Chapter 8
Spectral Element Methods

The main feature of nodal polynomial spectral methods that give them their power, namely the global polynomial approximation with grid points at the nodes of a Gauss quadrature rule, also limits them to a fairly small class of problems. The global nature of the approximation makes it difficult for us to apply the methods to complex geometries or to problems with discontinuities. The fixed node placement makes it difficult to refine the grid locally as may be needed. The cost to compute the derivatives gets large in large problems that need high order polynomials to resolve all the features in a problem. An example of that is the benchmark problem of Sect. 7.4.3. Coupled with the small time steps to which explicit methods are limited or the large condition numbers that adversely affect the solution of the linear systems that arise from implicit methods, the spectral methods that we have presented so far can become expensive, although very accurate.

To compute problems in geometries that are more complex than those we have presented so far, we introduce multidomain methods. In multidomain spectral methods, we divide the domain of interest into smaller subdomains that we can map individually onto the square. We can then apply a spectral method like one of those discussed in Chap. 7 to each of the subdomains. Multidomain spectral methods have become so useful and so common that the methods that we have derived so far in this book are now often called "single domain" or "mono-domain" methods.

We can extend any of the nodal spectral methods that we have presented so far to a multidomain version. We only need to develop methods to couple the subdomains together. However, in this book we will only discuss a subclass of multidomain methods that starts from the weak form of the equations, that is, the nodal Galerkin methods. The Galerkin based nodal methods have natural coupling that follows from the weak form. We call this subclass "Spectral Element Methods" because of its similarity to finite element methods. We note, however, that some authors refer to spectral multidomain methods in general as spectral element methods. For a taxonomy of multidomain methods and presentations of other methods that use the strong form of the equations, see the book by Canuto et al. [8].

Multidomain methods, and spectral element methods in particular, have many advantages over their single domain counterparts. Let us quickly review situations where we should or must use a multidomain method. They include:

- *Problems in complex geometries.* We can use multiple domain decompositions to solve problems in domains that are difficult to map or cannot be mapped onto a single square. Figure 8.1 shows a decomposition of such a domain into four elements. A more convincing geometry might be what we'd need to compute flow over a three element airfoil like that shown in Fig. 8.2.

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

© Springer Science + Business Media B.V. 2009

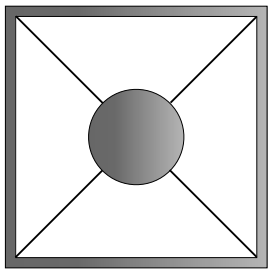

Fig. 8.1 A decomposition of an interior domain by four subdomains

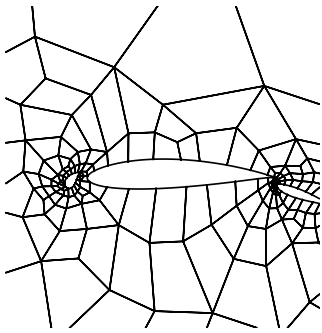

Fig. 8.2 A portion of a decomposition of the exterior of a three element airfoil into multiple subdomains

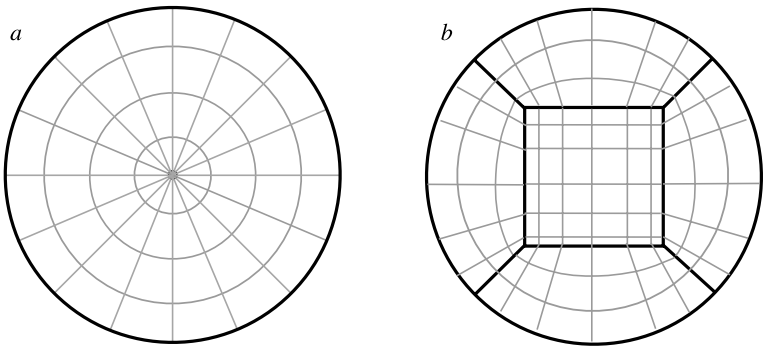

Fig. 8.3 Single (**a**) and multidomain (**b**) decompositions of a disk

- *Problems with coordinate singularities.* Even if the domain is simple, the coordinate mapping onto the square may have singularities. We can use a single domain to approximate an equation on the disk using a cylindrical coordinate system, as seen in Fig. 8.3a. We can eliminate the singularity at the origin by subdividing the disk as in Fig. 8.3b. We can use subdivision to eliminate pole singularities on the sphere as well, as in Fig. 8.4.
- *Problems with discontinuous coefficients or solutions.* Since the convergence rate of spectral methods depends on the smoothness of the solutions, we would not expect accurate solutions when the coefficients in an equation or the solutions are discontinuous. A simple problem on which we should not use a single domain

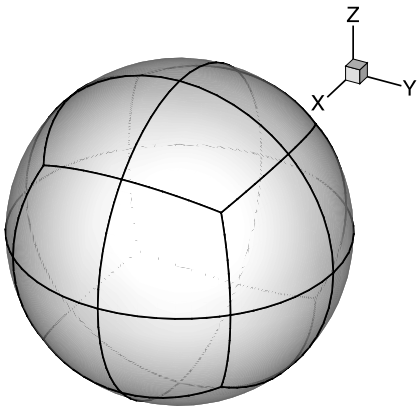

Fig. 8.4 A multidomain decomposition of the surface of a sphere

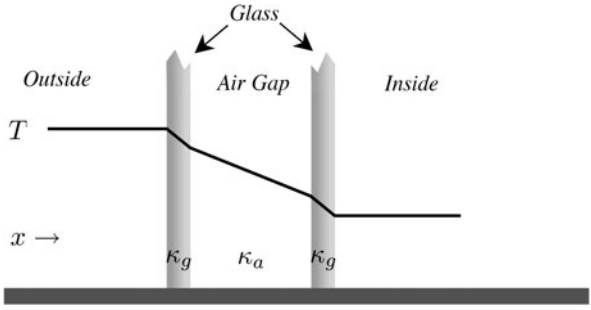

Fig. 8.5 A double glazed window model as an example of a problem with discontinuous media

spectral method is to compute the heat flux through a double glazed window as we sketch in Fig. 8.5. The temperature satisfies the equation

$$\frac{\partial}{\partial x}\left(\kappa \frac{\partial T}{\partial x}\right) = 0. \tag{8.1}$$

The steady temperature, also shown in the figure, is continuous even though the diffusion coefficient, κ, is discontinuous at the material boundaries. If we use a single domain spectral method to solve the problem, the convergence rate of temperature will be slow. On the other hand, if we break the domain into subintervals whose boundaries are at the material interfaces, we could represent the solution exactly by piecewise linear polynomials.

Other problems where the presence of material boundaries would lead us to spectral element methods include electromagnetic wave propagation through different media, electromagnetic scattering off optically coated surfaces, or ultrasound detection of tumors where waves must propagate through muscle, fat and bone.

- *Solution and efficiency driven considerations.* Finally, even if neither the geometry nor the solution dictate that we should use multiple domains, efficiency con-

siderations may instead. To find a technical discussion of the issues we suggest the book by Canuto et al. [8]. We will only give a rough idea here how work and accuracy are related.

The work required to compute spectral solutions depends strongly on the polynomial order. If fast transforms are not available, the cost to compute the derivatives increases as N^2. If we use an explicit time integration method, the time step is limited by the size of the eigenvalues of the problem. The fact that the eigenvalues grow as N^2 for first order and N^4 for second order derivatives means that the number of time steps, which is inversely proportional to Δt, grows at these rates. Overall, the work to compute a time dependent solution with explicit time integration goes as

$$W \sim \begin{cases} N^4 & \textit{Advection dominated}, \\ N^6 & \textit{Diffusion dominated}. \end{cases} \tag{8.2}$$

On the other hand, if we subdivide into K subdomains, with only N/K order polynomials in each, then the work grows more slowly as

$$W \sim \begin{cases} K(\frac{N}{K})^4 & \textit{Advection dominated}, \\ K(\frac{N}{K})^6 & \textit{Diffusion dominated}. \end{cases} \tag{8.3}$$

For instance for a diffusion dominated problem, splitting a domain into only two subdomains reduces the work by a significant factor of $2/2^6 = 1/32$.

We need to balance the savings in work with the fact that the error will likely grow if the total number of points is fixed and the number of subdomains is increased. Roughly speaking, the error will vary with N and K as

$$E \sim \frac{e^{-\alpha N}}{K^{N+1}}. \tag{8.4}$$

So for smooth solutions where the error decays exponentially with polynomial order, the convergence rate is faster if we increase the polynomial order instead of the number of subdomains. We therefore need to balance accuracy and work to decide on the optimal strategy.

The solution itself may also direct us to use multidomain methods. When the solutions have sharp fonts, boundary layers, or other regions where local refinement is needed, along with other regions where the solution has little variation, we can localize the effort needed to compute the solution. We can use large regions with low order polynomials where the solution varies little, and place small subdomains with high order where the solution varies significantly.

8.1 Spectral Element Methods in One Space Dimension

To introduce the additional complexity of a multidomain method, we begin with a presentation of the spectral element method in one space dimension. In one dimension, we subdivide the interval into multiple non-overlapping subintervals (Fig. 8.6).

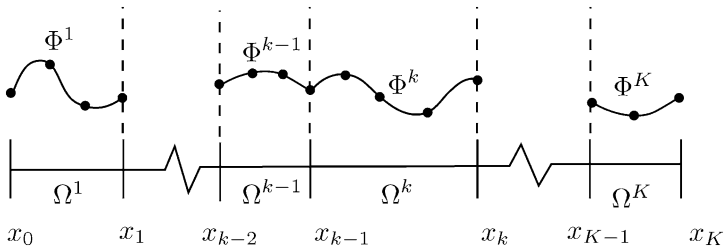

Fig. 8.6 Subdivision of the interval $[0, L] = [x_0, x_K]$ into K elements. The nodal Galerkin approximation is continuous and solutions are defined on Gauss-Lobatto points

Each interval will become an element. We couple an element to its neighbors at a single point, which we will call the element interface, or end point, in one space dimension. We will derive and implement two spectral element approximations. The first uses the continuous Galerkin approximation. The second uses the discontinuous version.

8.1.1 The Continuous Galerkin Spectral Element Method

As our first illustration of a spectral element method, we re-visit the approximation of the diffusion problem

$$\begin{aligned} \varphi_t &= \varphi_{xx} + f(x), \quad 0 < x < L, \\ \varphi(0,t) &= \varphi(L,t) = 0, \quad t > 0, \\ \varphi(x,0) &= \varphi_0(x), \quad 0 \le x \le L, \end{aligned} \tag{8.5}$$

that we solved with a single domain method on $[-1, 1]$ in Sect. 4.6.

As we did with the single domain approximation, we derive the continuous Galerkin spectral element method from a weak form of the equation. To get the weak form, we multiply the equation by a sufficiently smooth test function, ϕ, that satisfies the boundary conditions $\phi(0,t) = \phi(L,t) = 0$, is continuous, and has square integrable first and second derivatives. We then integrate over $[0, L]$ to get

$$\int_0^L \varphi_t \phi dx = \int_0^L \varphi_{xx} \phi dx + \int_0^L f \phi dx. \tag{8.6}$$

When we integrate the first integral on the right of (8.6) by parts and apply the boundary conditions,

$$\int_0^L \varphi_t \phi dx = -\int_0^L \varphi_x \phi_x dx + \int_0^L f \phi dx. \tag{8.7}$$

At this point, we would approximate the solution by a polynomial and replace the integrals by quadrature to derive a single domain spectral method, as we did in Sect. 4.6.

To derive the spectral element approximation, we subdivide the interval $[0, L]$ into K subintervals $\Omega^k = [x_{k-1}, x_k]$ (Fig. 8.6). These subintervals will be our spectral elements, e^k. We do not need to place any restrictions on the relative sizes of the elements.

Once we have subdivided the interval, we can break the integrals into sums of integrals over the elements

$$\sum_{k=1}^{K} \int_{x_{k-1}}^{x_k} \varphi_t \phi dx = -\sum_{k=1}^{K} \int_{x_{k-1}}^{x_k} \varphi_x \phi_x dx + \sum_{k=1}^{K} \int_{x_{k-1}}^{x_k} f \phi dx, \tag{8.8}$$

so that on each element the solution satisfies

$$\int_{x_{k-1}}^{x_k} \varphi_t \phi dx = -\int_{x_{k-1}}^{x_k} \varphi_x \phi_x dx + \int_{x_{k-1}}^{x_k} f \phi dx. \tag{8.9}$$

To use a Legendre polynomial approximation on each element, we map each element onto the reference interval $\xi \in [-1, 1]$, which will serve as our reference element, by the affine mapping

$$x = X^k(\xi) = x_{k-1} + \frac{\xi + 1}{2} \Delta x_k, \tag{8.10}$$

where $\Delta x_k = x_k - x_{k-1}$. Then

$$dx = \frac{\Delta x_k}{2} d\xi, \qquad \frac{\partial}{\partial x} = \frac{2}{\Delta x_k} \frac{\partial}{\partial \xi} \tag{8.11}$$

so that on the reference element the weak form of the PDE becomes

$$\frac{\Delta x_k}{2} \int_{-1}^{1} \varphi_t \phi d\xi = -\frac{2}{\Delta x_k} \int_{-1}^{1} \varphi_\xi \phi_\xi d\xi + \frac{\Delta x_k}{2} \int_{-1}^{1} f \phi d\xi. \tag{8.12}$$

To derive the continuous Galerkin spectral element method we approximate the solution by a piecewise continuous polynomial (Fig. 8.6). On each element, e^k, we write the polynomial in Lagrange form with nodes at the Legendre Gauss-Lobatto points,

$$\Phi^k(\xi, t) = \sum_{j=0}^{N} \Phi_j^k(t) \ell_j(\xi). \tag{8.13}$$

To keep the notation as simple as possible, we will keep the polynomial order N to be the same across all elements, but this is not necessary in practice. With N constant, the Lagrange interpolating polynomials are the same in each element. For

the approximation to be continuous and the first derivatives to be square integrable, we require that

$$\Phi_0^k = \Phi_N^{k-1}. \tag{8.14}$$

Similarly, we require that

$$\phi^k(\xi) = \sum_{j=0}^{N} \phi_j^k \ell_j(\xi), \qquad \phi_0^k = \phi_N^{k-1}. \tag{8.15}$$

Other than the continuity constraint, the values of ϕ_j^k are arbitrary since the weak form holds for any sufficiently smooth function ϕ.

We now replace each integral in (8.12) by Legendre Gauss-Lobatto quadrature. On the left,

$$\frac{\Delta x_k}{2} \int_{-1}^{1} \varphi_l \phi d\xi \approx \frac{\Delta x_k}{2} \sum_{j=0}^{N} w_j \dot{\Phi}_j^k \phi_j^k. \tag{8.16}$$

Since we are assuming that the same polynomial order is used in each element, the quadrature weights do not change with k. We approximate the last integral on the right of (8.12) in a similar manner. Finally, we follow the derivation of (4.117) to write the diffusion term in the compact form

$$\frac{2}{\Delta x_k} \int_{-1}^{1} \varphi_\xi \phi_x d\xi \approx \frac{2}{\Delta x_k} \sum_{j=0}^{N} \left\{ \sum_{m=0}^{N} \Phi_m^k G_{jm} \right\} \phi_j^k, \tag{8.17}$$

where

$$G_{jm} = G_{mj} = \sum_{l=0}^{N} w_l \ell_m'(\xi_l) \ell_j'(\xi_l) \tag{8.18}$$

is a symmetric matrix. When we gather the terms, we see that the solution satisfies the single domain approximation (4.118) element by element

$$\sum_{j=0}^{N} \left\{ w_j \frac{\Delta x_k}{2} \dot{\Phi}_j^k + \frac{2}{\Delta x_k} \sum_{m=0}^{N} \Phi_m^k G_{jm} - w_j \frac{\Delta x_k}{2} F_j^k \right\} \phi_j^k = 0. \tag{8.19}$$

By (8.8), we get the complete spectral element approximation when we sum over all of the elements

$$\sum_{k=1}^{K} \sum_{j=0}^{N} \left\{ w_j \frac{\Delta x_k}{2} \dot{\Phi}_j^k + \frac{2}{\Delta x_k} \sum_{m=0}^{N} \Phi_m^k G_{jm} - w_j \frac{\Delta x_k}{2} F_j^k \right\} \phi_j^k = 0. \tag{8.20}$$

We use (8.20) to derive the equations that the approximate values of the solution satisfy. Since the values of ϕ_j^k are independent except at the element boundaries, the

equations for interior node solution values depend only on others within their own element. If we take the test function ϕ_j^k to equal one at the single interior node ξ_j in element k and zero at all the others, we find that the interior node solutions satisfy

$$w_j \frac{\Delta x_k}{2} \dot{\Phi}_j^k = -\frac{2}{\Delta x_k} \sum_{m=0}^{N} \Phi_m^k G_{jm} + w_j \frac{\Delta x_k}{2} F_j^k,$$
$$j = 1, 2, \ldots, N-1; \; k = 1, 2, \ldots, K. \tag{8.21}$$

Elements couple because the solution and the test functions are continuous at the points shared by two elements. When we choose ϕ_j^k to equal one at the point shared by the kth and $k-1$st elements and zero elsewhere, then

$$w_N \frac{\Delta x_{k-1}}{2} \dot{\Phi}_N^{k-1} + w_0 \frac{\Delta x_k}{2} \dot{\Phi}_0^k = -\frac{2}{\Delta x_{k-1}} \sum_{m=0}^{N} \Phi_m^{k-1} G_{Nm} + w_N \frac{\Delta x_{k-1}}{2} F_N^{k-1}$$
$$- \frac{2}{\Delta x_k} \sum_{m=0}^{N} \Phi_m^k G_{0m} + w_0 \frac{\Delta x_k}{2} F_0^k. \tag{8.22}$$

When we impose the continuity constraint (8.14), $\Phi_0^k = \Phi_N^{k-1} \equiv \Phi_*^k$, we get the element boundary equations

$$\left[w_N \frac{\Delta x_{k-1}}{2} + w_0 \frac{\Delta x_k}{2} \right] \dot{\Phi}_*^k = -\frac{2}{\Delta x_{k-1}} \sum_{m=0}^{N} \Phi_m^{k-1} G_{Nm} + w_N \frac{\Delta x_{k-1}}{2} F_N^{k-1}$$
$$- \frac{2}{\Delta x_k} \sum_{m=0}^{N} \Phi_m^k G_{0m} + w_0 \frac{\Delta x_k}{2} F_0^k \tag{8.23}$$

for $k = 2, 3, \ldots, K-1$. Finally, we add the two boundary conditions $\Phi_0^1 = \Phi_N^K = 0$ to complete the system of equations that we integrate in time.

When we compare (8.21) to the single domain approximation (4.122), we see that we have the same equations to compute the interior and boundary points, except that now we have not yet divided by the quadrature weights and the element length can vary. The spectral element approximation differs only because we add the interior element boundary equations (8.23) that couple the elements. Notice that (8.22) is simply the sum of two equations (8.21), one from each element that shares the point. We will use this observation when we implement the approximations.

