must construct the edges from the elements and the nodes, so we will not know how many there are to start. To be completely general, we should store the edges in some dynamic data structure like a Linked List, which can have an arbitrary length. We could use Algorithm 144 (LinkedList) in Appendix E to store and manipulate the edge list. However, we can simplify our presentation here significantly if we store the edges in an array, with the edge *id* denoted by the location in the array.

To use a fixed size array to store the edges, we need to find a reasonable upper bound on the number of edges that the mesh can have. A result from algebraic topology tells us that the number of edges, N_{edge}, the number of elements, K, and the number of nodes, N_{node}, are related to the *Euler characteristic*, χ, by the relation

$$\chi = N_{node} + K - N_{edge}. \tag{8.51}$$

Algorithm 126: *QuadMesh:* Mesh Definition for Two-Dimensional Spectral Element Methods

Class QuadMesh
Uses Algorithms:
Algorithm 63 (Nodal2DStorage)
Data:
K, N_{node}, N_{edge} ; // # Elements, corner nodes, and edges
$\{elements_k\}_{k=1}^{K}$; // Array of Elements
$\{nodes_k\}_{k=1}^{N_{node}}$; // Array of CornerNodes
$\{edges_k\}_{k=1}^{edgeDim}$; // Array of Edges
$\{cornerMap_{i,j}\}_{i=1,j=1}^{2,4}$; // Convenience array
$\{sideMap_i\}_{i=1}^{4}$; // Convenience array
$\{edgeMap_{i,j}\}_{i,j=1}^{2,4}$; // Convenience array
Procedures:
Construct(*spA*, *meshFile*); // Algorithm 127
End Class QuadMesh

In turn, the Euler characteristic is related to the number of holes, N_h, in the mesh by

$$\chi = 1 - N_h. \tag{8.52}$$

(Try it on the decompositions shown in Figs. 8.1 and 8.3b.) Of course, we don't know the number of holes in the mesh simply by looking at the nodes and element definitions, so we will only try to find a reasonable upper bound. For a "reasonable" mesh without pinched holes, we can take $N_h \leq N_{node}/3$. In that case, we expect that

$$N_{edge} \leq K + \frac{4}{3}N_{node} - 1 \equiv edgeDim. \tag{8.53}$$

The number of holes that we've assumed is large compared to the number of nodes. Typically there will be only a few holes in a mesh. The penalty, though, is only 30% of the number of nodes in the mesh. Note, however, that we can come up with pathological and unlikely meshes that will have more edges. If we expect a large number of such cases, we should switch to a linked list or other dynamic data structure to store the edges.

We will also store three convenience arrays with the mesh. The first is the *sideMap*. We will use it to tell us what the fixed index value is along a given side. For instance, by our definition shown in Fig. 8.12, Side 1 corresponds to $j = 0$ and varying i, Side 2 corresponds to $i = N$ and varying j, etc. Therefore the four values of the *sideMap* will be $\{0, N, N, 0\}$. The second convenience array is the *cornerMap* that will tell us the values of the local grid indices for the four corners. Looking back again at Fig. 8.12, we see that Corner 1 corresponds to $i = j = 0$ and Corner 2 is $i = N$, $j = 0$, etc. The *cornerMap* array is $\{\{0, 0\}, \{N, 0\}, \{N, N\}, \{0, N\}\}$. Finally, we define the *edgeMap* array to make the correspondence between an ele-

ment side and the two CornerNodes that start and terminate the side. For instance, in Fig. 8.12 we see that Side 1 is constructed from CornerNodes 1 and 2, whereas we construct Side 2 from CornerNodes 2 and 3. The elements of the *edgeMap* are $\{\{1,2\},\{2,3\},\{4,3\},\{1,4\}\}$.

We show the procedure to construct the mesh in Algorithm 127 (QuadMesh: Construct). As input, it takes an already constructed *Nodal2DStorage* object and a mesh file to read. After it constructs the convenience arrays, it reads and constructs the nodes and elements from the mesh file. Once the array of elements is constructed, the procedure sets the node connectivity. The data variable d that is added to the linked list is of type *CornerConnectivity* that we defined in Algorithm 123 (CornerNodeClass). The procedure constructs the array of edges by Algorithm 148 (ConstructMeshEdges). Once the edges have been created, the procedure goes through each edge and sets boundary types for those edges that only have one neighboring element. If two elements share a edge, the direction variables of the second element are then set.

8.2.2 Benchmark Solution: A Spectral Element Mesh for a Disk

In the following sections we will solve problems on a disk by a spectral element method to avoid the coordinate singularity at the origin. We can construct a simple mesh file for the disk by hand using the topology shown in Fig. 8.14, which has five elements and eight corner nodes. The outer boundary needs to be approximated by a polynomial of degree N at the Gauss-Lobatto nodes to be represented accurately. The mesh generated by Algorithm 127 (QuadMesh:Construct) appears in Fig. 8.15 for $N = 8$.

8.3 The Spectral Element Method in Two Space Dimensions

The spectral element method is a continuous nodal spectral Galerkin approximation. We have used the continuous Galerkin approximation before to approximate potential and advection-diffusion problems. As a Galerkin method, we start the derivation from a weak form of the equation that we wish to solve.

We will introduce the spectral element method for two-dimensional geometries by approximating the potential equation with Dirichlet boundary conditions,

$$\begin{aligned} \nabla^2 \varphi &= s, \quad x \in \Omega, \\ \varphi &= \varphi_b, \quad x \in \partial\Omega. \end{aligned} \tag{8.54}$$

To convert the equation to the time dependent heat equation, we let $s = \partial\varphi/\partial t$. To approximate the advection-diffusion equation we will add an advection term as we did in Sect. 5.3 so that $s = \partial\varphi/\partial t + \mathbf{q} \cdot \nabla\varphi$.

Algorithm 127: *QuadMesh:Construct:* Constructor for a Two Dimensional Spectral Element Mesh

Procedure Construct
Input: Mesh File
Input: *spA* ; `// Nodal2DStorage`
Uses Algorithms:
Algorithm 63 (Nodal2DStorage)
Algorithm 148 (ConstructMeshEdges)
$N \leftarrow this.spA.N$
$this.\left\{sideMap_k\right\}_{k=1}^{4} \leftarrow \{0, N, N, 0\}$
$this.\left\{cornerMap_{1,k}\right\}_{k=1}^{4} \leftarrow \{0, N, N, 0\}$
$this.\left\{cornerMap_{2,k}\right\}_{k=1}^{4} \leftarrow \{0, 0, N, N\}$
$this.\left\{edgeMap_{1,k}\right\}_{k=1}^{4} \leftarrow \{1, 2, 4, 1\}$
$this.\left\{edgeMap_{2,k}\right\}_{k=1}^{4} \leftarrow \{2, 3, 3, 4\}$
Read from Mesh File: $this.N_{node}$, $this.K$
Allocate memory for nodes, elements and for edge array using (8.53)
$this.N_{edge} \leftarrow 0$
for $k = 1$ **to** $this.N_{node}$ **do**
  Read from Mesh File: x, y
  $this.nodes_k.Construct(x, y)$
end
for $k = 1$ **to** $this.K$ **do**
  Read from Mesh File: $\{cornerNodes_k\}_{k=1}^{4}$
  Read from Mesh File and Construct: $\{\mathbf{\Gamma}_j\}_{j=1}^{4}$
  $this.elements_k.Construct(spA, \{cornerNodes_k\}_{k=1}^{4}, \{\mathbf{\Gamma}_j\}_{j=1}^{4})$
end
for $eId = 1$ **to** $this.K$ **do**
  for $k = 1$ **to** *4* **do**
    $d.id \leftarrow eId$; $d.i \leftarrow this.cornerMap_{1,k}$; $d.j \leftarrow this.cornerMap_{2,k}$
    $n \leftarrow this.elements_{eId}.nodes_k$
    $this.nodes_n.NodeConnectivity.Add(d)$
  end
end
$this \leftarrow ConstructMeshEdges(this)$
for $k = 1$ **to** $this.N_{edge}$ **do**
  if $this.edges_k.elementID_2 = NONE$ **then**
    $this.edges_k.type \leftarrow BOUNDARY$
    $n1 \leftarrow this.edges_k.nodes_1$
    $n2 \leftarrow this.edges_k.nodes_2$
    $this.nodes_{n1}.type \leftarrow BOUNDARY$
    $this.nodes_{n2}.type \leftarrow BOUNDARY$
  else
    if $this.edges_k.elementSides_2 > 0$ **then**
      $this.edges_k.start = 1$; $this.edges_k.inc = 1$
    else
      $this.edges_k.start = N - 1$; $this.edges_k.inc = -1$
    end
  end
end
End Procedure Construct

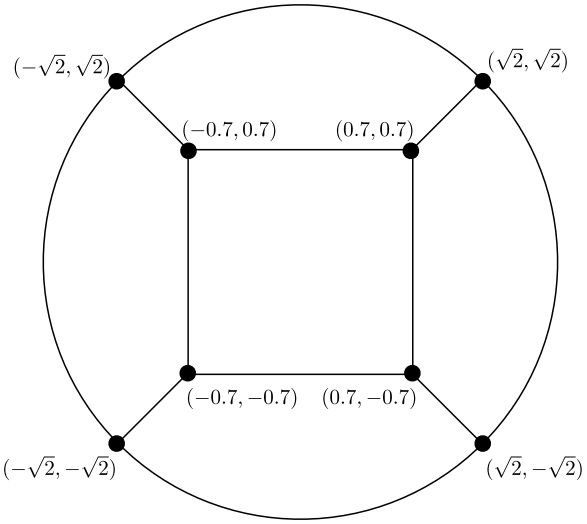

Fig. 8.14 A decomposition of a disk into five elements

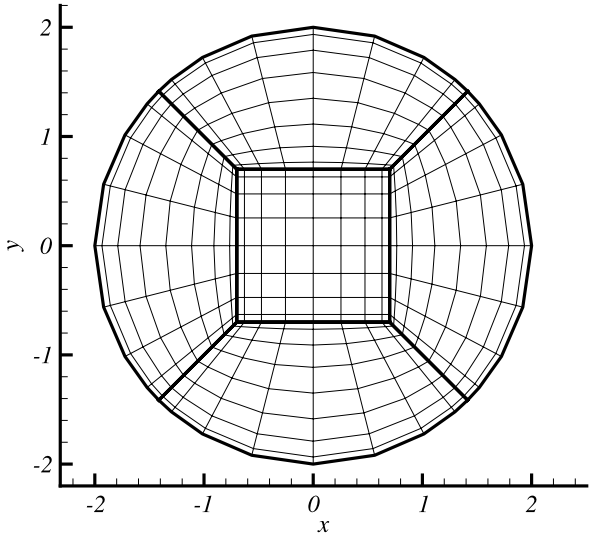

Fig. 8.15 Spectral element mesh for discretization of the disk

To get the weak form of the potential equation, we multiply it by an arbitrary, sufficiently smooth test function ϕ and integrate over the entire domain

$$\iint_{\Omega} \nabla^2 \varphi \phi dx dy = \iint_{\Omega} s\phi dx dy. \tag{8.55}$$

We then apply Green's identity to the integral with the Laplacian to rewrite it as

$$\iint_{\Omega} \nabla^2 \varphi \phi dxdy = \int_{\partial\Omega} \phi \frac{\partial \varphi}{\partial n} dL - \iint_{\Omega} \nabla\varphi \cdot \nabla\phi dxdy, \tag{8.56}$$

where dL is the differential along the boundary curve and $\partial\varphi/\partial n = \nabla\varphi \cdot \hat{n}$ is the normal derivative. As usual, for Dirichlet conditions we require that $\phi = 0$ along the boundary. Therefore the boundary integral vanishes to leave us with

$$-\iint_{\Omega} \nabla\varphi \cdot \nabla\phi dxdy = \iint_{\Omega} s\phi dxdy. \tag{8.57}$$

To get the spectral element method, we subdivide the domain Ω into K nonoverlapping quadrilateral elements, e^k, as we described in the introduction to this chapter. By breaking the domain into elements, we can break the integrals over the domain into the sum of integrals over the elements

$$\sum_{k=1}^{K} \left\{ -\iint_{e^k} \nabla\varphi \cdot \nabla\phi dxdy \right\} = \sum_{k=1}^{K} \left\{ \iint_{e^k} s\phi dxdy \right\}. \tag{8.58}$$

Therefore, each element contributes

$$-\iint_{e^k} \nabla\varphi \cdot \nabla\phi dxdy = \iint_{e^k} s\phi dxdy \tag{8.59}$$

to the total integral. For convenience, let us call

$$\begin{aligned} I_1^k &= -\iint_{e^k} \nabla\varphi \cdot \nabla\phi dxdy, \\ I_2^k &= \iint_{e^k} s\phi dxdy. \end{aligned} \tag{8.60}$$

The local element contributions (8.59) are exactly what we approximated in Sect. 5.2.2 on a single domain, so we already know how to approximate them. To recap, we first map the element e^k onto the reference square by a mapping $\mathbf{x} = \mathbf{X}(\xi, \eta)$. On the reference square, I_2^k transforms to

$$I_2^k = \int_{-1}^{1} \int_{-1}^{1} J^k s\phi d\xi d\eta. \tag{8.61}$$

To transform I_1^k, we write the gradient in the mapped coordinate as

$$\nabla\varphi = \frac{1}{J} \left\{ \left(Y_\eta \varphi_\xi - Y_\xi \varphi_\eta \right) \hat{x} + \left(X_\xi \varphi_\eta - X_\eta \varphi_\xi \right) \hat{y} \right\} \tag{8.62}$$

so that, with a little algebra, we get the integrand

$$\nabla\varphi\cdot\nabla\phi=\frac{1}{J}\left\{\left[\frac{Y_\eta^2+X_\eta^2}{J}\varphi_\xi-\frac{Y_\xi Y_\eta+X_\xi X_\eta}{J}\varphi_\eta\right]\phi_\xi\right.$$
$$\left.+\left[\frac{Y_\xi^2+X_\xi^2}{J}\varphi_\eta-\frac{Y_\xi Y_\eta+X_\xi X_\eta}{J}\varphi_\xi\right]\phi_\eta\right\}. \tag{8.63}$$

Ultimately, we write (8.63) in the familiar form

$$\nabla\varphi\cdot\nabla\phi=\frac{1}{J}\left\{f\phi_\xi+g\phi_\eta\right\}. \tag{8.64}$$

When we replace the integrand in (8.59) with (8.64), we write the element contribution as

$$-\int_{-1}^{1}\int_{-1}^{1}\left(f\phi_\xi+g\phi_\eta\right)d\xi d\eta=\int_{-1}^{1}\int_{-1}^{1}sJ^k\phi d\xi d\eta. \tag{8.65}$$

Therefore, we've already seen (8.59) on a mapped domain; the contribution of element k written on the reference square is just (7.17).

To approximate (8.65), we replace s by a piecewise polynomial approximation S and replace φ by a piecewise continuous polynomial approximation Φ. As an extension to what we did in one space dimension, (8.15), the test functions are going to be the continuous, piecewise polynomials

$$\phi^k=\sum_{i,j=0}^{N}\phi_{ij}^k\ell_i(\xi)\,\ell_j(\eta). \tag{8.66}$$

We enforce continuity of the test functions by requiring that the nodal values ϕ_{ij}^k be the same along each edge and at each element corner in the mesh. The same goes for the solution and source term polynomials.

When we substitute (8.66) for ϕ in (8.65),

$$I_1^k\approx\sum_{i,j}\phi_{i,j}^k\left[-\int_{-1}^{1}\int_{-1}^{1}\left(F\ell_i'\ell_j+G\ell_i\ell_j'\right)d\xi d\eta\right] \tag{8.67}$$

and

$$I_2^k\approx\sum_{i,j}\phi_{i,j}^k\left[\int_{-1}^{1}\int_{-1}^{1}S\ell_i\ell_j d\xi d\eta\right]. \tag{8.68}$$

The integrals in the square brackets are the same as in the single domain problem, and we already have their nodal Galerkin approximations worked out in (7.18)–(7.21). Therefore, the nodal Galerkin approximation of (8.65) is

$$\sum_{i,j}\phi_{i,j}^k\left[\left(\nabla^2\Phi,\ell_i\ell_j\right)_N-S_{i,j}^kJ_{i,j}^kw_iw_j\right]=0, \tag{8.69}$$

where $(\nabla^2\Phi, \ell_i\ell_j)_N$ is given by (7.21). The global sum over all the elements gives us our final approximation

$$\sum_{k=1}^{K}\left\{\sum_{i,j}\phi_{i,j}^k\left[\left(\nabla^2\Phi, \ell_i\ell_j\right)_N - S_{i,j}^k J_{i,j}^k w_i w_j\right]\right\} = 0. \tag{8.70}$$

We get the spectral element approximations to the time dependent diffusion and advection-diffusion equations from (8.70) if we replace $S_{i,j}^k$ by the appropriate approximations. For instance, the approximation to the time dependent diffusion equation is

$$\sum_{k=1}^{K}\left\{\sum_{i,j}\phi_{i,j}^k\left[\left(\nabla^2\Phi, \ell_i\ell_j\right)_N - \dot{\Phi}_{i,j}^k J_{i,j}^k w_i w_j\right]\right\} = 0, \tag{8.71}$$

whereas the approximation to the advection-diffusion equation is

$$\sum_{k=1}^{K}\left\{\sum_{i,j}\phi_{i,j}^k\left[\left(\nabla^2\Phi, \ell_i\ell_j\right)_N - \dot{\Phi}_{i,j}^k J_{i,j}^k w_i w_j - \left(\mathbf{q}\cdot\nabla\Phi, \ell_i\ell_j\right)_N\right]\right\} = 0 \tag{8.72}$$

with the advection term given by (7.77).

