Appendix A
Pseudocode Conventions

"How to play the flute. (picking up a flute) Well here we are.
You blow there and you move your fingers up and down here."
in "How to do it", Monty Python's Flying Circus, Episode 28.

We use a pseudocode in this book to show how to implement spectral methods. Pseudocode is a commonly used device to present algorithms. It represents an informal high level description of what one would program with a computer language. Pseudocodes omit details like variable declarations, memory allocations, and computer language specific syntax. Too high a level, however, and we risk missing important details. The goal of pseudocode is to give enough cues to allow the reader to write a working computer program, no matter what programming language will be ultimately used to implement it.

We use the LaTex macro "Algorithm2e" written by Christophe Fiorio to typeset our pseudocode. The macro provides commonly used keywords and ways to represent flow control such as conditional statements and loops. Comments, however, look like C/C++ statements. Since comments are typeset with a different font, it should be pretty clear what is a comment and what is not. Beyond these basic and common statements, we need to decide how to express both high level and low level concepts.

To help to read the almost 150 algorithms that we present, we outline some of the conventions that we use. On the lower level, these conventions include how we represent variables, arithmetic operations, and arrays. On the higher level, they include an object oriented philosophy to organize data and procedures.

Variables and Arithmetic Operations We use pseudocode to provide a bridge between the mathematics and a computer program. To make that bridge, we try to make the statements look as closely as possible to the equations that they are trying to implement. Therefore, if we compute something that has a well-known mathematical notation, such as the Chebyshev polynomial of degree n, we write it that way in the pseudocode, T_n. If the quantity does not have a common name, we make up a variable name for it. We denote constants by all uppercase names, e.g. NONE, with underscores to separate words.

As much as possible we write equations in the pseudocode just as we write them mathematically in the text. Sometimes, however, we represent multiplication by " * " when leaving it out can cause confusion.

Arrays The most common data structure that we use in the spectral methods algorithms is the array. Mathematical vectors and grid values in one space dimension can be stored as singly dimensioned arrays. Full matrices and grid values in two space dimensions can be stored as doubly dimensioned arrays. In this book, we represent

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

single arrays by $\{u_j\}_{j=s}^{e}$ and double arrays by $\{u_{i,j}\}_{i,j=s}^{e}$, where s and e represent the start and end indices. When the indices may mean something different, say if we want to think of an array of two dimensional arrays, we will separate indices with a semicolon. In any case, since different computer languages have different levels of support for arrays, we do not imply a particular data model by our arrays except in one circumstance. That circumstance is when we use iterative methods to solve linear systems, where we assume that arrays are contiguous in memory and therefore all representations are equivalent to a singly dimensioned array of the appropriate length.

Functions and Other Procedures With most computer languages it is difficult to tell which variables are input, which are output, and which are both. To make these clear, we write procedure calls like

$$result \leftarrow function(input1, input2).$$

If an argument is both an input and output variable, it appears both as a result and as an input

$$result \leftarrow function(input1, input2, result).$$

Since we work with tensor product spectral approximations in this book, most of the operations in two dimensions that work on doubly dimensioned arrays reduce to performing multiple operations on singly dimensioned arrays. We denote a one dimensional slice of a two dimensional array, say, $\{U_{i,j}\}_{i,j=0}^{N,M}$, by $\{U_{i,j}\}_{i=0}^{N}$ for slices along columns and $\{U_{i,j}\}_{j=0}^{M}$ along rows. In general, we assume that if an array with particular extents is passed into a procedure, then those extents are too.

Pointers We express setting a pointer to point to a record or another pointer by the notation "$\Rightarrow$". For simplicity, we assume that dereferencing a pointer, i.e., getting the record to which a pointer points, is automatic. That is, if a pointer p points to a record in memory that contains a structure *data*, then we reference that data by $p.data$. This is how pointers work in Fortran (with % in place of ".") and how references work in C++, but not how pointers work in C/C++, where one would use $p \rightarrow data$.

Object Oriented Algorithms We take an object oriented view of data and procedure organization. This doesn't mean that one has to use an object oriented language to implement these algorithms. As with arrays, different computer languages have different levels of support for automatically programming this way, so we use object orientation here to allow us to group variables, simplify procedure arguments, and reuse procedures that we have already developed. The fundamental construct is the *class*, which gathers data in the form of an abstract data type (or structure) and procedures that work on that data. Member variables and procedures are accessed in the common, though not universal, dot notation. Therefore if *obj* is an instance of a given class, a is a member variable, and $f(x)$ is a member procedure, then

we access a by $obj.a$ and invoke f by $obj.f(x)$. For procedure calls, the implicit assumption is that the object is passed as an argument to the procedure, whether this is done automatically within the computer language, or explicitly in the argument list as necessary. Thus, $obj.f(x)$ by itself means

$$obj \leftarrow f(obj, x).$$

Within the procedure, the object is named *this*, again common but not universal. We use the keyword **Extends** if we want to add data to, or replace procedures in, a class. In a sense, this represents subclassing that is present in fully object oriented languages.

Appendix B
Floating Point Arithmetic

Computers use floating point numbers, which behave differently than real numbers. Discussions of floating point arithmetic in general [16] and the IEEE implementation [18] used on most computers today can be found in the references. We are most interested in one number: ϵ, which represents the relative error due to rounding. Several computer languages now have this number as an available parameter. For instance, in Fortran, it is given by the function EPSILON( ). In C/C++ it is defined in the float.h header file as FLT_EPSILON, with a similar definition for doubles.

It is well-known that we should never do direct comparisons of equality for floating point numbers ([16], Vol. II, Chap. 4). On the other hand, it is not that obvious how to write a robust algorithm to test when two floating point numbers are "close enough" to be considered to be equal. The basic (strong) test for two floating point numbers, a and b, to be "essentially equal to" each other is that

$$|b-a| \leq \epsilon\,|a| \quad \text{and} \quad |b-a| \leq \epsilon\,|b|\,. \tag{B.1}$$

It is possible, however, for the products on the right to overflow or underflow. We can avoid those situations by scaling the numbers, or explicitly handling the overflow situations.

To test the equality of two floating point numbers in the algorithms developed for this book, such as is necessary for instance in Algorithm 31 (LagrangeInterpolation),

Algorithm 139: *AlmostEqual:* Testing Equality of Two Floating Point Numbers

```
Procedure AlmostEqual
Input: a, b
if a = 0 or b = 0 then
    if |a − b| ≤ 2ϵ then
        AlmostEqual ← TRUE
    else
        AlmostEqual ← FALSE
    end
else
    if |a − b| ≤ ϵ|a| and |a − b| ≤ ϵ|b| then
        AlmostEqual ← TRUE
    else
        AlmostEqual ← FALSE
    end
end
return AlmostEqual
End Procedure AlmostEqual
```

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

we propose Algorithm 139 (AlmostEqual). Note that we do not have to worry about all exceptional cases here, particularly since the numbers that we work with are in $[-1, 1]$ or $[0, 2\pi]$. Thus, the only exceptional cases we need to deal with are near the origin.