To integrate (8.21) and (8.23) in time, let us now use the second order implicit trapezoidal rule. Let's define

$$\left(G\Phi^k\right)_j \equiv \sum_{m=0}^{N} \Phi_m^k G_{jm}, \tag{8.24}$$

$$D_j^k \equiv -\frac{2}{\Delta x_k} \left(G\Phi^k\right)_j, \tag{8.25}$$

and let $\Phi_j^{k,n}$ denote the solution at point j in element k at time t_n. Then at interior points the solutions satisfy

$$w_j \frac{\Delta x_k}{2} \Phi_j^{k,n+1} - \frac{\Delta t}{2} D_j^{k,n+1} = w_j \frac{\Delta x_k}{2} \Phi_j^{k,n} + \frac{\Delta t}{2} D_j^{k,n} + \Delta t w_j \frac{\Delta x_k}{2} F_j^k$$
$$\equiv RHS_j^k. \tag{8.26}$$

To get the equations for the element interface points, we sum the contributions from neighboring elements, as we are instructed to do by (8.22) and (8.23).

We need to compute the residual to use Algorithm 80 (PreconditionedConjugateGradientSolve) to solve the system of equations at each time step. Let us define the local element residual at each node, including the endpoints, by

$$\tilde{r}_j^k = RHS_j^k - w_j \frac{\Delta x_k}{2} \Phi_j^k + \frac{\Delta t}{2} D_j^k, \quad j = 0, 1, \ldots, N;\ k = 1, 2, \ldots, K. \tag{8.27}$$

Since the global residuals at the interior element boundaries are just the sum of the local residuals at those points, we can compute the global residuals after we compute all the local residuals by

$$r_j^k = \begin{cases} \tilde{r}_j^k, & j = 1, 2, \ldots, N;\ k = 1, 2, \ldots, K, \\ \tilde{r}_N^k + \tilde{r}_0^{k+1}, & j = 0, N;\ k = 1, 2, \ldots, K, \\ 0, & k = 0, j = 0;\ k = K, j = N. \end{cases} \tag{8.28}$$

The boundary values on the last line are, of course, for Dirichlet boundary conditions. Algorithm 80 needs to be modified slightly because the limits on k start from one rather than zero, however we could accommodate this by a change of index in the data.

8.1.2 How to Implement the Continuous Galerkin Spectral Element Method

The first decision that we must make to evolve the nodal Galerkin approximation from a single domain to the spectral element approximation is to choose the data structures. We can choose between two extremes. One is to store all data locally. In this scheme, the solution, element size, Δx, etc. are stored by element within a structure or class. The other structure is "flat", where data is stored globally. The advantage of the local scheme is that the operations on the data look like the single domain approximations that we have developed in earlier chapters. The global scheme is useful if we want to use BLAS routines effectively. With the global implementation, we can use the Conjugate Gradient solver, Algorithm 80 (PreconditionedConjugateGradientSolve), at each time step to solve the symmetric system of equations that (8.26) and the boundary equations generate.

8.1.2.1 The Spectral Element Class

We will take a global view of the data with some organization to simplify local computations. In Sect. 8.1.4, where we will use explicit integration in time, we will show how to use local data structures. Much of the effort here about organization of the data in one space dimension is overkill, but it will prepare us for multidimensional problems where the effort is justified.

To use the iterative solver efficiently, we will store the solution of the diffusion problem in an array. Since we have assumed the same polynomial degree in each element and since Algorithm 80 has already been designed for two dimensional arrays, we will store the solution for each point on each element in a two dimensional array, $\{\Phi_{j,k}\}_{j=0;k=1}^{N;K}$. This array will be a member of a spectral element class, along with the data that defines the mesh, as we show in Algorithm 116 (SEM1DClass).

The array storage duplicates the solution at the interior element boundaries. It does, however, enable us to perform operations such as (8.24) locally on an element

Algorithm 116: *SEM1DClass:* Data Storage for the One-Dimensional Spectral Element Method

Class SEM1D
Data:
N, K
$\{\xi_j\}_{j=0}^{N}$; // Node locations
$\{w_j\}_{j=0}^{N}$; // Legendre Gauss-Lobatto quadrature weights
$\{x_k\}_{k=1}^{K}$; // Element boundary locations
$\{\Delta x_k\}_{k=1}^{K}$; // Element sizes
$\{G_{ij}\}_{i,j=0}^{N}$; // Derivative Matrix
$\{p_k\}_{k=1}^{K}$; // Shared node pointers
$\{\Phi_{j,k}\}_{j=0;k=1}^{N;K}$; // Solution
$\{RHS_{j,k}\}_{j=0;k=1}^{N;K}$; // For implicit integration
Procedures:
Construct$(N, K, \{x_k\}_{k=1}^{K})$
Mask$(\{a_{j,k}\}_{j=0;k=1}^{N,K})$; // Algorithm 117
UnMask$(\{a_{j,k}\}_{j=0;k=1}^{N,K})$; // Algorithm 117
GlobalSum$(\{a_{j,k}\}_{j=0;k=1}^{N,K})$; // Algorithm 117
LaplaceOperator; // Algorithm 118
MatrixAction$(s, \Delta t)$; // Algorithm 118
End Class SEM1D

Structure sharedNodePtrs
eLeft ; // index or pointer of element to the left
eRight ; // index or pointer of element to the right
nodeLeft ; // index of node to the left
nodeRight ; // index of node to the right
End Structure sharedNodePtrs

as if we were still doing a single domain computation. The fact that the element boundary equations are sums of the local element contributions from each side allows us to go back and "clean up" globally afterwards. We will store RHS_j^k and the residual r_j^k in the same kind of array as the solution.

We need a data structure to connect the elements. We assumed the simplest relationship between elements in Fig. 8.6, where elements are ordered from left to right. With that ordering we could assume that the continuity of the shared point is given by (8.14). In two dimensional problems we cannot assume such a simple ordering of elements. (See Fig. E.2 in Appendix E for a look ahead.) To allow for more general relationships, we create a structure that stores the values of k and j for the elements on the left and on the right that contribute to the solution at an interface (Algorithm 116, sharedNodePtrs).

We will store the sharedNodePtrs structures in an array, $\{p^k\}_{k=1}^{K-1}$, to mark the element interface points. When we need to do operations at interface points, we can access these points indirectly through the elements of this array. For the mesh that we show in Fig. 8.6, where elements are ordered left to right,

$$\begin{aligned} p^k.eleft &= k, \\ p^k.eRight &= k+1, \\ p^k.nodeLeft &= N, \\ p^k.nodeRight &= 0. \end{aligned} \tag{8.29}$$

With flat storage, we store all the data for the spectral element approximation, including the solutions, element relationships, element sizes, etc., in a single class or structure as we show in Algorithm 116 (SEM1DClass). We will not present a detailed algorithm for the constructor of this class since it looks much like constructors that we have presented earlier.

8.1.2.2 Global Operations

Before we can solve the system with the Conjugate Gradient solver, we must account for the fact that we have duplicated the solution and residuals at the element boundaries to perform the local computations as if on a single domain. One way to delete the duplicates is to copy the arrays to a new array that does not include them. We will use a mask instead. After we compute the residuals, we will simply set one of the duplicate solutions and the corresponding residuals to zero so that they have no contribution to the inner products or direction vectors in the Conjugate Gradient solver. Then to perform the local computations, we remove the mask by setting the solution value that has been zeroed to its duplicate. The number of masked points, $K-1$, is very small compared to the total number of degrees of freedom, $K \times N$, so the extra work to mask, unmask and evaluate masked quantities in the Conjugate Gradient algorithm is negligible. What we gain in return is an algorithm that is straightforward to implement.

We perform the global operations with the utility procedures *Mask*, *UnMask* and *GlobalSum*. It doesn't matter which duplicate we mask, so we will arbitrarily choose

Algorithm 117: *SEMGlobalProcedures1D:* Global Operations for the One-Dimensional Spectral Element Method

Uses Algorithms:
Algorithm 116 (SEM1DClass)
Procedure Mask
Input: $\left\{a_{j,k}\right\}_{j=0;k=1}^{N;K}$

for $k = 1$ **to** $K - 1$ **do**
  $jR \leftarrow this.p^k.nodeRight$; $eR \leftarrow this.p^k.eRight$
  $a_{jR,eR} \leftarrow 0$
end
return $\left\{a_{j,k}\right\}_{j=0;k=1}^{N;K}$
End Procedure Mask

Procedure UnMask
Input: $\left\{a_{j,k}\right\}_{j=0;k=1}^{N;K}$

for $k = 1$ **to** $K - 1$ **do**
  $jR \leftarrow this.p^k.nodeRight$; $eR \leftarrow this.p^k.eRight$
  $jL \leftarrow this.p^k.nodeLeft$; $eL \leftarrow this.p^k.eLeft$
  $a_{jR,eR} \leftarrow a_{jL,eL}$
end
return $\left\{a_{j,k}\right\}_{j=0;k=1}^{N;K}$
End Procedure UnMask

Procedure GlobalSum
Input: $\left\{a_{j,k}\right\}_{j=0;k=1}^{N;K}$

for $k = 1$ **to** $K - 1$ **do**
  $jR \leftarrow this.p^k.nodeRight$; $eR \leftarrow this.p^k.eRight$
  $jL \leftarrow this.p^k.nodeLeft$; $eL \leftarrow this.p^k.eLeft$
  $tmp \leftarrow a_{jR,eR} + a_{jL,eL}$
  $a_{jR,eR} \leftarrow tmp$
  $a_{jL,eL} \leftarrow tmp$
end
return $\left\{a_{j,k}\right\}_{j=0;k=1}^{N;K}$
End Procedure GlobalSum

the value on the right. We show the global operation procedures in Algorithm 117 (SEMGlobalProcedures1D). Each procedure uses the shared node array to perform its operation globally on an array of mesh values, which may be the solution or residual, for example.

8.1.2.3 The Diffusion Approximation

Both the right and left sides of the system (8.26) require us to compute a matrix-vector multiply to evaluate the contribution from the diffusion term of the equation.

We use Algorithm 57 (CGDerivativeMatrix) to compute the matrix G. We show the implementation of the diffusion term in the procedure *LaplaceOperator* in Algorithm 118 (SEM1DProcedures).

8.1.2.4 Side Operators and Residual Procedures

When we use the trapezoidal rule to integrate in time, the only difference between the matrices whose actions are given in (8.26) is the sign in front of the diffusion operator. For that reason, we will implement only one procedure for the matrix action in Algorithm 118 (SEM1DProcedures) and pass a sign variable with values ± 1 as we did with Fourier transforms. We set the residual to zero at the physical boundaries to implement Dirichlet conditions in the procedure *Residual* of Algorithm 118.

8.1.2.5 Iterative Solver

We need only make minor modifications to Algorithm 80 (PreconditionedConjugateGradientSolve). The first we have already mentioned, namely that the arrays now have different extents. The second is that the residual computation section will be replaced by the procedure *Residual* in Algorithm 118. We could still use a finite element preconditioner or not precondition at all.

8.1.2.6 The Time Integration Procedure

The trapezoidal rule integrator that we show in Algorithm 119 (TrapezoidalRuleIntegration) is straightforward to implement. At each time step we compute the *RHS* array and use Algorithm 80 to compute the new values. Since the solution values are no longer needed after *RHS* is evaluated, we need only one time level of storage for the solution, unlike if we were to use Algorithm 87 (MultistepIntegration). Note that as in Algorithm 87, we have hidden the boundary condition implementation in a procedure *SetBoundaryConditions* that needs to be provided. It will do nothing more than set the solution values at the boundaries.

8.1.3 Benchmark Solution: Cooling of a Temperature Spot

We present one benchmark example of the solution of (8.5) with the spectral element method. We have chosen the initial and boundary conditions so that the exact solution as a function of time describes a cooling Gaussian spot,

$$\varphi(x,t) = \frac{e^{-x^2/(4t+1)}}{\sqrt{4t+1}}. \tag{8.30}$$

Algorithm 118: *SEM1DProcedures:* Spatial Approximations for the One-Dimensional Spectral Element Method

Uses Algorithms:
Algorithm 116 (SEM1DClass)
Algorithm 19 (MxVDerivative)
Algorithm 117 (SEMGlobalProcedures1D)

Procedure LaplaceOperator
Input: $\{U_{j,k}\}_{j=0;k=1}^{N;K}$
$N \leftarrow this.N$; $K \leftarrow this.K$
for $k = 1$ **to** K **do**
  $\{D_j^k\}_{j=0}^{N} \leftarrow MxVDerivative(this.\{G_{i,j}\}_{i=0;j=0}^{N;N}, \{U_{j,k}\}_{j=0}^{N})$
  for $j = 0$ **to** N **do**
    $D_j^k \leftarrow -2 * D_j^k / \Delta x_k$
  end
end
return $\{D_j^k\}_{j=0;k=1}^{N;K}$
End Procedure LaplaceOperator

Procedure MatrixAction
Input: s, Δt, $\{U_{j,k}\}_{j=0;k=1}^{N;K}$
$N \leftarrow this.N$; $K \leftarrow this.K$
$\{U_{j,k}\}_{j=0;k=1}^{N;K} \leftarrow this.UnMask(\{U_{j,k}\}_{j=0;k=1}^{N;K})$
$\{AU_{j,k}\}_{j=0;k=1}^{N;K} \leftarrow this.LaplaceOperator(\{U_{j,k}\}_{j=0;k=1}^{N;K})$
for $k = 1$ **to** $K - 1$ **do**
  for $j = 0$ **to** N **do**
    $AU_{j,k} \leftarrow this.w_j/2 * this.\Delta x_k * U_{j,k} + s * \Delta t/2 * AU_{j,k}$
  end
end
$\{AU_{j,k}\}_{j=0;k=1}^{N;K} \leftarrow this.GlobalSum(\{AU_{j,k}\}_{j=0;k=1}^{N;K})$
$\{U_{j,k}\}_{j=0;k=1}^{N;K} \leftarrow this.Mask(\{U_{j,k}\}_{j=0;k=1}^{N;K})$
return $\{AU_{j,k}\}_{j=0;k=1}^{N;K}$
End Procedure MatrixAction

Procedure Residual
Input: *sem* ; `// Of type SEM1D`
$\{r_{j,k}\}_{j=0;k=1}^{N;K} \leftarrow this.MatrixAction(+1, \Delta t, sem.\{\Phi_{j,k}\}_{j=0;k=1}^{N;K})$
for $k = 1$ **to** K **do**
  for $j = 0$ **to** N **do**
    $r_{j,k} \leftarrow sem.RHS_{j,k} - r_{j,k}$
  end
end
$\{r_{j,k}\}_{j=0;k=1}^{N;K} \leftarrow this.Mask(\{r_{j,k}\}_{j=0;k=1}^{N;K})$
$r_{0,1} \leftarrow 0$
$r_{N,K} \leftarrow 0$
return $\{r_{j,k}\}_{j=0;k=1}^{N;K}$
End Procedure Residual

Algorithm 119: *TrapezoidalRuleIntegration:* Integration of the One-Dimensional Spectral Element Method in Time

Procedure TrapezoidalRuleIntegration
Input: N_T, N_{it}, TOL
Input: *sem* ; `// Of type SEM1D class`
Uses Algorithms:
Algorithm 116 (SEM1DClass)
Algorithm 117 (SEMGlobalProcedures1D)
Algorithm 118 (SEM1DProcedures)
Algorithm 80 (PreconditionedConjugateGradientSolve) //Modified

for $n = 0$ **to** $N_T - 1$ **do**
  $t \leftarrow n * \Delta t$
  $sem.\{RHS_{j,k}\}_{j=0;k=1}^{N;K} \leftarrow sem.MatrixAction(-1, \Delta t, sem.\{\Phi_{j,k}\}_{j=0;k=1}^{N;K})$
  $sem.\{\Phi_{j,k}\}_{j=0;k=1}^{N;K} \leftarrow SetBoundaryConditions(sem, t + \Delta t)$
  $sem \leftarrow PreconditionedConjugateGradientSolve(N_{it}, TOL, sem)$
end
$sem.\{\Phi_{j,k}\}_{j=0;k=1}^{N;K} \leftarrow sem.UnMask(sem.\{\Phi_{j,k}\}_{j=0;k=1}^{N;K})$
$t \leftarrow t + \Delta t$
return *sem*
End Procedure TrapezoidalRuleIntegration

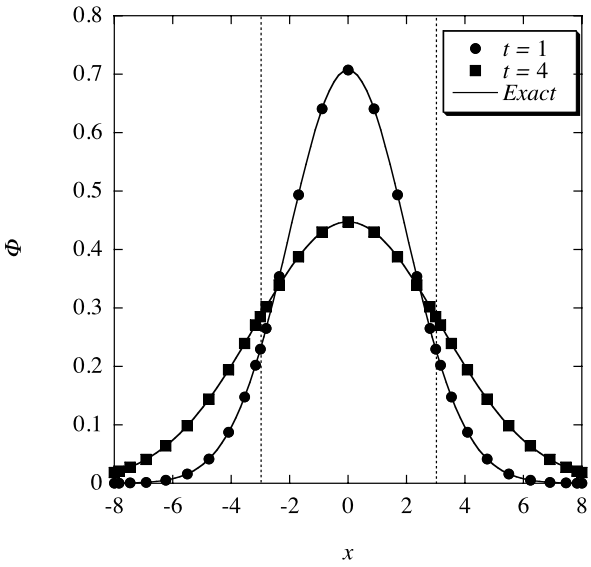

Fig. 8.7 Three element spectral element solution of the diffusion equation at two times. Element boundaries are marked with vertical dotted lines

We solve the equation on the interval $[-8, 8]$ with three elements $\Omega^1 = [-8, -3]$, $\Omega^2 = [-3, 3]$, and $\Omega^3 = [3, -8]$, $N = 10$, and $\Delta t = 0.05$. Figure 8.7 compares the computed and exact solutions at times $t = 1$ and $t = 4$. We could also make comparisons of multiple element approximations to the results of Sect. 4.6.

8.1.4 The Discontinuous Galerkin Spectral Element Method

We will motivate the development and implementation of the discontinuous Galerkin spectral element method with the approximation of the one-dimensional conservation law

$$\mathbf{q}_t + \mathbf{f}_x = 0, \quad x \in (0, L)\,. \tag{8.31}$$

As we discussed in Sect. 5.4, we can write the wave equation and others in conservation form.

Like the continuous spectral element approximation, we get the discontinuous approximation by dividing the interval into segments or elements (Fig. 8.8). Now, however, we will choose the nodes to be the Legendre Gauss points inside each element. More importantly, we will not assume that the solution is continuous at the element boundaries.