The last thing for us to do is to use the fact that the $\phi_{i,j}^k$ are independent except along element edges and at element corners to get the pointwise equations that the solution unknowns must satisfy. Let's focus on finding those equations for (8.70). The equations for (8.71) and (8.72) will follow directly.

To get the pointwise equations for the approximation (8.70), the first thing to notice is that the terms within the square brackets are simply the single domain approximation applied to the element e^k. At points interior to the elements, the $\phi_{i,j}^k$ are independent, which means that the approximation in an element is just the single domain approximation applied to it. Along edges, the approximation is the sum of the single domain values from the two contributing elements, just as we saw in the one dimensional spectral element method. Finally, at corners, all elements that share the point contribute to the sum. It is not as easy to write the summations explicitly as it was in one space dimension, but we will see that it is relatively easy to implement.

8.3.1 How to Implement the Spectral Element Method

The approximations (8.70)–(8.72) show that the spectral element method has local operations, gathered in the brackets, and global operations, gathered in the braces, so we will organize the algorithms as we did in one space dimension as local and global. The local operations are the single domain approximations that we have

already developed for quadrilateral domains. The global operations tie the local approximations together. The two-level form of the approximations shows that we can form a global framework that is essentially independent of the equations that we want to solve; We can compute the spectral approximation of the potential equation, the diffusion equation and the advection-diffusion equation with the same framework. Likewise, we have already developed the local operations for potentials, diffusion and advection-diffusion.

We will describe how to implement the approximation for the Dirichlet problem for the potential equation, (8.54) in detail. The implementation of Neumann boundary conditions and the time dependent problems are just extensions that we will pose as exercises.

8.3.1.1 The Potential Class

We organize the solution of the potential equation in a class, which we show in Algorithm 128 (SEMPotentialClass). Notice that the class is just an extension of the single domain class that we developed in Sect. 7.1.2. Since we now require that the polynomial order be the same in each element and in each direction within an element to guarantee most easily that the approximation is conforming, we still only need one instance of Nodal2DStorage, which we denote by the variable *spA*. However, now that there are multiple domains, the mesh will now store the geometry and mapping information with its elements. The mesh also stores the connectivity. Finally, we will use a global scheme for the solution and source terms to make it easy to use the Conjugate Gradient solver.

Algorithm 128: *SEMPotentialClass:* A Class Definition for the Spectral Element Approximation of the Potential Problem

Class SEMPotentialClass
Uses Algorithms:
Algorithm 63 (Nodal2DStorage)
Algorithm 101 (MappedGeometryClass)
Algorithm 107 (MappedLaplacian); Algorithm 126 (QuadMesh)
Data:
spA ; `// Of type Nodal2DStorage`
mesh ; `// Of type QuadMesh`
$\{\Phi_{i,j,k}\}_{i,j=0;k=1}^{N,N;K}$; `// Solution`
$\{s_{i,j,k}\}_{i,j=0;k=1}^{N,N;K}$; `// Source`
Procedures:
Construct$(N, meshFile)$; `// Algorithm 129`
MappedLaplacian$(\{U_{i,j,k}\}_{i,j=0}^{N}, geom)$; `// Algorithm 107`
MatrixAction$(\{U_{ij}\}_{i,j=0}^{N})$; `// Algorithm 133`
End Class SEMPotentialClass

Algorithm 129: *SEMPotentialClass:Construct:* Constructor for the Spectral Element Approximation of the Potential Problem

Procedure Construct
Input: N, *meshFile*
Uses Algorithms:
Algorithm 25 (LegendreGaussLobattoNodesAndWeights)
Algorithm 37 (PolynomialDerivativeMatrix)
Algorithm 127 (QuadMesh:Construct)

$this.spA.N \leftarrow N$; $this.spA.M \leftarrow N$
$\left\{this.spA.\{\xi_i\}_{i=0}^N, this.spA.\left\{w_i^{(\xi)}\right\}_{i=0}^N\right\} \leftarrow LegendreGaussLobattoNodesAndWeights(N)$
$this.spA.\left\{D_{ij}^{\xi}\right\}_{i,j=0}^N \leftarrow PolynomialDerivativeMatrix(this.spA.\{\xi_i\}_{i=0}^N)$
Copy arrays to η direction...
$mesh.Construct(spA, meshFile)$
End Procedure Construct

We must implement three procedures to compute the potential approximation. The constructor, Algorithm 129 (SEMPotentialClass:Construct), computes the spatial approximation array. One simplification here is that the node, weight and derivative matrix arrays are the same in both directions, so they need to be computed only once. The other action the constructor must take is to construct the mesh from a mesh file using Algorithm 127 (QuadMesh:Construct). We have already implemented the second procedure, *MappedLaplacian* in Algorithm 107 (MappedNodalGalerkinLaplacian). This is the local operation that computes the Laplacian term in the brackets in (8.70). The procedure works only on the section of the solution array for a given element, e^k. The geometry object that we pass with the array will be the geometry for the element. The final procedure computes the matrix action to be used with an iterative solver. The matrix action includes global operations, so we will wait to show what it does until after we develop the global procedures.

8.3.1.2 Global Procedures

As in one space dimension, we implement three global operations: Mask, UnMask and GlobalSum. The Mask operation will set duplicate node values in an array to zero so that they will have no contribution to the iterative solver. The UnMask operation will distribute the values back to the duplicate nodes. The global sum will add the contributions together at duplicate nodes, performing the operation in braces in (8.70). The procedures for all three operations have a similar structure. They first do their operations for each element side, then for each corner node.

We use the global Mask operation to set array values on duplicate nodes and boundary nodes to zero. We show an implementation in Algorithm 130 (SEMMask). The first section of the procedure loops over the array of edges and distinguishes between boundary an internal edges. Dirichlet conditions require the values to be

Algorithm 130: *SEMMask:* Mask Edges and Corners for the Spectral Element Method

Procedure Mask
Input: $mesh$, $\{a_{i,j,k}\}_{i,j=0;k=1}^{N;K}$

Uses Algorithms:
Algorithm 126 (QuadMesh)

for $j = 1$ **to** $mesh.N_{edge}$ **do**
  if $mesh.edges_j.type = BOUNDARY$ **then**
    $e \leftarrow mesh.edges_j.elementIDs_1$
    $s \leftarrow mesh.edges_j.elementSides_1$
  else
    $e \leftarrow mesh.edges_j.elementIDs_2$
    $s \leftarrow |mesh.edges_j.elementSides_2|$
  end
  $\{a_{i,j,k}\}_{i,j=0;k=1}^{N;K} \leftarrow MaskSide(e, s, mesh, \{a_{i,j,k}\}_{i,j=0;k=1}^{N;K})$
end
for $n = 1$ **to** $mesh.N_{node}$ **do**
  $pElements \Rightarrow mesh.nodes_n.nodeConnectivity$
  $pElements.current \Rightarrow pElements.head$
  if $mesh.nodes_n.type \neq BOUNDARY$ **then** pElements.MoveToNext()
  while $pElements.current \not\Rightarrow NULL$ **do**
    $d \leftarrow pElements.GetCurrentData()$
    $i \leftarrow d.i$; $j \leftarrow d.j$; $id \leftarrow d.id$
    $a_{i,j,id} \leftarrow 0$
    $pElements.MoveToNext()$
  end
end
return $\{a_{i,j,k}\}_{i,j=0;k=1}^{N;K}$
End Procedure Mask

Procedure MaskSide
Input: $id, side, mesh, \{a_{i,j,k}\}_{i,j=0;k=1}^{N;K}$

if $side = 2$ **or** $side = 4$ **then**
  $i \leftarrow mesh.sideMap_{side}$
  for $j = 1$ **to** $mesh.spA.N - 1$ **do**
    $a_{i,j,id} \leftarrow 0$
  end
else
  $j \leftarrow mesh.sideMap_{side}$
  for $i = 1$ **to** $mesh.spA.N - 1$ **do**
    $a_{i,j,id} \leftarrow 0$
  end
end
return $\{a_{i,j,k}\}_{i,j=0;k=1}^{N;K}$
End Procedure MaskSide

masked along boundary edges. So if the edge is a boundary edge, we mask the primary (and only), index 1, side. Otherwise, if the edge is on the interior, we mask the secondary (index 2) element side. We use the sideMap array to select the correct

local index when masking the secondary side in the *MaskSide* procedure. We apply the same philosophy to the corner nodes. For each corner node, we select the first contributing element to be the primary. If the corner node is on a physical boundary, we mask it to implement the Dirichlet condition. Otherwise, we mask the array entry for the nodes from each of the remaining elements that share the corner node.

The global UnMask operation, which we implement in Algorithm 131 (SEMUnmask), undoes the action of the Mask operation. It copies the data from the primary node to the secondary nodes. Again, the structure of the procedure is the same as *SEMMask*. However, we must account for the fact that the indices along the contributing sides do not have to increase in the same direction. In the implementation that we show here, we copy the values from the primary side into a temporary array to make the operations as clear as possible. We then use the edge's *start* and *inc* values to copy the edge values to the correct location for the secondary side in the global array.

The final global operation is the global summation, which we implement in Algorithm 132 (SEMGlobalSum). The global summation takes values from all contributing nodes, adds them together, and then distributes them back. Like the *SEMUnmask* procedure, the global summation must account for the fact that the element edges do not have to have indices that vary in the same direction. To be clear, we create two temporary arrays that store the summed values in the orders needed by the two sides. Then we simply copy those two arrays to their contributing sides.

8.3.1.3 Procedures for the Iterative Solver

Now that we have implemented the global operations, we implement the *MatrixAction* and *Residual* procedures that we need to use the Conjugate Gradient algorithm, Algorithm 80 (PreconditionedConjugateGradientSolve), to solve the system of equations. Algorithm 133 (SEMPotentialClass:MatrixAction) shows the matrix action. It first unmasks the solution values so that the Laplace approximations can be computed locally in a loop over each of the elements. Once the local actions are computed they are summed globally and then masked. To ensure that the procedure produces no side effects, it re-masks the input array. The procedure *Residual* that we show in Algorithm 134 is similar to the *MatrixAction* procedure and computes the global residual.

Since we use the Conjugate Gradient method to solve the linear system for the potentials, we should mention preconditioners for the spectral element method. The similarity of the spectral element method to the finite element method allows us to generalize the finite element preconditioner that we have already implemented to multiple domains. We showed in Sect. 7.1.3 how to modify the finite element preconditioner to work on a transformed domain that now forms one of our elements. If we use an iterative solver for the preconditioner, like the *SSORSweep* procedure that we presented in Algorithm 79, then we can generalize the finite element approximation to the multidomain discretization in the same way that we did the spectral method. That is, we compute the local preconditioners, and perform the

Algorithm 131: *SEMUnMask:* UnMask for the Spectral Element Method

```
Procedure UnMask
Input: mesh, {a_{i,j,k}}_{i,j=0;k=1}^{N;K}
Uses Algorithms:
  Algorithm 126 (QuadMesh)
for j = 1 to mesh.N_edge do
    if mesh.edges_j.type ≠ BOUNDARY then
        {a_{i,j,k}}_{i,j=0;k=1}^{N;K} ← UnMaskSide(mesh.edges_j, mesh, {a_{i,j,k}}_{i,j=0;k=1}^{N;K})
    end
end
for n = 1 to mesh.N_nodes do
    if mesh.nodes_n.type ≠ BOUNDARY then
        pElements ⇒ mesh.nodes_n.nodeConnectivity
        pElements.current ⇒ pElements.head
        d ← pElements.GetCurrentData()
        i1 ← d.i; j1 ← d.j; id1 ← d.id
        pElements.MoveToNext()
        while pElements.current ⇏ NULL do
            d ← pElements.GetCurrentData()
            i ← d.i; j ← d.j; id ← d.id
            a_{i,j,id} ← a_{i1,j1,id1}
            pElements.MoveToNext()
        end
    end
end
return {a_{i,j,k}}_{i,j=0;k=1}^{N;K}
End Procedure UnMask
```

```
Procedure UnMaskSide
Input: edge, mesh, {a_{i,j,k}}_{i,j=0;k=1}^{N;K}
id ← edge.elementIDs_1; side ← edge.elementSides_1
if side = 2 or side = 4 then
    i ← mesh.sideMap_side
    for j = 1 to mesh.spA.N − 1 do
        tmp_j ← a_{i,j,id}
    end
else
    j ← mesh.sideMap_side
    for i = 1 to mesh.spA.N − 1 do
        tmp_i ← a_{i,j,id}
    end
end
id ← edge.elementIDs_2; side ← |edge.elementSides_2|
if side = 2 or side = 4 then
    i ← mesh.sideMap_side; j ← edge.start
    for n = 1 to mesh.spA.N − 1 do
        a_{i,j,id} ← tmp_n
        j ← j + edge.inc
    end
else
    j ← mesh.sideMap_side; i ← edge.start
    for n = 1 to mesh.spA.N − 1 do
        a_{i,j,id} ← tmp_n
        i ← i + edge.inc
    end
end
return {a_{i,j,k}}_{i,j=0;k=1}^{N;K}
End Procedure UnMaskSide
```

Algorithm 132: *SEMGlobalSum:* Sum Edge Contributions for the Two-Dimensional Spectral Element Method

```
Procedure GlobalSum
Input: mesh, {a_{i,j,k}}_{i,j=0;k=1}^{N;K}
Uses Algorithms:
  Algorithm 126 (QuadMesh)
for j = 1 to mesh.N_edge do
    if mesh.edges_j.type ≠ BOUNDARY then
        {a_{i,j,k}}_{i,j=0;k=1}^{N;K} ← SumSide(mesh.edges_j, mesh, {a_{i,j,k}}_{i,j=0;k=1}^{N;K})
    end
end
for n = 1 to mesh.N_node do
    if mesh.nodes_n.type ≠ BOUNDARY then
        pElements ⇒ mesh.nodes_n.nodeConnectivity
        pElements.current ⇒ pElements.head
        sum = 0
        while pElements.current ≠ NULL do
            d ← pElements.GetCurrentData()
            i ← d.i; j ← d.j; id ← d.id
            sum ← sum + a_{i,j,id}
            pElements.MoveToNext()
        end
        pElements.current ⇒ pElements.head
        while pElements.current ≠ NULL do
            d ← pElements.GetCurrentData()
            i ← d.i; j ← d.j; id ← d.id
            a_{i,j,id} ← sum
            pElements.MoveToNext()
        end
    end
end
return {a_{i,j,k}}_{i,j=0;k=1}^{N;K}
End Procedure GlobalSum
```

masks and global sums on those to get the global action and residuals. This is not the way we would implement a finite element method from scratch, but it fits easily into the framework that we have developed so far. One problem with the finite element preconditioner is that as the meshes get large its convergence rate slows down, too. More sophisticated and complex spectral element preconditioners have been developed over the years. For those we point to Chap. 6 of the book [8], which describes several strategies including alternating Schwartz and Schur complement techniques.

8.3.1.4 The Driver

The driver to solve the potential problem with the spectral element method has the same structure as the driver on the square, Algorithm 76 (CollocationPoten-

Algorithm 132: *SEMGlobalSum:* Sum Edge Contributions for the Two-Dimensional Spectral Element Method (continued)

```
Procedure SumSide
Input: edge, mesh, {a_{i,j,k}}_{i,j=0;k=1}^{N;K}

for k = 1 to 2 do
    id ← edge.elementIDs_k; side ← |edge.elementSides_k|
    if side = 2 or side = 4 then
        i ← mesh.sideMap_side
        for j = 1 to mesh.spA.N − 1 do
            tmp_{j,k} ← a_{i,j,id}
        end
    else
        j ← mesh.sideMap_side
        for i = 1 to mesh.spA.N − 1 do
            tmp_{i,k} ← a_{i,j,id}
        end
    end
end
n ← edge.start
for j = 1 to mesh.spA.N − 1 do
    sum ← tmp_{j,1} + tmp_{j,2}
    tmp_{j,1} ← sum; tmp_{n,2} ← sum
    n ← n + edge.inc
end
for k = 1 to 2 do
    id ← edge.elementIDs_k; side ← |edge.elementSides_k|
    if side = 2 or side = 4 then
        i ← mesh.sideMap_side
        for j = 1 to mesh.spA.N − 1 do
            a_{i,j,id} ← tmp_{j,k}
        end
    else
        j ← mesh.sideMap_side
        for i = 1 to mesh.spA.N − 1 do
            a_{i,j,id} ← tmp_{i,k}
        end
    end
end
return {a_{i,j,k}}_{i,j=0;k=1}^{N;K}
End Procedure SumSide
```

tialDriver) and the quadrilateral, Algorithm 108 (MappedCollocationDriver). As before, a *SourceValue* function must be provided to compute the source, s. The boundary mask array is not needed because we have assumed only Dirichlet boundary conditions and have incorporated them into the global mask and unmask functions themselves. We do need a routine to set the boundary values. For that, we present an implementation in Algorithm 135 (SetBoundaryValues). It assumes that the actual function that provides the solution value as a function of x, y (and t for time dependent problems) is provided as input.