Appendix C
Basic Linear Algebra Subroutines (BLAS)

The BLAS provide standard building blocks to perform basic vector and matrix operations. There are three levels of BLAS, with each level performing more and more complex operations. The first level, BLAS Level 1, is a collection of routines that compute common operations such as the Euclidean norm, the dot product, and vector operations such as $y = \alpha x + y$, known as AXPY. A PDF reference of the available routines, blasqr.pdf, can be found at http://www.netlib.org/blas/.

BLAS libraries are freely available on the web, for example at www.netlib.org, and specifically optimized versions are often included with commercial compilers. Furthermore, the ATLAS (Automatically Tuned Linear Algebra Software) library found at http://math-atlas.sourceforge.net can be compiled to create a portably efficient BLAS library.

The BLAS routines are named with the format xNAME, where "x" denotes the precision of the arithmetic, either "D" for double precision or "S" for single. (For many routines, complex or complex*16 versions are available with the prefixes "C" and "Z", respectively.) For example, the single precision dot product is named SDOT.

BLAS routines that are of use in this book include DOT, for the Euclidean inner product, NRM2, for the Euclidean norm, AXPY, for scalar times vector plus vector, COPY and SCAL. The standard calling arguments are

```
_DOT(N,X,INCX,Y,INCY)
_NRM2(N,X,INCX)
_AXPY(N,ALPHA,X,INCX,Y,INCY)
_COPY(N,X,INCX,Y,INCY)
_SCAL(N,ALPHA,X,INCX)
```

In each case, N corresponds to the total number of elements. The arguments X and Y correspond to the input/output arrays. The integers *INCX* and *INCY* indicate the stride of the data, and enable the computation of subarray operations. The argument ALPHA corresponds to the scalar parameter.

All arrays used in BLAS routines are *constrained to be contiguous*, a constraint that should be observed when using languages that define arrays as arrays of pointers. As such, the BLAS routines, now called the dense versions, don't distinguish between inputs that represent one or two-dimensional arrays.

For completeness, and by way of example, we present prototype algorithms to compute the dot product, the Euclidean norm, $y = \alpha x + y$, copy, and scale by a parameter. In practice, one should use optimized library versions. Consistent with the dense BLAS philosophy, we constrain the input arrays to be contiguous, so it does not matter whether arguments represent single or multidimensional arrays.

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

Algorithm 140: *BLAS_Level1:* A Selection of Basic Linear Algebra Subroutines

Procedure BLAS_NRM2
Input: $N, \{x_j\}_{j=1}^{N*INCX}$, INCX

$z \leftarrow 0; i \leftarrow 1$
for $j = 1$ **to** N **do**
  $z \leftarrow z + x_i^2$
  $i \leftarrow i + INCX$
end
return $\sqrt{z}$
Procedure BLAS_NRM2

Procedure BLAS_DOT
Input: $N, \{x_j\}_{j=1}^{N*INCX}$, INCX, $\{y_j\}_{j=1}^{N*INCY}$, INCY

$z \leftarrow 0; i \leftarrow 1; j \leftarrow 1$
for $k = 1$ **to** N **do**
  $z \leftarrow z + x_i * y_j$
  $i \leftarrow i + INCX$
  $j \leftarrow j + INCY$
end
return z
Procedure BLAS_DOT

Procedure BLAS_AXPY
Input: $N, \alpha, \{x_j\}_{j=1}^{N*INCX}$, INCX, $\{y_j\}_{j=1}^{N*INCY}$, INCY

$i \leftarrow 1; j \leftarrow 1$
for $k = 1$ **to** N **do**
  $y_j \leftarrow y_j + \alpha * x_i$
  $i \leftarrow i + INCX$
  $j \leftarrow j + INCY$
end
return $\{y_j\}_{j=1}^{N}$
End Procedure BLAS_AXPY

Procedure BLAS_COPY
Input: $N, \{x_j\}_{j=1}^{N*INCX}$, INCX, $\{y_j\}_{j=1}^{N*INCY}$, INCY

$i \leftarrow 1; j \leftarrow 1$
for $k = 1$ **to** N **do**
  $y_j \leftarrow x_i$
  $i \leftarrow i + INCX$
  $j \leftarrow j + INCY$
end
return $\{y_j\}_{j=1}^{N}$
End Procedure BLAS_COPY

Procedure BLAS_SCAL
Input: $N, \alpha, \{x_j\}_{j=1}^{N*INCX}$, INCX

$i \leftarrow 1$
for $k = 1$ **to** N **do**
  $x_i \leftarrow \alpha x_i$
  $i \leftarrow i + INCX$
end
return $\{x_i\}_{i=1}^{N}$
End Procedure BLAS_SCAL

Appendix D
Linear Solvers

Approximations of potential problems and implicit discretizations of time dependent partial differential equations lead to linear systems of equations to solve. We solve the systems either directly, typically by some variant of Gauss elimination, or iteratively. In this appendix, we motivate the algorithms that we use to solve the systems that appear in this book. In no way can a short appendix like this survey the entire field of numerical linear algebra and all the issues related to efficiency, parallelism, etc. For further study, we suggest the books [13] or [19].

D.1 Direct Solvers

For small enough systems, or in special cases, direct linear system solvers are efficient. In this section, we discuss two, namely the Thomas algorithm to solve tri-diagonal systems, and LU factorization to solve full, general systems. In both cases, well-tested and portable code with multiple language bindings is available in the LAPACK [2] library. Some compiler vendors supply precompiled versions of LAPACK, which should be used if possible. Otherwise, it is possible to download the source code from www.netlib.org (see, in particular, http://www.netlib.org/lapack/index.html) and compile it oneself.

D.1.1 Tri-Diagonal Solver

Tri-diagonal matrix problems are ubiquitous in numerical analysis, and appear, for example in Sect. 4.5 with the Legendre Galerkin approximation. For completeness, therefore, we include the Thomas algorithm for the solution of tri-diagonal systems. We represent the elements of the matrix by the three vectors ℓ, d and u, for the subdiagonal, diagonal and superdiagonal elements, numbered as

$$\begin{bmatrix} d_0 & u_0 & 0 & \dots & 0 \\ \ell_1 & d_1 & u_1 & \ddots & \vdots \\ 0 & \ell_2 & d_2 & \ddots & 0 \\ \vdots & \ddots & \ddots & & u_{N-1} \\ 0 & \dots & 0 & \ell_N & d_N \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \\ x_N \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \\ y_N \end{bmatrix}. \tag{D.1}$$

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

Algorithm 141: *TriDiagonalSolve:*

Procedure TriDiagonalSolve
Input: $\{\ell_j\}_{j=1}^{N}, \{d_j\}_{j=0}^{N}, \{u_j\}_{j=0}^{N-1}, \{y_j\}_{j=0}^{N}$
for $j = 0$ **to** N **do**
  $\hat{d}_j = d_j$
end
// Forward Elimination
for $j = 1$ **to** N **do**
  $\hat{d}_j = \hat{d}_j - \ell_j / \hat{d}_{j-1} * u_{j-1}$
  $y_j = y_j - \ell_j / \hat{d}_{j-1} * y_{j-1}$
end
// Backward Substitution
$x_N = y_N / \hat{d}_N$
for $j = N - 1$ **to** 0 **step** -1 **do**
  $x_j = (y_j - u_j * x_{j+1}) / \hat{d}_j$
end
return $\{x_j\}_{j=0}^{N}$
End Procedure TriDiagonalSolve

The Thomas algorithm is a variant of Gauss elimination, and a description of it can be found in any elementary numerical analysis book. It has two phases, the forward elimination and the backward substitution.