The spectral element approximation starts from the weak form of (8.31). After we multiply by a suitable test function, integrate, and subdivide into elements as we did in the previous section, the weak form is

$$\sum_{k=1}^{K} \left[\int_{x_{k-1}}^{x_k} (\mathbf{q}_t + \mathbf{f}_x)\, \phi dx \right] = 0. \tag{8.32}$$

On each element then,

$$\int_{x_{k-1}}^{x_k} (\mathbf{q}_t + \mathbf{f}_x)\, \phi dx = 0. \tag{8.33}$$

When we map (8.33) onto the reference element by the affine map (8.10),

$$\frac{\Delta x_k}{2} \int_{-1}^{1} \mathbf{q}_t \phi d\xi + \int_{-1}^{1} \mathbf{f}_\xi \phi d\xi = 0. \tag{8.34}$$

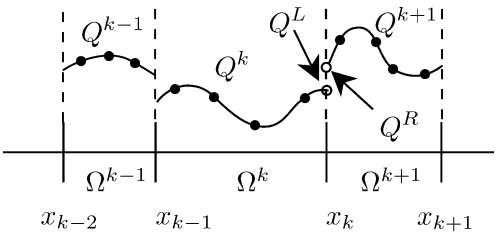

Fig. 8.8 Subdivision of the interval $[0, L] = [x_0, x_K]$ into K elements. The nodal discontinuous Galerkin approximation is not continuous at element boundaries and is defined on Gauss points

As usual for a nodal method, we approximate the solution and fluxes by polynomials in Lagrange form

$$\begin{aligned}\mathbf{q}\big(X^k(\xi)\big) &\approx \mathbf{Q}^k(\xi) = \sum_{j=0}^{N} \mathbf{Q}_j^k \ell_j(\xi),\\ \mathbf{f}\big(X^k(\xi)\big) &\approx \mathbf{F}^k(\xi) = \sum_{j=0}^{N} \mathbf{f}\big(\mathbf{Q}_j^k\big) \ell_j(\xi).\end{aligned} \tag{8.35}$$

Since the discontinuous Galerkin approximation does not enforce continuity at the element boundaries, we can take advantage of the higher precision of the Gauss rules and place the nodes at the Legendre Gauss points on the reference element. When we substitute the polynomial approximations into (8.12), the approximate solution satisfies

$$\frac{\Delta x_k}{2} \int_{-1}^{1} \mathbf{Q}_t \phi d\xi + \int_{-1}^{1} \mathbf{F}_\xi \phi d\xi = 0. \tag{8.36}$$

We write the test function ϕ also as a polynomial,

$$\phi = \sum_{j=0}^{N} \phi_j \ell_j(\xi). \tag{8.37}$$

The discontinuous Galerkin approximation does not require the test functions to be continuous at element boundaries either. Therefore to say that (8.33) holds for any piecewise continuous function ϕ is equivalent to saying that the ϕ_j's are independent of each other, without any constraints.

The approximation and test functions are continuous within any element. Therefore, we can integrate the second integral in (8.36) by parts to get

$$\frac{\Delta x_k}{2} \int_{-1}^{1} \mathbf{Q}_t \phi d\xi + \mathbf{F}\phi\big|_{-1}^{1} - \int_{-1}^{1} \mathbf{F}\phi_\xi d\xi = 0. \tag{8.38}$$

We've done the next two steps in the derivation of a nodal spectral approximation several times before. We replace the integrals by Gauss quadrature and use the fact that the ϕ_j's are independent. These steps lead us to

$$\frac{\Delta x_k}{2} \dot{\mathbf{Q}}_j + \mathbf{F} \frac{\ell_j}{w_j}\bigg|_{-1}^{1} - \sum_{n=0}^{N} \mathbf{F}_n \frac{w_n \ell_j'(\xi_n)}{w_j} = 0 \tag{8.39}$$

or, using the definition (4.139),

$$\frac{\Delta x_k}{2} \dot{\mathbf{Q}}_j + \left\{ \mathbf{F} \frac{\ell_j}{w_j}\bigg|_{-1}^{1} + \sum_{n=0}^{N} \mathbf{F}_n \hat{D}_{jn} \right\} = 0. \tag{8.40}$$

In the final step, we couple the elements. This step is peculiar to the discontinuous Galerkin approximation. We replace the fluxes at the element boundaries by the numerical flux $\mathbf{F}^* \cdot \hat{n}$ as we did in Sect. 5.4.1.1 at physical boundaries. Recall that the numerical flux is a function of two states, one to the left and one to the right, where we define the direction according to the normal at the boundary. At element boundaries, the left and right states are what we get when we evaluate the solution polynomial from the neighboring elements, as we show in Fig. 8.8. For the element ordering in Fig. 8.8, the numerical fluxes at the two element boundaries are

$$\begin{aligned} \mathbf{F}(-1) &\leftarrow \mathbf{F}^* \left(Q^{k-1}(1), Q^k(-1), -\hat{x} \right), \\ \mathbf{F}(+1) &\leftarrow \mathbf{F}^* \left(Q^k(1), Q^{k+1}(-1), +\hat{x} \right). \end{aligned} \tag{8.41}$$

At a physical boundary, we use the external state, just as in a single domain approximation.

When we replace the fluxes at the element boundaries by the numerical fluxes, we get the final version of the discontinuous Galerkin spectral element approximation for the kth element

$$\begin{aligned} \dot{\mathbf{Q}}_j^k + \frac{2}{\Delta x_k} \Bigg\{ & \mathbf{F}^* \left(Q^k(1), Q^{k+1}(-1), +\hat{x} \right) \frac{\ell_j(1)}{w_j} \\ & + \mathbf{F}^* \left(Q^{k-1}(1), Q^k(-1), -\hat{x} \right) \frac{\ell_j(-1)}{w_j} + \sum_{n=0}^{N} \mathbf{F}_n \hat{D}_{jn} \Bigg\} = 0. \end{aligned} \tag{8.42}$$

We make two observations about (8.42). First, except for the element size factor, $2/\Delta x_k$, the approximation is identical to the single domain approximation. Therefore we still compute the derivative approximation with Algorithm 92 (SystemDGDerivative). Second, the only coupling between elements occurs when we compute the boundary flux values. Therefore, we need only to supply the boundary values from the immediate neighbor to compute the time derivative of the solution.

8.1.5 How to Implement the Discontinuous Galerkin Spectral Element Method

The fact that the discontinuous Galerkin spectral element approximation (8.42) is virtually the same as the single domain approximation except for information to be set from the nearest neighbor elements suggests that we use a two level view for the implementation. The lowest level view will be that of an element on which the local operations such as the spatial derivative approximation are performed. The higher level view will be that of the mesh, which will keep track of the elements and perform global operations such as the computation of the numerical fluxes. To implement the two level view, we create the two classes *Element* and *Mesh* shown in

Algorithm 120: *DGSEM1DClasses:* Element and Mesh Definitions for the One-Dimensional Discontinuous Galerkin Spectral Element Method

Class Element
Uses Algorithms:
Algorithm 58 (NodalDiscontinuousGalerkin);

Data:
$\Delta x, x_L, x_R$; // Size and left and right boundaries of this element
dG ; // Of type NodalDiscontinuousGalerkin
$nEqn$; // # of equations in the physical system
$\{Q_{j,n}\}_{j=0;n=1}^{N;nEqn}$ $\{\dot{Q}_{j,n}\}_{j=0;n=1}^{N;nEqn}$; // Solution and time derivative
$\{G_{j,n}\}_{j=0;n=1}^{N;nEqn}$; // For low storage Runge-Kutta
$\left\{Q_n^L\right\}_{n=1}^{nEqn}, \left\{Q_n^R\right\}_{n=1}^{nEqn}$; // Solution on left/right element boundary
$\left\{F_n^{*,L}\right\}_{n=1}^{nEqn}, \left\{F_n^{*,R}\right\}_{n=1}^{nEqn}$; // Flux on left/right element boundary
Procedures:
Construct$(dG, nEqn, x_L, x_R)$; // See text.
InterpolateToBoundaries(); // Algorithm 121
LocalTimeDerivative(); // Algorithm 121
AffineMap(ξ); // Equation (8.10)
End Class Element

Class Mesh1D
Data:
K ; // # Elements
$\{e_k\}_{k=1}^{K}$; // Elements
$\{p^k\}_{k=0}^{K}$; // sharedNodePointers from Algorithm 116
Procedures:
Construct$(K, N, \{x_k\}_{n=0}^{K})$; // Algorithm 122
GlobalTimeDerivative(); // Algorithm 122
End Class Mesh1D

Algorithm 120 (DGSEM1DClasses). Again, the organization that we present here is overkill for one-dimensional problems, but it is helpful to see it on a simpler problem before moving to multidimensional problems.

8.1.5.1 The Elements

The element class (Algorithm 120) stores the element's geometry data and solution. The geometry data for the one dimensional elements are the left and right boundary locations and the length. To compute the spatial derivative in (8.42), the element needs access to the data stored in the *NodalDiscontinuousGalerkin* class (Algorithm 58). For simplicity, we show the Element class storing an instance of that class, though to do so clearly wastes storage if N is the same in each element. Even if N varies, there would likely only be a few unique values. In a large implementation, we would store the *NodalDiscontinuousGalerkin* instances in a collection (e.g.

a linked list discussed in Appendix E.1) and have the element only store a pointer to a member of that collection.

Other data includes the numerical fluxes, $F^{*,L/R}$, on the left and the right of the element and the interpolated values of the solution, $Q^{L/R}$, from which to compute the numerical flux. In practice, once we compute the numerical fluxes, we no longer need the interpolated values of the solution. The boundary solutions and fluxes could use the same storage. For clarity, however, we will store them separately. We also have the element store the number of equations rather than write a separate "physics" class. Finally, we store the time derivative with each element. If we were to modify the time integrator appropriately, we could reuse the storage of only one array.

The basic methods that the element needs to implement are also shown in Algorithm 120 (SEM1DClasses). We won't show the constructor explicitly since it only needs to set the local data. The interpolation and time derivative methods are implemented in Algorithm 121 (LocalDSEMProcedures). We have seen the use of the interpolation routines before in Algorithms 61 (InterpolateToBoundary) and 114

Algorithm 121: *LocalDSEMProcedures:* Local Procedures for the Discontinuous Galerkin Spectral Element Method

Procedure InterpolateToBoundaries
Uses Algorithms:
Algorithm 61 (InterpolateToBoundary);

$N \leftarrow this.dG.N$; $nEqn \leftarrow this.nEqn$
for $n = 1$ **to** $nEqn$ **do**
  $this.Q_n^L \leftarrow InterpolateToBoundary(this.\{Q_{j,n}\}_{j=0}^N, this.dG.\{\ell_j(-1)\}_{j=0}^N)$
  $this.Q_n^R \leftarrow InterpolateToBoundary(this.\{Q_{j,n}\}_{j=0}^N, this.dG.\{\ell_j(+1)\}_{j=0}^N)$
end
End Procedure InterpolateToBoundaries

Procedure LocalTimeDerivative
Uses Algorithms:
Algorithm 92 (SystemDGDerivative);

$N \leftarrow this.N$; $nEqn \leftarrow this.nEqn$
for $j = 0$ **to** N **do**
  $\{F_{j,n}\}_{n=1}^{nEqn} \leftarrow Flux(this.\{Q_{j,n}\}_{n=1}^{nEqn})$
end
$\left\{F'_{j,n}\right\}_{j=0;n=1}^{N;nEqn} \leftarrow$
$SystemDGDerivative\big(this.\left\{F_n^{*,L}\right\}_{n=1}^{nEqn}, this.\left\{F_n^{*,R}\right\}_{n=1}^{nEqn}, \{F_{j,n}\}_{j=0;n=1}^{N;nEqn}, this.dG.\{D_{i,j}\}_{i,j=0}^N,$
$this.dG.\{\ell_i(-1)\}_{i=0}^N, this.dG.\{\ell_i(1)\}_{i=0}^N, this.dG.\{w_i\}_{i=0}^N\big)$
for $j = 0$ **to** N **do**
  for $n = 1$ **to** $nEqn$ **do**
    $this.\dot{Q}_{j,n} \leftarrow -2F'_{j,n}/this.\Delta x$
  end
end
End Procedure LocalTimeDerivative

(MappedDGSystemTimeDerivative). The local time derivative procedure is of the form we have already seen in those two algorithms. We do not show an implementation of the *Flux* function, but it is similar to what we used in Algorithm 94 (WaveEquationFluxes). Finally, the *AffineMap* procedure merely implements (8.10) so we don't provide an implementation for it, either.

8.1.5.2 The Mesh

We manage global data at the mesh level. The mesh, also described in Algorithm 120 (DGSEM1DClasses), stores the number of elements, the elements themselves, and the connections between the elements. As in Sect. 8.1.1, the *sharedNodePointers* store pointers to the elements on the left and the right of an interface. To simplify the presentation, we will assume here that $x_R > x_L$ so that the Q^L and Q^R arrays are on the left and right of the elements. That way we do not have to store information in the *sharedNodePointers* to distinguish between which corresponds to the left and which to the right. When we go to two dimensional problems later, we will have to be more general.

We use the constructor for the mesh class to create the elements and connections. The constructor will take the number of elements and the location of the element boundaries as input. It constructs an instance of the *NodalDiscontinuousGalerkin* class and uses that and the element boundary information to construct the elements. The element connections are constructed next. Since there is essentially no difference between a physical boundary and an element boundary, the limits on the p^k array include the endpoints. At the physical boundaries we set the neighboring element to a defined constant *NONE*. Later, we can test for being on a boundary by checking to see if one of the elements equals *NONE*.

The global time derivative procedure in Algorithm 122 (GlobalMeshProcedures) performs four basic operations. First, it interpolates the solutions to the boundaries on each of the elements. It then computes the physical boundary values by way of a procedure *ExternalState* whose implementation is problem dependent. We pass a parameter with defined values of *LEFT* or *RIGHT* to the procedure so that different boundary conditions can be applied at the left and right boundaries. We also pass the boundary value of the solution to allow reflection boundary conditions like (5.168) to be implemented. Then the numerical fluxes are computed for each element boundary point and sent to the appropriate element. Again, we have assumed that the elements are laid out left to right. Finally, each element computes its local time derivative values.

8.1.5.3 Time Integration

We can easily modify the third order explicit Runge-Kutta algorithm Algorithm 62 (DGStepByRK3) to accommodate the spectral element approximation. We first change the inputs to the procedure to be an instance of the *Mesh* class, the time

Algorithm 122: *GlobalMeshProcedures:* Mesh Global Procedures for the Discontinuous Galerkin Spectral Element Approximation

Procedure Construct
Input: $K, N, \{x_k\}_{n=0}^{K}$
Uses Algorithms:
Algorithm 120 (SEM1DClasses)
Algorithm 58 (NodalDiscontinuousGalerkin)

$this.K \leftarrow K$
$dG.Construct(N)$; `// Of type NodalDiscontinuousGalerkin`
for $k = 1$ **to** $this.K$ **do**
  $this.e^k.Construct(dG, nEqn, x_{k-1}, x_k)$
end
for $k = 1$ **to** $K - 1$ **do**
  $this.p^k.eLeft \leftarrow k$
  $this.p^k.eRight \leftarrow k + 1$
end
$this.p^0.eLeft \leftarrow NONE$
$this.p^0.eRight \leftarrow 1$
$this.p^K.eLeft \leftarrow K$
$this.p^K.eRight \leftarrow NONE$
End Procedure Construct

Procedure GlobalTimeDerivative
Input: t
Uses Algorithms:
Algorithm 88 (RiemannSolver)

for $k = 1$ **to** $this.K$ **do**
  $this.e^k.InterpolateToBoundaries()$
end
$k \leftarrow this.p^0.eRight$
$\{Q_n^{ext,L}\}_{n=1}^{nEqn} \leftarrow ExternalState(this.e^k.\{Q_n^L\}_{n=1}^{nEqn}, this.e^k.AffineMap(-1), LEFT)$
$k \leftarrow this.p^K.eLeft$
$\{Q_n^{ext,R}\}_{n=1}^{nEqn} \leftarrow ExternalState(this.e^k.\{Q_n^R\}_{n=1}^{nEqn}, this.e^k.AffineMap(+1), RIGHT)$
for $k = 0$ **to** $this.K$ **do**
  $idL \leftarrow this.p^k.eLeft$; $idR \leftarrow this.p^k.eRight$
  if $idL = NONE$ **then**
    $this.e^{idR}.\{F_n^L\}_{n=1}^{nEqn} \leftarrow RiemannSolver(\{Q_n^{ext,L}\}_{n=1}^{nEqn}, this.e^{idR}.\{Q_n^L\}_{n=1}^{nEqn}, -1)$
  else if $idR = NONE$ **then**
    $this.e^{idL}.\{F_n^R\}_{n=1}^{nEqn} \leftarrow RiemannSolver(this.e^{idL}.\{Q_n^R\}_{n=1}^{nEqn}, \{Q_n^{ext,R}\}_{n=1}^{nEqn}, +1)$
  else
    $\{F_n^L\}_{n=1}^{nEqn} \leftarrow RiemannSolver(this.e^{idL}.\{Q_n^R\}_{n=1}^{nEqn}, this.e^{idR}.\{Q_n^L\}_{n=1}^{nEqn}, +1)$
    $this.e^{idR}.\{F_n^L\}_{n=1}^{nEqn} \leftarrow -\{F_n^L\}_{n=1}^{nEqn}$
    $this.e^{idL}.\{F_n^R\}_{n=1}^{nEqn} \leftarrow \{F_n^L\}_{n=1}^{nEqn}$
  end
end
for $k = 1$ **to** $this.K$ **do**
  $this.e^k.LocalTimeDerivative()$
end
End Procedure GlobalTimeDerivative

and the time step. Next, the time derivative computation is performed by the global mesh procedure *GlobalTimeDerivative* in Algorithm 122 (GlobalMeshProcedures). Finally, we update the solution element by element by replacing

$$\begin{array}{l}\textbf{for } j = 0 \textbf{ to } N \textbf{ do}\\ \quad G_j \leftarrow a_m G_j + \dot{\Phi}_j^n\\ \quad \Phi_j^{n+1} \leftarrow \Phi_j^{n+1} + g_m \Delta t G_j\\ \textbf{end}\end{array} \tag{8.43}$$

with

$$\begin{array}{l}\textbf{for } k = 1 \textbf{ to } K \textbf{ do}\\ \quad \textbf{for } j = 0 \textbf{ to } N \textbf{ do}\\ \quad\quad \textbf{for } n = 1 \textbf{ to } nEqn \textbf{ do}\\ \quad\quad\quad mesh.e^k.G_{j,n} \leftarrow a_m * mesh.e^k.G_{j,n} + mesh.e^k.\dot{Q}_{j,n}\\ \quad\quad\quad mesh.e^k.Q_{j,n} \leftarrow mesh.e^k.Q_{j,n} + g_m * \Delta t * mesh.e^k.G_{j,n}\\ \quad\quad \textbf{end}\\ \quad \textbf{end}\\ \textbf{end}\end{array} \tag{8.44}$$

where *mesh* is the mesh class instance.

8.1.6 Benchmark Solution: Wave Propagation and Reflection

We now solve the one-dimensional wave equation in system form

$$\begin{bmatrix} u \\ v \end{bmatrix}_t + \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}_x = \begin{bmatrix} u \\ v \end{bmatrix}_t + \begin{bmatrix} v \\ u \end{bmatrix}_x = 0, \quad x \in [0, L] \tag{8.45}$$

with a reflection boundary at $x = 0$ and a nonreflecting boundary at $x = L$ as the benchmark problem to illustrate the use of the discontinuous Galerkin spectral element approximation on a conservation law. We can use the transformation matrix

$$S = \frac{1}{2} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{8.46}$$

and its inverse to convert the system to the decoupled equation

$$\begin{bmatrix} w^+ \\ w^- \end{bmatrix}_t + \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} w^+ \\ w^- \end{bmatrix}_x = 0 \tag{8.47}$$

to see that the system has a leftgoing and a rightgoing wave, each of which moves with unit speed. The wave components are

$$\begin{aligned} w^+ &= u + v, \\ w^- &= u - v. \end{aligned} \tag{8.48}$$

In terms of the wave variables, the reflection boundary condition at the left is $w^+ = w^-$ and the non-reflection boundary condition at the right is $w^- = 0$.