Algorithm 133: *SEMPotentialClass:MatrixAction:* Matrix Action for the Spectral Element Approximation to the Potential Equation

Procedure MatrixAction
Uses Algorithms:
Algorithm 126 (QuadMesh)
Algorithm 107 (MappedLaplacian)
Input: $\left\{\Phi_{i,j,k}\right\}_{i,j=0;k=1}^{N;K}$

$\left\{\Phi_{i,j,k}\right\}_{i,j=0;k=1}^{N;K} \leftarrow UnMask\left(this.mesh, \left\{\Phi_{i,j,k}\right\}_{i,j=0;k=1}^{N;K}\right)$
for $k = 1$ **to** $this.mesh.K$ **do**
  $\left\{action_{i,j,k}\right\}_{i,j=0}^{N,M} \leftarrow MappedLaplacian\left(this.mesh.elements_k.geom, \left\{\Phi_{i,j,k}\right\}_{i,j=0}^{N,M}\right)$
end
$\left\{action_{i,j,k}\right\}_{i,j=0;k=1}^{N;K} \leftarrow GlobalSum\left(this.mesh, \left\{action_{i,j,k}\right\}_{i,j=0;k=1}^{N;K}\right)$
$\left\{action_{i,j,k}\right\}_{i,j=0;k=1}^{N;K} \leftarrow Mask\left(this.mesh, \left\{action_{i,j,k}\right\}_{i,j=0;k=1}^{N;K}\right)$
$\left\{\Phi_{i,j,k}\right\}_{i,j=0;k=1}^{N;K} \leftarrow Mask\left(this.mesh, \left\{\Phi_{i,j,k}\right\}_{i,j=0;k=1}^{N;K}\right)$
return $\left\{action_{i,j,k}\right\}_{i,j=0;k=1}^{N;K}$
End Procedure MatrixAction

Algorithm 134: *Residual:* Residual Computation for the Spectral Element Approximation to the Potential Equation

Procedure Residual
Input: pA ; `// Of type SEMPotentialClass`
Uses Algorithms:
Algorithm 128 (SEMPotentialClass)
Algorithm 107 (MappedLaplacian)

$pA.\left\{\Phi_{i,j,k}\right\}_{i,j=0;k=1}^{N;K} \leftarrow UnMask\left(pA.mesh, pA.\left\{\Phi_{i,j,k}\right\}_{i,j=0;k=1}^{N;K}\right)$
for $k = 1$ **to** $mesh.K$ **do**
  $\left\{r_{i,j,k}\right\}_{i,j=0}^{N,M} \leftarrow MappedLaplacian\left(pA.mesh.elements_k.geom, pA.\left\{\Phi_{i,j,k}\right\}_{i,j=0}^{N,M}\right)$
  for $j = 0$ **to** $pA.spA.N$ **do**
    for $i = 0$ **to** $pA.spA.M$ **do**
      $r_{i,j,k} \leftarrow pA.spA.w_i^{(\xi)} * pA.spA.w_j^{(\eta)} * pA.source_{i,j,k} * pA.mesh.elements_k.geom.J_{i,j} - r_{i,j,k}$
    end
  end
end
$\left\{r_{i,j,k}\right\}_{i,j=0;k=1}^{N;K} \leftarrow GlobalSum\left(pA.mesh, \left\{r_{i,j,k}\right\}_{i,j=0;k=1}^{N;K}\right)$
$\left\{r_{i,j,k}\right\}_{i,j=0;k=1}^{N;K} \leftarrow Mask\left(pA.mesh, \left\{r_{i,j,k}\right\}_{i,j=0;k=1}^{N;K}\right)$
$pA.\left\{\Phi_{i,j,k}\right\}_{i,j=0;k=1}^{N;K} \leftarrow Mask\left(pA.mesh, pA.\left\{\Phi_{i,j,k}\right\}_{i,j=0;k=1}^{N;K}\right)$
return $\left\{r_{i,j,k}\right\}_{i,j=0;k=1}^{N;K}$
End Procedure Residual

Algorithm 135: *SetBoundaryValues:* Set Dirichlet Boundary Conditions for the Two-Dimensional Spectral Element Method

Procedure SEMSetDirichletBoundaries
Input: $mesh$, $\{\Phi_{i,j,k}\}_{i,j=0;k=1}^{N;K}$, t, $BCFunction$
Uses Algorithms:
Algorithm 126 (QuadMesh)

for $k = 1$ **to** $mesh.N_{edge}$ **do**
  if $mesh.edges_j.type = BOUNDARY$ **then**
    $id \leftarrow mesh.edges_k.elementIDs_1$; $side \leftarrow mesh.edges_k.elementSides_1$
    if $side = 2$ **or** $side = 4$ **then**
      $i \leftarrow mesh.sideMap_{side}$
      for $j = 1$ **to** $mesh.spA.N - 1$ **do**
        $x \leftarrow mesh.elements_{id}.geom.x_{i,j}$
        $y \leftarrow mesh.elements_{id}.geom.y_{i,j}$
        $\Phi_{i,j,id} \leftarrow BCFunction(x, y, t)$
      end
    else
      $j \leftarrow mesh.sideMap_{side}$
      for $i = 1$ **to** $mesh.spA.N - 1$ **do**
        $x \leftarrow mesh.elements_{id}.geom.x_{i,j}$
        $y \leftarrow mesh.elements_{id}.geom.y_{i,j}$
        $\Phi_{i,j,id} \leftarrow BCFunction(x, y, t)$
      end
    end
  end
end
for $n = 1$ **to** $mesh.N_{node}$ **do**
  if $mesh.nodes_n.type = BOUNDARY$ **then**
    $pElements \Rightarrow mesh.nodes_n.nodeConnectivity$
    $x \leftarrow mesh.nodes_n.x$; $y \leftarrow mesh.nodes_n.y$
    $\Phi_b \leftarrow BCFunction(x, y, t)$
    $pElements.current \Rightarrow pElements.head$
    while $pElements.current \Rightarrow NULL$ **do**
      $d \leftarrow pElements.GetCurrentData()$
      $i \leftarrow d.i$; $j \leftarrow d.j$; $id \leftarrow d.id$
      $\Phi_{i,j,id} \leftarrow \Phi_b$
      $pElements.MoveToNext()$
    end
  end
end
return $\{\Phi_{i,j,k}\}_{i,j=0;k=1}^{N;K}$
End Procedure SEMSetDirichletBoundaries

8.3.2 Benchmark Solution: Steady Temperatures in a Long Cylindrical Rod

The benchmark problem for the spectral element method is to compute the steady temperature in a long cylindrical rod that is heated uniformly along its length. The simple physical model reduces to the solution of the potential equation on a circular

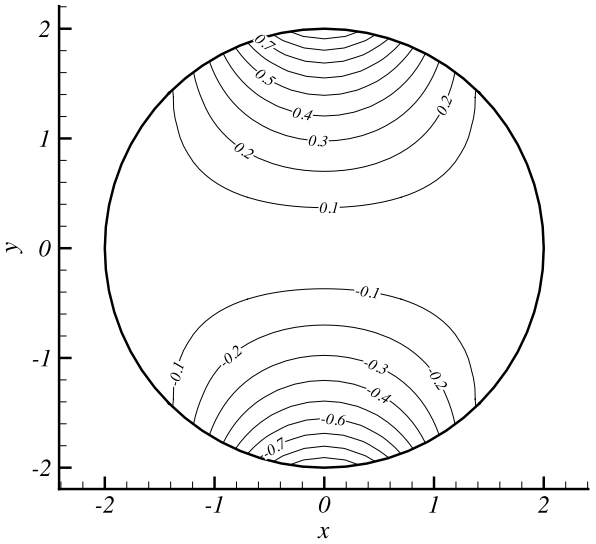

Fig. 8.16 Spectral element solution of temperatures on a disk

domain with a prescribed temperature around its perimeter. The model problem has an exact solution known as the Poisson Integral against which we can compare the spectral element solution. The steady temperature as a function of the polar coordinates (r, θ) on a disk of radius R that is kept at a temperature $\varphi(R, \theta) = F(\theta)$ along the outer boundary is

$$\varphi(r, \theta) = \frac{R^2 - r^2}{2\pi} \int_{-\pi}^{\pi} \frac{F(s)}{R^2 - 2rR\cos(s - \theta) + r^2} ds. \tag{8.73}$$

We solve for the temperature in the disk on the mesh shown in Fig. 8.15 with $N = 8$ that is heated and cooled along the outer edge according to

$$F(\theta) = e^{-4(\theta - \pi/2)^2} - e^{-4(\theta + \pi/2)^2}.$$

We show a contour plot of the computed solution in Fig. 8.16. We make a direct comparison to the analytical solution in Fig. 8.17, which shows the computed and exact solutions along the line $x = 0$.

8.4 The Discontinuous Galerkin Spectral Element Method

Finally, we derive the discontinuous Galerkin spectral element approximation of the system of conservation laws in two dimensions. We will see that the discontinuous approximation will give us simpler algorithms to implement than what we had in

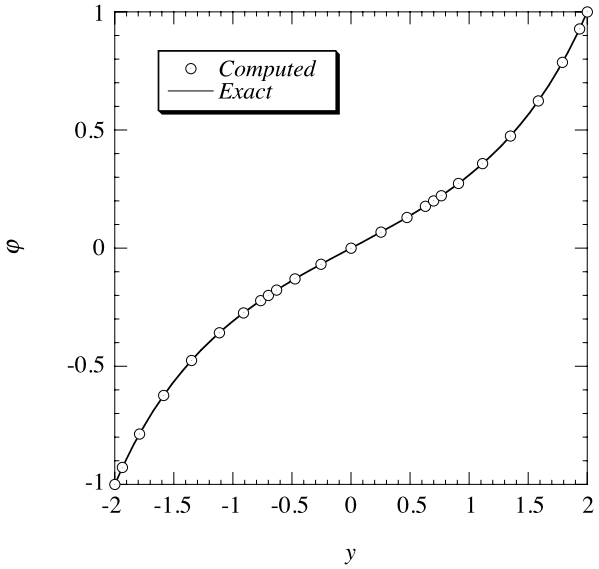

Fig. 8.17 Spectral element solution of temperatures on a disk. Cut along $x = 0$

the previous section, and that we have already developed all the machinery that we need to implement the method.

Our starting point is the two dimension conservation law (7.89), which we reproduce here,

$$\mathbf{q}_t + \mathbf{f}_x + \mathbf{g}_y = 0, \tag{8.74}$$

and which we convert to the weak form

$$\iint_{\Omega} \phi \left(\mathbf{q}_t + \mathbf{f}_x + \mathbf{g}_y\right) dxdy = 0. \tag{8.75}$$

We get the equation that the solution satisfies on an element when we break the integral into the sum of subintegrals over the elements

$$\sum_{k=1}^{K} \iint_{e^k} \phi \left(\mathbf{q}_t + \mathbf{f}_x + \mathbf{g}_y\right) dxdy = 0 \tag{8.76}$$

and examine each subintegral individually

$$\iint_{e^k} \phi \left(\mathbf{q}_t + \mathbf{f}_x + \mathbf{g}_y\right) dxdy = 0. \tag{8.77}$$

Since the element e^k is a quadrilateral, we have already derived the nodal discontinuous Galerkin approximation for (8.77) in Sect. 7.4, specifically, (7.99).

In the discontinuous Galerkin approximation, we couple the approximation to the boundary and, in the spectral element version, to neighboring elements through the boundary fluxes using the Riemann solver. If the boundary of the element is not on a physical boundary, then the external state is simply the solution on the neighboring element.

8.4.1 How to Implement the Discontinuous Galerkin Spectral Element Method

We see that the discontinuous Galerkin spectral element approximation is just the single domain mapped quadrilateral approximation applied to each element individually, plus coupling of the fluxes at the boundaries. This suggests that to implement the approximation we can reuse the algorithms of Sect. 7.4.2 for the element local operations. To compute the fluxes at the element boundaries, we will use the information stored in the mesh data structure to tell us which common nodes we need to use to compute the element boundary fluxes. As a spectral element method, we can arrange the data as we did in Sect. 8.3.1.

Our basic data structure will be the *DGSEMClass* that we present in Algorithm 136. It extends the single domain *MappedNodalDG2DClass* (Algorithm 111) to manage multiple domains. We assume a conforming mesh, as we did with the spectral element approximation, so there is still only one instance of the structure that stores the Gauss quadrature nodes, weights and the derivative matrices. Instead of a single mapping, each element has its own, so we replace the geometry object in the *MappedNodalDG2DClass* with the mesh object. Finally, we must now have an array of *DGSolutionStorage* structures to store the solution on each element. The structure of the new class is basically the same as the *SEMPotentialClass* of Algorithm 128, so the constructor for the class is the same as Algorithm 129 except for one change. We use the Gauss, not the Gauss-Lobatto, points for the

Algorithm 136: *DGSEMClass:* A Discontinuous Galerkin Class Definition

Class DGSEMClass
Uses Algorithms:
Algorithm 89 (NodalDG2DStorage)
Algorithm 110 (DGSolutionStorage)
Algorithm 126 (QuadMesh)
Data:
spA ; `// Of type NodalDG2DStorage`
mesh ; `// Of type QuadMesh`
$\{dGS\}_{k=1}^{K}$; `// Of type DGSolutionStorage`
Procedures:
Construct(*N*, *meshFile*); `// Algorithm 129, modified. See text.`
TimeDerivative(*t*); `// Algorithm 138`
End Class DGSEMClass

nodes in the discontinuous Galerkin approximation. Therefore, the constructor must use Algorithm 23 (LegendreGaussNodesAndWeights) to compute the nodes and weights.

The time derivative procedure for the spectral element approximation will follow Algorithm 115 (DG2DTimeDerivative) except that it must do its work on all of the elements. Since the boundary solutions need to be available before the boundary fluxes are computed, we interpolate all of the solutions to the faces first. We then compute the boundary fluxes for each of the edges. Finally, the local time derivatives need to be computed.

Before we present the time derivative procedure, however, we must develop a procedure to compute the boundary fluxes. Fortunately, all the information that we need is available in the *QuadMesh* structure. Recall that an edge stores the *id*'s of two neighboring elements and their associated sides. (See Algorithm 125 (EdgeClass).) The first one we call the primary element and the other the secondary. The edge also stores how the index of the secondary side varies with the index of the primary. Therefore, to compute the element boundary fluxes, we can loop through each of the edges, and for each, get the solutions and normal to feed to the Riemann solver to compute the flux. Once we compute the normal flux, we compute the contravariant flux, e.g., by (7.102). Algorithm 137 (EdgeFluxes) implements this procedure. Note that the outward normals for the two elements point in opposite directions. Therefore the edge flux must be negated to get the contravariant flux for the secondary side. If the edge is on a physical boundary, then the flux is computed just as it was for the single domain approximation.

Now that we have the edge flux procedure, we can implement the global time derivative procedure shown in Algorithm 138 (SEMGlobalTimeDerivative).

Unlike the continuous Galerkin spectral element method, we do not have to perform any operations on the corner nodes with the discontinuous version. Recall that our use of the Gauss, rather than Gauss-Lobatto points places the solution unknowns entirely within an element. (Cf. Fig. 5.7.) The boundary flux values are then located at the Gauss points along an edge. The corner nodes are not part of the approximation, thereby simplifying the implementation. It is for this simplification and for the increased quadrature precision that we have chosen the Gauss over the Gauss-Lobatto points for the location of the nodes.

To integrate in time, we will still use an explicit time integrator. The only difference from the single domain implementation is the need to loop over each element. See Sect. 8.1.4.

8.4.2 Benchmark Solution: Propagation of a Circular Wave in a Circular Domain

In Sect. 5.4.4 we computed the solution of the wave equation in a square domain with initial conditions that create an outward propagating circular sound wave. In this section, we will re-do the problem with the mesh shown in Fig. 8.15. For this mesh, we present solutions with $w = 0.15$.