The first phase of the algorithm is the forward elimination, where we remove the subdiagonal entries. In the second row, ℓ_1 is eliminated when we multiply the first row by $-\ell_1/d_0$ and add the result to the second. The diagonal entry on the second element of the modified row and the right hand side become

$$\begin{aligned} \hat{d}_1 &= d_1 - u_0 \ell_1 / d_0, \\ \hat{y}_1 &= y_1 - y_0 \ell_1 / d_0. \end{aligned} \tag{D.2}$$

We eliminate the ℓ_2 entry in the third row and all succeeding values of ℓ_j by repeating the procedure for $j = 2, \ldots, N$. When complete, the matrix is upper bi-diagonal, so that $x_N = \hat{y}_N / \hat{d}_N$ and for $j < N$,

$$x_j = \left(\hat{y}_j - u_j x_{j+1}\right) / \hat{d}_j. \tag{D.3}$$

Since the right hand side vector is rarely needed again in practice, it is usually safe just to overwrite the original array $\mathbf{y}$ with $\hat{\mathbf{y}}$. Algorithm 141 (TriDiagonalSolve) shows the whole procedure.

D.1.2 LU Factorization

An LU factorization with partial row pivoting computes a lower triangular, L, upper triangular, U, and permutation, P, matrix to rewrite a square matrix A in the form

$A = PLU$. The permutation matrix is there to swap rows to ensure that the diagonal contains the largest element in a column. The attraction of the algorithm is that once we compute and store the factorization, we solve the system $A\mathbf{x} = \mathbf{y}$ efficiently for multiple right hand sides, $\mathbf{y}$, by a triangular forward substitution followed by a triangular backward substitution.

To motivate the algorithm, let's assume that row swapping (pivoting) is not needed. Then $A = LU$ and we write the matrix multiplication component-wise as

$$a_{ij} = \sum_{n=1}^{N} \ell_{in} u_{nj} = \sum_{n=1}^{\min(i,j)} \ell_{in} u_{nj}, \tag{D.4}$$

since L is lower triangular and U is upper triangular. If we choose $\ell_{kk} = 1$ (giving us what is known as the Doolittle Method), we can write

$$\begin{aligned} a_{kj} &= \sum_{n=1}^{k-1} \ell_{kn} u_{nj} + u_{kj}, \quad j = k, \ldots, N, \\ a_{ik} &= \sum_{n=1}^{k-1} \ell_{in} u_{nk} + \ell_{ik} u_{kk}, \quad i = k+1, \ldots, N. \end{aligned} \tag{D.5}$$

We rearrange these to solve for the unknowns

$$\begin{aligned} u_{kj} &= a_{kj} - \sum_{n=1}^{k-1} \ell_{kn} u_{nj}, \quad j = k, \ldots, N, \\ \ell_{ik} &= \frac{1}{u_{kk}} \left(a_{ik} - \sum_{n=1}^{k-1} \ell_{in} u_{nk} \right), \quad i = k+1, \ldots, N. \end{aligned} \tag{D.6}$$

Typically, one destroys the original matrix by writing U to the upper triangular part of the A, and L to the lower. Since we chose the diagonal of L to be one, it doesn't need to be stored, which allows the diagonal part of U to be stored there instead.

In general, we must swap rows to move the largest element in a row to the diagonal. The information can be stored in a single pivot vector, $\{p_j\}_{j=1}^{N}$, that simply tells which row must be swapped with the current row. The procedure *Factorize* in Algorithm 142 (LUFactorization) prototypes how to decompose a matrix into its $A = PLU$ factorization for columnwise storage of the matrix A. Note that this procedure, like most library factorization procedures, destroys the original matrix to save storage. An easy mistake to make is to forget and try to use the matrix again after it has been factorized.

We break the solve operation into two steps. Since $A = PLU$,

$$PLU\mathbf{x} = \mathbf{y} \tag{D.7}$$

or

$$LU\mathbf{x} = P\mathbf{y} \tag{D.8}$$

Algorithm 142: *LUFactorization:* Factorization and Solve Procedures to Solve $A\mathbf{x} = \mathbf{y}$

```
Procedure Factorize
Input: {A_{i,j}}_{i,j=1}^{N}

for k = 1 to N do
    p(k) ← k
    for i = k + 1 to N do
        if |A_{i,k}| > |A_{p_k,k}| then p_k ← i
    end
    if p_k ≠ k then
        for j = 1 to N do
            t ← A_{k,j}; A_{k,j} ← A_{p_k,j}; A_{p_k,j} ← t
        end
    end
end
for j = k to N do
    s ← 0
    for n = 1 to N do
        s ← s + A_{k,n} * A_{n,j}
    end
    A_{k,j} ← A_{k,j} − s
end
for i = k + 1 to N do
    s ← 0
    for n = 1 to k − 1 do
        s ← s + A_{i,n} * A_{n,k}
    end
    A_{i,k} ← (A_{i,k} − s)/A_{k,k}
end
return {A_{i,j}}_{i,j=1}^{N}, {p_j}_{j=1}^{N}
End Procedure Factorize
```

```
Procedure LUSolve
Input: {A_{i,j}}_{i,j=1}^{N}, {p_j}_{j=1}^{N}, {y_{j,m}}_{j,m=1}^{N,N_RHS}

for i = 1 to N do
    if p_i ≠ i then
        for m = 1 to N_RHS do
            t ← y_{i,m}; y_{i,m} ← y_{p_i,m}; y_{p_i,m} ← t
        end
    end
end
for m = 1 to N_RHS do
    w_1 ← y_{1,m}
    for i = 2 to N do
        s = 0
        for j = 1 to i − 1 do
            s ← s + A_{i,j} * w_j
        end
        w_i ← y_{i,m} − s
    end
    y_{N,m} ← w_N/A_{N,N}
    for i = N − 1 to 1 step −1 do
        s = 0
        for j = i + 1 to N do
            s ← s + A_{i,j} * y_{j,m}
        end
        y_{i,m} ← (w_i − s)/A_{i,j}
    end
end
return {y_{j,m}}_{j,m=1}^{N,N_RHS}
End Procedure LUSolve
```

since swapping a row then swapping again returns rows to their original state and implies $P^{-1} = P$. If we define $\mathbf{w} = U\mathbf{x}$, we can solve two triangular systems in succession

$$\begin{aligned} L\mathbf{w} &= P\mathbf{y}, \\ U\mathbf{x} &= \mathbf{w}. \end{aligned} \tag{D.9}$$

The first is simply forward substitution. Since

$$\sum_{j=1}^{i} L_{ij} w_j = (Py)_i \,, \tag{D.10}$$

we solve $w_1 = (Py)_1 / L_{11} = (Py)_1$ and

$$w_i = (Py)_i - \sum_{j=1}^{i-1} L_{ij} w_j, \quad i = 2, 3, \ldots, N. \tag{D.11}$$

We derive a similar back substitution formula to solve $U\mathbf{x} = \mathbf{w}$. The combination of these two form the *LUSolve* procedure in Algorithm 142 (LUFactorization). The input to the procedure is the *factorized* matrix, i.e. the output of *Factorize*. To accommodate multiple right hand sides, we assume that a two dimensional array with N_{RHS} columns is supplied, as is done with the LAPACK routines. The output is then an array whose columns are the solution vector for each right hand side.

