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),

$$\begin{aligned} \Phi\left(x, y\right) &= \sum_{i,j=0}^{N} \Phi_{i,j} \ell_i\left(x\right) \ell_j\left(y\right), \\ S\left(x, y\right) &= \sum_{i,j=0}^{N} s\left(x_i, y_j\right) \ell_i\left(x\right) \ell_j\left(y\right). \end{aligned} \tag{5.18}$$

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

$$\frac{\partial^2 \Phi}{\partial x^2} = \frac{\partial^2}{\partial x^2} \sum_{k,l=0}^{N} \Phi_{k,l} \ell_k\left(x\right) \ell_l\left(y\right) = \sum_{k,l=0}^{N} \Phi_{k,l} \ell_k''\left(x\right) \ell_l\left(y\right). \tag{5.20}$$

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

$$\begin{aligned}
&\sum_{k=1}^{N-1} D_{ik}^{(2),x} \Phi_{k,j} + \sum_{k=1}^{M-1} D_{ik}^{(2),y} \Phi_{i,k} \\
&\quad = s_{i,j} - D_{i0}^{(2),x} \Phi_{0,j} - D_{iN}^{(2),x} \Phi_{N,j} - D_{j0}^{(2),y} \Phi_{i,0} - D_{jM}^{(2),y} \Phi_{i,M} \\
&\quad \equiv RHS_{i,j}, \quad i = 1, 2, \ldots, N-1; \;\; j = 1, 2, \ldots, N-1.
\end{aligned} \tag{5.29}$$

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

$$n = index(i, j) \equiv \begin{cases} i + (j-1)(N-1) & columnwise/Fortran, \\ j + (i-1)(M-1) & rowwise/C. \end{cases} \tag{5.30}$$

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,

$$\begin{aligned}
&D_{i1}^{(2),x} \Phi_{1,j} + D_{i2}^{(2),x} \Phi_{2,j} + \cdots + D_{i(N-1)}^{(2),x} \Phi_{N-1,j} \\
&\quad + D_{j1}^{(2),y} \Phi_{i,1} + D_{j2}^{(2),y} \Phi_{i,2} + \cdots + D_{j(M-1)}^{(2),y} \Phi_{i,M-1} = RHS_n, \\
&\quad i = 1, 2, \ldots, N-1, \;\; j = 1, 2, \ldots, M-1.
\end{aligned} \tag{5.31}$$

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

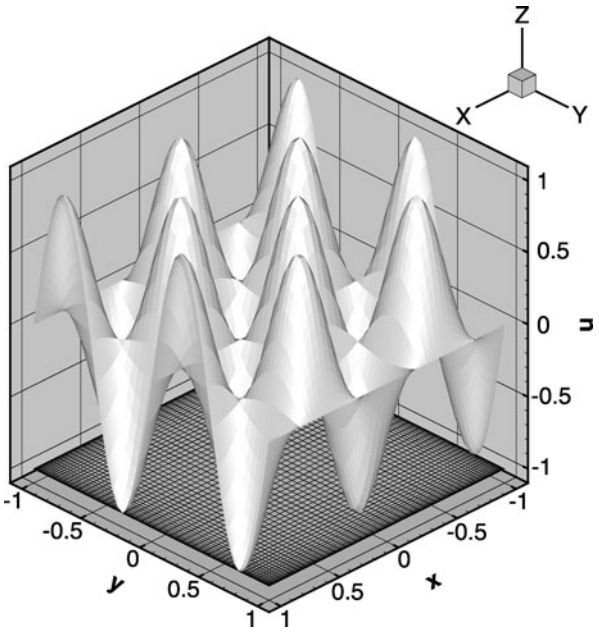

Fig. 5.1 Solution and grid for the Chebyshev collocation approximation of the steady potential in a square with a sinusoidal heat source

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-

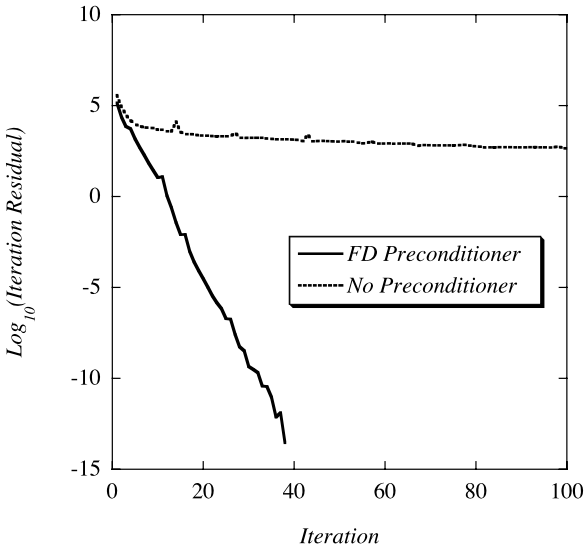

Fig. 5.2 Iteration convergence of the Bi-CGSTAB iterative solver for the Chebyshev collocation approximation to the steady potential in a square with a source when $N = 72$