To implement the discontinuous Galerkin spectral element method, we need to provide the numerical flux, which for this problem is

$$F^*\left(Q^L, Q^R; \hat{n}\right) = \frac{1}{2}\begin{bmatrix} u^L - u^R + v^L + v^R \\ u^L + u^R + v^L - v^R \end{bmatrix} \hat{n}. \tag{8.49}$$

We implement the boundary condition on the left by specifying an external state with $u^L = u^R$ and $v^L = -v^R$. We use the exact solution as the external state on the right.

The initial condition to the benchmark problem is

$$\begin{aligned} u &= 2^{-(x-1)^2/b^2}, \\ v &= 0 \end{aligned} \tag{8.50}$$

with $b = 0.15$. We derive the exact solution using the methods of characteristics and images.

Figure 8.9 shows the computed and exact solutions for five elements of equal size on the interval $[0, 5]$. For benchmark purposes, the other parameters for the computation were $N = 14$ and $\Delta t = 4 \times 10^{-2}$.

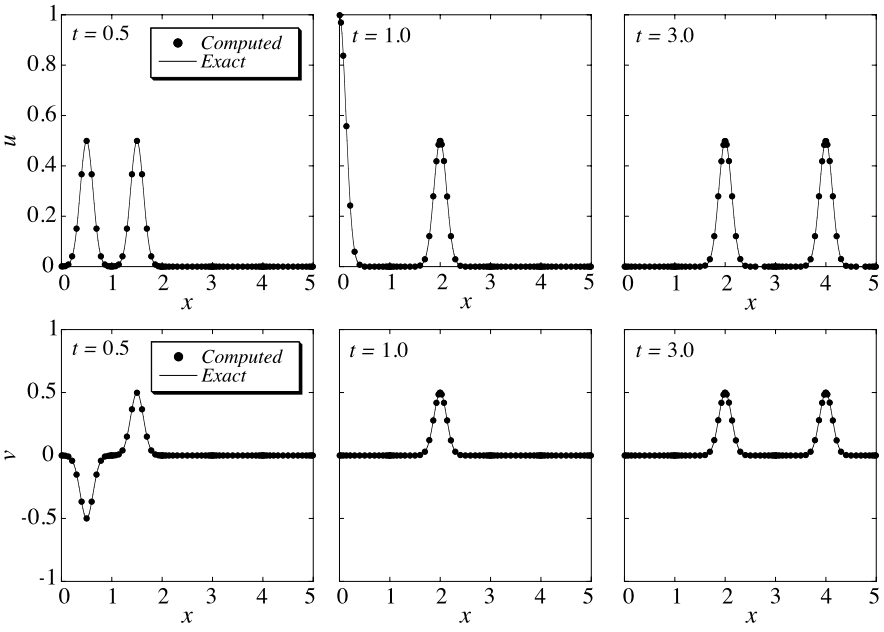

Fig. 8.9 Discontinuous Galerkin spectral element solution of the wave equation showing propagation and reflection of a wave with a five element approximation. A Gaussian spot with initial maximum at $x = 1$ splits into a leftgoing and rightgoing wave. The leftgoing wave reflects at the left boundary and then propagates to the right

8.2 The Two-Dimensional Mesh and Its Specification

In two space dimensions, we divide the physical domain into non-overlapping quadrilaterals like we have sketched in Fig. 8.10. These quadrilaterals will be our elements and the collection of elements is the mesh. The elements can have straight or curved sides, just like a single domain. Within each element we will place an $N \times N$ set of nodes—Gauss-Lobatto or Gauss, depending on the spatial approximation—at which we will approximate the solution. To avoid confusion, we explicitly refer to the set of nodes within an element as a *grid* and to the set of elements as the *mesh*.

The elements do not have to be tiled in any regular pattern or numbered in any particular order. In other words, the mesh can be *unstructured*. The only absolute restriction that we must place on the elements is that they each must have a shape that we can map onto the reference square like we did in Chap. 6. There are, however, other issues of *mesh quality* that affect the accuracy of solutions. Mesh quality measures are not as developed yet for spectral methods as they are for low order finite element methods. For a discussion of mesh quality, we defer to more technical books like [20]. As a rule of thumb, we try to keep the angles in the mesh to be as close to 90 degrees as possible.

To make the methods easier to implement, we will make one restriction on how the elements tile a domain. We will require that neighboring elements share either an entire side or a corner point. Such a mesh is called *geometrically conforming*. Figures 8.2, 8.3b, and 8.4 show examples of geometrically conforming meshes. We contrast a conforming mesh to nonconforming meshes in Fig. 8.11. The geometrically nonconforming mesh of Fig. 8.11b has elements that share a partial side. The mesh of Fig. 8.11c is geometrically conforming, but has different order polynomials on each side so that the nodes do not match across the element boundaries. Such meshes are *functionally nonconforming*. Of course, a mesh could be both geometrically and functionally nonconforming. Nonconforming meshes are sometimes easier to generate, particularly if we want to refine the mesh locally, but they will lead

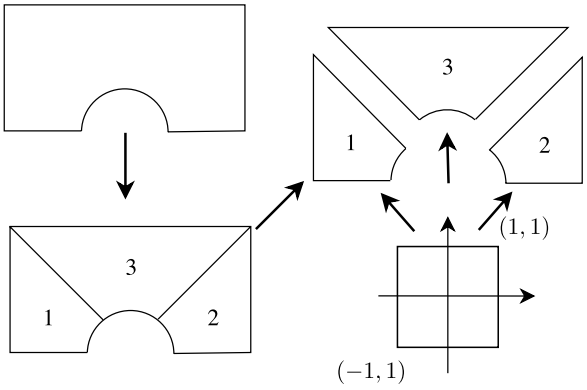

Fig. 8.10 Subdivision of a domain into quadrilateral elements, which are mapped individually from the reference square

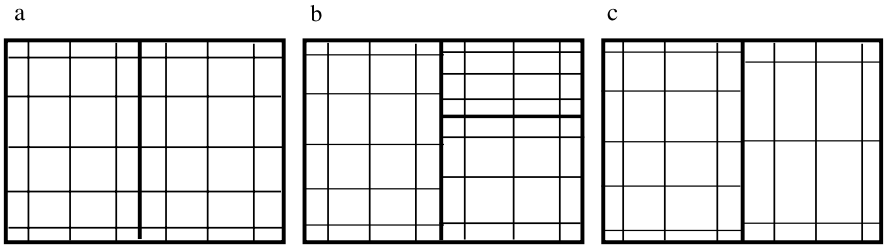

Fig. 8.11 A comparison of a conforming mesh (**a**) to a geometrically nonconforming mesh (**b**) and a functionally nonconforming mesh (**c**). Within each element of the meshes, we show grids of Gauss-Lobatto nodes at which a spectral element approximation might be approximated

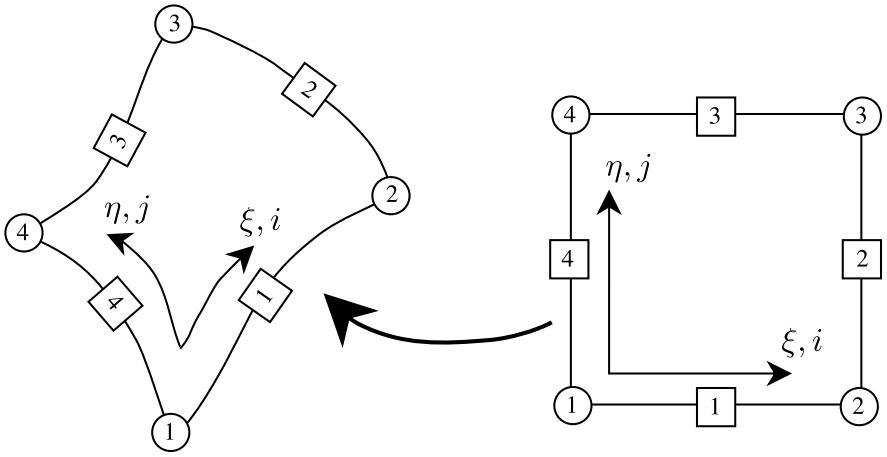

Fig. 8.12 Location of boundary curves and corner nodes that define an element shown in physical space and on the reference square

to more complex spatial approximations. Spectral approximations can be derived for both geometrically and functionally nonconforming meshes. For a discussion of how, see [8].

Let us start our specification of the mesh with the local definition of an element. An element in two dimensions is nothing but a quadrilateral domain that we used in Chap. 7. Therefore, we need four bounding curves to specify the geometry of an element. If a side is straight, then it is sufficient to specify only the two endpoints of the curve, which will be two *corner nodes* of the element. In finite element applications, the boundary curves are called *edges* and the corner nodes are called *nodes*. Since we will also have nodes associated with the Gauss or Gauss-Lobatto grids within each element, we make a distinction here between the two types. To fully define an element, we will now specify the four boundary curves and the four corner nodes, numbered counter clockwise, as shown in Fig. 8.12. We will use the corner nodes to determine how the elements are coupled.

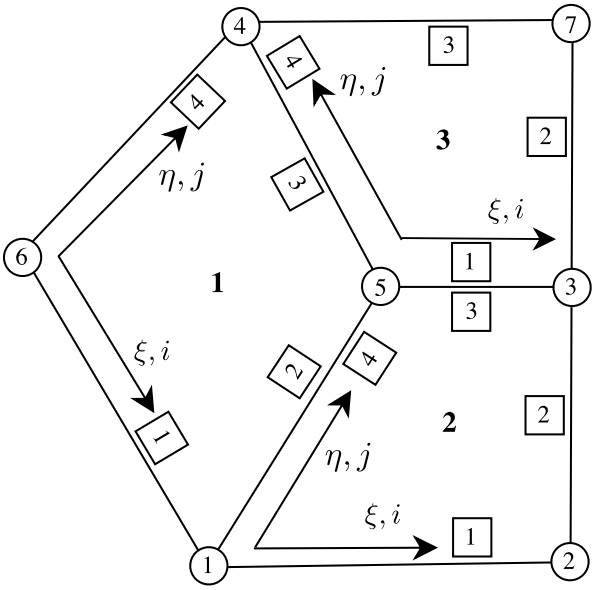

Fig. 8.13 A three element mesh that shows the local element structure constructed from seven globally numbered corner nodes (*circles*). Element sides and grid orientation are located as in Fig. 8.12

We need to know how elements are coupled if the mesh has more than one. Since we have restricted ourselves to conforming meshes, an element is coupled to a neighbor either along a boundary curve or at a corner node. What we need, then, is a list of which elements share a common side and a list of which elements share a common corner node. We can generate both of these lists (see the examples in Appendix E) if we are given a globally numbered list of corner nodes and a list of which corner nodes are used to construct each element. The list of nodes and the list of element connectivity, as the latter is known, are standard output of finite element mesh generators.

To see how to build a mesh, let us practice with the three elements shown in Fig. 8.13. The mesh has seven globally numbered corner nodes from which we construct the three elements. (If the sides were not straight, we would also have to include boundary curve information.) The elements are defined by four corners numbered counterclockwise. The choice of the first node is arbitrary, but once we choose it, we have specified the element topology (Fig. 8.12). For the mesh in Fig. 8.13, we have chosen the element connectivity as shown in Table 8.1.

The elements couple through their corner nodes and their sides. At the corners, we will need to know the elements that contribute and the location in the local nodal grid within these elements. This information is the equivalent to what was stored in the *SharedNodePointer* that we introduced in the previous section. We will use the information similarly to mask, unmask and do global sums on the solution. The

Table 8.1 Element definitions for Fig. 8.13

Element	Node 1	Node 2	Node 3	Node 4
1	6	1	5	4
2	1	2	3	5
3	5	3	7	4

Table 8.2 Corner node connectivity and local grid index

Node	Element	i	j
1	2	0	0
	1	N	0
2	2	N	0
3	2	N	N
	3	N	0
4	1	0	N
	3	0	N
5	1	N	N
	2	0	N
	3	0	0
6	1	0	0
7	3	N	N

corner connectivity is expressed by Table 8.2. Note each corner node can have a different number of adjacent elements.

The elements are also connected along their sides. We index the sides by an ordered pair of two corner nodes, which we will call an *edge*, even if the sides are not straight. When we look at Fig. 8.13, we see what information we should extract to be able to perform mask, unmask and global sum operations. First, we should find which elements border a side. If only one does, then we conclude that edge is along a boundary. Next, we need to know which sides of the neighboring elements are adjacent so that we know which part of the local element grids are connected. The last piece of information concerns the direction in which the local grid index varies along the side. For instance, along the edge indexed by Nodes 4 and 5, Side 3 of Element 1 and Side 4 of Element 3 are the neighbors. The order in which we list the two neighbors is arbitrary, but we have to choose something. Let us say that Element 1/Side 3 is the first (primary) and Element 3/Side 4 is the second (secondary). Then the nodes of Element 1 that lay along the edge are those with index (i, N) for $i = 1, 2, \ldots, N-1$ in order. We don't include the $i = 0, N$ nodes because they are corner nodes of the element. The nodes of Element 3 are those with index (N, j) with j running in reverse order, $j = N-1, N-2 \ldots, 1$. We need to flag this situation. The approach that we will take here is to set the sign of the secondary element side to be negative if the index runs in the direction opposite of the primary side. If we remember the starting index ($N-1$ or 1) and

Table 8.3 Edge information for the mesh in Fig. 8.13

Edge	Node 1	Node 2	Element 1	Side 1	Element 2	Side 2	Start	Inc.
1	6	1	1	1				
2	1	5	1	2	2	4	1	1
3	5	4	1	3	3	−4	$N-1$	−1
4	4	6	1	4				
5	1	2	2	1				
6	2	3	2	2				
7	3	5	2	3	3	1	1	1
8	3	7	3	2				
9	7	4	3	3				

the increment (± 1), we can simplify the logic when we perform computations along element edges. The edge data of the mesh in Fig. 8.13 is shown in Table 8.3. Note that the ordering of the edges is arbitrary.

8.2.1 How to Construct a Two-Dimensional Mesh

Although we can construct the node, edge, and element information by hand for a simple three element mesh just as we've done for the mesh in Fig. 8.13, it gets tedious and error prone very fast as the number of elements grows. It is better to automate the process as much as possible. How much we have to automate depends on what mesh generators we have available. For the purposes of this book, we have decided to choose the lowest common denominator so that simple mesh files can be created by hand. The minimum information needed is a list of corner node locations and a list of elements, with each element defined by its four corner nodes. To allow sides to be curves, we will include boundary curves that are constructed using Algorithm 96 (CurveInterpolant). We would normally read node, element, and boundary curve information from a file.

8.2.1.1 Nodes

The first data structure that we need to define is a corner node. At a minimum, a corner node stores its (x, y) values. To define the element connectivity so that we can perform mask, unmask and sum operations at nodes, we also need a list of what elements share the node. Since different numbers of elements can share a common node in an unstructured mesh, it is better to store this information in a linked list than in an array of fixed size. (See Appendix E.) Most mesh generators will try to keep this number, called the *valence*, low, typically less than or equal to six. So we could reasonably choose to store the adjacent element information in a

Algorithm 123: *CornerNodeClass:* Corner Node for Two-Dimensional Spectral Element Methods

Class CornerNode
Uses Algorithms:
Algorithm 144 (LinkedList)
Data:
type ; // Kind of node—INTERIOR or BOUNDARY
x, y ; // location
nodeConnectivity ; // Linked list of type CornerConnectivity
Procedures:
Construct(x, y)
End Class CornerNode

Procedure Construct
Input: x, y
$this.x \leftarrow x$; $this.y \leftarrow y$
$this.type \leftarrow INTERIOR$
nodeConnectivity.Construct()
End Procedure Construct

Structure CornerConnectivity
id, i, j
End Structure CornerConnectivity

fixed array of length six instead of a linked list. However, since we will not have to search for a particular element in this list, we will use the linked list. Finally, for convenience, we will also store the type of node, either boundary or interior, to help when we set boundary conditions. We show a corner node definition in Algorithm 123 (CornerNode).

A mesh file will typically contain a sequence of (x, y) locations that correspond to corner nodes. As each is read, we construct the node with the procedure *Construct* in Algorithm 123, which simply stores the location of the node and a default value of the type of node. We number the nodes with a node number/identifier, *id*, according to their order in the file, and will access them by their location in an array of nodes stored by the mesh. We will construct the list of adjacent elements, which is the data we collected for Table 8.2, after we construct the elements and edges.

8.2.1.2 Elements

An element is usually defined in a mesh file by an array of the *id*'s of its four corner nodes, ordered counter-clockwise. This is the data that we gathered in Table 8.1 for our three element example. For straight sided elements, the location of the four corner nodes is enough information to compute the element's geometry using Algorithm 95 (QuadMap) and metric terms using Algorithm 100 (QuadMapMetrics). For curved sides, we need additional information to define the curves.

Algorithm 124: *QuadElementClass:* Quadrilateral Element Definition for Two-Dimensional Spectral Element Methods

Class QuadElement
Uses Algorithms:
Algorithm 101 (MappedGeometry)
Algorithm 63 (Nodal2DStorage)
Data:
$\{nodeIds_j\}_{j=1}^4$; `// Corner node id's in node array.`
geom ; `// MappedGeometry to store metrics, etc.`
Procedures:
Construct(*spA*, $\{nodeIds_j\}_{j=1}^4$, $\{\mathbf{\Gamma}_j\}_{j=1}^4$)
End Class QuadElement

Procedure Construct
Input: $\{nodeIds_j\}_{j=1}^4$
Input: $\{\mathbf{\Gamma}_j\}_{j=1}^4$; `// CurveInterpolant`
Input: *spA* ; `// Nodal2DStorage`
this.$\{nodeIds_j\}_{j=1}^4 \leftarrow \{nodeIds_j\}_{j=1}^4$
this.geom.Construct(*spA*, $\{\mathbf{\Gamma}_j\}_{j=1}^4$)
End Procedure Construct

This is information not typically provided by finite element mesh generators. We usually specify Chebyshev polynomial interpolants only for those sides that are curved. From the four curves, we will store the grid and metric arrays in our standard *MappedGeometry* structure of Algorithm 101 (MappedGeometryClass). We show the storage that we need to define an element in Algorithm 124 (QuadElementClass). The constructor takes the array that lists the four corner nodes and, as we did to define a quadrilateral single domain, an array of the four boundary curves.

8.2.1.3 Edges

We will use an edge class to store the information along a row in Table 8.3. Our implementation of an edge is Algorithm 125 (EdgeClass). The constructor is simple. It takes the *id*'s of two nodes, an element of which the edge is a side, and the number of the side. We will find the identity of any additional elements that may share that side later as we construct the mesh.