Rather than use these prototype procedures in production, which we present here to understand how the algorithm works, it is better to use optimized library routines such as those provided by LAPACK [2]. The LAPACK routines can take advantage of optimized BLAS routines and run efficiently on parallel systems. The two routines useful for general matrices are

```
xGETRF(M, N, A, LDA, IPIV, INFO)
xGETRS(TRANS, N, NRHS, A, LDA, IPIV, B, LDB, INFO)
```

where "x" denotes the data type of the variables, "D" for double, "S" for single, for instance. The procedures perform the factorization (F) and Solve (S) phases separately. The arguments are the number of rows and columns, M and N, the matrix A, the leading dimension of A, LDA, the pivot vector, IPIV and an error flag. The solve routine takes a character variable, TRANS, that specifies if the transpose of A is to be used. The argument N is the order of the matrix A and NRHS is N_{RHS}. The next three arguments are the same as for xGETRF, but A is the factorized matrix. Finally, B corresponds to $\{y_{j,m}\}_{j,m=1}^{N,N_{RHS}}$ where LDB is the leading dimension of the array B.

D.2 Iterative Solvers

We construct the solution of a system of equations $A\mathbf{x} = \mathbf{y}$ iteratively by adding a correction to a current iterate, $\mathbf{x}^k$,

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \omega^k \mathbf{v}^k. \tag{D.12}$$

The vector $\mathbf{v}^k$ is the search direction and ω is a parameter that says how far to step in that direction. The simplest choice of direction is the direction of the iteration residual

$$\mathbf{v}^k = \mathbf{r}^k \equiv \mathbf{y} - A\mathbf{x}^k \tag{D.13}$$

giving what is known as the *Richardson Iteration* method,

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \omega^k \mathbf{r}^k. \tag{D.14}$$

For a point of reference, we can write the well-known Jacobi method in this form with $\mathbf{v}^k = D^{-1}\mathbf{r}^k$, where D is the diagonal part of the matrix A.

We should choose the step parameter ω^k so that the iterate, $\mathbf{x}^{k+1}$, is in some sense closer to the solution than the previous iterate. One choice, for example, is to find ω^k so that the Euclidean norm of the residual at the next step, $\|\mathbf{r}^{k+1}\|$ is minimized along the search direction. We relate the residual at the next iteration to that of the current residual by

$$\mathbf{r}^{k+1} = \mathbf{y} - A\mathbf{x}^{k+1} = \mathbf{y} - A\mathbf{x}^k - \omega^k A\mathbf{r}^k = \mathbf{r}^k - \omega^k A\mathbf{r}^k. \tag{D.15}$$

The *Euclidean norm* of the new residual,

$$\left\|\mathbf{r}^{k+1}\right\| = \sqrt{\left\langle \mathbf{r}^{k+1}, \mathbf{r}^{k+1}\right\rangle} = \sqrt{\sum_{i=1}^{N} \left(\mathbf{r}_i^{k+1}\right)^2} \tag{D.16}$$

is therefore related to the old residual and ω^k by the relation

$$\begin{aligned} \left\|\mathbf{r}^{k+1}\right\|_2^2 &= \left\langle \mathbf{r}^{k+1}, \mathbf{r}^{k+1}\right\rangle \\ &= \left\langle \mathbf{r}^k, \mathbf{r}^k\right\rangle - 2\omega^k \left\langle \mathbf{r}^k, A\mathbf{r}^k\right\rangle + \left(\omega^k\right)^2 \left\langle A\mathbf{r}^k, A\mathbf{r}^k\right\rangle, \end{aligned} \tag{D.17}$$

which is quadratic in ω^k. We find the minimum as a function of ω^k where the derivative is zero,

$$\frac{d\|\mathbf{r}^{k+1}\|_2^2}{d\omega^k} = 0 = -2\left\langle \mathbf{r}^k, A\mathbf{r}^k\right\rangle + 2\omega^k \left\langle A\mathbf{r}^k, A\mathbf{r}^k\right\rangle, \tag{D.18}$$

giving

$$\omega^k = \frac{\langle \mathbf{r}^k, A\mathbf{r}^k\rangle}{\langle A\mathbf{r}^k, A\mathbf{r}^k\rangle}. \tag{D.19}$$

Alternatively, we could choose to minimize some functional other than the next residual along the search direction. For example, we could minimize

$$\Psi(x) = \frac{1}{2}\langle \mathbf{x}, A\mathbf{x}\rangle - \langle \mathbf{x}, \mathbf{y}\rangle, \tag{D.20}$$

which gives

$$\omega^k = \frac{\langle \mathbf{r}^k, \mathbf{r}^k\rangle}{\langle \mathbf{r}^k, A\mathbf{r}^k\rangle}. \tag{D.21}$$

The functional Ψ has as its minimum the solution of the linear system, and this choice of ω^k gives the *method of steepest descent*.

The convergence rate is determined by the *condition number* of the matrix A, $\kappa(A) = \|A\|\|A^{-1}\|$. The larger the condition number, the slower the convergence. Spectral collocation matrices have condition numbers that grow rapidly with N. It is therefore important to mitigate this growth.

We accelerate convergence by introducing a matrix factor to the search direction in addition to the parameter, ω^k, to lower the condition number of the system. We change the iteration (D.12) to

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \omega H^{-1}\mathbf{v}^k, \tag{D.22}$$

where H is the *preconditioning matrix* or *preconditioner*. To see the effect of the preconditioning matrix, consider the Richardson method for which $\mathbf{v}^k = \mathbf{r}^k$. The (exact) solution to the system is the fixed point of the iteration that satisfies

$$\mathbf{x} = \mathbf{x} + \omega^k H^{-1}(\mathbf{y} - A\mathbf{x}), \tag{D.23}$$

so $\mathbf{x}$ also satisfies the modified (preconditioned) system of equations

$$H^{-1}A\mathbf{x} = H^{-1}\mathbf{y}. \tag{D.24}$$

We require the matrix H to have two properties: It should be easy to invert, and it should approximate the original matrix A in such a way that the condition number $\kappa(H^{-1}A)$ of the modified system is lower than $\kappa(A)$. The choice $H = A$ is optimal from the point of view of conditioning since $\kappa(I) = 1$. On the other hand, if A was that easy to invert in the first place, this whole exercise would be pointless. Instead, we settle, and choose H to be some easily invertible approximation of A. In fact, the Jacobi method noted above can be viewed as the preconditioned Richardson method with the preconditioner $H = D$, the diagonal part of A, which is clearly easy to invert.