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

$$
\begin{aligned}
&- \sum_{n,m=0}^{N} \Phi_{n,m} \left[\int_{-1}^{1} \int_{-1}^{1} \ell_n'(x) \ell_i'(x) \ell_m(y) \ell_j(y) dx dy \right] \\
&\quad - \sum_{n,m=0}^{N} \Phi_{n,m} \left[\int_{-1}^{1} \int_{-1}^{1} \ell_i(x) \ell_n(x) \ell_j'(y) \ell_m'(y) dx dy \right] \\
&= \int_{-1}^{1} \int_{-1}^{1} \ell_i(x) \ell_j(y) S dx dy, \quad i, j = 1, 2, \ldots, N-1.
\end{aligned} \tag{5.70}
$$

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

$$
\begin{aligned}
\int_{-1}^{1} \int_{-1}^{1} \ell_i(x) \ell_j(y) S dx dy &\approx \sum_{n,m=0}^{N} w_n w_m \ell_i(x_n) \ell_j(y_m) S(x_n, y_m) \\
&= w_i w_j S(x_i, y_j).
\end{aligned} \tag{5.71}
$$

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

$$
\begin{aligned}
&\int_{-1}^{1} \int_{-1}^{1} \ell_n'(x) \ell_i'(x) \ell_m(y) \ell_j(y) dx dy \\
&\quad = \left(\int_{-1}^{1} \ell_m(y) \ell_j(y) dy \right) \left(\int_{-1}^{1} \ell_n'(x) \ell_i'(x) dx \right) \\
&\quad \approx \left(\delta_{j,m} w_m \right) \left(\sum_{k=0}^{N} \ell_n'(x_k) \ell_i'(x_k) w_k \right).
\end{aligned} \tag{5.72}
$$

As before (e.g. Sect. 3.5.2), let us call

$$
D_{nk}^{(x)} = \ell_n'(x_k) \tag{5.73}
$$

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

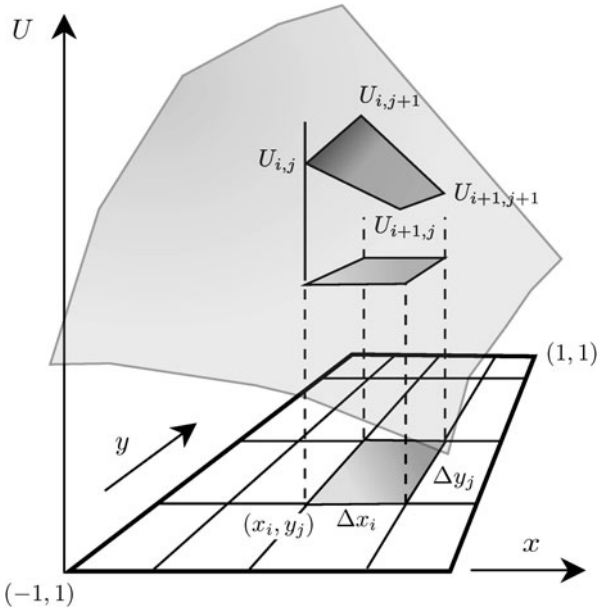

Fig. 5.3 Linear finite element approximation on a quadrilateral element created from four points on the grid

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 $y = x$.