8.2.1.4 The Mesh

Lastly, we need to define our *Mesh* data structure. The mesh will store the nodes, the elements and the edges. Since we are going to assume that the mesh is conforming, we will also assume that the polynomial order in all elements will be

Algorithm 125: *EdgeClass:* Edge Definition for Two-Dimensional Spectral Element Methods

Class Edge
Data:
$type$; // Kind of edge—interior or boundary
$\{nodes_k\}_{k=1}^2$; // start and end node id's
$\{elementIDs_k\}_{k=1}^2$; // Elements that share this edge
$\{elementSides_k\}_{k=1}^2$; // Sides of Elements that share this edge
$start, inc$; // Loop start and increment for the secondary side
Procedures:
$Construct(\{nodes_k\}_{k=1}^2, elementID, side)$
End Class Edge

Procedure Construct
Input: $\{nodes_k\}_{k=1}^2, elementID, side$
$this.\{nodeIds_j\}_{j=1}^2 \leftarrow \{nodeIds_j\}_{j=1}^2$
$this.type \leftarrow INTERIOR$
$this.elementIDs_1 \leftarrow elementID$
$this.elementIDs_2 \leftarrow NONE$
$this.elementSides_1 \leftarrow side$
$this.elementSides_2 \leftarrow NONE$
End Procedure Construct

the same. Therefore, we need to store only one instance of a *Nodal2DStorage* object to hold the quadrature nodes, weights, and the derivative matrices in the mesh structure. Since the number of elements and the number of corner nodes are usually listed in, or can be determined from, the mesh file, we store an array of CornerNodes and an array of Elements in the mesh structure. Finally we will store three convenience arrays that we will describe below to help navigate local data structures. We show the structure for the *Mesh* class in Algorithm 126 (QuadMesh).

We do not know the number of edges beforehand. We must construct the edges from the elements and the nodes, so we will not know how many there are to start. To be completely general, we should store the edges in some dynamic data structure like a Linked List, which can have an arbitrary length. We could use Algorithm 144 (LinkedList) in Appendix E to store and manipulate the edge list. However, we can simplify our presentation here significantly if we store the edges in an array, with the edge *id* denoted by the location in the array.

To use a fixed size array to store the edges, we need to find a reasonable upper bound on the number of edges that the mesh can have. A result from algebraic topology tells us that the number of edges, N_{edge}, the number of elements, K, and the number of nodes, N_{node}, are related to the *Euler characteristic*, χ, by the relation

$$\chi = N_{node} + K - N_{edge}. \tag{8.51}$$

Algorithm 126: *QuadMesh:* Mesh Definition for Two-Dimensional Spectral Element Methods

Class QuadMesh
Uses Algorithms:
Algorithm 63 (Nodal2DStorage)
Data:
K, N_{node}, N_{edge} ; // # Elements, corner nodes, and edges
$\{elements_k\}_{k=1}^{K}$; // Array of Elements
$\{nodes_k\}_{k=1}^{N_{node}}$; // Array of CornerNodes
$\{edges_k\}_{k=1}^{edgeDim}$; // Array of Edges
$\{cornerMap_{i,j}\}_{i=1,j=1}^{2,4}$; // Convenience array
$\{sideMap_i\}_{i=1}^{4}$; // Convenience array
$\{edgeMap_{i,j}\}_{i,j=1}^{2,4}$; // Convenience array
Procedures:
Construct($spA, meshFile$); // Algorithm 127
End Class QuadMesh

In turn, the Euler characteristic is related to the number of holes, N_h, in the mesh by

$$\chi = 1 - N_h. \tag{8.52}$$

(Try it on the decompositions shown in Figs. 8.1 and 8.3b.) Of course, we don't know the number of holes in the mesh simply by looking at the nodes and element definitions, so we will only try to find a reasonable upper bound. For a "reasonable" mesh without pinched holes, we can take $N_h \leq N_{node}/3$. In that case, we expect that

$$N_{edge} \leq K + \frac{4}{3}N_{node} - 1 \equiv edgeDim. \tag{8.53}$$

The number of holes that we've assumed is large compared to the number of nodes. Typically there will be only a few holes in a mesh. The penalty, though, is only 30% of the number of nodes in the mesh. Note, however, that we can come up with pathological and unlikely meshes that will have more edges. If we expect a large number of such cases, we should switch to a linked list or other dynamic data structure to store the edges.

We will also store three convenience arrays with the mesh. The first is the *sideMap*. We will use it to tell us what the fixed index value is along a given side. For instance, by our definition shown in Fig. 8.12, Side 1 corresponds to $j = 0$ and varying i, Side 2 corresponds to $i = N$ and varying j, etc. Therefore the four values of the *sideMap* will be $\{0, N, N, 0\}$. The second convenience array is the *cornerMap* that will tell us the values of the local grid indices for the four corners. Looking back again at Fig. 8.12, we see that Corner 1 corresponds to $i = j = 0$ and Corner 2 is $i = N$, $j = 0$, etc. The *cornerMap* array is $\{\{0, 0\}, \{N, 0\}, \{N, N\}, \{0, N\}\}$. Finally, we define the *edgeMap* array to make the correspondence between an ele-

ment side and the two CornerNodes that start and terminate the side. For instance, in Fig. 8.12 we see that Side 1 is constructed from CornerNodes 1 and 2, whereas we construct Side 2 from CornerNodes 2 and 3. The elements of the *edgeMap* are $\{\{1,2\},\{2,3\},\{4,3\},\{1,4\}\}$.

We show the procedure to construct the mesh in Algorithm 127 (QuadMesh: Construct). As input, it takes an already constructed *Nodal2DStorage* object and a mesh file to read. After it constructs the convenience arrays, it reads and constructs the nodes and elements from the mesh file. Once the array of elements is constructed, the procedure sets the node connectivity. The data variable d that is added to the linked list is of type *CornerConnectivity* that we defined in Algorithm 123 (CornerNodeClass). The procedure constructs the array of edges by Algorithm 148 (ConstructMeshEdges). Once the edges have been created, the procedure goes through each edge and sets boundary types for those edges that only have one neighboring element. If two elements share a edge, the direction variables of the second element are then set.

8.2.2 Benchmark Solution: A Spectral Element Mesh for a Disk

In the following sections we will solve problems on a disk by a spectral element method to avoid the coordinate singularity at the origin. We can construct a simple mesh file for the disk by hand using the topology shown in Fig. 8.14, which has five elements and eight corner nodes. The outer boundary needs to be approximated by a polynomial of degree N at the Gauss-Lobatto nodes to be represented accurately. The mesh generated by Algorithm 127 (QuadMesh:Construct) appears in Fig. 8.15 for $N = 8$.

8.3 The Spectral Element Method in Two Space Dimensions

The spectral element method is a continuous nodal spectral Galerkin approximation. We have used the continuous Galerkin approximation before to approximate potential and advection-diffusion problems. As a Galerkin method, we start the derivation from a weak form of the equation that we wish to solve.

We will introduce the spectral element method for two-dimensional geometries by approximating the potential equation with Dirichlet boundary conditions,

$$\begin{aligned} \nabla^2 \varphi &= s, \quad x \in \Omega, \\ \varphi &= \varphi_b, \quad x \in \partial\Omega. \end{aligned} \tag{8.54}$$

To convert the equation to the time dependent heat equation, we let $s = \partial\varphi/\partial t$. To approximate the advection-diffusion equation we will add an advection term as we did in Sect. 5.3 so that $s = \partial\varphi/\partial t + \mathbf{q} \cdot \nabla\varphi$.

Algorithm 127: *QuadMesh:Construct:* Constructor for a Two Dimensional Spectral Element Mesh

Procedure Construct
Input: Mesh File
Input: *spA* ; `// Nodal2DStorage`
Uses Algorithms:
Algorithm 63 (Nodal2DStorage)
Algorithm 148 (ConstructMeshEdges)
$N \leftarrow this.spA.N$
$this.\left\{sideMap_k\right\}_{k=1}^{4} \leftarrow \{0, N, N, 0\}$
$this.\left\{cornerMap_{1,k}\right\}_{k=1}^{4} \leftarrow \{0, N, N, 0\}$
$this.\left\{cornerMap_{2,k}\right\}_{k=1}^{4} \leftarrow \{0, 0, N, N\}$
$this.\left\{edgeMap_{1,k}\right\}_{k=1}^{4} \leftarrow \{1, 2, 4, 1\}$
$this.\left\{edgeMap_{2,k}\right\}_{k=1}^{4} \leftarrow \{2, 3, 3, 4\}$
Read from Mesh File: $this.N_{node}$, $this.K$
Allocate memory for nodes, elements and for edge array using (8.53)
$this.N_{edge} \leftarrow 0$
for $k = 1$ **to** $this.N_{node}$ **do**
  Read from Mesh File: x, y
  $this.nodes_k.Construct(x, y)$
end
for $k = 1$ **to** $this.K$ **do**
  Read from Mesh File: $\{cornerNodes_k\}_{k=1}^{4}$
  Read from Mesh File and Construct: $\{\mathbf{\Gamma}_j\}_{j=1}^{4}$
  $this.elements_k.Construct(spA, \{cornerNodes_k\}_{k=1}^{4}, \{\mathbf{\Gamma}_j\}_{j=1}^{4})$
end
for $eId = 1$ **to** $this.K$ **do**
  for $k = 1$ **to** *4* **do**
    $d.id \leftarrow eId$; $d.i \leftarrow this.cornerMap_{1,k}$; $d.j \leftarrow this.cornerMap_{2,k}$
    $n \leftarrow this.elements_{eId}.nodes_k$
    $this.nodes_n.NodeConnectivity.Add(d)$
  end
end
$this \leftarrow ConstructMeshEdges(this)$
for $k = 1$ **to** $this.N_{edge}$ **do**
  if $this.edges_k.elementID_2 = NONE$ **then**
    $this.edges_k.type \leftarrow BOUNDARY$
    $n1 \leftarrow this.edges_k.nodes_1$
    $n2 \leftarrow this.edges_k.nodes_2$
    $this.nodes_{n1}.type \leftarrow BOUNDARY$
    $this.nodes_{n2}.type \leftarrow BOUNDARY$
  else
    if $this.edges_k.elementSides_2 > 0$ **then**
      $this.edges_k.start = 1$; $this.edges_k.inc = 1$
    else
      $this.edges_k.start = N - 1$; $this.edges_k.inc = -1$
    end
  end
end
End Procedure Construct

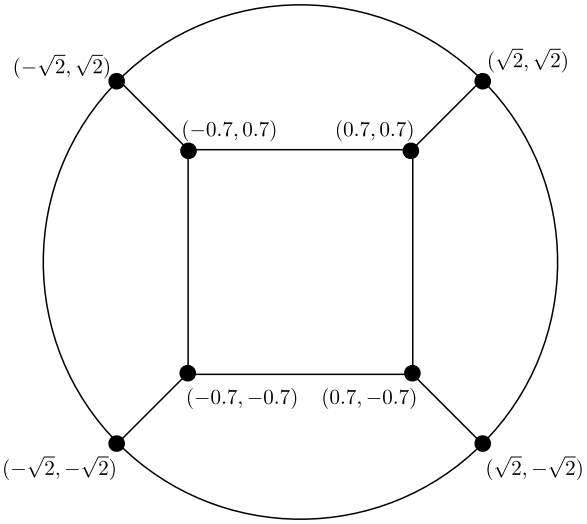

Fig. 8.14 A decomposition of a disk into five elements

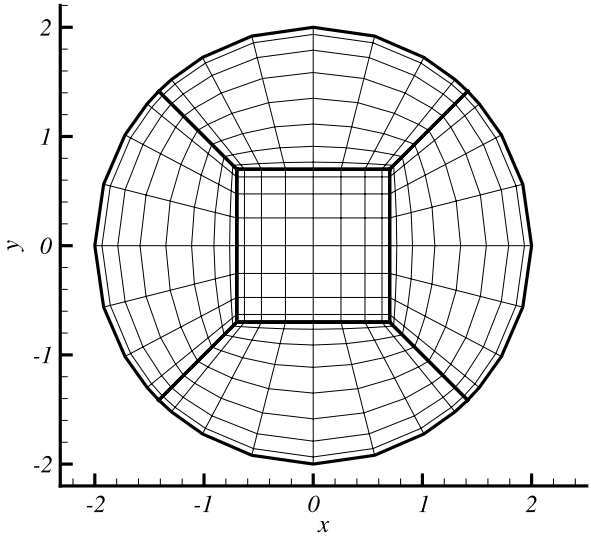

Fig. 8.15 Spectral element mesh for discretization of the disk

To get the weak form of the potential equation, we multiply it by an arbitrary, sufficiently smooth test function ϕ and integrate over the entire domain

$$\iint_{\Omega} \nabla^2 \varphi \phi dx dy = \iint_{\Omega} s\phi dx dy. \tag{8.55}$$

We then apply Green's identity to the integral with the Laplacian to rewrite it as

$$\iint_{\Omega} \nabla^2 \varphi \phi dxdy = \int_{\partial\Omega} \phi \frac{\partial \varphi}{\partial n} dL - \iint_{\Omega} \nabla\varphi \cdot \nabla\phi dxdy, \tag{8.56}$$

where dL is the differential along the boundary curve and $\partial\varphi/\partial n = \nabla\varphi \cdot \hat{n}$ is the normal derivative. As usual, for Dirichlet conditions we require that $\phi = 0$ along the boundary. Therefore the boundary integral vanishes to leave us with

$$-\iint_{\Omega} \nabla\varphi \cdot \nabla\phi dxdy = \iint_{\Omega} s\phi dxdy. \tag{8.57}$$

To get the spectral element method, we subdivide the domain Ω into K nonoverlapping quadrilateral elements, e^k, as we described in the introduction to this chapter. By breaking the domain into elements, we can break the integrals over the domain into the sum of integrals over the elements

$$\sum_{k=1}^{K} \left\{ -\iint_{e^k} \nabla\varphi \cdot \nabla\phi dxdy \right\} = \sum_{k=1}^{K} \left\{ \iint_{e^k} s\phi dxdy \right\}. \tag{8.58}$$

Therefore, each element contributes

$$-\iint_{e^k} \nabla\varphi \cdot \nabla\phi dxdy = \iint_{e^k} s\phi dxdy \tag{8.59}$$

to the total integral. For convenience, let us call

$$\begin{aligned} I_1^k &= -\iint_{e^k} \nabla\varphi \cdot \nabla\phi dxdy, \\ I_2^k &= \iint_{e^k} s\phi dxdy. \end{aligned} \tag{8.60}$$

The local element contributions (8.59) are exactly what we approximated in Sect. 5.2.2 on a single domain, so we already know how to approximate them. To recap, we first map the element e^k onto the reference square by a mapping $\mathbf{x} = \mathbf{X}(\xi, \eta)$. On the reference square, I_2^k transforms to

$$I_2^k = \int_{-1}^{1} \int_{-1}^{1} J^k s\phi d\xi d\eta. \tag{8.61}$$

To transform I_1^k, we write the gradient in the mapped coordinate as

$$\nabla\varphi = \frac{1}{J} \left\{ \left(Y_\eta \varphi_\xi - Y_\xi \varphi_\eta \right) \hat{x} + \left(X_\xi \varphi_\eta - X_\eta \varphi_\xi \right) \hat{y} \right\} \tag{8.62}$$

so that, with a little algebra, we get the integrand

$$\nabla\varphi\cdot\nabla\phi=\frac{1}{J}\left\{\left[\frac{Y_\eta^2+X_\eta^2}{J}\varphi_\xi-\frac{Y_\xi Y_\eta+X_\xi X_\eta}{J}\varphi_\eta\right]\phi_\xi\right.$$
$$\left.+\left[\frac{Y_\xi^2+X_\xi^2}{J}\varphi_\eta-\frac{Y_\xi Y_\eta+X_\xi X_\eta}{J}\varphi_\xi\right]\phi_\eta\right\}. \tag{8.63}$$

Ultimately, we write (8.63) in the familiar form

$$\nabla\varphi\cdot\nabla\phi=\frac{1}{J}\left\{f\phi_\xi+g\phi_\eta\right\}. \tag{8.64}$$

When we replace the integrand in (8.59) with (8.64), we write the element contribution as

$$-\int_{-1}^{1}\int_{-1}^{1}\left(f\phi_\xi+g\phi_\eta\right)d\xi d\eta=\int_{-1}^{1}\int_{-1}^{1}sJ^k\phi d\xi d\eta. \tag{8.65}$$

Therefore, we've already seen (8.59) on a mapped domain; the contribution of element k written on the reference square is just (7.17).

To approximate (8.65), we replace s by a piecewise polynomial approximation S and replace φ by a piecewise continuous polynomial approximation Φ. As an extension to what we did in one space dimension, (8.15), the test functions are going to be the continuous, piecewise polynomials

$$\phi^k=\sum_{i,j=0}^{N}\phi_{ij}^k\ell_i(\xi)\,\ell_j(\eta). \tag{8.66}$$

We enforce continuity of the test functions by requiring that the nodal values ϕ_{ij}^k be the same along each edge and at each element corner in the mesh. The same goes for the solution and source term polynomials.

When we substitute (8.66) for ϕ in (8.65),

$$I_1^k\approx\sum_{i,j}\phi_{i,j}^k\left[-\int_{-1}^{1}\int_{-1}^{1}\left(F\ell_i'\ell_j+G\ell_i\ell_j'\right)d\xi d\eta\right] \tag{8.67}$$

and

$$I_2^k\approx\sum_{i,j}\phi_{i,j}^k\left[\int_{-1}^{1}\int_{-1}^{1}S\ell_i\ell_j d\xi d\eta\right]. \tag{8.68}$$

The integrals in the square brackets are the same as in the single domain problem, and we already have their nodal Galerkin approximations worked out in (7.18)–(7.21). Therefore, the nodal Galerkin approximation of (8.65) is

$$\sum_{i,j}\phi_{i,j}^k\left[\left(\nabla^2\Phi,\ell_i\ell_j\right)_N-S_{i,j}^kJ_{i,j}^kw_iw_j\right]=0, \tag{8.69}$$

where $(\nabla^2\Phi, \ell_i\ell_j)_N$ is given by (7.21). The global sum over all the elements gives us our final approximation

$$\sum_{k=1}^{K}\left\{\sum_{i,j}\phi_{i,j}^k\left[\left(\nabla^2\Phi, \ell_i\ell_j\right)_N - S_{i,j}^k J_{i,j}^k w_i w_j\right]\right\} = 0. \tag{8.70}$$

We get the spectral element approximations to the time dependent diffusion and advection-diffusion equations from (8.70) if we replace $S_{i,j}^k$ by the appropriate approximations. For instance, the approximation to the time dependent diffusion equation is

$$\sum_{k=1}^{K}\left\{\sum_{i,j}\phi_{i,j}^k\left[\left(\nabla^2\Phi, \ell_i\ell_j\right)_N - \dot{\Phi}_{i,j}^k J_{i,j}^k w_i w_j\right]\right\} = 0, \tag{8.71}$$

whereas the approximation to the advection-diffusion equation is

$$\sum_{k=1}^{K}\left\{\sum_{i,j}\phi_{i,j}^k\left[\left(\nabla^2\Phi, \ell_i\ell_j\right)_N - \dot{\Phi}_{i,j}^k J_{i,j}^k w_i w_j - \left(\mathbf{q}\cdot\nabla\Phi, \ell_i\ell_j\right)_N\right]\right\} = 0 \tag{8.72}$$

with the advection term given by (7.77).