If we re-trace the steps that we used to determine ω^k and gather the relations, we get the preconditioned minimum residual Richardson method

```
Compute r = y − Ax^0
for k = 1 to N_it do
    Solve Hz = r for z
    ω ← ⟨r, Az⟩ / ⟨Az, Az⟩
    x ← x + ωz
    r ← r − ωAz
end
```

We convert the procedure to the method of steepest descent by changing the definition of ω to (D.21).

Even with preconditioning, the relaxations of the Richardson and steepest descent methods can be slow. The main culprit is that the residual does not necessarily have to point in the direction of the solution. Instead, there may be a large amount of backtracking. The problem is easy to imagine in the context of the steepest descent algorithm and two variables. If the functional is shaped more like a long thin valley instead of a round basin, the downhill direction does not point directly to the bottom.

For symmetric systems, the popular *Conjugate Gradient* method eliminates the backtracking that slows down the convergence of the steepest descent method. Instead of choosing the residual as the search direction, it chooses a direction that is conjugate to all of the previous directions. This guarantees that each search direction does not contain components in the previous directions already searched.

Descriptions of the Conjugate Gradient method can be found in many sources. For those not familiar with it, we recommend [13] or [19] for background. In form, it looks similar to the methods we just discussed,

```
Compute r = y − Ax^0
Solve Hz = r for z
v ← z
c ← ⟨r, z⟩
for k = 1 to N_it do
    z ← Av
    ω ← c / ⟨v, z⟩
    x ← x + ωv
    r ← r − ωz
    Solve Hz = r for z
    d ← ⟨r, z⟩
    v ← z + (d/c) v
    c ← d
end
```

The Conjugate Gradient algorithm is guaranteed to work, however, only for matrices that are symmetric. For non-symmetric systems, it can fail to converge. If the matrix is nonsymmetric we could solve the problem $A^T Ax = A^T y$ so that the coefficient matrix is symmetric. Unfortunately, squaring the matrix increases the condition number, which slows down the convergence rate. For non-symmetric systems, we should use methods specifically derived for them, such as the BiCGStab or the GMRES method. The BiCGStab, for instance is

```
Compute r = y − Ax^0
r̄ ← r
v ← 0; p ← 0
ρ ← 1; α ← 1; ω ← 1
for k = 1 to N_it do
    ρ̂ ← ρ
    ρ = ⟨r̄, r⟩
    β = ρα/(ρ̂ω)
    p = r + β(p − ωv)
    Solve Hy = p for y
    v ← Ay
    α = ρ/⟨r̄, v⟩
    s = r − αv
    Solve Hz = s for z
    t ← Az
    ω = ⟨t, s⟩ / ⟨t, t⟩
    x ← x + αy + ωz
    r ← s − ωt
    if ‖r‖_2 < Tol then Exit
end
```

The GMRES method requires significantly more storage and we will not describe it here. We recommend the book [19] for a description of the GMRES method, should it be needed.

Appendix E
Data Structures

Arrays have both advantages and disadvantages as structures in which to store data. Their main advantage, beyond their simplicity, is that operations on arrays can be computed very efficiently. Standardization of such operations has led to the basic linear algebra subroutines (BLAS) that we discussed in Appendix C. Also, access to a particular element of an array is fast. The main disadvantage, which goes hand in hand with simplicity and efficiency, is that arrays are not very flexible. For best efficiency, we typically must fix the size of an array. If we don't know the size beforehand, the addition of new elements or the deletion of existing elements can require costly allocation and deallocation of blocks of memory, plus the time to copy data from old to new versions of the array. To enable flexible data storage and retrieval, we need more sophisticated data structures.

In this appendix we describe two useful data structures. The first is the linked list. In contrast to arrays, linked lists do not have a specified ordering of the data. They have the advantage that we can easily add and delete elements of the list, so we don't have to know the size of a list in advance. On the negative side, because there is no structured ordering of the data, it is expensive to find a particular element of a list. The second data structure is the hash table. Hash tables are flexible structures that are efficient to search.

Our discussion of data structures will be necessarily brief, and we will discuss only aspects that we need to implement the spectral element algorithms in this book. In particular, our discussion of hash tables is limited to an example of a sparse matrix. Further discussion of the subject of data structures can be found in many books, such as that of Knuth [16]. Note that it is particularly easy to find detailed discussions of linked lists, since they are often used in programming language books for examples of how to use pointers.

E.1 Linked Lists

A *linked list* is a data structure that consists of a collection of *records*. In a *singly linked list*, the record consists of two parts, namely the data that it holds and a pointer to the next record. (Variations that we do not need to consider here include doubly linked lists where a record also has a pointer to the previous record, and circularly linked lists whose last record points to the first record of the list.) We show a diagram of a singly linked list in Fig. E.1.

The records linked in a singly linked list data structure contain data and a pointer (or pointers) to other records. The data stored in the record can be something as simple as an integer value, or as complicated as a structure that contains a variety

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

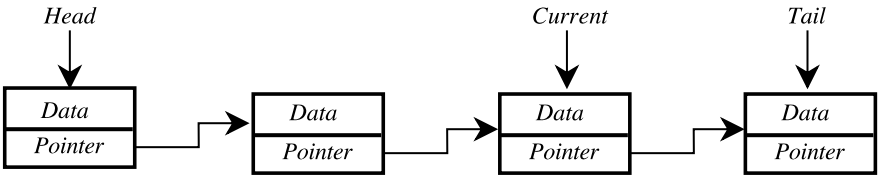

Fig. E.1 Schematics of a singly linked list. Records, represented by *boxes*, are linked by a pointer to the next record in the list. A list can be accessed by its head, tail or current pointer

Algorithm 143: *Record:* An Example Linked List Record Definition

```
Structure Record
  listData // Primitive data type, structure or pointer
  next // Pointer to a Record data type
End Structure Record
```

Algorithm 144: *LinkedList:* A Linked List Class Definition

```
Class LinkedList
  Data:
    head, tail, current // Pointers to Record type
  Procedures:
    Construct(); // Algorithm 145
    Add(data); // Algorithm 145
    GetCurrentData(); // Algorithm 145
    MoveToNext(); // Algorithm 145
    Destruct(); // Algorithm 145
    Print(); // Algorithm 145
End Class LinkedList
```

of other data structures, including arrays and linked lists. To allow for some abstraction, let us assume that the data is stored in a *structure*, such as is provided as a *struct* in C/C++ or a *TYPE* in Fortran. Otherwise, any data that we want to include is appropriate. We can implement the record itself as a structure as well, or as a class. We show an example of a *Record* implemented as a structure in Algorithm 143 (Record). It contains another structure, *listData* that organizes the actual data. It also contains a pointer to a *Record*, which is typically called *next*.

A list itself is referenced by its *head* and *tail* pointers. For convenience, we add a *current* pointer that marks a particular record in the list. These pointers comprise the only data that we need for the list itself shown in Algorithm 144 (LinkedList).