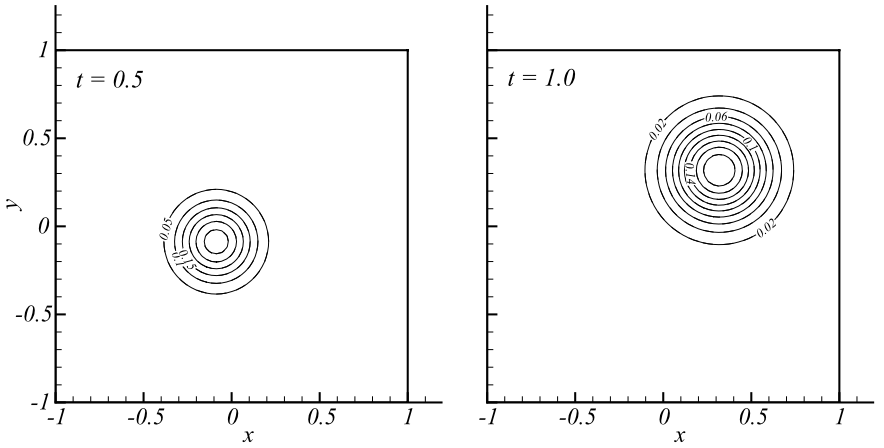

Fig. 5.4 Advection and diffusion of a Gaussian spot by Legendre collocation with $N = 28$. Exact contours drawn with *dashed lines* are indistinguishable from the *solid contours* of the computed solutions

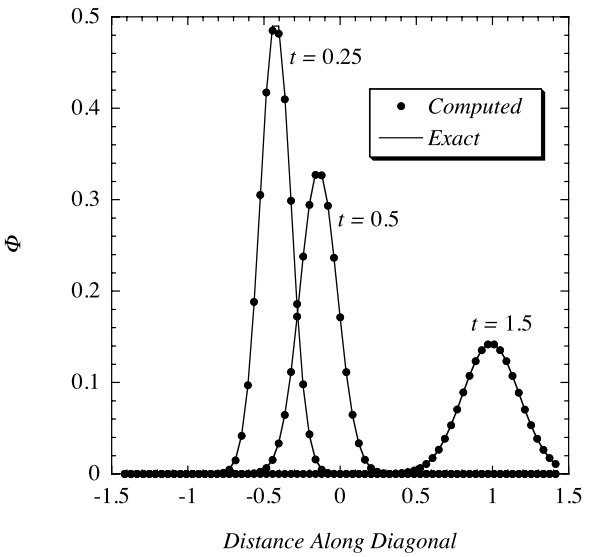

Fig. 5.5 Comparison of computed and exact solutions at three times for the advection and diffusion of a Gaussian spot by Legendre collocation with $N = M = 28$, interpolated to 70 points along the line $y = x$

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

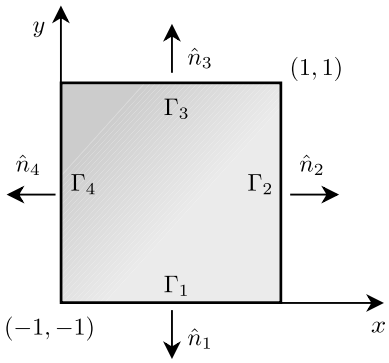

Fig. 5.6 The reference square with normals and boundary curves

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

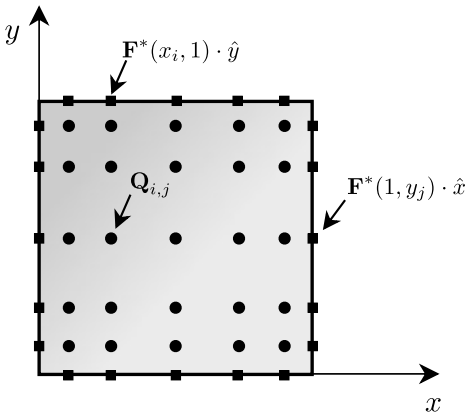

Fig. 5.7 Location of nodes for the discontinuous Galerkin approximation of the wave equation. *Circles* represent nodes where the solution is approximated and are located at the two dimensional Legendre Gauss quadrature points. *Squares* represent nodes where the boundary fluxes are approximated, and 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

$$\mathbf{F}^*\big(\mathbf{Q}^L, \mathbf{Q}^R; \hat{n}\big) = A^+\mathbf{Q}^L + A^-\mathbf{Q}^R. \tag{5.161}$$

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].

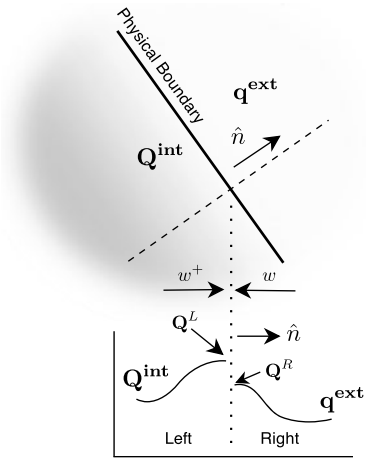

Fig. 5.8 Interior and exterior states at a boundary viewed along the normal direction