Algorithm 137: *EdgeFluxes:* Compute the Riemann Problem Along Mesh Edges

Procedure EdgeFluxes
Input: t
Input: $edge$; `// Of type Edge`
Input: $\{elements\}_{k=1}^{K}$; `// Of type QuadElement`
Input: $\{dGS\}_{k=1}^{K}$; `// Of type DGSolutionStorage`
Uses Algorithms:
Algorithm 125 (EdgeClass)
Algorithm 110 (DGSolutionStorage)
Algorithm 124 (QuadElementClass)

if $edge.type = INTERIOR$ **then**
  $k \leftarrow edge.start - edge.inc$
  for $j = 0$ **to** N **do**
    $e1 \leftarrow edge.elementIDs_1$
    $s1 \leftarrow edge.elementSides_1$
    $e2 \leftarrow edge.elementIDs_2$
    $s2 \leftarrow |edge.elementSides_2|$
    $\{F_n\}_{n=1}^{nEqn} \leftarrow$
    $RiemannSolver(dGS_{e1}.\{Qb_{j,n,s1}\}_{n=1}^{nEqn}, dGS_{e2}.\{Qb_{k,n,s2}\}_{n=1}^{nEqn}, elements_{e1}.geom.\hat{n}_j^{s1})$
    for $n = 1$ **to** $nEqn$ **do**
      $dGS_{e1}.F^*_{j,n,s1} \leftarrow F_n * elements_{e1}.geom.scal_j^{s1}$
      $dGS_{e2}.F^*_{k,n,s2} \leftarrow -F_n * elements_{e2}.geom.scal_k^{s2}$
    end
    $k \leftarrow k + edge.inc$
  end
else
  $e1 \leftarrow edge.elementIDs_1$
  $s1 \leftarrow edge.elementSides_1$
  for $j = 0$ **to** N **do**
    $\left\{Q_n^{ext}\right\}_{n=1}^{nEqn} \leftarrow$
    $ExternalState(dGS_{e1}.\{Qb_{j,n,s1}\}_{n=1}^{nEqn}, elements_{e1}.geom.x_j^{s1}, elements_{e1}.geom.y_j^{s1}, t)$
    $dGS_{e1}.\left\{F^*_{j,n,s1}\right\}_{n=1}^{nEqn} \leftarrow elements_{e1}.geom.scal_j^{s1} *$
    $RiemannSolver\left(dGS_{e1}.\{Qb_{j,n,s1}\}_{n=1}^{nEqn}, \left\{Q_n^{ext}\right\}_{n=1}^{nEqn}, elements_{e1}.geom.\hat{n}_j^{s1}\right)$
  end
end
return $\{dGS\}_{k=1}^{K}$
End Procedure EdgeFluxes

We present solutions for the propagating circular wave at time $t = 1.25$ in Figs. 8.18 and 8.19. The solutions were computed with $N = 20$ and a time step of $\Delta t = 1 \times 10^{-3}$. To present the solutions in Fig. 8.18, we interpolated the solution in each element to 30 points in each direction using Algorithm 35 (2DCoarseToFineInterpolation). Figure 8.18 shows contours of the pressure, which illustrates that the circular shape of the wave is retained. Figure 8.19 shows the comparison of the exact and computed solutions along the line $y = 0$.

Algorithm 138: *DGSEMClass:TimeDerivative:* Compute the Time Derivative for the Discontinuous Galerkin Approximation

Procedure TimeDerivative
Input: t
Uses Algorithms:
Algorithm 112 (DG2DProlongToFaces)
Algorithm 137 (EdgeFluxes)
Algorithm 114 (MappedDGSystemTimeDerivative)

for $k = 1$ **to** $this.mesh.K$ **do**
  $this.dGS_k \leftarrow DG2DProlongToFaces(this.spA, this.elements_k.geom, this.dGS_k)$
end
for $i = 1$ **to** $this.mesh.N_{edge}$ **do**
  $this.\{dGS\}_{k=1}^{K} \leftarrow$
  $EdgeFluxes(t, this.mesh.edges_i, this.mesh.\{elements\}_{k=1}^{K}, this.\{dGS\}_{k=1}^{K})$
end
for $k = 1$ **to** $this.mesh.K$ **do**
  $this.dGS_k \leftarrow$
  $MappedDG2DTimeDerivative(this.spA, this.elements_k.geom, this.dGS_k)$
end
End Procedure TimeDerivative

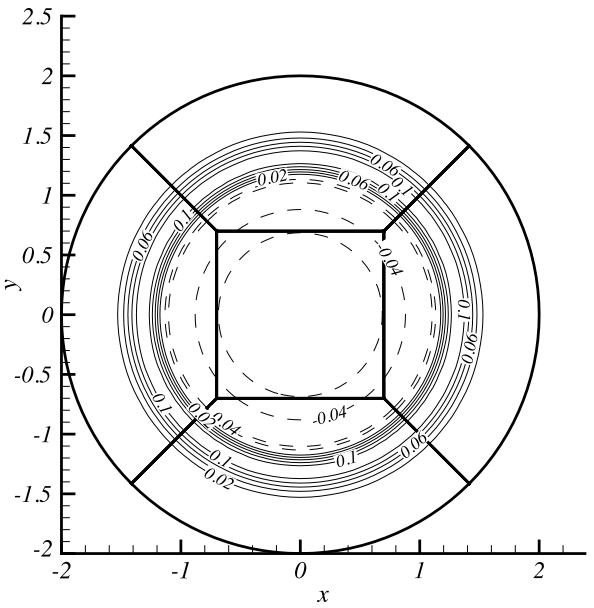

Fig. 8.18 Computed pressure contours at time $t = 1.25$ for a propagating circular wave when $N = 20$ and $\Delta t = 1 \times 10^{-3}$. The solutions were interpolated to 30 uniformly spaced points in each direction on each element. Heavy lines show the element boundaries

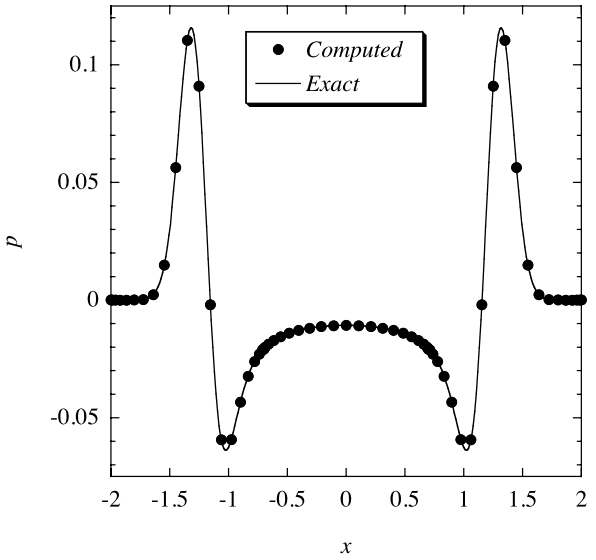

Fig. 8.19 Comparison of the computed circular wave pressure with the exact solution along the line $y = 0$ at $t = 1.25$

8.4.3 Benchmark Solution: Transmission and Reflection from a Material Interface

In the introduction to this chapter, we listed four reasons why we might want or need to use a spectral element approximation instead of a single domain method. In the previous benchmark, the reason was to avoid the coordinate singularity that a cylindrical coordinate mesh would produce. In this benchmark, we have two reasons for using a spectral element method. The first is that we can use the method when there are singularities in the coefficients. The second is to increase efficiency by using smaller elements of lower order. With this benchmark solution we explore a unique feature of the discontinuous Galerkin spectral element method, namely its ability to approximate discontinuous solutions at element interfaces accurately.

When a wave propagates across an interface where the wave speed abruptly changes, part of the wave is transmitted and part of it is reflected. The phenomenon is familiar in daily life. In electrodynamics, we study the reflection and transmission of electromagnetic waves at dielectric interfaces. In ultrasound tests, ultrasonic waves reflect as they propagate through different tissues.

To extend the wave propagation model that we have used so far to include propagation through multiple materials, we now allow the density of the material, ρ, and the wave speed, c, to vary as a function of location. With these changes, the

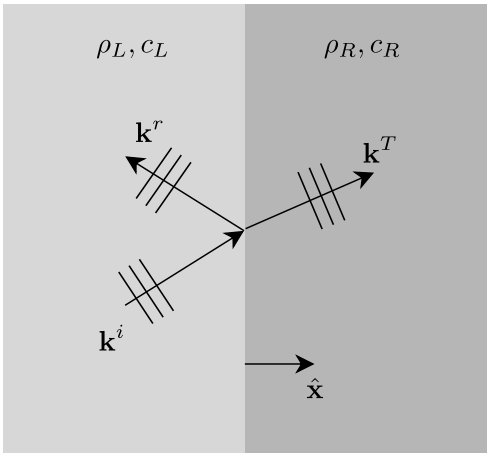

Fig. 8.20 Model for plane wave reflection at a material interface

conservation law form of the wave equation is

$$\begin{bmatrix} p \\ u \\ v \end{bmatrix}_t + \begin{bmatrix} \rho c^2 u \\ p/\rho \\ 0 \end{bmatrix}_x + \begin{bmatrix} \rho c^2 v \\ 0 \\ p/\rho \end{bmatrix}_y = 0. \tag{8.78}$$

As our benchmark problem, we solve for the transmission and reflection of a plane wave at a plane interface between two materials that have uniform properties within each, as we show in Fig. 8.20.

The problem has an analytic solution against which we can compare our computed solutions. It is the kind of problem that is solved in electrodynamics texts for scattering at an interface between two dielectrics. Let $\psi(\xi)$ be a waveform with maximum value of one, and a be an amplitude. Then each of the incident, reflected and transmitted plane waves is of the form

$$\mathbf{q} = a\psi\left(\mathbf{k}\cdot\mathbf{x} - \omega\left(t - t_0\right)\right)\begin{bmatrix} 1 \\ \frac{k_x}{\rho c} \\ \frac{k_y}{\rho c} \end{bmatrix}. \tag{8.79}$$

To define a particular wave, we simply replace $\mathbf{k}$ by the appropriate wavevector and a by the appropriate amplitude. The wavevectors and amplitudes of the reflected and transmitted waves depend on the incident wave and must satisfy the correct jump and phase matching conditions at the interface. Let us define the incident wavevector to be

$$\mathbf{k}^i = \frac{\omega}{c_L}\left(k_x^i \hat{x} + k_y^i \hat{y}\right), \tag{8.80}$$

where $(k_x^i)^2 + (k_y^i)^2 = 1$. Then the reflected and transmitted wavevectors are

$$\begin{aligned} \mathbf{k}^r &= \frac{\omega}{c_L}\left(-k_x^i \hat{x} + k_y^i \hat{y}\right), \\ \mathbf{k}^T &= \frac{\omega}{c_R}\left[\sqrt{1 - \left(\frac{c_R}{c_L}\right)^2 \left(k_y^i\right)^2}\,\hat{x} + \frac{c_R}{c_L} k_y^i \hat{y}\right]. \end{aligned} \tag{8.81}$$

The corresponding amplitudes are

$$\begin{aligned} \frac{a^r}{a^i} &= \frac{1}{J}\left(\rho_R c_R k_x^T / k^T - \rho_L c_L k_x^i / k^i\right), \\ \frac{a^T}{a^i} &= \frac{1}{J}\left(\rho_L c_L k_x^r / k^r - \rho_R c_L k_x^i / k^i\right), \end{aligned} \tag{8.82}$$

where

$$J = -\rho_R c_R k_x^T / k^T + \rho_L c_L k_x^r / k^r. \tag{8.83}$$

To use the discontinuous Galerkin approximation, we must define the flux functions and derive a Riemann solver. We get the flux functions from (8.78). To derive the Riemann solver, remember that it computes the numerical flux $\mathbf{F}^*(\mathbf{q}^L, \mathbf{q}^R; \hat{n})$ given two possibly different states, $\mathbf{q}^L$ and $\mathbf{q}^R$, where left and right are defined according to the normal direction $\hat{n} = \alpha\hat{x} + \beta\hat{y}$. To start, we construct the coefficient matrix for the system

$$A = \begin{bmatrix} 0 & \alpha\rho c^2 & \beta\rho c^2 \\ \alpha/\rho & 0 & 0 \\ \beta/\rho & 0 & 0 \end{bmatrix}. \tag{8.84}$$

The eigenvalues of this system remain the same as before, $\lambda = \pm c, 0$. What makes the problem interesting now is that the characteristic variables have a jump discontinuity when the wave speeds and the density jump across an interface. Instead of requiring the characteristic variables to be continuous at an interface, we apply the *Rankine-Hugoniot* condition, which says that the normal flux must be continuous. For the wave equation, this means

$$A_L \mathbf{q}_L - A_R \mathbf{q}_R = 0, \tag{8.85}$$

since A depends on both the density and the wave speed. Nevertheless, waves that propagate to the right are evaluated from $\mathbf{q}^L$ and waves that propagate to the left are evaluated from $\mathbf{q}^R$, just as before. Under those constraints, a fair amount of algebra shows that

$$\mathbf{F}^*\left(\mathbf{q}_L, \mathbf{q}_R; \hat{n}\right) = \begin{bmatrix} z_R[c(p + \rho c(nu + n_y v))]_L - z_L[c(p - \rho c(n_x u + n_y v))]_R \\ n_x\{\frac{z_L}{\rho_L}[p + \rho c(n_x u + n_y v)]_L + \frac{z_R}{\rho_R}[p - \rho c(n_x u + n_y v)]_R\} \\ n_y\{\frac{z_L}{\rho_L}[p + \rho c(n_x u + n_y v)]_L + \frac{z_R}{\rho_R}[p - \rho c(n_x u + n_y v)]_R\} \end{bmatrix}, \tag{8.86}$$

where

$$z_L = \frac{\rho_L c_L}{\rho_L c_L + \rho_R c_R}, \qquad z_R = \frac{\rho_R c_R}{\rho_L c_L + \rho_R c_R}. \tag{8.87}$$

Note that when the densities and the wave speeds are the same on both sides, (8.86) reduces to (5.164). If, in addition, the solution is the same on both sides, it reduces to the normal flux $A\mathbf{q}$.

Since the discontinuous Galerkin spectral element approximation allows discontinuities in the solution at element boundaries, it is a natural choice for problems with material discontinuities, as long as we place element boundaries along material boundaries. The only modifications that we need to add beyond the new flux functions to replace the procedures in Algorithm 94 (WaveEquationFluxes) and the new Riemann solver to replace Algorithm 88 (RiemannSolver), is to have the element class store the element's material properties, ρ and c. When the Riemann solver is called in Algorithm 137 (EdgeFluxes), it will be passed the material values from the left and the right elements.

For the benchmark solution, we compute the reflection and transmission of a plane wave through a vertical material interface, as pictured in Fig. 8.20. We take the domain to be the square $[-5, 5] \times [-5, 5]$ with the material interface along $x = 0$. We subdivide the domain into a structured mesh of 20 elements in each direction so that each element has a length and width equal to 0.5. We present solutions for $N = 10$ in each element. For plotting they are interpolated to 12 uniformly spaced points in each direction in each element. We integrate to $t = 3.0$ in time with $\Delta t = 0.05$.

We model the incident wave as an approximation of a typical ultrasound pulse,

$$\psi(t) = \sin(\omega t)\, e^{-t^2/(\omega\sigma)^2}, \tag{8.88}$$

where $\omega = 2\pi f$ and f is the frequency. For the envelope, $\sigma^2 = -(MT)^2/(4\ln(10^{-4}))$ where M is the number of modes in the significant part of the envelope and $T = 1/f$ is the period. We present the specific parameters in Table 8.4. With these parameters and the mesh, we resolve the sine waves with an average of about seven points per wavelength. The external state and the initial condition are set using the exact solution.

We show contours of the computed and exact values of p in Fig. 8.21. Clearly visible are the incident wave, above on the left, the reflected wave below on the left, and transmitted wave on the right. Notice that p itself is discontinuous at the interface between the two materials as is allowed by the discontinuous Galerkin method. We show p as a function of y in Figs. 8.22 and 8.23. We chose the locations to be at the nearest Gauss points to $x = -1$ and $x = 0.5$.

Table 8.4 Parameters for plane wave reflection problem

Parameter	M	f	k_x^i	k_y^i	ρ_L	ρ_R	c_L	c_R	t_0
Value	4	2.5	0.5	$\sqrt{3}/2$	1	0.4	1	0	3

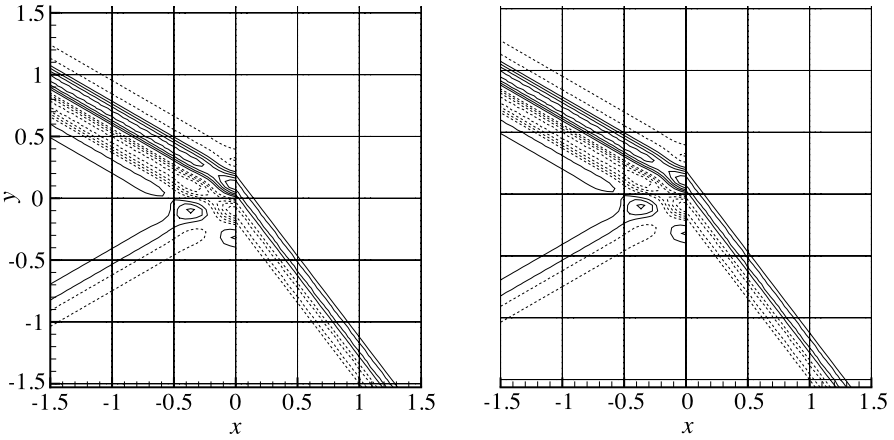

Fig. 8.21 Comparison of computed (*left*) and exact (*right*) pressure contours at $t = 3$ for plane wave reflection at a material interface. *Dashed lines* are negative contours. The contour levels range from -0.8 to 0.8 with a step of 0.2. Note the discontinuity in the pressure at the material discontinuity along $x = 0$. The overlay of squares shows the locations of the element boundaries

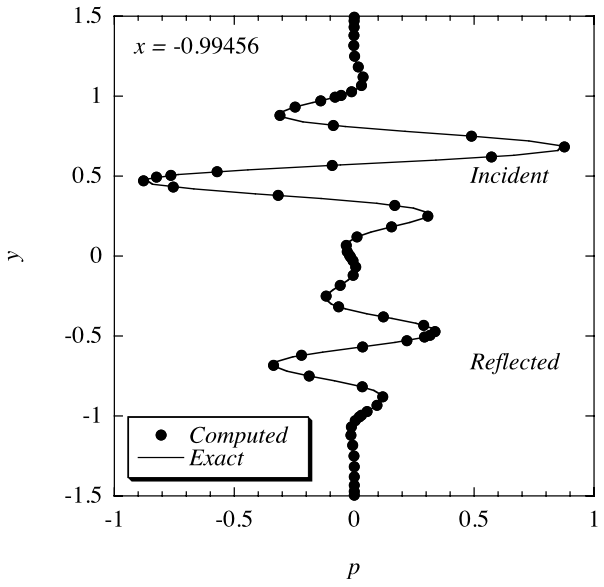

Fig. 8.22 Comparison of the computed and exact pressures along a vertical line to the left of the material interface

Exercises

8.1 Show that if N and Δx are constants, then the time derivative, (8.22), is the simple average of the approximations from either side.