Typical operations to define for a linked list include those that add and delete records and that traverse the list. Other procedures of importance to more general problems might sort the records according to some relation in the stored data, provide copy or merge functions, and the like. For the applications in this book, we need only to add records to the list, traverse the list, and destroy the list. With these needs in mind, we define the *LinkedList* class in Algorithm 144 (LinkedList). It has a con-

Algorithm 145: *LinkedList:Procedures:*

```
Procedure Construct
this.head ⇒ NULL
this.tail ⇒ NULL
this.current ⇒ NULL
End Procedure Construct

Procedure Add
Input: data // is either a primitive, a structure, or a pointer
Allocate newRecord
if this.tail ⇒ NULL then
    this.head ⇒ newRecord
    this.tail ⇒ newRecord
else
    this.tail.next ⇒ newRecord
    this.tail ⇒ newRecord
end
this.current ⇒ this.tail
this.current.next ⇒ NULL
this.current.listData ← data
End Procedure Add

Procedure GetCurrentData
return this.current.listData
End Procedure GetCurrentData

Procedure MoveToNext
this.current ⇒ this.current.next
End Procedure MoveToNext

Procedure Destruct
this.current ⇒ this.head
while this.current ⇏ NULL do
    pNext ⇒ this.current.next
    Deallocate memory pointed to by this.current
    this.current ⇒ pNext
end
End Procedure Destruct

Procedure Print
this.current ⇒ this.head
while this.current ⇏ NULL do
    Print this.current.data
    this.MoveToNext()
end
End Procedure Print
```

structor, destructor, a procedure to add new data, a procedure to get the data in the current record, and a procedure that moves the current pointer to the next element of the list. The implementations of those procedures are shown in Algorithm 145 (LinkList:Procedures).

The constructor for the linked list in Algorithm 145 (LinkList:Procedures) has little to do. It simply sets the pointers to point to *NULL*, which denotes that the pointer does not yet point to any record.

The *Add* procedure takes data as input, creates (allocates) a pointer to a new record and adds the new record to the end of the list, if the list is not empty. If the list is empty, then the new record becomes the start of the list and all the list's pointers point to it.

The *MoveToNext* and *Destruct* procedures illustrate how to navigate the list. To move one step from any position in the list, the *current* pointer is pointed to it's *next* pointer. The *Destruct* procedure shows how to navigate through the entire list. There, we set a pointer to the head and another to its *next* pointer (to keep the next record accessible). A *while* loop then steps through the entire list until the end, which is detected by the current pointer pointing to *NULL*. At each step, the memory to which the current record points is destroyed, and the current pointer is set to point to what was the next record in the list.

We could construct new procedures to perform whole list actions, such as printing the list or searching for a record whose data matches a given criterion, by modifying the *Destruct* procedure. The *Print* procedure in Algorithm 145 (LinkList:Procedures), for instance, loops through the list and prints the data associated with the record pointed to by the *current* pointer. To search for a record with particular data, we would replace the print line in the *Print* procedure with a test on the data.

The *Add* and *Delete* procedures illustrate the flexibility of a linked list to store data when we do not know the number of records ahead of time. The *Destruct* and *Print* procedures show the weakness of a linked list. To access a particular record, we must start at the beginning and search for it sequentially. There is no mechanism for random access to the records, say, to access the fifth element directly, as can be done with an array. Neither can the list find a particular record with a given data value without stepping through the list from the beginning.

E.1.1 Example: Elements that Share a Node

One example of where a linked list is useful is to collect the *id*'s of all the spectral elements in a mesh that share a common corner node. In a structured mesh, a node might be shared by one, two or four elements. In an unstructured mesh a node may in principle be shared by any number of elements (Fig. E.2). Operations on each list are typically of a "ForAll" type, so the entire list must be traversed, but the order in which elements are accessed is not often important. For these needs, a linked list is appropriate to store the shared *elementID*'s for each node.

Let us assume that we have an array of nodes stored by their *nodeID*, and an array of elements stored by their *elementID*. We create such arrays by reading the information from a data file created by a mesh generator. Each node will store (in addition to its other data, such as location) a linked list whose data is just an *elementID* (Fig. E.2). We will not need to modify this shared element data after it is created,

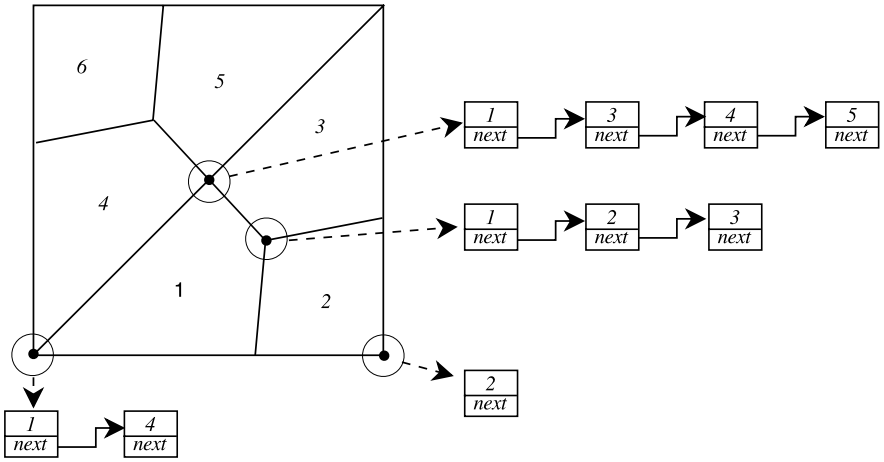

Fig. E.2 Linked lists associated with nodes that store neighboring element id's

so a singly linked list is sufficient to store the data. Each element will have an array that contains the *nodeID*s of the four nodes that define the corners of the element. Each element knows the *nodeID*'s of its corners, since that information is provided by the mesh file. The nodes, however, do not know which elements share them.

To construct the lists of shared elements, we would loop through each element and for each node in the element, add the element's *elementID* to the list associated with the node with that *nodeID*. Specifically, given an array of elements $\{e_j\}_{j=1}^{N_e}$, which stores the *nodeID*s of its nodes in the array $\{nodeID_j\}_{j=1}^{4}$, and array of nodes $\{node_i\}_{i=1}^{N_{node}}$, which stores a linked list of *elementID*s, the logic to create a list of shared elements is

```
for elementID = 1 to N_e do
    for k = 1 to 4 do
        n = e_j.nodeID_k
        node_n.elementList.Add(elementID)
    end
end
```

The shared elements for each node can then be accessed and printed by a simple loop

```
for i = 1 to N_node do
    node_i.elementList.Print()
end
```

E.2 Hash Tables

Hash tables are data structures designed to enable storage and fast retrieval of key-value pairs. An example of a key-value pair is a variable name ("gamma") and its

associated value ("1.4"). The table itself is typically an array. The location of the value in a hash table associated with a key, k, is specified by way of a *hash function*, $H(k)$. In the case of a variable name and value, the hash function converts the name into an integer that tells it where to find the associated value in the table.

A very simple example of a hash table is, in fact, a singly dimensioned array. The key is the array index and the value is what is stored at that index. Multiple keys can be used to identify data; a two dimensional array provides an example where two keys are used to access memory and retrieve the value at that location. If we view a singly dimensioned array as a special case of a hash table, its hash function is just the array index, $H(j) = j$. A doubly dimensioned array could be (and often is) stored columnwise as a singly dimensioned array by creating a hash function that maps the two indices to a single location in the array, e.g., $H(i, j) = i + j * N$, where N is the range of the first index, i.