The last thing for us to do is to use the fact that the $\phi_{i,j}^k$ are independent except along element edges and at element corners to get the pointwise equations that the solution unknowns must satisfy. Let's focus on finding those equations for (8.70). The equations for (8.71) and (8.72) will follow directly.

To get the pointwise equations for the approximation (8.70), the first thing to notice is that the terms within the square brackets are simply the single domain approximation applied to the element e^k. At points interior to the elements, the $\phi_{i,j}^k$ are independent, which means that the approximation in an element is just the single domain approximation applied to it. Along edges, the approximation is the sum of the single domain values from the two contributing elements, just as we saw in the one dimensional spectral element method. Finally, at corners, all elements that share the point contribute to the sum. It is not as easy to write the summations explicitly as it was in one space dimension, but we will see that it is relatively easy to implement.

8.3.1 How to Implement the Spectral Element Method

The approximations (8.70)–(8.72) show that the spectral element method has local operations, gathered in the brackets, and global operations, gathered in the braces, so we will organize the algorithms as we did in one space dimension as local and global. The local operations are the single domain approximations that we have

already developed for quadrilateral domains. The global operations tie the local approximations together. The two-level form of the approximations shows that we can form a global framework that is essentially independent of the equations that we want to solve; We can compute the spectral approximation of the potential equation, the diffusion equation and the advection-diffusion equation with the same framework. Likewise, we have already developed the local operations for potentials, diffusion and advection-diffusion.

We will describe how to implement the approximation for the Dirichlet problem for the potential equation, (8.54) in detail. The implementation of Neumann boundary conditions and the time dependent problems are just extensions that we will pose as exercises.

8.3.1.1 The Potential Class

We organize the solution of the potential equation in a class, which we show in Algorithm 128 (SEMPotentialClass). Notice that the class is just an extension of the single domain class that we developed in Sect. 7.1.2. Since we now require that the polynomial order be the same in each element and in each direction within an element to guarantee most easily that the approximation is conforming, we still only need one instance of Nodal2DStorage, which we denote by the variable *spA*. However, now that there are multiple domains, the mesh will now store the geometry and mapping information with its elements. The mesh also stores the connectivity. Finally, we will use a global scheme for the solution and source terms to make it easy to use the Conjugate Gradient solver.

Algorithm 128: *SEMPotentialClass:* A Class Definition for the Spectral Element Approximation of the Potential Problem

Class SEMPotentialClass
Uses Algorithms:
Algorithm 63 (Nodal2DStorage)
Algorithm 101 (MappedGeometryClass)
Algorithm 107 (MappedLaplacian); Algorithm 126 (QuadMesh)
Data:
spA ; `// Of type Nodal2DStorage`
mesh ; `// Of type QuadMesh`
$\{\Phi_{i,j,k}\}_{i,j=0;k=1}^{N,N;K}$; `// Solution`
$\{s_{i,j,k}\}_{i,j=0;k=1}^{N,N;K}$; `// Source`
Procedures:
Construct$(N, meshFile)$; `// Algorithm 129`
MappedLaplacian$(\{U_{i,j,k}\}_{i,j=0}^{N}, geom)$; `// Algorithm 107`
MatrixAction$(\{U_{ij}\}_{i,j=0}^{N})$; `// Algorithm 133`
End Class SEMPotentialClass

Algorithm 129: *SEMPotentialClass:Construct:* Constructor for the Spectral Element Approximation of the Potential Problem

Procedure Construct
Input: N, *meshFile*
Uses Algorithms:
Algorithm 25 (LegendreGaussLobattoNodesAndWeights)
Algorithm 37 (PolynomialDerivativeMatrix)
Algorithm 127 (QuadMesh:Construct)

$this.spA.N \leftarrow N$; $this.spA.M \leftarrow N$
$\left\{this.spA.\{\xi_i\}_{i=0}^N, this.spA.\left\{w_i^{(\xi)}\right\}_{i=0}^N\right\} \leftarrow LegendreGaussLobattoNodesAndWeights(N)$
$this.spA.\left\{D_{ij}^{\xi}\right\}_{i,j=0}^N \leftarrow PolynomialDerivativeMatrix(this.spA.\{\xi_i\}_{i=0}^N)$
Copy arrays to η direction...
$mesh.Construct(spA, meshFile)$
End Procedure Construct

We must implement three procedures to compute the potential approximation. The constructor, Algorithm 129 (SEMPotentialClass:Construct), computes the spatial approximation array. One simplification here is that the node, weight and derivative matrix arrays are the same in both directions, so they need to be computed only once. The other action the constructor must take is to construct the mesh from a mesh file using Algorithm 127 (QuadMesh:Construct). We have already implemented the second procedure, *MappedLaplacian* in Algorithm 107 (MappedNodalGalerkinLaplacian). This is the local operation that computes the Laplacian term in the brackets in (8.70). The procedure works only on the section of the solution array for a given element, e^k. The geometry object that we pass with the array will be the geometry for the element. The final procedure computes the matrix action to be used with an iterative solver. The matrix action includes global operations, so we will wait to show what it does until after we develop the global procedures.

8.3.1.2 Global Procedures

As in one space dimension, we implement three global operations: Mask, UnMask and GlobalSum. The Mask operation will set duplicate node values in an array to zero so that they will have no contribution to the iterative solver. The UnMask operation will distribute the values back to the duplicate nodes. The global sum will add the contributions together at duplicate nodes, performing the operation in braces in (8.70). The procedures for all three operations have a similar structure. They first do their operations for each element side, then for each corner node.

We use the global Mask operation to set array values on duplicate nodes and boundary nodes to zero. We show an implementation in Algorithm 130 (SEMMask). The first section of the procedure loops over the array of edges and distinguishes between boundary an internal edges. Dirichlet conditions require the values to be

Algorithm 130: *SEMMask:* Mask Edges and Corners for the Spectral Element Method

Procedure Mask
Input: $mesh$, $\{a_{i,j,k}\}_{i,j=0;k=1}^{N;K}$

Uses Algorithms:
Algorithm 126 (QuadMesh)

for $j = 1$ **to** $mesh.N_{edge}$ **do**
  if $mesh.edges_j.type = BOUNDARY$ **then**
    $e \leftarrow mesh.edges_j.elementIDs_1$
    $s \leftarrow mesh.edges_j.elementSides_1$
  else
    $e \leftarrow mesh.edges_j.elementIDs_2$
    $s \leftarrow |mesh.edges_j.elementSides_2|$
  end
  $\{a_{i,j,k}\}_{i,j=0;k=1}^{N;K} \leftarrow MaskSide(e, s, mesh, \{a_{i,j,k}\}_{i,j=0;k=1}^{N;K})$
end
for $n = 1$ **to** $mesh.N_{node}$ **do**
  $pElements \Rightarrow mesh.nodes_n.nodeConnectivity$
  $pElements.current \Rightarrow pElements.head$
  if $mesh.nodes_n.type \neq BOUNDARY$ **then** pElements.MoveToNext()
  while $pElements.current \not\Rightarrow NULL$ **do**
    $d \leftarrow pElements.GetCurrentData()$
    $i \leftarrow d.i$; $j \leftarrow d.j$; $id \leftarrow d.id$
    $a_{i,j,id} \leftarrow 0$
    $pElements.MoveToNext()$
  end
end
return $\{a_{i,j,k}\}_{i,j=0;k=1}^{N;K}$
End Procedure Mask

Procedure MaskSide
Input: $id, side, mesh, \{a_{i,j,k}\}_{i,j=0;k=1}^{N;K}$

if $side = 2$ **or** $side = 4$ **then**
  $i \leftarrow mesh.sideMap_{side}$
  for $j = 1$ **to** $mesh.spA.N - 1$ **do**
    $a_{i,j,id} \leftarrow 0$
  end
else
  $j \leftarrow mesh.sideMap_{side}$
  for $i = 1$ **to** $mesh.spA.N - 1$ **do**
    $a_{i,j,id} \leftarrow 0$
  end
end
return $\{a_{i,j,k}\}_{i,j=0;k=1}^{N;K}$
End Procedure MaskSide

masked along boundary edges. So if the edge is a boundary edge, we mask the primary (and only), index 1, side. Otherwise, if the edge is on the interior, we mask the secondary (index 2) element side. We use the sideMap array to select the correct

local index when masking the secondary side in the *MaskSide* procedure. We apply the same philosophy to the corner nodes. For each corner node, we select the first contributing element to be the primary. If the corner node is on a physical boundary, we mask it to implement the Dirichlet condition. Otherwise, we mask the array entry for the nodes from each of the remaining elements that share the corner node.

The global UnMask operation, which we implement in Algorithm 131 (SEMUnmask), undoes the action of the Mask operation. It copies the data from the primary node to the secondary nodes. Again, the structure of the procedure is the same as *SEMMask*. However, we must account for the fact that the indices along the contributing sides do not have to increase in the same direction. In the implementation that we show here, we copy the values from the primary side into a temporary array to make the operations as clear as possible. We then use the edge's *start* and *inc* values to copy the edge values to the correct location for the secondary side in the global array.

The final global operation is the global summation, which we implement in Algorithm 132 (SEMGlobalSum). The global summation takes values from all contributing nodes, adds them together, and then distributes them back. Like the *SEMUnmask* procedure, the global summation must account for the fact that the element edges do not have to have indices that vary in the same direction. To be clear, we create two temporary arrays that store the summed values in the orders needed by the two sides. Then we simply copy those two arrays to their contributing sides.

8.3.1.3 Procedures for the Iterative Solver

Now that we have implemented the global operations, we implement the *MatrixAction* and *Residual* procedures that we need to use the Conjugate Gradient algorithm, Algorithm 80 (PreconditionedConjugateGradientSolve), to solve the system of equations. Algorithm 133 (SEMPotentialClass:MatrixAction) shows the matrix action. It first unmasks the solution values so that the Laplace approximations can be computed locally in a loop over each of the elements. Once the local actions are computed they are summed globally and then masked. To ensure that the procedure produces no side effects, it re-masks the input array. The procedure *Residual* that we show in Algorithm 134 is similar to the *MatrixAction* procedure and computes the global residual.

Since we use the Conjugate Gradient method to solve the linear system for the potentials, we should mention preconditioners for the spectral element method. The similarity of the spectral element method to the finite element method allows us to generalize the finite element preconditioner that we have already implemented to multiple domains. We showed in Sect. 7.1.3 how to modify the finite element preconditioner to work on a transformed domain that now forms one of our elements. If we use an iterative solver for the preconditioner, like the *SSORSweep* procedure that we presented in Algorithm 79, then we can generalize the finite element approximation to the multidomain discretization in the same way that we did the spectral method. That is, we compute the local preconditioners, and perform the

Algorithm 131: *SEMUnMask:* UnMask for the Spectral Element Method

```
Procedure UnMask
Input: mesh, {a_{i,j,k}}_{i,j=0;k=1}^{N;K}
Uses Algorithms:
  Algorithm 126 (QuadMesh)
for j = 1 to mesh.N_edge do
    if mesh.edges_j.type ≠ BOUNDARY then
        {a_{i,j,k}}_{i,j=0;k=1}^{N;K} ← UnMaskSide(mesh.edges_j, mesh, {a_{i,j,k}}_{i,j=0;k=1}^{N;K})
    end
end
for n = 1 to mesh.N_nodes do
    if mesh.nodes_n.type ≠ BOUNDARY then
        pElements ⇒ mesh.nodes_n.nodeConnectivity
        pElements.current ⇒ pElements.head
        d ← pElements.GetCurrentData()
        i1 ← d.i; j1 ← d.j; id1 ← d.id
        pElements.MoveToNext()
        while pElements.current ⇏ NULL do
            d ← pElements.GetCurrentData()
            i ← d.i; j ← d.j; id ← d.id
            a_{i,j,id} ← a_{i1,j1,id1}
            pElements.MoveToNext()
        end
    end
end
return {a_{i,j,k}}_{i,j=0;k=1}^{N;K}
End Procedure UnMask
```

```
Procedure UnMaskSide
Input: edge, mesh, {a_{i,j,k}}_{i,j=0;k=1}^{N;K}
id ← edge.elementIDs_1; side ← edge.elementSides_1
if side = 2 or side = 4 then
    i ← mesh.sideMap_side
    for j = 1 to mesh.spA.N − 1 do
        tmp_j ← a_{i,j,id}
    end
else
    j ← mesh.sideMap_side
    for i = 1 to mesh.spA.N − 1 do
        tmp_i ← a_{i,j,id}
    end
end
id ← edge.elementIDs_2; side ← |edge.elementSides_2|
if side = 2 or side = 4 then
    i ← mesh.sideMap_side; j ← edge.start
    for n = 1 to mesh.spA.N − 1 do
        a_{i,j,id} ← tmp_n
        j ← j + edge.inc
    end
else
    j ← mesh.sideMap_side; i ← edge.start
    for n = 1 to mesh.spA.N − 1 do
        a_{i,j,id} ← tmp_n
        i ← i + edge.inc
    end
end
return {a_{i,j,k}}_{i,j=0;k=1}^{N;K}
End Procedure UnMaskSide
```

Algorithm 132: *SEMGlobalSum:* Sum Edge Contributions for the Two-Dimensional Spectral Element Method

```
Procedure GlobalSum
Input: mesh, {a_{i,j,k}}_{i,j=0;k=1}^{N;K}
Uses Algorithms:
  Algorithm 126 (QuadMesh)
for j = 1 to mesh.N_edge do
    if mesh.edges_j.type ≠ BOUNDARY then
        {a_{i,j,k}}_{i,j=0;k=1}^{N;K} ← SumSide(mesh.edges_j, mesh, {a_{i,j,k}}_{i,j=0;k=1}^{N;K})
    end
end
for n = 1 to mesh.N_node do
    if mesh.nodes_n.type ≠ BOUNDARY then
        pElements ⇒ mesh.nodes_n.nodeConnectivity
        pElements.current ⇒ pElements.head
        sum = 0
        while pElements.current ≠ NULL do
            d ← pElements.GetCurrentData()
            i ← d.i; j ← d.j; id ← d.id
            sum ← sum + a_{i,j,id}
            pElements.MoveToNext()
        end
        pElements.current ⇒ pElements.head
        while pElements.current ≠ NULL do
            d ← pElements.GetCurrentData()
            i ← d.i; j ← d.j; id ← d.id
            a_{i,j,id} ← sum
            pElements.MoveToNext()
        end
    end
end
return {a_{i,j,k}}_{i,j=0;k=1}^{N;K}
End Procedure GlobalSum
```

masks and global sums on those to get the global action and residuals. This is not the way we would implement a finite element method from scratch, but it fits easily into the framework that we have developed so far. One problem with the finite element preconditioner is that as the meshes get large its convergence rate slows down, too. More sophisticated and complex spectral element preconditioners have been developed over the years. For those we point to Chap. 6 of the book [8], which describes several strategies including alternating Schwartz and Schur complement techniques.

8.3.1.4 The Driver

The driver to solve the potential problem with the spectral element method has the same structure as the driver on the square, Algorithm 76 (CollocationPoten-

Algorithm 132: *SEMGlobalSum:* Sum Edge Contributions for the Two-Dimensional Spectral Element Method (continued)

```
Procedure SumSide
Input: edge, mesh, {a_{i,j,k}}_{i,j=0;k=1}^{N;K}

for k = 1 to 2 do
    id ← edge.elementIDs_k; side ← |edge.elementSides_k|
    if side = 2 or side = 4 then
        i ← mesh.sideMap_side
        for j = 1 to mesh.spA.N − 1 do
            tmp_{j,k} ← a_{i,j,id}
        end
    else
        j ← mesh.sideMap_side
        for i = 1 to mesh.spA.N − 1 do
            tmp_{i,k} ← a_{i,j,id}
        end
    end
end
n ← edge.start
for j = 1 to mesh.spA.N − 1 do
    sum ← tmp_{j,1} + tmp_{j,2}
    tmp_{j,1} ← sum; tmp_{n,2} ← sum
    n ← n + edge.inc
end
for k = 1 to 2 do
    id ← edge.elementIDs_k; side ← |edge.elementSides_k|
    if side = 2 or side = 4 then
        i ← mesh.sideMap_side
        for j = 1 to mesh.spA.N − 1 do
            a_{i,j,id} ← tmp_{j,k}
        end
    else
        j ← mesh.sideMap_side
        for i = 1 to mesh.spA.N − 1 do
            a_{i,j,id} ← tmp_{i,k}
        end
    end
end
return {a_{i,j,k}}_{i,j=0;k=1}^{N;K}
End Procedure SumSide
```

tialDriver) and the quadrilateral, Algorithm 108 (MappedCollocationDriver). As before, a *SourceValue* function must be provided to compute the source, s. The boundary mask array is not needed because we have assumed only Dirichlet boundary conditions and have incorporated them into the global mask and unmask functions themselves. We do need a routine to set the boundary values. For that, we present an implementation in Algorithm 135 (SetBoundaryValues). It assumes that the actual function that provides the solution value as a function of x, y (and t for time dependent problems) is provided as input.