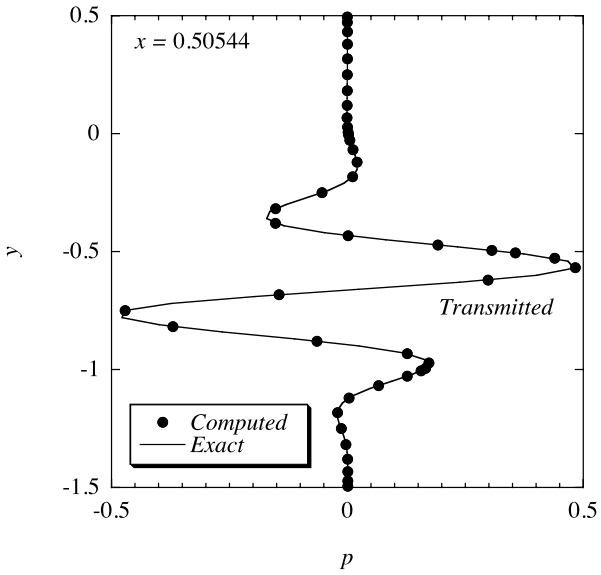

Fig. 8.23 Comparison of the computed and exact pressures along a vertical line to the right of the material interface

8.2 Derive the one dimensional spectral element method for the variable coefficient problem,

$$\varphi_t = (\nu(x)\varphi_x)_x \,.$$

8.3 Derive the spectral element approximation to the advection-diffusion equation.

$$\varphi_t + a\varphi_x = \nu\varphi_{xx}.$$

If $\nu = 0$ and N and Δx are constant, show that the time derivative at an element interface point is the simple average of the spatial derivatives from either side. Compare the approximation of the advection term to the discontinuous Galerkin approximation for the same equation.

8.4 Derive the spectral element approximation to the equation

$$\varphi_{xx} = s$$

with Dirichlet boundary conditions. Develop the algorithms that you need to solve the equations iteratively.

8.5 Derive the numerical flux, (8.49).

8.6 In general one would want to impose different boundary conditions along different boundaries of a physical problem. In Sect. 5.2.1, we did this by defining an

array that specified whether or not to mask a particular boundary. Apply the mask idea to the spectral element approximation by assigning to each edge in the mesh a mask variable that is set from information in the mesh file. Show how to modify Algorithms 130–134 to incorporate both Dirichlet boundary conditions and radiation conditions of the form $\nabla\varphi \cdot \hat{n} = \gamma\varphi$.

8.7 Design algorithms to integrate the spectral element approximation to the diffusion equation, (8.71) using the trapezoidal rule for the time integrator. Implement and test your algorithms and solve the problem with solution

$$\varphi(x, y, t) = \frac{1}{4t+1} e^{-\frac{((x-x_0)^2+(y-y_0)^2)}{4t+1}}$$

on the disk.

8.8 The steady incompressible viscous flow in a circular pipe is known as *Poiseuille flow* and is a special case of the flows computed in Problem 7.12. If the pipe has radius R, then the axial velocity of the Poiseuille flow is given by

$$u(r) = -\frac{\gamma}{4}\left(R^2 - r^2\right).$$

1. Use the spectral element method to compute the Poiseuille flow and compare the computed solutions to the exact for $\gamma = -1$.
2. Compute to spectral accuracy the *volume flow rate* Q defined by

$$Q = \int_{disk} u dA$$

and compare to the exact analytical value

$$Q_{pipe} = -\frac{\pi R^4}{8}\gamma.$$

8.9 Do Problem 7.13 with a spectral element mesh.

8.10 Develop and implement the algorithms for the spectral element approximation to the advection-diffusion equation, (8.72) with the semi-implicit time integration of Sect. 5.3.3. Solve the benchmark problem of Sect. 7.3.6 and compare the computation time between single and multidomain approximations for a given accuracy.

8.11 Modify Algorithm 137 (EdgeFluxes) to allow it to apply different boundary conditions to different edges. (Cf. Problem 8.6.)

8.12 In Sect. 5.4.3 we saw that single domain wave propagation requires large order polynomials to resolve the two main spatial scales, namely the size of the domain

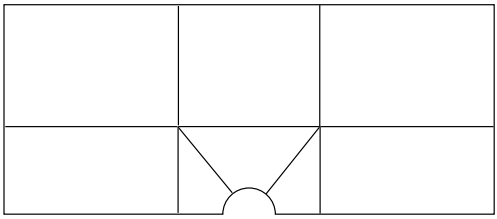

Fig. 8.24 A Mesh topology for scattering of an acoustic wave off a circular cylinder

and the length scale of the propagating wave. A spectral element approximation enables us to resolve both scales by subdividing the square into a mesh of smaller square elements and keep the cost down by using lower order polynomials on the elements. Redo the solution of the wave equation for both the planewave and cylindrical wave problems of Sect. 5.4.3 with multiple square elements. Compare the cost to compute the solutions to a desired accuracy for several subdivisions of the square.

8.13 In Sect. 7.4.3 we computed the scattering of an acoustic wave off a cylinder. As we discussed in Problem 8.12, the differences in scales required us to compute the solution with very high order polynomials, and with correspondingly small time steps required by the explicit time differencing. A more efficient approach is to use a spectral element approximation. Compute the scattering problem with a mesh topology like that shown in Fig. 8.24 and compare the cost to the single domain computation.

Appendix A
Pseudocode Conventions

"How to play the flute. (picking up a flute) Well here we are.
You blow there and you move your fingers up and down here."
in "How to do it", Monty Python's Flying Circus, Episode 28.

We use a pseudocode in this book to show how to implement spectral methods. Pseudocode is a commonly used device to present algorithms. It represents an informal high level description of what one would program with a computer language. Pseudocodes omit details like variable declarations, memory allocations, and computer language specific syntax. Too high a level, however, and we risk missing important details. The goal of pseudocode is to give enough cues to allow the reader to write a working computer program, no matter what programming language will be ultimately used to implement it.

We use the LaTex macro "Algorithm2e" written by Christophe Fiorio to typeset our pseudocode. The macro provides commonly used keywords and ways to represent flow control such as conditional statements and loops. Comments, however, look like C/C++ statements. Since comments are typeset with a different font, it should be pretty clear what is a comment and what is not. Beyond these basic and common statements, we need to decide how to express both high level and low level concepts.

To help to read the almost 150 algorithms that we present, we outline some of the conventions that we use. On the lower level, these conventions include how we represent variables, arithmetic operations, and arrays. On the higher level, they include an object oriented philosophy to organize data and procedures.

Variables and Arithmetic Operations We use pseudocode to provide a bridge between the mathematics and a computer program. To make that bridge, we try to make the statements look as closely as possible to the equations that they are trying to implement. Therefore, if we compute something that has a well-known mathematical notation, such as the Chebyshev polynomial of degree n, we write it that way in the pseudocode, T_n. If the quantity does not have a common name, we make up a variable name for it. We denote constants by all uppercase names, e.g. NONE, with underscores to separate words.

As much as possible we write equations in the pseudocode just as we write them mathematically in the text. Sometimes, however, we represent multiplication by " * " when leaving it out can cause confusion.

Arrays The most common data structure that we use in the spectral methods algorithms is the array. Mathematical vectors and grid values in one space dimension can be stored as singly dimensioned arrays. Full matrices and grid values in two space dimensions can be stored as doubly dimensioned arrays. In this book, we represent

D.A. Kopriva, *Implementing Spectral Methods for Partial Differential Equations,*
Scientific Computation,

single arrays by $\{u_j\}_{j=s}^{e}$ and double arrays by $\{u_{i,j}\}_{i,j=s}^{e}$, where s and e represent the start and end indices. When the indices may mean something different, say if we want to think of an array of two dimensional arrays, we will separate indices with a semicolon. In any case, since different computer languages have different levels of support for arrays, we do not imply a particular data model by our arrays except in one circumstance. That circumstance is when we use iterative methods to solve linear systems, where we assume that arrays are contiguous in memory and therefore all representations are equivalent to a singly dimensioned array of the appropriate length.

Functions and Other Procedures With most computer languages it is difficult to tell which variables are input, which are output, and which are both. To make these clear, we write procedure calls like

$$result \leftarrow function(input1, input2).$$

If an argument is both an input and output variable, it appears both as a result and as an input

$$result \leftarrow function(input1, input2, result).$$

Since we work with tensor product spectral approximations in this book, most of the operations in two dimensions that work on doubly dimensioned arrays reduce to performing multiple operations on singly dimensioned arrays. We denote a one dimensional slice of a two dimensional array, say, $\{U_{i,j}\}_{i,j=0}^{N,M}$, by $\{U_{i,j}\}_{i=0}^{N}$ for slices along columns and $\{U_{i,j}\}_{j=0}^{M}$ along rows. In general, we assume that if an array with particular extents is passed into a procedure, then those extents are too.

Pointers We express setting a pointer to point to a record or another pointer by the notation "$\Rightarrow$". For simplicity, we assume that dereferencing a pointer, i.e., getting the record to which a pointer points, is automatic. That is, if a pointer p points to a record in memory that contains a structure *data*, then we reference that data by $p.data$. This is how pointers work in Fortran (with % in place of ".") and how references work in C++, but not how pointers work in C/C++, where one would use $p \rightarrow data$.

Object Oriented Algorithms We take an object oriented view of data and procedure organization. This doesn't mean that one has to use an object oriented language to implement these algorithms. As with arrays, different computer languages have different levels of support for automatically programming this way, so we use object orientation here to allow us to group variables, simplify procedure arguments, and reuse procedures that we have already developed. The fundamental construct is the *class*, which gathers data in the form of an abstract data type (or structure) and procedures that work on that data. Member variables and procedures are accessed in the common, though not universal, dot notation. Therefore if *obj* is an instance of a given class, a is a member variable, and $f(x)$ is a member procedure, then

we access a by $obj.a$ and invoke f by $obj.f(x)$. For procedure calls, the implicit assumption is that the object is passed as an argument to the procedure, whether this is done automatically within the computer language, or explicitly in the argument list as necessary. Thus, $obj.f(x)$ by itself means

$$obj \leftarrow f(obj, x).$$

Within the procedure, the object is named *this*, again common but not universal. We use the keyword **Extends** if we want to add data to, or replace procedures in, a class. In a sense, this represents subclassing that is present in fully object oriented languages.

Appendix B
Floating Point Arithmetic

Computers use floating point numbers, which behave differently than real numbers. Discussions of floating point arithmetic in general [16] and the IEEE implementation [18] used on most computers today can be found in the references. We are most interested in one number: ϵ, which represents the relative error due to rounding. Several computer languages now have this number as an available parameter. For instance, in Fortran, it is given by the function EPSILON( ). In C/C++ it is defined in the float.h header file as FLT_EPSILON, with a similar definition for doubles.

It is well-known that we should never do direct comparisons of equality for floating point numbers ([16], Vol. II, Chap. 4). On the other hand, it is not that obvious how to write a robust algorithm to test when two floating point numbers are "close enough" to be considered to be equal. The basic (strong) test for two floating point numbers, a and b, to be "essentially equal to" each other is that

$$|b-a| \leq \epsilon\,|a| \quad \text{and} \quad |b-a| \leq \epsilon\,|b|\,. \tag{B.1}$$

It is possible, however, for the products on the right to overflow or underflow. We can avoid those situations by scaling the numbers, or explicitly handling the overflow situations.

To test the equality of two floating point numbers in the algorithms developed for this book, such as is necessary for instance in Algorithm 31 (LagrangeInterpolation),

Algorithm 139: *AlmostEqual:* Testing Equality of Two Floating Point Numbers

```
Procedure AlmostEqual
Input: a, b
if a = 0 or b = 0 then
    if |a − b| ≤ 2ϵ then
        AlmostEqual ← TRUE
    else
        AlmostEqual ← FALSE
    end
else
    if |a − b| ≤ ϵ|a| and |a − b| ≤ ϵ|b| then
        AlmostEqual ← TRUE
    else
        AlmostEqual ← FALSE
    end
end
return AlmostEqual
End Procedure AlmostEqual
```

D.A. Kopriva, *Implementing Spectral Methods for Partial Differential Equations,* Scientific Computation,

we propose Algorithm 139 (AlmostEqual). Note that we do not have to worry about all exceptional cases here, particularly since the numbers that we work with are in $[-1, 1]$ or $[0, 2\pi]$. Thus, the only exceptional cases we need to deal with are near the origin.

Appendix C
Basic Linear Algebra Subroutines (BLAS)

The BLAS provide standard building blocks to perform basic vector and matrix operations. There are three levels of BLAS, with each level performing more and more complex operations. The first level, BLAS Level 1, is a collection of routines that compute common operations such as the Euclidean norm, the dot product, and vector operations such as $y = \alpha x + y$, known as AXPY. A PDF reference of the available routines, blasqr.pdf, can be found at http://www.netlib.org/blas/.

BLAS libraries are freely available on the web, for example at www.netlib.org, and specifically optimized versions are often included with commercial compilers. Furthermore, the ATLAS (Automatically Tuned Linear Algebra Software) library found at http://math-atlas.sourceforge.net can be compiled to create a portably efficient BLAS library.

The BLAS routines are named with the format xNAME, where "x" denotes the precision of the arithmetic, either "D" for double precision or "S" for single. (For many routines, complex or complex*16 versions are available with the prefixes "C" and "Z", respectively.) For example, the single precision dot product is named SDOT.

BLAS routines that are of use in this book include DOT, for the Euclidean inner product, NRM2, for the Euclidean norm, AXPY, for scalar times vector plus vector, COPY and SCAL. The standard calling arguments are

```
_DOT(N,X,INCX,Y,INCY)
_NRM2(N,X,INCX)
_AXPY(N,ALPHA,X,INCX,Y,INCY)
_COPY(N,X,INCX,Y,INCY)
_SCAL(N,ALPHA,X,INCX)
```

In each case, N corresponds to the total number of elements. The arguments X and Y correspond to the input/output arrays. The integers *INCX* and *INCY* indicate the stride of the data, and enable the computation of subarray operations. The argument ALPHA corresponds to the scalar parameter.

All arrays used in BLAS routines are *constrained to be contiguous*, a constraint that should be observed when using languages that define arrays as arrays of pointers. As such, the BLAS routines, now called the dense versions, don't distinguish between inputs that represent one or two-dimensional arrays.

For completeness, and by way of example, we present prototype algorithms to compute the dot product, the Euclidean norm, $y = \alpha x + y$, copy, and scale by a parameter. In practice, one should use optimized library versions. Consistent with the dense BLAS philosophy, we constrain the input arrays to be contiguous, so it does not matter whether arguments represent single or multidimensional arrays.

D.A. Kopriva, *Implementing Spectral Methods for Partial Differential Equations,* Scientific Computation,

Algorithm 140: *BLAS_Level1:* A Selection of Basic Linear Algebra Subroutines

Procedure BLAS_NRM2
Input: $N, \{x_j\}_{j=1}^{N*INCX}$, INCX

$z \leftarrow 0; i \leftarrow 1$
for $j = 1$ **to** N **do**
  $z \leftarrow z + x_i^2$
  $i \leftarrow i + INCX$
end
return $\sqrt{z}$
Procedure BLAS_NRM2

Procedure BLAS_DOT
Input: $N, \{x_j\}_{j=1}^{N*INCX}$, INCX, $\{y_j\}_{j=1}^{N*INCY}$, INCY

$z \leftarrow 0; i \leftarrow 1; j \leftarrow 1$
for $k = 1$ **to** N **do**
  $z \leftarrow z + x_i * y_j$
  $i \leftarrow i + INCX$
  $j \leftarrow j + INCY$
end
return z
Procedure BLAS_DOT

Procedure BLAS_AXPY
Input: $N, \alpha, \{x_j\}_{j=1}^{N*INCX}$, INCX, $\{y_j\}_{j=1}^{N*INCY}$, INCY

$i \leftarrow 1; j \leftarrow 1$
for $k = 1$ **to** N **do**
  $y_j \leftarrow y_j + \alpha * x_i$
  $i \leftarrow i + INCX$
  $j \leftarrow j + INCY$
end
return $\{y_j\}_{j=1}^{N}$
End Procedure BLAS_AXPY

Procedure BLAS_COPY
Input: $N, \{x_j\}_{j=1}^{N*INCX}$, INCX, $\{y_j\}_{j=1}^{N*INCY}$, INCY

$i \leftarrow 1; j \leftarrow 1$
for $k = 1$ **to** N **do**
  $y_j \leftarrow x_i$
  $i \leftarrow i + INCX$
  $j \leftarrow j + INCY$
end
return $\{y_j\}_{j=1}^{N}$
End Procedure BLAS_COPY

Procedure BLAS_SCAL
Input: $N, \alpha, \{x_j\}_{j=1}^{N*INCX}$, INCX

$i \leftarrow 1$
for $k = 1$ **to** N **do**
  $x_i \leftarrow \alpha x_i$
  $i \leftarrow i + INCX$
end
return $\{x_i\}_{i=1}^{N}$
End Procedure BLAS_SCAL

Appendix D
Linear Solvers

Approximations of potential problems and implicit discretizations of time dependent partial differential equations lead to linear systems of equations to solve. We solve the systems either directly, typically by some variant of Gauss elimination, or iteratively. In this appendix, we motivate the algorithms that we use to solve the systems that appear in this book. In no way can a short appendix like this survey the entire field of numerical linear algebra and all the issues related to efficiency, parallelism, etc. For further study, we suggest the books [13] or [19].