Although arrays provide fast access to their data, they allocate storage for all possible keys, and only set the value for a key if the data associated with a particular key is present. A matrix, for instance, can be stored as a two-dimensional array. The value of the (i, j)th element of the matrix is stored at a location in memory associated with the two keys, (i, j). A sparse matrix can require much more memory than necessary when stored this way, since most of the elements are zero and don't need to be stored at all. It is for sparse data structures like this that hash tables are useful.

To create a hash table, we need a storage model and a hash function. Each has practical issues. Often it is convenient and efficient to store the data in an array of fixed size. The hash function will then map the keys to an element of that array. However, when we are done, we want that array to be fully populated so that there is no wasted space. Therefore, we don't want to allocate an array that can hold all possible values for all possible keys (as in the matrix storage problem above), only ones that can hold the values for keys that occur. It is not generally desirable to create a "perfect" hash function that generates a unique index for a given set of keys. Instead, *collisions* will be allowed where different keys can give the same hash value, i.e., point to the same location in the array. For instance, it is unrealistic to create a dictionary (word + definition) by allocating storage for every possible combination of letters. Instead we might define a finite array and create a hash function to map to that array. We could create the hash function by assigning a value to each letter, like its position in the alphabet, and adding the values in the word. Thus the word "dad" would be hashed to index $4 + 1 + 4 = 9$. But so would "fab".

Of the many ways to resolve collisions, *chaining* is common. To use chaining, each entry in the array stores a pointer to a linked list, instead of storing values in the hash table array itself. As collisions occur, the new entry is added to the linked list. Then, when it is time to retrieve the value associated with a key, the key is hashed and the linked list at the location given by the hash value is then searched (sequentially) for the actual key. Yes, we have already said that it is slow to search a linked list. But if the table is of a reasonable size, and the hash function is reasonable, then the number of collisions, and hence the number of entries in the linked list, will be small and quickly searched.

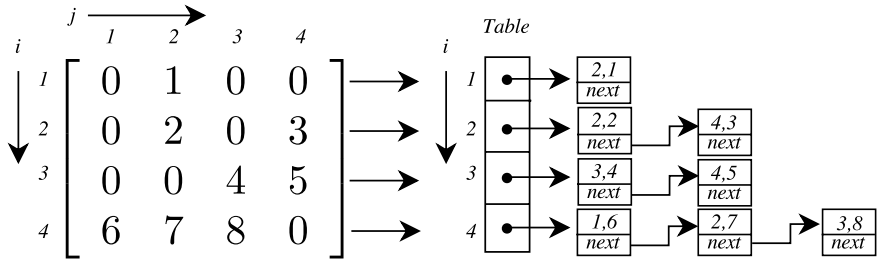

Fig. E.3 Example of sparse matrix representation by a hash table

Algorithm 146: *SparseMatrix:* A Sparse Matrix Class Definition

```
Class SparseMatrix
  Data:
    {table_i} // Array of pointers to LinkedList
  Procedures:
    Construct(N) ; // Algorithm 147
    AddDataForKeys(data, i, j); // Algorithm 147
    DataForKeys(i, j); // Algorithm 147
    ContainsKeys(i, j); // Algorithm 147
    Destruct(); // Algorithm 147
End Class SparseMatrix

Structure TableData
  key // an integer
  data // a primitive type, structure, or pointer
End Structure TableData
```

Since matrices are easy to understand, we show how to use a simple hash table with chaining to store and retrieve values from a sparse matrix. The algorithm will be useful to delete duplicate edges in a mesh. We show a diagram of the data structure in Fig. E.3. To start, note that if the matrix is invertible, then it cannot have a row that is all zeros, that is, every row must have at least one entry. Therefore, the table itself should be an array of length N, where N is the order of the matrix. Each location in the table will correspond to a row in the matrix and the hash function for a matrix element (i, j) is simply $H(i, j) = i$. Clearly there will be collisions, since each row hashes to the same table entry. For this reason, we will use chaining so that each item of the table array will be a pointer to a linked list. A record in the linked list will store two pieces of data, namely the column j and the value of the matrix entry in the ith column and jth row.

We show a sparse matrix class and data definition that describe these requirements in Algorithm 146 (SparseMatrix). Note that we allocate the storage for the array of pointers in the constructor to allow variable size arrays to be created, see Algorithm 147 (SparseMatrix:Procedures). The destructor, *Destruct* tells each of the linked lists to destruct themselves and then deallocates the memory for the array.

Algorithm 147: *SparseMatrix:Procedures:*

```
Procedure Construct
Input: N // Order of the matrix
Allocate memory for this.{table_i}_{i=1}^{N}
for i = 1 to N do
    this.table_i ⇒ NULL
end
End Procedure Construct
```

```
Procedure AddDataForKeys
Input: inData, i, j
if this.table_i ⇒ NULL then this.table_i.Construct()
if this.ContainsKeys(i, j) = FALSE then
    d.key ← j; d.data ← inData // d is of type TableData
    this.table_i.Add(d)
end
End Procedure AddDataForKeys
```

```
Procedure ContainsKeys
Input: i, j
if this.table_i ⇒ NULL then return FALSE
this.table_i.current ⇒ this.table_i.head
while this.table_i.current ⇏ NULL do
    d ← this.table_i.GetCurrentData()
    if d.key = j then return TRUE
    this.table_i.MoveToNext()
end
return FALSE
End Procedure ContainsKeys
```

```
Procedure GetDataForKeys
Input: i, j
if this.table_i ⇒ NULL then setError
this.table_i.current ⇒ this.table_i.head
while this.table_i.current ⇏ NULL do
    d ← this.table_i.GetCurrentData()
    if d.key = j then return d.data
    this.table_i.MoveToNext()
end
setError
End Procedure GetDataForKeys
```

```
Procedure Destruct
for i = 1 to N do
    this.table_i.Destruct()
    this.table_i ⇒ NULL
end
Deallocate memory for this.{table_i}_{i=1}^{N}
End Procedure Destruct
```

We need three basic procedures for the SparseMatrix class. We present implementations of them also in Algorithm 147. The first procedure is the *AddDataForKeys* procedure, which adds the matrix entry for the pair (i, j). It first checks to see if the ith row has been created and constructs a linked list to which it will point. It then checks (just to be safe) to see if the current entry has already been added. If not, the column, j, and value are added to the linked list for row i. The *ContainsKeys* function does a linear search from the head of the list for the ith row and checks to see if the *key* matches the column number. If so, the (i, j)th component is in the table. Finally, the *DataForKeys* procedure returns the data that is stored for the matrix entry, if the entry exists. For safety, it first checks to see if the ith column has been created. If not, an error condition must be set, which we represent generically by a call to some *setError* procedure. (Setting an error could include setting an output "flag" variable, creating and returning an "error handler" structure, or throwing an exception, depending on the computer language being used.) If the ith row is present, the *DataForKeys* procedure then searches through the list for a record whose *key* matches the desired column. On the chance that the *current* position in the list already corresponds to the desired column, the value there is returned. Otherwise, it searches from the beginning by using the *ContainsKeys* procedure.