Algorithm 133: *SEMPotentialClass:MatrixAction:* Matrix Action for the Spectral Element Approximation to the Potential Equation

Procedure MatrixAction
Uses Algorithms:
Algorithm 126 (QuadMesh)
Algorithm 107 (MappedLaplacian)
Input: $\left\{\Phi_{i,j,k}\right\}_{i,j=0;k=1}^{N;K}$

$\left\{\Phi_{i,j,k}\right\}_{i,j=0;k=1}^{N;K} \leftarrow UnMask\left(this.mesh, \left\{\Phi_{i,j,k}\right\}_{i,j=0;k=1}^{N;K}\right)$
for $k = 1$ **to** $this.mesh.K$ **do**
  $\left\{action_{i,j,k}\right\}_{i,j=0}^{N,M} \leftarrow MappedLaplacian\left(this.mesh.elements_k.geom, \left\{\Phi_{i,j,k}\right\}_{i,j=0}^{N,M}\right)$
end
$\left\{action_{i,j,k}\right\}_{i,j=0;k=1}^{N;K} \leftarrow GlobalSum\left(this.mesh, \left\{action_{i,j,k}\right\}_{i,j=0;k=1}^{N;K}\right)$
$\left\{action_{i,j,k}\right\}_{i,j=0;k=1}^{N;K} \leftarrow Mask\left(this.mesh, \left\{action_{i,j,k}\right\}_{i,j=0;k=1}^{N;K}\right)$
$\left\{\Phi_{i,j,k}\right\}_{i,j=0;k=1}^{N;K} \leftarrow Mask\left(this.mesh, \left\{\Phi_{i,j,k}\right\}_{i,j=0;k=1}^{N;K}\right)$
return $\left\{action_{i,j,k}\right\}_{i,j=0;k=1}^{N;K}$
End Procedure MatrixAction

Algorithm 134: *Residual:* Residual Computation for the Spectral Element Approximation to the Potential Equation

Procedure Residual
Input: pA ; `// Of type SEMPotentialClass`
Uses Algorithms:
Algorithm 128 (SEMPotentialClass)
Algorithm 107 (MappedLaplacian)

$pA.\left\{\Phi_{i,j,k}\right\}_{i,j=0;k=1}^{N;K} \leftarrow UnMask\left(pA.mesh, pA.\left\{\Phi_{i,j,k}\right\}_{i,j=0;k=1}^{N;K}\right)$
for $k = 1$ **to** $mesh.K$ **do**
  $\left\{r_{i,j,k}\right\}_{i,j=0}^{N,M} \leftarrow MappedLaplacian\left(pA.mesh.elements_k.geom, pA.\left\{\Phi_{i,j,k}\right\}_{i,j=0}^{N,M}\right)$
  for $j = 0$ **to** $pA.spA.N$ **do**
    for $i = 0$ **to** $pA.spA.M$ **do**
      $r_{i,j,k} \leftarrow pA.spA.w_i^{(\xi)} * pA.spA.w_j^{(\eta)} * pA.source_{i,j,k} * pA.mesh.elements_k.geom.J_{i,j} - r_{i,j,k}$
    end
  end
end
$\left\{r_{i,j,k}\right\}_{i,j=0;k=1}^{N;K} \leftarrow GlobalSum\left(pA.mesh, \left\{r_{i,j,k}\right\}_{i,j=0;k=1}^{N;K}\right)$
$\left\{r_{i,j,k}\right\}_{i,j=0;k=1}^{N;K} \leftarrow Mask\left(pA.mesh, \left\{r_{i,j,k}\right\}_{i,j=0;k=1}^{N;K}\right)$
$pA.\left\{\Phi_{i,j,k}\right\}_{i,j=0;k=1}^{N;K} \leftarrow Mask\left(pA.mesh, pA.\left\{\Phi_{i,j,k}\right\}_{i,j=0;k=1}^{N;K}\right)$
return $\left\{r_{i,j,k}\right\}_{i,j=0;k=1}^{N;K}$
End Procedure Residual

Algorithm 135: *SetBoundaryValues:* Set Dirichlet Boundary Conditions for the Two-Dimensional Spectral Element Method

Procedure SEMSetDirichletBoundaries
Input: $mesh$, $\{\Phi_{i,j,k}\}_{i,j=0;k=1}^{N;K}$, t, $BCFunction$
Uses Algorithms:
Algorithm 126 (QuadMesh)

for $k = 1$ **to** $mesh.N_{edge}$ **do**
  if $mesh.edges_j.type = BOUNDARY$ **then**
    $id \leftarrow mesh.edges_k.elementIDs_1$; $side \leftarrow mesh.edges_k.elementSides_1$
    if $side = 2$ **or** $side = 4$ **then**
      $i \leftarrow mesh.sideMap_{side}$
      for $j = 1$ **to** $mesh.spA.N - 1$ **do**
        $x \leftarrow mesh.elements_{id}.geom.x_{i,j}$
        $y \leftarrow mesh.elements_{id}.geom.y_{i,j}$
        $\Phi_{i,j,id} \leftarrow BCFunction(x, y, t)$
      end
    else
      $j \leftarrow mesh.sideMap_{side}$
      for $i = 1$ **to** $mesh.spA.N - 1$ **do**
        $x \leftarrow mesh.elements_{id}.geom.x_{i,j}$
        $y \leftarrow mesh.elements_{id}.geom.y_{i,j}$
        $\Phi_{i,j,id} \leftarrow BCFunction(x, y, t)$
      end
    end
  end
end
for $n = 1$ **to** $mesh.N_{node}$ **do**
  if $mesh.nodes_n.type = BOUNDARY$ **then**
    $pElements \Rightarrow mesh.nodes_n.nodeConnectivity$
    $x \leftarrow mesh.nodes_n.x$; $y \leftarrow mesh.nodes_n.y$
    $\Phi_b \leftarrow BCFunction(x, y, t)$
    $pElements.current \Rightarrow pElements.head$
    while $pElements.current \Rightarrow NULL$ **do**
      $d \leftarrow pElements.GetCurrentData()$
      $i \leftarrow d.i$; $j \leftarrow d.j$; $id \leftarrow d.id$
      $\Phi_{i,j,id} \leftarrow \Phi_b$
      $pElements.MoveToNext()$
    end
  end
end
return $\{\Phi_{i,j,k}\}_{i,j=0;k=1}^{N;K}$
End Procedure SEMSetDirichletBoundaries

8.3.2 Benchmark Solution: Steady Temperatures in a Long Cylindrical Rod

The benchmark problem for the spectral element method is to compute the steady temperature in a long cylindrical rod that is heated uniformly along its length. The simple physical model reduces to the solution of the potential equation on a circular

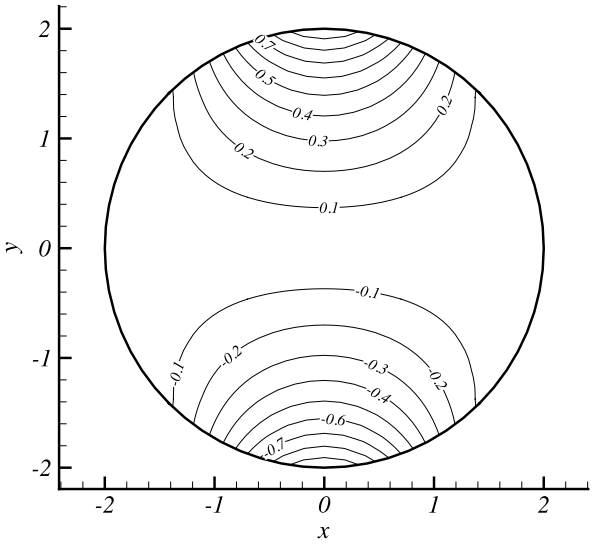

Fig. 8.16 Spectral element solution of temperatures on a disk

domain with a prescribed temperature around its perimeter. The model problem has an exact solution known as the Poisson Integral against which we can compare the spectral element solution. The steady temperature as a function of the polar coordinates (r, θ) on a disk of radius R that is kept at a temperature $\varphi(R, \theta) = F(\theta)$ along the outer boundary is

$$\varphi(r, \theta) = \frac{R^2 - r^2}{2\pi} \int_{-\pi}^{\pi} \frac{F(s)}{R^2 - 2rR\cos(s - \theta) + r^2} ds. \tag{8.73}$$

We solve for the temperature in the disk on the mesh shown in Fig. 8.15 with $N = 8$ that is heated and cooled along the outer edge according to

$$F(\theta) = e^{-4(\theta - \pi/2)^2} - e^{-4(\theta + \pi/2)^2}.$$

We show a contour plot of the computed solution in Fig. 8.16. We make a direct comparison to the analytical solution in Fig. 8.17, which shows the computed and exact solutions along the line $x = 0$.

8.4 The Discontinuous Galerkin Spectral Element Method

Finally, we derive the discontinuous Galerkin spectral element approximation of the system of conservation laws in two dimensions. We will see that the discontinuous approximation will give us simpler algorithms to implement than what we had in

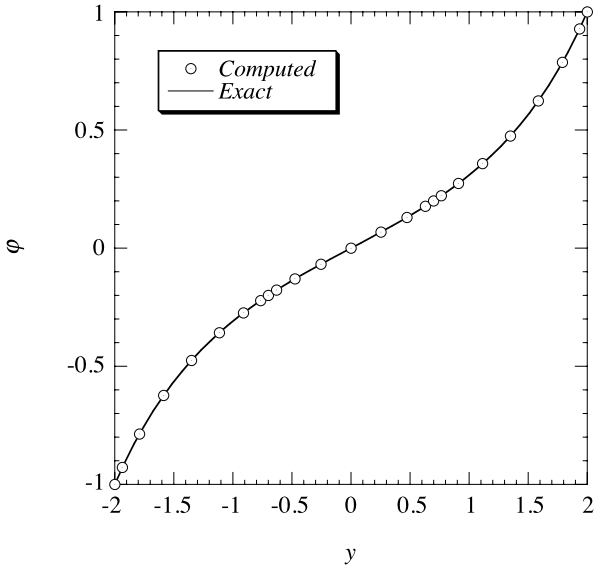

Fig. 8.17 Spectral element solution of temperatures on a disk. Cut along $x = 0$

the previous section, and that we have already developed all the machinery that we need to implement the method.

Our starting point is the two dimension conservation law (7.89), which we reproduce here,

$$\mathbf{q}_t + \mathbf{f}_x + \mathbf{g}_y = 0, \tag{8.74}$$

and which we convert to the weak form

$$\iint_{\Omega} \phi \left(\mathbf{q}_t + \mathbf{f}_x + \mathbf{g}_y\right) dxdy = 0. \tag{8.75}$$

We get the equation that the solution satisfies on an element when we break the integral into the sum of subintegrals over the elements

$$\sum_{k=1}^{K} \iint_{e^k} \phi \left(\mathbf{q}_t + \mathbf{f}_x + \mathbf{g}_y\right) dxdy = 0 \tag{8.76}$$

and examine each subintegral individually

$$\iint_{e^k} \phi \left(\mathbf{q}_t + \mathbf{f}_x + \mathbf{g}_y\right) dxdy = 0. \tag{8.77}$$

Since the element e^k is a quadrilateral, we have already derived the nodal discontinuous Galerkin approximation for (8.77) in Sect. 7.4, specifically, (7.99).

In the discontinuous Galerkin approximation, we couple the approximation to the boundary and, in the spectral element version, to neighboring elements through the boundary fluxes using the Riemann solver. If the boundary of the element is not on a physical boundary, then the external state is simply the solution on the neighboring element.

8.4.1 How to Implement the Discontinuous Galerkin Spectral Element Method

We see that the discontinuous Galerkin spectral element approximation is just the single domain mapped quadrilateral approximation applied to each element individually, plus coupling of the fluxes at the boundaries. This suggests that to implement the approximation we can reuse the algorithms of Sect. 7.4.2 for the element local operations. To compute the fluxes at the element boundaries, we will use the information stored in the mesh data structure to tell us which common nodes we need to use to compute the element boundary fluxes. As a spectral element method, we can arrange the data as we did in Sect. 8.3.1.

Our basic data structure will be the *DGSEMClass* that we present in Algorithm 136. It extends the single domain *MappedNodalDG2DClass* (Algorithm 111) to manage multiple domains. We assume a conforming mesh, as we did with the spectral element approximation, so there is still only one instance of the structure that stores the Gauss quadrature nodes, weights and the derivative matrices. Instead of a single mapping, each element has its own, so we replace the geometry object in the *MappedNodalDG2DClass* with the mesh object. Finally, we must now have an array of *DGSolutionStorage* structures to store the solution on each element. The structure of the new class is basically the same as the *SEMPotentialClass* of Algorithm 128, so the constructor for the class is the same as Algorithm 129 except for one change. We use the Gauss, not the Gauss-Lobatto, points for the

Algorithm 136: *DGSEMClass:* A Discontinuous Galerkin Class Definition

Class DGSEMClass
Uses Algorithms:
Algorithm 89 (NodalDG2DStorage)
Algorithm 110 (DGSolutionStorage)
Algorithm 126 (QuadMesh)
Data:
spA ; `// Of type NodalDG2DStorage`
mesh ; `// Of type QuadMesh`
$\{dGS\}_{k=1}^{K}$; `// Of type DGSolutionStorage`
Procedures:
Construct$(N, meshFile)$; `// Algorithm 129, modified. See text.`
TimeDerivative(t); `// Algorithm 138`
End Class DGSEMClass

nodes in the discontinuous Galerkin approximation. Therefore, the constructor must use Algorithm 23 (LegendreGaussNodesAndWeights) to compute the nodes and weights.

The time derivative procedure for the spectral element approximation will follow Algorithm 115 (DG2DTimeDerivative) except that it must do its work on all of the elements. Since the boundary solutions need to be available before the boundary fluxes are computed, we interpolate all of the solutions to the faces first. We then compute the boundary fluxes for each of the edges. Finally, the local time derivatives need to be computed.

Before we present the time derivative procedure, however, we must develop a procedure to compute the boundary fluxes. Fortunately, all the information that we need is available in the *QuadMesh* structure. Recall that an edge stores the *id*'s of two neighboring elements and their associated sides. (See Algorithm 125 (EdgeClass).) The first one we call the primary element and the other the secondary. The edge also stores how the index of the secondary side varies with the index of the primary. Therefore, to compute the element boundary fluxes, we can loop through each of the edges, and for each, get the solutions and normal to feed to the Riemann solver to compute the flux. Once we compute the normal flux, we compute the contravariant flux, e.g., by (7.102). Algorithm 137 (EdgeFluxes) implements this procedure. Note that the outward normals for the two elements point in opposite directions. Therefore the edge flux must be negated to get the contravariant flux for the secondary side. If the edge is on a physical boundary, then the flux is computed just as it was for the single domain approximation.

Now that we have the edge flux procedure, we can implement the global time derivative procedure shown in Algorithm 138 (SEMGlobalTimeDerivative).

Unlike the continuous Galerkin spectral element method, we do not have to perform any operations on the corner nodes with the discontinuous version. Recall that our use of the Gauss, rather than Gauss-Lobatto points places the solution unknowns entirely within an element. (Cf. Fig. 5.7.) The boundary flux values are then located at the Gauss points along an edge. The corner nodes are not part of the approximation, thereby simplifying the implementation. It is for this simplification and for the increased quadrature precision that we have chosen the Gauss over the Gauss-Lobatto points for the location of the nodes.

To integrate in time, we will still use an explicit time integrator. The only difference from the single domain implementation is the need to loop over each element. See Sect. 8.1.4.

8.4.2 Benchmark Solution: Propagation of a Circular Wave in a Circular Domain

In Sect. 5.4.4 we computed the solution of the wave equation in a square domain with initial conditions that create an outward propagating circular sound wave. In this section, we will re-do the problem with the mesh shown in Fig. 8.15. For this mesh, we present solutions with $w = 0.15$.

Algorithm 137: *EdgeFluxes:* Compute the Riemann Problem Along Mesh Edges

Procedure EdgeFluxes
Input: t
Input: $edge$; `// Of type Edge`
Input: $\{elements\}_{k=1}^{K}$; `// Of type QuadElement`
Input: $\{dGS\}_{k=1}^{K}$; `// Of type DGSolutionStorage`
Uses Algorithms:
Algorithm 125 (EdgeClass)
Algorithm 110 (DGSolutionStorage)
Algorithm 124 (QuadElementClass)

if $edge.type = INTERIOR$ **then**
  $k \leftarrow edge.start - edge.inc$
  for $j = 0$ **to** N **do**
    $e1 \leftarrow edge.elementIDs_1$
    $s1 \leftarrow edge.elementSides_1$
    $e2 \leftarrow edge.elementIDs_2$
    $s2 \leftarrow |edge.elementSides_2|$
    $\{F_n\}_{n=1}^{nEqn} \leftarrow$
    $RiemannSolver(dGS_{e1}.\{Qb_{j,n,s1}\}_{n=1}^{nEqn}, dGS_{e2}.\{Qb_{k,n,s2}\}_{n=1}^{nEqn}, elements_{e1}.geom.\hat{n}_j^{s1})$
    for $n = 1$ **to** $nEqn$ **do**
      $dGS_{e1}.F^*_{j,n,s1} \leftarrow F_n * elements_{e1}.geom.scal_j^{s1}$
      $dGS_{e2}.F^*_{k,n,s2} \leftarrow -F_n * elements_{e2}.geom.scal_k^{s2}$
    end
    $k \leftarrow k + edge.inc$
  end
else
  $e1 \leftarrow edge.elementIDs_1$
  $s1 \leftarrow edge.elementSides_1$
  for $j = 0$ **to** N **do**
    $\left\{Q_n^{ext}\right\}_{n=1}^{nEqn} \leftarrow$
    $ExternalState(dGS_{e1}.\{Qb_{j,n,s1}\}_{n=1}^{nEqn}, elements_{e1}.geom.x_j^{s1}, elements_{e1}.geom.y_j^{s1}, t)$
    $dGS_{e1}.\left\{F^*_{j,n,s1}\right\}_{n=1}^{nEqn} \leftarrow elements_{e1}.geom.scal_j^{s1} *$
    $RiemannSolver\left(dGS_{e1}.\{Qb_{j,n,s1}\}_{n=1}^{nEqn}, \left\{Q_n^{ext}\right\}_{n=1}^{nEqn}, elements_{e1}.geom.\hat{n}_j^{s1}\right)$
  end
end
return $\{dGS\}_{k=1}^{K}$
End Procedure EdgeFluxes

We present solutions for the propagating circular wave at time $t = 1.25$ in Figs. 8.18 and 8.19. The solutions were computed with $N = 20$ and a time step of $\Delta t = 1 \times 10^{-3}$. To present the solutions in Fig. 8.18, we interpolated the solution in each element to 30 points in each direction using Algorithm 35 (2DCoarseToFineInterpolation). Figure 8.18 shows contours of the pressure, which illustrates that the circular shape of the wave is retained. Figure 8.19 shows the comparison of the exact and computed solutions along the line $y = 0$.

Algorithm 138: *DGSEMClass:TimeDerivative:* Compute the Time Derivative for the Discontinuous Galerkin Approximation

Procedure TimeDerivative
Input: t
Uses Algorithms:
Algorithm 112 (DG2DProlongToFaces)
Algorithm 137 (EdgeFluxes)
Algorithm 114 (MappedDGSystemTimeDerivative)

for $k = 1$ **to** $this.mesh.K$ **do**
  $this.dGS_k \leftarrow DG2DProlongToFaces(this.spA, this.elements_k.geom, this.dGS_k)$
end
for $i = 1$ **to** $this.mesh.N_{edge}$ **do**
  $this.\{dGS\}_{k=1}^{K} \leftarrow$
  $EdgeFluxes(t, this.mesh.edges_i, this.mesh.\{elements\}_{k=1}^{K}, this.\{dGS\}_{k=1}^{K})$
end
for $k = 1$ **to** $this.mesh.K$ **do**
  $this.dGS_k \leftarrow$
  $MappedDG2DTimeDerivative(this.spA, this.elements_k.geom, this.dGS_k)$
end
End Procedure TimeDerivative

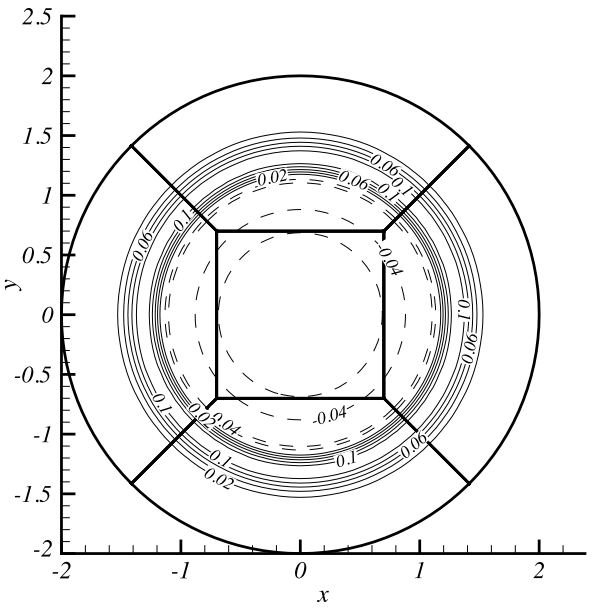

Fig. 8.18 Computed pressure contours at time $t = 1.25$ for a propagating circular wave when $N = 20$ and $\Delta t = 1 \times 10^{-3}$. The solutions were interpolated to 30 uniformly spaced points in each direction on each element. Heavy lines show the element boundaries

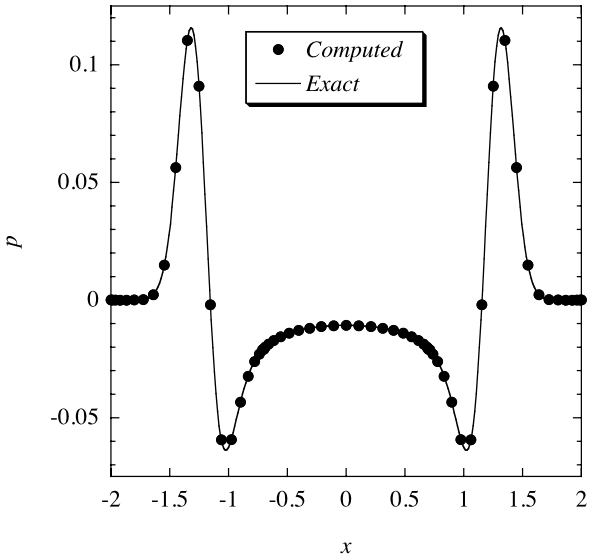

Fig. 8.19 Comparison of the computed circular wave pressure with the exact solution along the line $y = 0$ at $t = 1.25$

8.4.3 Benchmark Solution: Transmission and Reflection from a Material Interface

In the introduction to this chapter, we listed four reasons why we might want or need to use a spectral element approximation instead of a single domain method. In the previous benchmark, the reason was to avoid the coordinate singularity that a cylindrical coordinate mesh would produce. In this benchmark, we have two reasons for using a spectral element method. The first is that we can use the method when there are singularities in the coefficients. The second is to increase efficiency by using smaller elements of lower order. With this benchmark solution we explore a unique feature of the discontinuous Galerkin spectral element method, namely its ability to approximate discontinuous solutions at element interfaces accurately.