D.1 Direct Solvers

For small enough systems, or in special cases, direct linear system solvers are efficient. In this section, we discuss two, namely the Thomas algorithm to solve tri-diagonal systems, and LU factorization to solve full, general systems. In both cases, well-tested and portable code with multiple language bindings is available in the LAPACK [2] library. Some compiler vendors supply precompiled versions of LAPACK, which should be used if possible. Otherwise, it is possible to download the source code from www.netlib.org (see, in particular, http://www.netlib.org/lapack/index.html) and compile it oneself.

D.1.1 Tri-Diagonal Solver

Tri-diagonal matrix problems are ubiquitous in numerical analysis, and appear, for example in Sect. 4.5 with the Legendre Galerkin approximation. For completeness, therefore, we include the Thomas algorithm for the solution of tri-diagonal systems. We represent the elements of the matrix by the three vectors ℓ, d and u, for the subdiagonal, diagonal and superdiagonal elements, numbered as

$$\begin{bmatrix} d_0 & u_0 & 0 & \dots & 0 \\ \ell_1 & d_1 & u_1 & \ddots & \vdots \\ 0 & \ell_2 & d_2 & \ddots & 0 \\ \vdots & \ddots & \ddots & & u_{N-1} \\ 0 & \dots & 0 & \ell_N & d_N \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \\ x_N \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \\ y_N \end{bmatrix}. \tag{D.1}$$

D.A. Kopriva, *Implementing Spectral Methods for Partial Differential Equations,* Scientific Computation,

Algorithm 141: *TriDiagonalSolve:*

Procedure TriDiagonalSolve
Input: $\{\ell_j\}_{j=1}^{N}, \{d_j\}_{j=0}^{N}, \{u_j\}_{j=0}^{N-1}, \{y_j\}_{j=0}^{N}$
for $j = 0$ **to** N **do**
  $\hat{d}_j = d_j$
end
// Forward Elimination
for $j = 1$ **to** N **do**
  $\hat{d}_j = \hat{d}_j - \ell_j / \hat{d}_{j-1} * u_{j-1}$
  $y_j = y_j - \ell_j / \hat{d}_{j-1} * y_{j-1}$
end
// Backward Substitution
$x_N = y_N / \hat{d}_N$
for $j = N - 1$ **to** 0 **step** -1 **do**
  $x_j = (y_j - u_j * x_{j+1}) / \hat{d}_j$
end
return $\{x_j\}_{j=0}^{N}$
End Procedure TriDiagonalSolve

The Thomas algorithm is a variant of Gauss elimination, and a description of it can be found in any elementary numerical analysis book. It has two phases, the forward elimination and the backward substitution.

The first phase of the algorithm is the forward elimination, where we remove the subdiagonal entries. In the second row, ℓ_1 is eliminated when we multiply the first row by $-\ell_1/d_0$ and add the result to the second. The diagonal entry on the second element of the modified row and the right hand side become

$$\begin{aligned} \hat{d}_1 &= d_1 - u_0 \ell_1 / d_0, \\ \hat{y}_1 &= y_1 - y_0 \ell_1 / d_0. \end{aligned} \tag{D.2}$$

We eliminate the ℓ_2 entry in the third row and all succeeding values of ℓ_j by repeating the procedure for $j = 2, \ldots, N$. When complete, the matrix is upper bi-diagonal, so that $x_N = \hat{y}_N / \hat{d}_N$ and for $j < N$,

$$x_j = \left(\hat{y}_j - u_j x_{j+1}\right) / \hat{d}_j. \tag{D.3}$$

Since the right hand side vector is rarely needed again in practice, it is usually safe just to overwrite the original array $\mathbf{y}$ with $\hat{\mathbf{y}}$. Algorithm 141 (TriDiagonalSolve) shows the whole procedure.

D.1.2 LU Factorization

An LU factorization with partial row pivoting computes a lower triangular, L, upper triangular, U, and permutation, P, matrix to rewrite a square matrix A in the form

$A = PLU$. The permutation matrix is there to swap rows to ensure that the diagonal contains the largest element in a column. The attraction of the algorithm is that once we compute and store the factorization, we solve the system $A\mathbf{x} = \mathbf{y}$ efficiently for multiple right hand sides, $\mathbf{y}$, by a triangular forward substitution followed by a triangular backward substitution.

To motivate the algorithm, let's assume that row swapping (pivoting) is not needed. Then $A = LU$ and we write the matrix multiplication component-wise as

$$a_{ij} = \sum_{n=1}^{N} \ell_{in} u_{nj} = \sum_{n=1}^{\min(i,j)} \ell_{in} u_{nj}, \tag{D.4}$$

since L is lower triangular and U is upper triangular. If we choose $\ell_{kk} = 1$ (giving us what is known as the Doolittle Method), we can write

$$\begin{aligned} a_{kj} &= \sum_{n=1}^{k-1} \ell_{kn} u_{nj} + u_{kj}, \quad j = k, \ldots, N, \\ a_{ik} &= \sum_{n=1}^{k-1} \ell_{in} u_{nk} + \ell_{ik} u_{kk}, \quad i = k+1, \ldots, N. \end{aligned} \tag{D.5}$$

We rearrange these to solve for the unknowns

$$\begin{aligned} u_{kj} &= a_{kj} - \sum_{n=1}^{k-1} \ell_{kn} u_{nj}, \quad j = k, \ldots, N, \\ \ell_{ik} &= \frac{1}{u_{kk}} \left(a_{ik} - \sum_{n=1}^{k-1} \ell_{in} u_{nk} \right), \quad i = k+1, \ldots, N. \end{aligned} \tag{D.6}$$

Typically, one destroys the original matrix by writing U to the upper triangular part of the A, and L to the lower. Since we chose the diagonal of L to be one, it doesn't need to be stored, which allows the diagonal part of U to be stored there instead.

In general, we must swap rows to move the largest element in a row to the diagonal. The information can be stored in a single pivot vector, $\{p_j\}_{j=1}^{N}$, that simply tells which row must be swapped with the current row. The procedure *Factorize* in Algorithm 142 (LUFactorization) prototypes how to decompose a matrix into its $A = PLU$ factorization for columnwise storage of the matrix A. Note that this procedure, like most library factorization procedures, destroys the original matrix to save storage. An easy mistake to make is to forget and try to use the matrix again after it has been factorized.

We break the solve operation into two steps. Since $A = PLU$,

$$PLU\mathbf{x} = \mathbf{y} \tag{D.7}$$

or

$$LU\mathbf{x} = P\mathbf{y} \tag{D.8}$$

Algorithm 142: *LUFactorization:* Factorization and Solve Procedures to Solve $A\mathbf{x} = \mathbf{y}$

```
Procedure Factorize
Input: {A_{i,j}}_{i,j=1}^{N}

for k = 1 to N do
    p(k) ← k
    for i = k + 1 to N do
        if |A_{i,k}| > |A_{p_k,k}| then p_k ← i
    end
    if p_k ≠ k then
        for j = 1 to N do
            t ← A_{k,j}; A_{k,j} ← A_{p_k,j}; A_{p_k,j} ← t
        end
    end
end
for j = k to N do
    s ← 0
    for n = 1 to N do
        s ← s + A_{k,n} * A_{n,j}
    end
    A_{k,j} ← A_{k,j} − s
end
for i = k + 1 to N do
    s ← 0
    for n = 1 to k − 1 do
        s ← s + A_{i,n} * A_{n,k}
    end
    A_{i,k} ← (A_{i,k} − s)/A_{k,k}
end
return {A_{i,j}}_{i,j=1}^{N}, {p_j}_{j=1}^{N}
End Procedure Factorize
```

```
Procedure LUSolve
Input: {A_{i,j}}_{i,j=1}^{N}, {p_j}_{j=1}^{N}, {y_{j,m}}_{j,m=1}^{N,N_RHS}

for i = 1 to N do
    if p_i ≠ i then
        for m = 1 to N_RHS do
            t ← y_{i,m}; y_{i,m} ← y_{p_i,m}; y_{p_i,m} ← t
        end
    end
end
for m = 1 to N_RHS do
    w_1 ← y_{1,m}
    for i = 2 to N do
        s = 0
        for j = 1 to i − 1 do
            s ← s + A_{i,j} * w_j
        end
        w_i ← y_{i,m} − s
    end
    y_{N,m} ← w_N/A_{N,N}
    for i = N − 1 to 1 step −1 do
        s = 0
        for j = i + 1 to N do
            s ← s + A_{i,j} * y_{j,m}
        end
        y_{i,m} ← (w_i − s)/A_{i,j}
    end
end
return {y_{j,m}}_{j,m=1}^{N,N_RHS}
End Procedure LUSolve
```

since swapping a row then swapping again returns rows to their original state and implies $P^{-1} = P$. If we define $\mathbf{w} = U\mathbf{x}$, we can solve two triangular systems in succession

$$\begin{aligned} L\mathbf{w} &= P\mathbf{y}, \\ U\mathbf{x} &= \mathbf{w}. \end{aligned} \tag{D.9}$$

The first is simply forward substitution. Since

$$\sum_{j=1}^{i} L_{ij} w_j = (Py)_i \,, \tag{D.10}$$

we solve $w_1 = (Py)_1 / L_{11} = (Py)_1$ and

$$w_i = (Py)_i - \sum_{j=1}^{i-1} L_{ij} w_j, \quad i = 2, 3, \ldots, N. \tag{D.11}$$

We derive a similar back substitution formula to solve $U\mathbf{x} = \mathbf{w}$. The combination of these two form the *LUSolve* procedure in Algorithm 142 (LUFactorization). The input to the procedure is the *factorized* matrix, i.e. the output of *Factorize*. To accommodate multiple right hand sides, we assume that a two dimensional array with N_{RHS} columns is supplied, as is done with the LAPACK routines. The output is then an array whose columns are the solution vector for each right hand side.

Rather than use these prototype procedures in production, which we present here to understand how the algorithm works, it is better to use optimized library routines such as those provided by LAPACK [2]. The LAPACK routines can take advantage of optimized BLAS routines and run efficiently on parallel systems. The two routines useful for general matrices are

```
xGETRF(M, N, A, LDA, IPIV, INFO)
xGETRS(TRANS, N, NRHS, A, LDA, IPIV, B, LDB, INFO)
```

where "x" denotes the data type of the variables, "D" for double, "S" for single, for instance. The procedures perform the factorization (F) and Solve (S) phases separately. The arguments are the number of rows and columns, M and N, the matrix A, the leading dimension of A, LDA, the pivot vector, IPIV and an error flag. The solve routine takes a character variable, TRANS, that specifies if the transpose of A is to be used. The argument N is the order of the matrix A and NRHS is N_{RHS}. The next three arguments are the same as for xGETRF, but A is the factorized matrix. Finally, B corresponds to $\{y_{j,m}\}_{j,m=1}^{N,N_{RHS}}$ where LDB is the leading dimension of the array B.

D.2 Iterative Solvers

We construct the solution of a system of equations $A\mathbf{x} = \mathbf{y}$ iteratively by adding a correction to a current iterate, $\mathbf{x}^k$,

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \omega^k \mathbf{v}^k. \tag{D.12}$$

The vector $\mathbf{v}^k$ is the search direction and ω is a parameter that says how far to step in that direction. The simplest choice of direction is the direction of the iteration residual

$$\mathbf{v}^k = \mathbf{r}^k \equiv \mathbf{y} - A\mathbf{x}^k \tag{D.13}$$

giving what is known as the *Richardson Iteration* method,

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \omega^k \mathbf{r}^k. \tag{D.14}$$

For a point of reference, we can write the well-known Jacobi method in this form with $\mathbf{v}^k = D^{-1}\mathbf{r}^k$, where D is the diagonal part of the matrix A.

We should choose the step parameter ω^k so that the iterate, $\mathbf{x}^{k+1}$, is in some sense closer to the solution than the previous iterate. One choice, for example, is to find ω^k so that the Euclidean norm of the residual at the next step, $\|\mathbf{r}^{k+1}\|$ is minimized along the search direction. We relate the residual at the next iteration to that of the current residual by

$$\mathbf{r}^{k+1} = \mathbf{y} - A\mathbf{x}^{k+1} = \mathbf{y} - A\mathbf{x}^k - \omega^k A\mathbf{r}^k = \mathbf{r}^k - \omega^k A\mathbf{r}^k. \tag{D.15}$$

The *Euclidean norm* of the new residual,

$$\left\|\mathbf{r}^{k+1}\right\| = \sqrt{\left\langle \mathbf{r}^{k+1}, \mathbf{r}^{k+1}\right\rangle} = \sqrt{\sum_{i=1}^{N} \left(\mathbf{r}_i^{k+1}\right)^2} \tag{D.16}$$

is therefore related to the old residual and ω^k by the relation

$$\begin{aligned} \left\|\mathbf{r}^{k+1}\right\|_2^2 &= \left\langle \mathbf{r}^{k+1}, \mathbf{r}^{k+1}\right\rangle \\ &= \left\langle \mathbf{r}^k, \mathbf{r}^k\right\rangle - 2\omega^k \left\langle \mathbf{r}^k, A\mathbf{r}^k\right\rangle + \left(\omega^k\right)^2 \left\langle A\mathbf{r}^k, A\mathbf{r}^k\right\rangle, \end{aligned} \tag{D.17}$$

which is quadratic in ω^k. We find the minimum as a function of ω^k where the derivative is zero,

$$\frac{d\|\mathbf{r}^{k+1}\|_2^2}{d\omega^k} = 0 = -2\left\langle \mathbf{r}^k, A\mathbf{r}^k\right\rangle + 2\omega^k \left\langle A\mathbf{r}^k, A\mathbf{r}^k\right\rangle, \tag{D.18}$$

giving

$$\omega^k = \frac{\langle \mathbf{r}^k, A\mathbf{r}^k\rangle}{\langle A\mathbf{r}^k, A\mathbf{r}^k\rangle}. \tag{D.19}$$

Alternatively, we could choose to minimize some functional other than the next residual along the search direction. For example, we could minimize

$$\Psi(x) = \frac{1}{2}\langle \mathbf{x}, A\mathbf{x}\rangle - \langle \mathbf{x}, \mathbf{y}\rangle, \tag{D.20}$$

which gives

$$\omega^k = \frac{\langle \mathbf{r}^k, \mathbf{r}^k\rangle}{\langle \mathbf{r}^k, A\mathbf{r}^k\rangle}. \tag{D.21}$$

The functional Ψ has as its minimum the solution of the linear system, and this choice of ω^k gives the *method of steepest descent*.

The convergence rate is determined by the *condition number* of the matrix A, $\kappa(A) = \|A\|\|A^{-1}\|$. The larger the condition number, the slower the convergence. Spectral collocation matrices have condition numbers that grow rapidly with N. It is therefore important to mitigate this growth.

We accelerate convergence by introducing a matrix factor to the search direction in addition to the parameter, ω^k, to lower the condition number of the system. We change the iteration (D.12) to

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \omega H^{-1}\mathbf{v}^k, \tag{D.22}$$

where H is the *preconditioning matrix* or *preconditioner*. To see the effect of the preconditioning matrix, consider the Richardson method for which $\mathbf{v}^k = \mathbf{r}^k$. The (exact) solution to the system is the fixed point of the iteration that satisfies

$$\mathbf{x} = \mathbf{x} + \omega^k H^{-1}(\mathbf{y} - A\mathbf{x}), \tag{D.23}$$

so $\mathbf{x}$ also satisfies the modified (preconditioned) system of equations

$$H^{-1}A\mathbf{x} = H^{-1}\mathbf{y}. \tag{D.24}$$

We require the matrix H to have two properties: It should be easy to invert, and it should approximate the original matrix A in such a way that the condition number $\kappa(H^{-1}A)$ of the modified system is lower than $\kappa(A)$. The choice $H = A$ is optimal from the point of view of conditioning since $\kappa(I) = 1$. On the other hand, if A was that easy to invert in the first place, this whole exercise would be pointless. Instead, we settle, and choose H to be some easily invertible approximation of A. In fact, the Jacobi method noted above can be viewed as the preconditioned Richardson method with the preconditioner $H = D$, the diagonal part of A, which is clearly easy to invert.

If we re-trace the steps that we used to determine ω^k and gather the relations, we get the preconditioned minimum residual Richardson method

```
Compute r = y − Ax^0
for k = 1 to N_it do
    Solve Hz = r for z
    ω ← ⟨r, Az⟩ / ⟨Az, Az⟩
    x ← x + ωz
    r ← r − ωAz
end
```

We convert the procedure to the method of steepest descent by changing the definition of ω to (D.21).