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

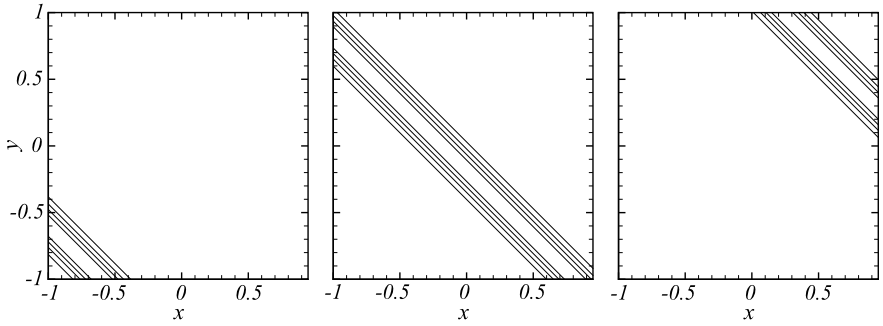

Fig. 5.9 Propagation of a plane Gaussian wave using the nodal discontinuous Galerkin approximation with $N = 40$ for $k_x = k_y = \sqrt{2}/2$ shown at times $t = 0.0$ (*left*), $t = 1.0$ (*center*), and $t = 2.0$ (*right*). Contour levels are 0.2, 0.4, 0.6, 0.8

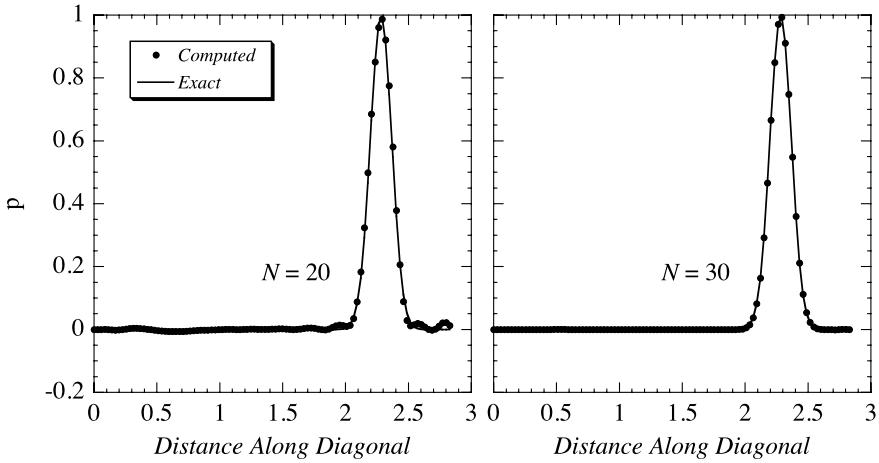

Fig. 5.10 Computed and exact values of the pressure interpolated at 100 points along the line $y = x$ for the plane wave shown in Fig. 5.9 at time $t = 2$

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.

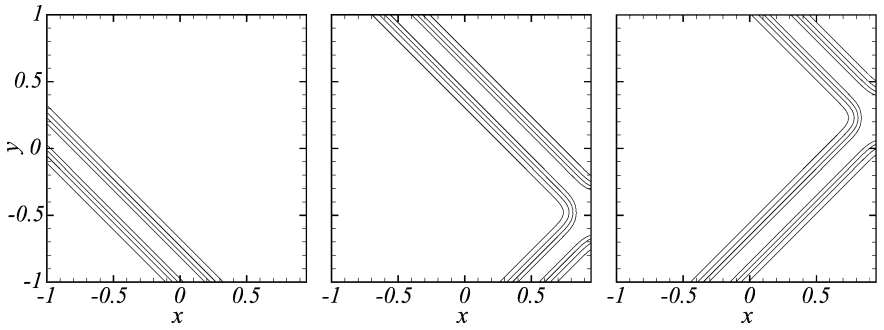

Fig. 5.11 Reflection of a plane Gaussian wave off the right wall boundary using the nodal discontinuous Galerkin approximation with $N = 40$ and $k_x = k_y = \sqrt{2}/2$. Contours are shown at times $t = 0.5$ (*left*), $t = 1.5$ (*center*), and $t = 2.0$ (*right*). Contour levels are 0.2, 0.4, 0.6, 0.8