When a wave propagates across an interface where the wave speed abruptly changes, part of the wave is transmitted and part of it is reflected. The phenomenon is familiar in daily life. In electrodynamics, we study the reflection and transmission of electromagnetic waves at dielectric interfaces. In ultrasound tests, ultrasonic waves reflect as they propagate through different tissues.

To extend the wave propagation model that we have used so far to include propagation through multiple materials, we now allow the density of the material, ρ, and the wave speed, c, to vary as a function of location. With these changes, the

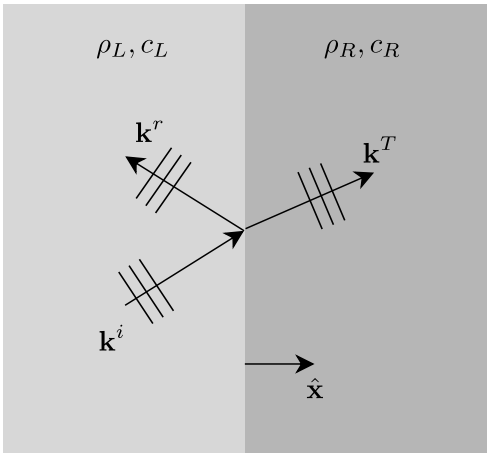

Fig. 8.20 Model for plane wave reflection at a material interface

conservation law form of the wave equation is

$$\begin{bmatrix} p \\ u \\ v \end{bmatrix}_t + \begin{bmatrix} \rho c^2 u \\ p/\rho \\ 0 \end{bmatrix}_x + \begin{bmatrix} \rho c^2 v \\ 0 \\ p/\rho \end{bmatrix}_y = 0. \tag{8.78}$$

As our benchmark problem, we solve for the transmission and reflection of a plane wave at a plane interface between two materials that have uniform properties within each, as we show in Fig. 8.20.

The problem has an analytic solution against which we can compare our computed solutions. It is the kind of problem that is solved in electrodynamics texts for scattering at an interface between two dielectrics. Let $\psi(\xi)$ be a waveform with maximum value of one, and a be an amplitude. Then each of the incident, reflected and transmitted plane waves is of the form

$$\mathbf{q} = a\psi\left(\mathbf{k}\cdot\mathbf{x} - \omega\left(t - t_0\right)\right)\begin{bmatrix} 1 \\ \frac{k_x}{\rho c} \\ \frac{k_y}{\rho c} \end{bmatrix}. \tag{8.79}$$

To define a particular wave, we simply replace $\mathbf{k}$ by the appropriate wavevector and a by the appropriate amplitude. The wavevectors and amplitudes of the reflected and transmitted waves depend on the incident wave and must satisfy the correct jump and phase matching conditions at the interface. Let us define the incident wavevector to be

$$\mathbf{k}^i = \frac{\omega}{c_L}\left(k_x^i \hat{x} + k_y^i \hat{y}\right), \tag{8.80}$$

where $(k_x^i)^2 + (k_y^i)^2 = 1$. Then the reflected and transmitted wavevectors are

$$\begin{aligned} \mathbf{k}^r &= \frac{\omega}{c_L}\left(-k_x^i \hat{x} + k_y^i \hat{y}\right), \\ \mathbf{k}^T &= \frac{\omega}{c_R}\left[\sqrt{1 - \left(\frac{c_R}{c_L}\right)^2 \left(k_y^i\right)^2}\,\hat{x} + \frac{c_R}{c_L} k_y^i \hat{y}\right]. \end{aligned} \tag{8.81}$$

The corresponding amplitudes are

$$\begin{aligned} \frac{a^r}{a^i} &= \frac{1}{J}\left(\rho_R c_R k_x^T / k^T - \rho_L c_L k_x^i / k^i\right), \\ \frac{a^T}{a^i} &= \frac{1}{J}\left(\rho_L c_L k_x^r / k^r - \rho_R c_L k_x^i / k^i\right), \end{aligned} \tag{8.82}$$

where

$$J = -\rho_R c_R k_x^T / k^T + \rho_L c_L k_x^r / k^r. \tag{8.83}$$

To use the discontinuous Galerkin approximation, we must define the flux functions and derive a Riemann solver. We get the flux functions from (8.78). To derive the Riemann solver, remember that it computes the numerical flux $\mathbf{F}^*(\mathbf{q}^L, \mathbf{q}^R; \hat{n})$ given two possibly different states, $\mathbf{q}^L$ and $\mathbf{q}^R$, where left and right are defined according to the normal direction $\hat{n} = \alpha\hat{x} + \beta\hat{y}$. To start, we construct the coefficient matrix for the system

$$A = \begin{bmatrix} 0 & \alpha\rho c^2 & \beta\rho c^2 \\ \alpha/\rho & 0 & 0 \\ \beta/\rho & 0 & 0 \end{bmatrix}. \tag{8.84}$$

The eigenvalues of this system remain the same as before, $\lambda = \pm c, 0$. What makes the problem interesting now is that the characteristic variables have a jump discontinuity when the wave speeds and the density jump across an interface. Instead of requiring the characteristic variables to be continuous at an interface, we apply the *Rankine-Hugoniot* condition, which says that the normal flux must be continuous. For the wave equation, this means

$$A_L \mathbf{q}_L - A_R \mathbf{q}_R = 0, \tag{8.85}$$

since A depends on both the density and the wave speed. Nevertheless, waves that propagate to the right are evaluated from $\mathbf{q}^L$ and waves that propagate to the left are evaluated from $\mathbf{q}^R$, just as before. Under those constraints, a fair amount of algebra shows that

$$\mathbf{F}^*\left(\mathbf{q}_L, \mathbf{q}_R; \hat{n}\right) = \begin{bmatrix} z_R[c(p + \rho c(nu + n_y v))]_L - z_L[c(p - \rho c(n_x u + n_y v))]_R \\ n_x\{\frac{z_L}{\rho_L}[p + \rho c(n_x u + n_y v)]_L + \frac{z_R}{\rho_R}[p - \rho c(n_x u + n_y v)]_R\} \\ n_y\{\frac{z_L}{\rho_L}[p + \rho c(n_x u + n_y v)]_L + \frac{z_R}{\rho_R}[p - \rho c(n_x u + n_y v)]_R\} \end{bmatrix}, \tag{8.86}$$

where

$$z_L = \frac{\rho_L c_L}{\rho_L c_L + \rho_R c_R}, \qquad z_R = \frac{\rho_R c_R}{\rho_L c_L + \rho_R c_R}. \tag{8.87}$$

Note that when the densities and the wave speeds are the same on both sides, (8.86) reduces to (5.164). If, in addition, the solution is the same on both sides, it reduces to the normal flux $A\mathbf{q}$.

Since the discontinuous Galerkin spectral element approximation allows discontinuities in the solution at element boundaries, it is a natural choice for problems with material discontinuities, as long as we place element boundaries along material boundaries. The only modifications that we need to add beyond the new flux functions to replace the procedures in Algorithm 94 (WaveEquationFluxes) and the new Riemann solver to replace Algorithm 88 (RiemannSolver), is to have the element class store the element's material properties, ρ and c. When the Riemann solver is called in Algorithm 137 (EdgeFluxes), it will be passed the material values from the left and the right elements.

For the benchmark solution, we compute the reflection and transmission of a plane wave through a vertical material interface, as pictured in Fig. 8.20. We take the domain to be the square $[-5, 5] \times [-5, 5]$ with the material interface along $x = 0$. We subdivide the domain into a structured mesh of 20 elements in each direction so that each element has a length and width equal to 0.5. We present solutions for $N = 10$ in each element. For plotting they are interpolated to 12 uniformly spaced points in each direction in each element. We integrate to $t = 3.0$ in time with $\Delta t = 0.05$.

We model the incident wave as an approximation of a typical ultrasound pulse,

$$\psi(t) = \sin(\omega t)\, e^{-t^2/(\omega\sigma)^2}, \tag{8.88}$$

where $\omega = 2\pi f$ and f is the frequency. For the envelope, $\sigma^2 = -(MT)^2/(4\ln(10^{-4}))$ where M is the number of modes in the significant part of the envelope and $T = 1/f$ is the period. We present the specific parameters in Table 8.4. With these parameters and the mesh, we resolve the sine waves with an average of about seven points per wavelength. The external state and the initial condition are set using the exact solution.

We show contours of the computed and exact values of p in Fig. 8.21. Clearly visible are the incident wave, above on the left, the reflected wave below on the left, and transmitted wave on the right. Notice that p itself is discontinuous at the interface between the two materials as is allowed by the discontinuous Galerkin method. We show p as a function of y in Figs. 8.22 and 8.23. We chose the locations to be at the nearest Gauss points to $x = -1$ and $x = 0.5$.

Table 8.4 Parameters for plane wave reflection problem

Parameter	M	f	k_x^i	k_y^i	ρ_L	ρ_R	c_L	c_R	t_0
Value	4	2.5	0.5	$\sqrt{3}/2$	1	0.4	1	0	3

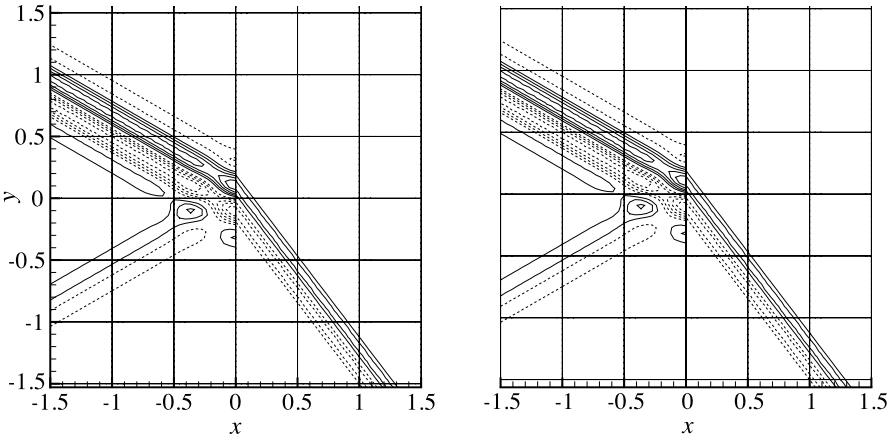

Fig. 8.21 Comparison of computed (*left*) and exact (*right*) pressure contours at $t = 3$ for plane wave reflection at a material interface. *Dashed lines* are negative contours. The contour levels range from -0.8 to 0.8 with a step of 0.2. Note the discontinuity in the pressure at the material discontinuity along $x = 0$. The overlay of squares shows the locations of the element boundaries

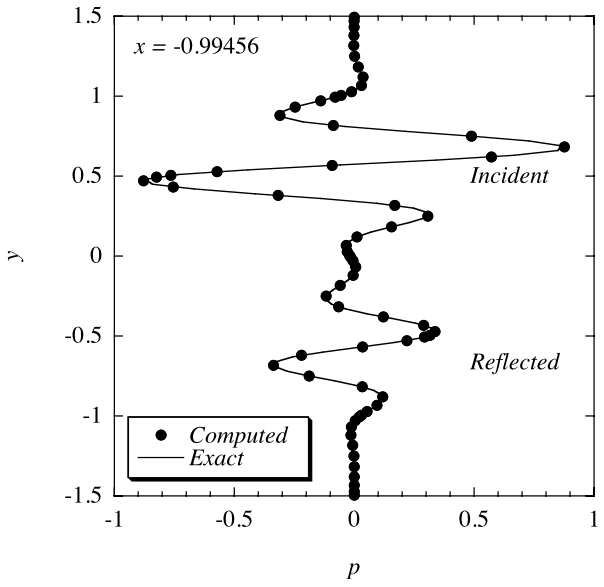

Fig. 8.22 Comparison of the computed and exact pressures along a vertical line to the left of the material interface

Exercises

8.1 Show that if N and Δx are constants, then the time derivative, (8.22), is the simple average of the approximations from either side.

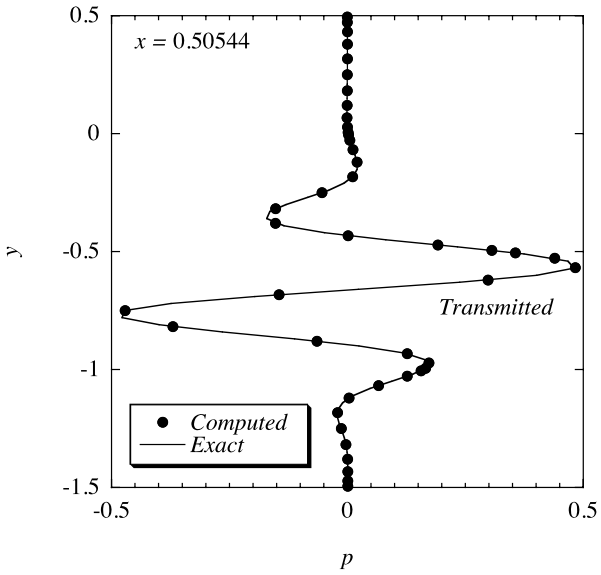

Fig. 8.23 Comparison of the computed and exact pressures along a vertical line to the right of the material interface

8.2 Derive the one dimensional spectral element method for the variable coefficient problem,

$$\varphi_t = (\nu(x)\varphi_x)_x \,.$$

8.3 Derive the spectral element approximation to the advection-diffusion equation.

$$\varphi_t + a\varphi_x = \nu\varphi_{xx}.$$

If $\nu = 0$ and N and Δx are constant, show that the time derivative at an element interface point is the simple average of the spatial derivatives from either side. Compare the approximation of the advection term to the discontinuous Galerkin approximation for the same equation.

8.4 Derive the spectral element approximation to the equation

$$\varphi_{xx} = s$$

with Dirichlet boundary conditions. Develop the algorithms that you need to solve the equations iteratively.

8.5 Derive the numerical flux, (8.49).

8.6 In general one would want to impose different boundary conditions along different boundaries of a physical problem. In Sect. 5.2.1, we did this by defining an

array that specified whether or not to mask a particular boundary. Apply the mask idea to the spectral element approximation by assigning to each edge in the mesh a mask variable that is set from information in the mesh file. Show how to modify Algorithms 130–134 to incorporate both Dirichlet boundary conditions and radiation conditions of the form $\nabla\varphi \cdot \hat{n} = \gamma\varphi$.

8.7 Design algorithms to integrate the spectral element approximation to the diffusion equation, (8.71) using the trapezoidal rule for the time integrator. Implement and test your algorithms and solve the problem with solution

$$\varphi(x, y, t) = \frac{1}{4t+1} e^{-\frac{((x-x_0)^2+(y-y_0)^2)}{4t+1}}$$

on the disk.

8.8 The steady incompressible viscous flow in a circular pipe is known as *Poiseuille flow* and is a special case of the flows computed in Problem 7.12. If the pipe has radius R, then the axial velocity of the Poiseuille flow is given by

$$u(r) = -\frac{\gamma}{4}\left(R^2 - r^2\right).$$

1. Use the spectral element method to compute the Poiseuille flow and compare the computed solutions to the exact for $\gamma = -1$.
2. Compute to spectral accuracy the *volume flow rate* Q defined by

$$Q = \int_{disk} u dA$$

and compare to the exact analytical value

$$Q_{pipe} = -\frac{\pi R^4}{8}\gamma.$$

8.9 Do Problem 7.13 with a spectral element mesh.

8.10 Develop and implement the algorithms for the spectral element approximation to the advection-diffusion equation, (8.72) with the semi-implicit time integration of Sect. 5.3.3. Solve the benchmark problem of Sect. 7.3.6 and compare the computation time between single and multidomain approximations for a given accuracy.

8.11 Modify Algorithm 137 (EdgeFluxes) to allow it to apply different boundary conditions to different edges. (Cf. Problem 8.6.)

8.12 In Sect. 5.4.3 we saw that single domain wave propagation requires large order polynomials to resolve the two main spatial scales, namely the size of the domain

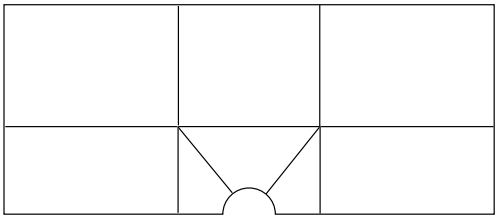

Fig. 8.24 A Mesh topology for scattering of an acoustic wave off a circular cylinder

and the length scale of the propagating wave. A spectral element approximation enables us to resolve both scales by subdividing the square into a mesh of smaller square elements and keep the cost down by using lower order polynomials on the elements. Redo the solution of the wave equation for both the planewave and cylindrical wave problems of Sect. 5.4.3 with multiple square elements. Compare the cost to compute the solutions to a desired accuracy for several subdivisions of the square.

8.13 In Sect. 7.4.3 we computed the scattering of an acoustic wave off a cylinder. As we discussed in Problem 8.12, the differences in scales required us to compute the solution with very high order polynomials, and with correspondingly small time steps required by the explicit time differencing. A more efficient approach is to use a spectral element approximation. Compute the scattering problem with a mesh topology like that shown in Fig. 8.24 and compare the cost to the single domain computation.