Even with preconditioning, the relaxations of the Richardson and steepest descent methods can be slow. The main culprit is that the residual does not necessarily have to point in the direction of the solution. Instead, there may be a large amount of backtracking. The problem is easy to imagine in the context of the steepest descent algorithm and two variables. If the functional is shaped more like a long thin valley instead of a round basin, the downhill direction does not point directly to the bottom.

For symmetric systems, the popular *Conjugate Gradient* method eliminates the backtracking that slows down the convergence of the steepest descent method. Instead of choosing the residual as the search direction, it chooses a direction that is conjugate to all of the previous directions. This guarantees that each search direction does not contain components in the previous directions already searched.

Descriptions of the Conjugate Gradient method can be found in many sources. For those not familiar with it, we recommend [13] or [19] for background. In form, it looks similar to the methods we just discussed,

```
Compute r = y − Ax^0
Solve Hz = r for z
v ← z
c ← ⟨r, z⟩
for k = 1 to N_it do
    z ← Av
    ω ← c / ⟨v, z⟩
    x ← x + ωv
    r ← r − ωz
    Solve Hz = r for z
    d ← ⟨r, z⟩
    v ← z + (d/c) v
    c ← d
end
```

The Conjugate Gradient algorithm is guaranteed to work, however, only for matrices that are symmetric. For non-symmetric systems, it can fail to converge. If the matrix is nonsymmetric we could solve the problem $A^T Ax = A^T y$ so that the coefficient matrix is symmetric. Unfortunately, squaring the matrix increases the condition number, which slows down the convergence rate. For non-symmetric systems, we should use methods specifically derived for them, such as the BiCGStab or the GMRES method. The BiCGStab, for instance is

```
Compute r = y − Ax^0
r̄ ← r
v ← 0; p ← 0
ρ ← 1; α ← 1; ω ← 1
for k = 1 to N_it do
    ρ̂ ← ρ
    ρ = ⟨r̄, r⟩
    β = ρα/(ρ̂ω)
    p = r + β(p − ωv)
    Solve Hy = p for y
    v ← Ay
    α = ρ/⟨r̄, v⟩
    s = r − αv
    Solve Hz = s for z
    t ← Az
    ω = ⟨t, s⟩ / ⟨t, t⟩
    x ← x + αy + ωz
    r ← s − ωt
    if ‖r‖_2 < Tol then Exit
end
```

The GMRES method requires significantly more storage and we will not describe it here. We recommend the book [19] for a description of the GMRES method, should it be needed.

Appendix E
Data Structures

Arrays have both advantages and disadvantages as structures in which to store data. Their main advantage, beyond their simplicity, is that operations on arrays can be computed very efficiently. Standardization of such operations has led to the basic linear algebra subroutines (BLAS) that we discussed in Appendix C. Also, access to a particular element of an array is fast. The main disadvantage, which goes hand in hand with simplicity and efficiency, is that arrays are not very flexible. For best efficiency, we typically must fix the size of an array. If we don't know the size beforehand, the addition of new elements or the deletion of existing elements can require costly allocation and deallocation of blocks of memory, plus the time to copy data from old to new versions of the array. To enable flexible data storage and retrieval, we need more sophisticated data structures.

In this appendix we describe two useful data structures. The first is the linked list. In contrast to arrays, linked lists do not have a specified ordering of the data. They have the advantage that we can easily add and delete elements of the list, so we don't have to know the size of a list in advance. On the negative side, because there is no structured ordering of the data, it is expensive to find a particular element of a list. The second data structure is the hash table. Hash tables are flexible structures that are efficient to search.

Our discussion of data structures will be necessarily brief, and we will discuss only aspects that we need to implement the spectral element algorithms in this book. In particular, our discussion of hash tables is limited to an example of a sparse matrix. Further discussion of the subject of data structures can be found in many books, such as that of Knuth [16]. Note that it is particularly easy to find detailed discussions of linked lists, since they are often used in programming language books for examples of how to use pointers.

E.1 Linked Lists

A *linked list* is a data structure that consists of a collection of *records*. In a *singly linked list*, the record consists of two parts, namely the data that it holds and a pointer to the next record. (Variations that we do not need to consider here include doubly linked lists where a record also has a pointer to the previous record, and circularly linked lists whose last record points to the first record of the list.) We show a diagram of a singly linked list in Fig. E.1.

The records linked in a singly linked list data structure contain data and a pointer (or pointers) to other records. The data stored in the record can be something as simple as an integer value, or as complicated as a structure that contains a variety

D.A. Kopriva, *Implementing Spectral Methods for Partial Differential Equations,* Scientific Computation,

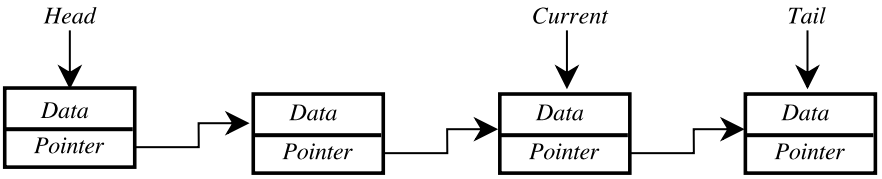

Fig. E.1 Schematics of a singly linked list. Records, represented by *boxes*, are linked by a pointer to the next record in the list. A list can be accessed by its head, tail or current pointer

Algorithm 143: *Record:* An Example Linked List Record Definition

```
Structure Record
  listData // Primitive data type, structure or pointer
  next // Pointer to a Record data type
End Structure Record
```

Algorithm 144: *LinkedList:* A Linked List Class Definition

```
Class LinkedList
  Data:
    head, tail, current // Pointers to Record type
  Procedures:
    Construct(); // Algorithm 145
    Add(data); // Algorithm 145
    GetCurrentData(); // Algorithm 145
    MoveToNext(); // Algorithm 145
    Destruct(); // Algorithm 145
    Print(); // Algorithm 145
End Class LinkedList
```

of other data structures, including arrays and linked lists. To allow for some abstraction, let us assume that the data is stored in a *structure*, such as is provided as a *struct* in C/C++ or a *TYPE* in Fortran. Otherwise, any data that we want to include is appropriate. We can implement the record itself as a structure as well, or as a class. We show an example of a *Record* implemented as a structure in Algorithm 143 (Record). It contains another structure, *listData* that organizes the actual data. It also contains a pointer to a *Record*, which is typically called *next*.

A list itself is referenced by its *head* and *tail* pointers. For convenience, we add a *current* pointer that marks a particular record in the list. These pointers comprise the only data that we need for the list itself shown in Algorithm 144 (LinkedList).

Typical operations to define for a linked list include those that add and delete records and that traverse the list. Other procedures of importance to more general problems might sort the records according to some relation in the stored data, provide copy or merge functions, and the like. For the applications in this book, we need only to add records to the list, traverse the list, and destroy the list. With these needs in mind, we define the *LinkedList* class in Algorithm 144 (LinkedList). It has a con-

Algorithm 145: *LinkedList:Procedures:*

```
Procedure Construct
this.head ⇒ NULL
this.tail ⇒ NULL
this.current ⇒ NULL
End Procedure Construct

Procedure Add
Input: data // is either a primitive, a structure, or a pointer
Allocate newRecord
if this.tail ⇒ NULL then
    this.head ⇒ newRecord
    this.tail ⇒ newRecord
else
    this.tail.next ⇒ newRecord
    this.tail ⇒ newRecord
end
this.current ⇒ this.tail
this.current.next ⇒ NULL
this.current.listData ← data
End Procedure Add

Procedure GetCurrentData
return this.current.listData
End Procedure GetCurrentData

Procedure MoveToNext
this.current ⇒ this.current.next
End Procedure MoveToNext

Procedure Destruct
this.current ⇒ this.head
while this.current ⇏ NULL do
    pNext ⇒ this.current.next
    Deallocate memory pointed to by this.current
    this.current ⇒ pNext
end
End Procedure Destruct

Procedure Print
this.current ⇒ this.head
while this.current ⇏ NULL do
    Print this.current.data
    this.MoveToNext()
end
End Procedure Print
```

structor, destructor, a procedure to add new data, a procedure to get the data in the current record, and a procedure that moves the current pointer to the next element of the list. The implementations of those procedures are shown in Algorithm 145 (LinkList:Procedures).

The constructor for the linked list in Algorithm 145 (LinkList:Procedures) has little to do. It simply sets the pointers to point to *NULL*, which denotes that the pointer does not yet point to any record.

The *Add* procedure takes data as input, creates (allocates) a pointer to a new record and adds the new record to the end of the list, if the list is not empty. If the list is empty, then the new record becomes the start of the list and all the list's pointers point to it.

The *MoveToNext* and *Destruct* procedures illustrate how to navigate the list. To move one step from any position in the list, the *current* pointer is pointed to it's *next* pointer. The *Destruct* procedure shows how to navigate through the entire list. There, we set a pointer to the head and another to its *next* pointer (to keep the next record accessible). A *while* loop then steps through the entire list until the end, which is detected by the current pointer pointing to *NULL*. At each step, the memory to which the current record points is destroyed, and the current pointer is set to point to what was the next record in the list.

We could construct new procedures to perform whole list actions, such as printing the list or searching for a record whose data matches a given criterion, by modifying the *Destruct* procedure. The *Print* procedure in Algorithm 145 (LinkList:Procedures), for instance, loops through the list and prints the data associated with the record pointed to by the *current* pointer. To search for a record with particular data, we would replace the print line in the *Print* procedure with a test on the data.

The *Add* and *Delete* procedures illustrate the flexibility of a linked list to store data when we do not know the number of records ahead of time. The *Destruct* and *Print* procedures show the weakness of a linked list. To access a particular record, we must start at the beginning and search for it sequentially. There is no mechanism for random access to the records, say, to access the fifth element directly, as can be done with an array. Neither can the list find a particular record with a given data value without stepping through the list from the beginning.

E.1.1 Example: Elements that Share a Node

One example of where a linked list is useful is to collect the *id*'s of all the spectral elements in a mesh that share a common corner node. In a structured mesh, a node might be shared by one, two or four elements. In an unstructured mesh a node may in principle be shared by any number of elements (Fig. E.2). Operations on each list are typically of a "ForAll" type, so the entire list must be traversed, but the order in which elements are accessed is not often important. For these needs, a linked list is appropriate to store the shared *elementID*'s for each node.

Let us assume that we have an array of nodes stored by their *nodeID*, and an array of elements stored by their *elementID*. We create such arrays by reading the information from a data file created by a mesh generator. Each node will store (in addition to its other data, such as location) a linked list whose data is just an *elementID* (Fig. E.2). We will not need to modify this shared element data after it is created,

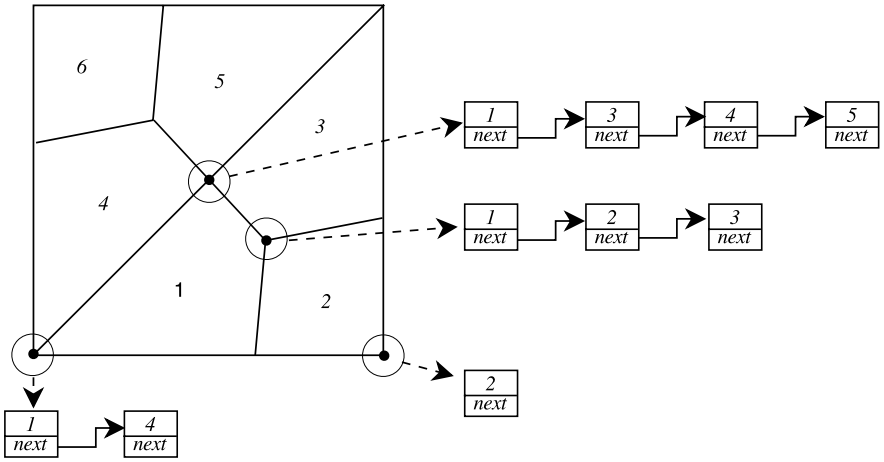

Fig. E.2 Linked lists associated with nodes that store neighboring element id's

so a singly linked list is sufficient to store the data. Each element will have an array that contains the *nodeID*s of the four nodes that define the corners of the element. Each element knows the *nodeID*'s of its corners, since that information is provided by the mesh file. The nodes, however, do not know which elements share them.

To construct the lists of shared elements, we would loop through each element and for each node in the element, add the element's *elementID* to the list associated with the node with that *nodeID*. Specifically, given an array of elements $\{e_j\}_{j=1}^{N_e}$, which stores the *nodeID*s of its nodes in the array $\{nodeID_j\}_{j=1}^{4}$, and array of nodes $\{node_i\}_{i=1}^{N_{node}}$, which stores a linked list of *elementID*s, the logic to create a list of shared elements is

```
for elementID = 1 to N_e do
    for k = 1 to 4 do
        n = e_j.nodeID_k
        node_n.elementList.Add(elementID)
    end
end
```

The shared elements for each node can then be accessed and printed by a simple loop

```
for i = 1 to N_node do
    node_i.elementList.Print()
end
```

E.2 Hash Tables

Hash tables are data structures designed to enable storage and fast retrieval of key-value pairs. An example of a key-value pair is a variable name ("gamma") and its

associated value ("1.4"). The table itself is typically an array. The location of the value in a hash table associated with a key, k, is specified by way of a *hash function*, $H(k)$. In the case of a variable name and value, the hash function converts the name into an integer that tells it where to find the associated value in the table.

A very simple example of a hash table is, in fact, a singly dimensioned array. The key is the array index and the value is what is stored at that index. Multiple keys can be used to identify data; a two dimensional array provides an example where two keys are used to access memory and retrieve the value at that location. If we view a singly dimensioned array as a special case of a hash table, its hash function is just the array index, $H(j) = j$. A doubly dimensioned array could be (and often is) stored columnwise as a singly dimensioned array by creating a hash function that maps the two indices to a single location in the array, e.g., $H(i, j) = i + j * N$, where N is the range of the first index, i.

Although arrays provide fast access to their data, they allocate storage for all possible keys, and only set the value for a key if the data associated with a particular key is present. A matrix, for instance, can be stored as a two-dimensional array. The value of the (i, j)th element of the matrix is stored at a location in memory associated with the two keys, (i, j). A sparse matrix can require much more memory than necessary when stored this way, since most of the elements are zero and don't need to be stored at all. It is for sparse data structures like this that hash tables are useful.

To create a hash table, we need a storage model and a hash function. Each has practical issues. Often it is convenient and efficient to store the data in an array of fixed size. The hash function will then map the keys to an element of that array. However, when we are done, we want that array to be fully populated so that there is no wasted space. Therefore, we don't want to allocate an array that can hold all possible values for all possible keys (as in the matrix storage problem above), only ones that can hold the values for keys that occur. It is not generally desirable to create a "perfect" hash function that generates a unique index for a given set of keys. Instead, *collisions* will be allowed where different keys can give the same hash value, i.e., point to the same location in the array. For instance, it is unrealistic to create a dictionary (word + definition) by allocating storage for every possible combination of letters. Instead we might define a finite array and create a hash function to map to that array. We could create the hash function by assigning a value to each letter, like its position in the alphabet, and adding the values in the word. Thus the word "dad" would be hashed to index $4 + 1 + 4 = 9$. But so would "fab".

Of the many ways to resolve collisions, *chaining* is common. To use chaining, each entry in the array stores a pointer to a linked list, instead of storing values in the hash table array itself. As collisions occur, the new entry is added to the linked list. Then, when it is time to retrieve the value associated with a key, the key is hashed and the linked list at the location given by the hash value is then searched (sequentially) for the actual key. Yes, we have already said that it is slow to search a linked list. But if the table is of a reasonable size, and the hash function is reasonable, then the number of collisions, and hence the number of entries in the linked list, will be small and quickly searched.

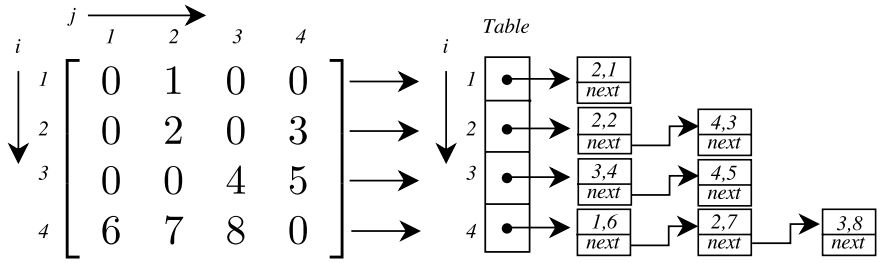

Fig. E.3 Example of sparse matrix representation by a hash table

Algorithm 146: *SparseMatrix:* A Sparse Matrix Class Definition

```
Class SparseMatrix
  Data:
    {table_i} // Array of pointers to LinkedList
  Procedures:
    Construct(N) ; // Algorithm 147
    AddDataForKeys(data, i, j); // Algorithm 147
    DataForKeys(i, j); // Algorithm 147
    ContainsKeys(i, j); // Algorithm 147
    Destruct(); // Algorithm 147
End Class SparseMatrix

Structure TableData
  key // an integer
  data // a primitive type, structure, or pointer
End Structure TableData
```

Since matrices are easy to understand, we show how to use a simple hash table with chaining to store and retrieve values from a sparse matrix. The algorithm will be useful to delete duplicate edges in a mesh. We show a diagram of the data structure in Fig. E.3. To start, note that if the matrix is invertible, then it cannot have a row that is all zeros, that is, every row must have at least one entry. Therefore, the table itself should be an array of length N, where N is the order of the matrix. Each location in the table will correspond to a row in the matrix and the hash function for a matrix element (i, j) is simply $H(i, j) = i$. Clearly there will be collisions, since each row hashes to the same table entry. For this reason, we will use chaining so that each item of the table array will be a pointer to a linked list. A record in the linked list will store two pieces of data, namely the column j and the value of the matrix entry in the ith column and jth row.