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

$$\varphi_{xx} + \varphi_{xy} + \varphi_{yy} = f.$$

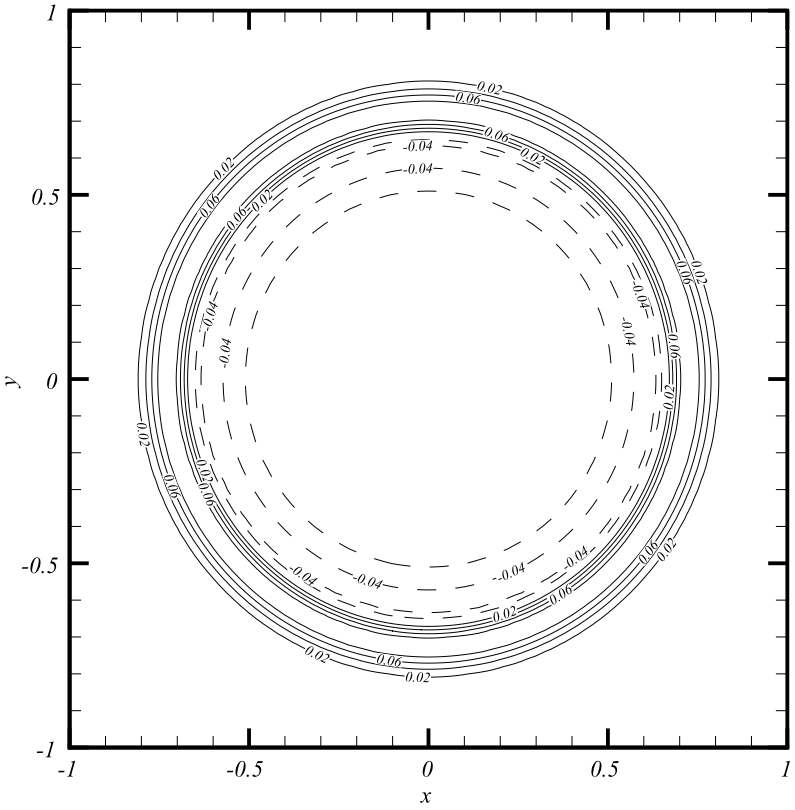

Fig. 5.12 Computed pressure contours at time $t = 0.7$ for a propagating circular wave for $N = M = 70$ and $\Delta t = 8.75 \times 10^{-4}$ interpolated to 100 uniformly spaced points in each direction

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.

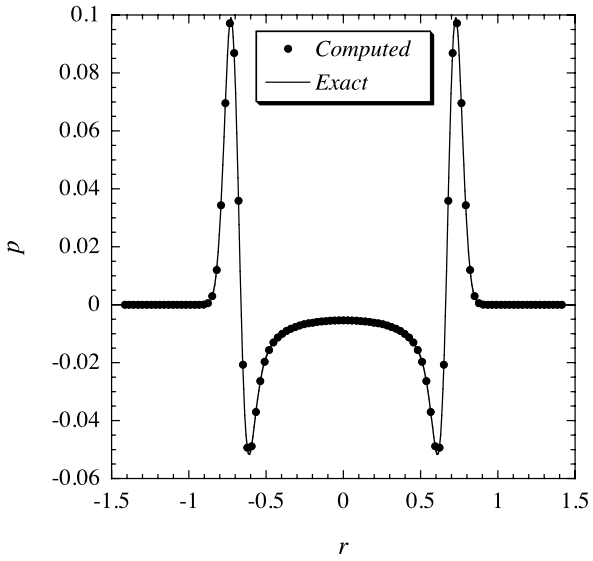

Fig. 5.13 Comparison of the computed circular wave pressure interpolated to 100 points with the exact solution along the line $y = x$ at $t = 0.7$

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 $f(x) = e^{-(x-a/2)^2}$.

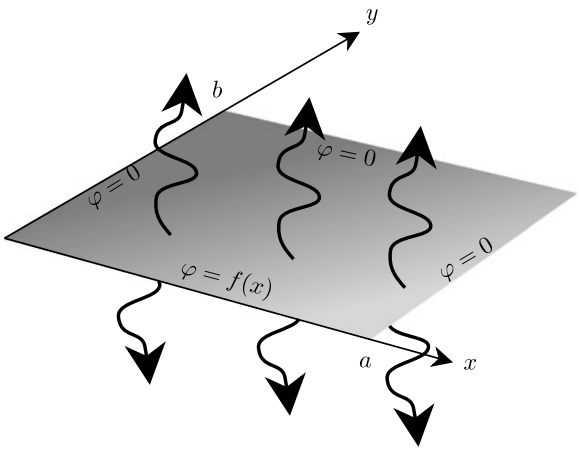

Fig. 5.14 Geometry and boundary conditions for steady temperatures on a thin plate with radiation

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.