We show a sparse matrix class and data definition that describe these requirements in Algorithm 146 (SparseMatrix). Note that we allocate the storage for the array of pointers in the constructor to allow variable size arrays to be created, see Algorithm 147 (SparseMatrix:Procedures). The destructor, *Destruct* tells each of the linked lists to destruct themselves and then deallocates the memory for the array.

Algorithm 147: *SparseMatrix:Procedures:*

```
Procedure Construct
Input: N // Order of the matrix
Allocate memory for this.{table_i}_{i=1}^{N}
for i = 1 to N do
    this.table_i ⇒ NULL
end
End Procedure Construct
```

```
Procedure AddDataForKeys
Input: inData, i, j
if this.table_i ⇒ NULL then this.table_i.Construct()
if this.ContainsKeys(i, j) = FALSE then
    d.key ← j; d.data ← inData // d is of type TableData
    this.table_i.Add(d)
end
End Procedure AddDataForKeys
```

```
Procedure ContainsKeys
Input: i, j
if this.table_i ⇒ NULL then return FALSE
this.table_i.current ⇒ this.table_i.head
while this.table_i.current ⇏ NULL do
    d ← this.table_i.GetCurrentData()
    if d.key = j then return TRUE
    this.table_i.MoveToNext()
end
return FALSE
End Procedure ContainsKeys
```

```
Procedure GetDataForKeys
Input: i, j
if this.table_i ⇒ NULL then setError
this.table_i.current ⇒ this.table_i.head
while this.table_i.current ⇏ NULL do
    d ← this.table_i.GetCurrentData()
    if d.key = j then return d.data
    this.table_i.MoveToNext()
end
setError
End Procedure GetDataForKeys
```

```
Procedure Destruct
for i = 1 to N do
    this.table_i.Destruct()
    this.table_i ⇒ NULL
end
Deallocate memory for this.{table_i}_{i=1}^{N}
End Procedure Destruct
```

We need three basic procedures for the SparseMatrix class. We present implementations of them also in Algorithm 147. The first procedure is the *AddDataForKeys* procedure, which adds the matrix entry for the pair (i, j). It first checks to see if the ith row has been created and constructs a linked list to which it will point. It then checks (just to be safe) to see if the current entry has already been added. If not, the column, j, and value are added to the linked list for row i. The *ContainsKeys* function does a linear search from the head of the list for the ith row and checks to see if the *key* matches the column number. If so, the (i, j)th component is in the table. Finally, the *DataForKeys* procedure returns the data that is stored for the matrix entry, if the entry exists. For safety, it first checks to see if the ith column has been created. If not, an error condition must be set, which we represent generically by a call to some *setError* procedure. (Setting an error could include setting an output "flag" variable, creating and returning an "error handler" structure, or throwing an exception, depending on the computer language being used.) If the ith row is present, the *DataForKeys* procedure then searches through the list for a record whose *key* matches the desired column. On the chance that the *current* position in the list already corresponds to the desired column, the value there is returned. Otherwise, it searches from the beginning by using the *ContainsKeys* procedure.

We could add other procedures to the sparse matrix class, but we don't need them here. For instance, if we wanted to write a sparse matrix-vector multiplication procedure, we would add a *GetNextData* procedure that steps to the next record in the list for the ith row and returns the row (to access the vector element) and the value for the factor.

E.2.1 Example: Avoiding Duplicate Edges in a Mesh

As an example that uses the sparse matrix algorithm, we create an array of unique edges in a mesh. (See Sect. 8.2.) Two *nodeID*'s define an edge, one for the start and one for the end, and hence, an edge is uniquely specified by two keys. If the only information in a mesh file are the nodes and the element connectivity that specifies which nodes are used to create the element, we can generate a collection of edges by looping through each element and creating the four edges from the corner nodes. Defined counter-clockwise on an element, the four edges correspond to the local index of the four corner nodes $(1, 2)$, $(2, 3)$, $(3, 4)$ and $(4, 1)$ (Fig. E.4). To simplify the edge generation procedure, we can create a local *mapping array* (which, itself, can be viewed as a simple hash table), $\{p_{i,k}\}_{i=1,k=1}^{2,4}$, where $(p_{1,1}, p_{2,1}) = (1, 2)$, $(p_{1,2}, p_{2,2}) = (2, 3)$, etc. Suppose an element has an array, $\{nodes_k\}_{k=1}^{4}$ that stores the global *id*'s of its corner nodes. Then the global *id*'s of the start and end nodes for side one of the element are $nodes_{p_{1,1}}$ and $nodes_{p_{2,1}}$. We can construct a linked list of edges, *edgeList* from an array of elements *elArray* by the algorithm

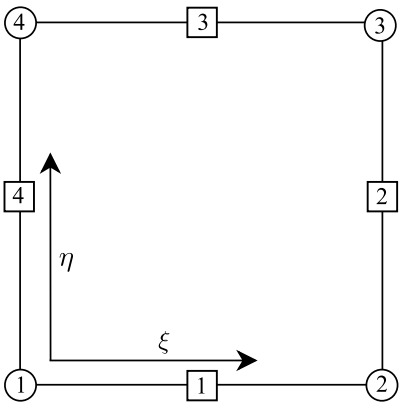

Fig. E.4 Organization of element nodes (*circles*) and edges (*squares*)

```
for j = 1 to N_e do
    for k = 1 to 4 do
        edge.startNode ← elArray_j.nodes_{p1,k}
        edge.endNode ← elArray_j.nodes_{p2,k}
        edgeList.Add(edge)
    end
end
```

Unfortunately, this procedure will create duplicate edges since it counts shared edges twice. To avoid duplicates, we need to test if an edge that is to be about created already exists in the list of edges, i.e., has the same two end nodes. Rather than search the entire list every time, we will create a parallel data structure, in the form of a sparse matrix that we implement as a hash table, so that a search for a given edge is fast. If the edge already exists in the table, we will not add it to the list. At the same time, we will find for free the neighbor to the element across that edge.

Since an edge is uniquely determined by its two end nodes, it is natural to use two keys and two hash functions. A simple choice for the two hash functions is $key1 = Hash1(i, j) = \min(i, j)$ and $key2 = Hash2(i, j) = \max(i, j)$ where i and j are the starting and ending node number. The data held by the table will be simply the edge number of the edge that has the two keys.

The hash table is an efficient way to store the edges so that they can be accessed quickly according to their endpoints. The first key will range from one to the number of nodes, so this will be the size of the array that we construct. Since most mesh generators keep the valence of a node low, ≤ 6, the number of edges with the same first key, which is equal to the valence, is small. Therefore the number of collisions in the table will be small relative to the size of the table.

We modify the procedure above that we wrote to create the list of edges to use the hash table to fill the edge array of the mesh structure of Algorithm 126 (QuadMesh). The result is Algorithm 148 (ConstructMeshEdges). As each edge is found, we first consult the table to see if that edge already exists in the edge array. If it doesn't, we add the edge to the array and the position of the edge in the edge array to the table. If the edge does already exist, then we do not create a new edge. At that

Algorithm 148: *ConstructMeshEdges:* Construct Edge Information for a Spectral Element Mesh

Procedure ConstructMeshEdges
Input: *mesh* `// A QuadMesh`
Uses Algorithms:
Algorithm 126 (QuadMesh)
Algorithm 144 (LinkedList)
Algorithm 146 (SparseMatrix)

edgeTable.*Construct*(*mesh*.N_{node}) `// A SparseMatrix`
for $eID = 1$ **to** *mesh*.K **do**
  for $k = 1$ **to** 4 **do**
    $l1 \leftarrow mesh.edgeMap_{1,k}$
    $l2 \leftarrow mesh.edgeMap_{2,k}$
    $startId \leftarrow mesh.elements_{eID}.nodeIds_{l1}$
    $endId \leftarrow mesh.elements_{eID}.nodeIds_{l2}$
    $key1 \leftarrow Hash1(startId, endId)$
    $key2 \leftarrow Hash2(startId, endId)$
    if *edgeTable*.*ContainsKeys*($key1, key2$) **then**
      $edgeId \leftarrow edgeTable.GetDataForKeys(key1, key2)$
      $e1 \leftarrow mesh.edges_{edgeId}.elementIds_1$
      $s1 \leftarrow mesh.edges_{edgeId}.elementSides_1$
      $l1 \leftarrow mesh.edgeMap_{1,s1}$
      $n1 \leftarrow mesh.elements_{e1}.nodeIDs_{l1}$
      $mesh.edges_{edgeId}.elementId_2 \leftarrow eID$
      if $startId = n1$ **then**
        $mesh.edges_{edgeId}.elementSide_2 \leftarrow k$
      else
        $mesh.edges_{edgeId}.elementSide_2 \leftarrow -k$
      end
    else
      $mesh.N_{edge} \leftarrow mesh.N_{edge} + 1$
      $edgeId \leftarrow mesh.N_{edge}$
      $\{nodes_n\}_{n=1}^{2} \leftarrow \{startId, endId\}$
      $mesh.edges_{edgeId}.Construct(\{nodes_n\}_{n=1}^{2}, eID, k)$
      edgeTable.*AddDataForKeys*($edgeId, key1, key2$)
    end
  end
end
edgeTable.*Destruct*()
return *mesh*
End Procedure ConstructMeshEdges

point in the algorithm, however, we know the current element and side that would have created the duplicate edge. From the array of edges we know the element and side that originally created the edge. Therefore we get the information about the two elements that contribute to an edge for free at the time the duplicate edge is rejected. This information is the element *id*, the element side, and whether or not the direction of the edge is swapped. When we finish creating the array of edges, we destroy it since we no longer need it.

References

1. Abramowitz, M., Stegun, I.A.: Handbook of Mathematical Functions: with Formulas, Graphs, and Mathematical Tables. Dover, New York (1965)
2. Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Croz, J.D., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.: LAPACK Users' Guide, 3rd edn. SIAM, Philadelphia (1999)
3. Ascher, U.M., Petzold, L.R.: Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations. SIAM, Philadelphia (1998)
4. Boyd, J.P.: Chebyshev and Fourier Spectral Methods, 2nd revised edn. Dover, New York (2001)
5. Bracewell, R.: The Fourier Transform and Its Applications. McGraw-Hill, New York (1999)
6. Brigham, E.: Fast Fourier Transform and Its Applications. Prentice Hall, New York (1988)
7. Canuto, C., Hussaini, M., Quarteroni, A., Zang, T.: Spectral Methods: Fundamentals in Single Domains. Springer, Berlin (2006)
8. Canuto, C., Hussaini, M., Quarteroni, A., Zang, T.: Spectral Methods: Evolution to Complex Geometries and Applications to Fluid Dynamics. Springer, Berlin (2007)
9. Deville, M., Fischer, P., Mund, E.: High Order Methods for Incompressible Fluid Flow. Cambridge University Press, Cambridge (2002)
10. Don, W.S., Solomonoff, A.: Accuracy and speed in computing the Chebyshev collocation derivative. SIAM J. Sci. Comput. **16**, 1253–1268 (1995)
11. Farrashkhalvat, M., Miles, J.P.: Basic Structured Grid Generation: With an Introduction to Unstructured Grid Generation. Butterworth-Heinemann, Stoneham (2003)
12. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proc. IEEE **93**(2), 216–231 (2005)
13. Golub, G.H., Loan, C.F.V.: Matrix Computations. Johns Hopkins University Press, Baltimore (1996)
14. Hesthaven, J.S., Gottlieb, S., Gottlieb, D.: Spectral Methods for Time-Dependent Problems. Cambridge University Press, Cambridge (2007)
15. Knupp, P.M., Steinberg, S.: Fundamentals of Grid Generation. CRC Press, Boca Raton (1993)
16. Knuth, D.E.: The Art of Computer Programming. Addison-Wesley, Reading (1998)
17. Lambert, J.D.: Numerical Methods for Ordinary Differential Systems: The Initial Value Problem. Wiley, New York (1991)
18. Overton, M.L.: Numerical Computing with IEEE Floating Point Arithmetic. SIAM, Philadelphia (2001)
19. Saad, Y.: Iterative Methods for Sparse Linear Systems. SIAM, Philadelphia (2003)
20. Schwab, C.: p- and hp-Finite Element Methods: Theory and Applications in Solid and Fluid Mechanics. Oxford University Press, London (1988)
21. Shen, J., Tang, T.: Spectral and High-Order Methods with Applications. Science Press, Beijing (2006)
22. Swarztrauber, P.: On computing the points and weights for Gauss-Legendre quadrature. SIAM J. Sci. Comput. **24**(3), 945–954 (2002)
23. Temperton, C.: Self-sorting in-place fast Fourier transforms. SIAM J. Sci. Stat. Comput. **12**(4), 808–823 (1991)
24. Toro, E.F.: Riemann Solvers and Numerical Methods for Fluid Dynamics. Springer, Berlin (1999)
25. Williamson, J.: Low storage Runge-Kutta schemes. J. Comput. Phys. **35**, 48–56 (1980)
26. Yakimiw, E.: Accurate computation of weights in classical Gauss-Christoffel quadrature rules. J. Comput. Phys. **129**, 406–430 (1996)

Index of Algorithms

D.A. Kopriva, *Implementing Spectral Methods for Partial Differential Equations,* Scientific Computation,

Subject Index

D.A. Kopriva, *Implementing Spectral Methods for Partial Differential Equations,* Scientific Computation,

Erratum

1 Chapter 3

1. Algorithm 22:

$$L_N \leftarrow \frac{2k-1}{k} x L_{N-1}(x) - \frac{k-1}{k} L_{N-2}$$

should read

$$L_N \leftarrow \frac{2k-1}{k} x L_{N-1} - \frac{k-1}{k} L_{N-2}.$$

(Thanks to Travis Johnson)

2 Chapter 4

1. Equation (4.6) should be φ_x in the boundary term. Should be

$$\int_0^L \varphi_t \phi dx = \int_0^L (\nu \varphi_x)_x \phi dx = \nu \varphi_x \phi|_0^L - \int_0^L \nu \varphi_x \phi_x dx.$$

2. After (4.14) $x_n = 2\pi n/N$. The n is missing on the right in the text.
3. Equation (4.39) is missing the ν. Should be

$$\dot{\hat{\Phi}}_k = -\left(ik + \nu k^2\right)\hat{\Phi}_k, \quad k = -N/2, \ldots, N/2.$$

4. After (4.141), the statement about integrate by parts once or twice is incorrect. New results show they are actually identical for either quadrature. See D.A. Kopriva and G. Gassner "On the Quadrature and Weak Form Choices in Collocation Type Discontinuous Galerkin Spectral Element Methods", J. Sci. Comput. (doi:10.1007/s10915-010-9372-3).
5. Equation (4.82) should be

$$\varphi(x,t) = \sin[\pi(x+1)]e^{-\pi^2 t}.$$

(Thanks to Travis Johnson)
6. In Algorithm 60, p. 139, the procedure should be called "DGDerivative", not "ComputeDGDerivative".
7. Page 127, Algorithm 53 should have $\Phi \leftarrow 0$ instead of $U \leftarrow 0$. (Thanks to Travis Johnson)
8. Page 133, Algorithm 57's summand in the inner loop should be $D_{k,n} D_{k,j} w_k$ to be consistent with (4.123). (Thanks to Travis Johnson)

D.A. Kopriva, *Implementing Spectral Methods for Partial Differential Equations,* Scientific Computation,

3 Chapter 5

1. Algorithm 87 needs t, N_{it}, TOL as an input.
2. Benchmark 5.3.5 $\Delta t = 5.0 \times 10^{-3}$.
3. Figure 5.8: w is missing the "$-$" superscript on the right of the boundary. It should be as shown in Fig. 1 below.

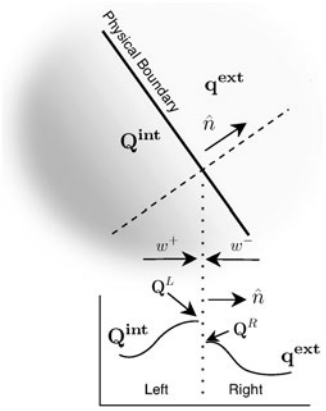

Fig. 1 Interior and exterior states at a boundary viewed along the normal direction

4 Chapter 6

1. In (6.79), the spatial derivative has been moved to the right hand side. Needs a minus sign. Should be

$$\dot{\Phi}_{i,j} = -\frac{1}{J}\left\{\frac{\partial}{\partial\xi} I_N[(Y_\eta - X_\eta)\Phi] + \frac{\partial}{\partial\eta} I_N[(-Y_\xi + X_\xi)\Phi]\right\}_{i,j}.$$

5 Chapter 8

1. In Algorithm 129, the next to the last line should read *mesh.Construct*(*this.spA*, *meshFile*)

6 Appendix E

1. In Algorithm 147, Procedure "GetDataForKeys" should be "DataForKeys".