We could add other procedures to the sparse matrix class, but we don't need them here. For instance, if we wanted to write a sparse matrix-vector multiplication procedure, we would add a *GetNextData* procedure that steps to the next record in the list for the ith row and returns the row (to access the vector element) and the value for the factor.

E.2.1 Example: Avoiding Duplicate Edges in a Mesh

As an example that uses the sparse matrix algorithm, we create an array of unique edges in a mesh. (See Sect. 8.2.) Two *nodeID*'s define an edge, one for the start and one for the end, and hence, an edge is uniquely specified by two keys. If the only information in a mesh file are the nodes and the element connectivity that specifies which nodes are used to create the element, we can generate a collection of edges by looping through each element and creating the four edges from the corner nodes. Defined counter-clockwise on an element, the four edges correspond to the local index of the four corner nodes $(1, 2)$, $(2, 3)$, $(3, 4)$ and $(4, 1)$ (Fig. E.4). To simplify the edge generation procedure, we can create a local *mapping array* (which, itself, can be viewed as a simple hash table), $\{p_{i,k}\}_{i=1,k=1}^{2,4}$, where $(p_{1,1}, p_{2,1}) = (1, 2)$, $(p_{1,2}, p_{2,2}) = (2, 3)$, etc. Suppose an element has an array, $\{nodes_k\}_{k=1}^{4}$ that stores the global *id*'s of its corner nodes. Then the global *id*'s of the start and end nodes for side one of the element are $nodes_{p_{1,1}}$ and $nodes_{p_{2,1}}$. We can construct a linked list of edges, *edgeList* from an array of elements *elArray* by the algorithm

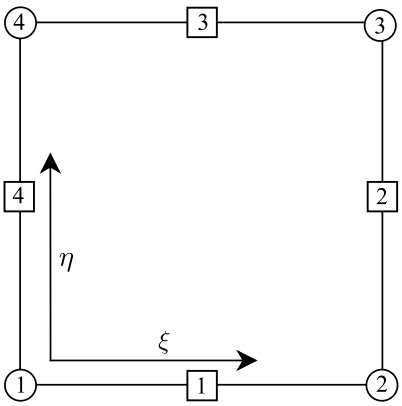

Fig. E.4 Organization of element nodes (*circles*) and edges (*squares*)

```
for j = 1 to N_e do
    for k = 1 to 4 do
        edge.startNode ← elArray_j.nodes_{p1,k}
        edge.endNode ← elArray_j.nodes_{p2,k}
        edgeList.Add(edge)
    end
end
```

Unfortunately, this procedure will create duplicate edges since it counts shared edges twice. To avoid duplicates, we need to test if an edge that is to be about created already exists in the list of edges, i.e., has the same two end nodes. Rather than search the entire list every time, we will create a parallel data structure, in the form of a sparse matrix that we implement as a hash table, so that a search for a given edge is fast. If the edge already exists in the table, we will not add it to the list. At the same time, we will find for free the neighbor to the element across that edge.

Since an edge is uniquely determined by its two end nodes, it is natural to use two keys and two hash functions. A simple choice for the two hash functions is $key1 = Hash1(i, j) = \min(i, j)$ and $key2 = Hash2(i, j) = \max(i, j)$ where i and j are the starting and ending node number. The data held by the table will be simply the edge number of the edge that has the two keys.

The hash table is an efficient way to store the edges so that they can be accessed quickly according to their endpoints. The first key will range from one to the number of nodes, so this will be the size of the array that we construct. Since most mesh generators keep the valence of a node low, ≤ 6, the number of edges with the same first key, which is equal to the valence, is small. Therefore the number of collisions in the table will be small relative to the size of the table.

We modify the procedure above that we wrote to create the list of edges to use the hash table to fill the edge array of the mesh structure of Algorithm 126 (QuadMesh). The result is Algorithm 148 (ConstructMeshEdges). As each edge is found, we first consult the table to see if that edge already exists in the edge array. If it doesn't, we add the edge to the array and the position of the edge in the edge array to the table. If the edge does already exist, then we do not create a new edge. At that

Algorithm 148: *ConstructMeshEdges:* Construct Edge Information for a Spectral Element Mesh

Procedure ConstructMeshEdges
Input: *mesh* `// A QuadMesh`
Uses Algorithms:
Algorithm 126 (QuadMesh)
Algorithm 144 (LinkedList)
Algorithm 146 (SparseMatrix)

edgeTable.*Construct*(*mesh*.N_{node}) `// A SparseMatrix`
for $eID = 1$ **to** *mesh*.K **do**
  for $k = 1$ **to** 4 **do**
    $l1 \leftarrow mesh.edgeMap_{1,k}$
    $l2 \leftarrow mesh.edgeMap_{2,k}$
    $startId \leftarrow mesh.elements_{eID}.nodeIds_{l1}$
    $endId \leftarrow mesh.elements_{eID}.nodeIds_{l2}$
    $key1 \leftarrow Hash1(startId, endId)$
    $key2 \leftarrow Hash2(startId, endId)$
    if *edgeTable*.*ContainsKeys*($key1, key2$) **then**
      $edgeId \leftarrow edgeTable.GetDataForKeys(key1, key2)$
      $e1 \leftarrow mesh.edges_{edgeId}.elementIds_1$
      $s1 \leftarrow mesh.edges_{edgeId}.elementSides_1$
      $l1 \leftarrow mesh.edgeMap_{1,s1}$
      $n1 \leftarrow mesh.elements_{e1}.nodeIDs_{l1}$
      $mesh.edges_{edgeId}.elementId_2 \leftarrow eID$
      if $startId = n1$ **then**
        $mesh.edges_{edgeId}.elementSide_2 \leftarrow k$
      else
        $mesh.edges_{edgeId}.elementSide_2 \leftarrow -k$
      end
    else
      $mesh.N_{edge} \leftarrow mesh.N_{edge} + 1$
      $edgeId \leftarrow mesh.N_{edge}$
      $\{nodes_n\}_{n=1}^{2} \leftarrow \{startId, endId\}$
      $mesh.edges_{edgeId}.Construct(\{nodes_n\}_{n=1}^{2}, eID, k)$
      edgeTable.*AddDataForKeys*($edgeId, key1, key2$)
    end
  end
end
edgeTable.*Destruct*()
return *mesh*
End Procedure ConstructMeshEdges

point in the algorithm, however, we know the current element and side that would have created the duplicate edge. From the array of edges we know the element and side that originally created the edge. Therefore we get the information about the two elements that contribute to an edge for free at the time the duplicate edge is rejected. This information is the element *id*, the element side, and whether or not the direction of the edge is swapped. When we finish creating the array of edges, we destroy it since we no longer need it.

References

1. Abramowitz, M., Stegun, I.A.: Handbook of Mathematical Functions: with Formulas, Graphs, and Mathematical Tables. Dover, New York (1965)
2. Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Croz, J.D., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.: LAPACK Users' Guide, 3rd edn. SIAM, Philadelphia (1999)
3. Ascher, U.M., Petzold, L.R.: Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations. SIAM, Philadelphia (1998)
4. Boyd, J.P.: Chebyshev and Fourier Spectral Methods, 2nd revised edn. Dover, New York (2001)
5. Bracewell, R.: The Fourier Transform and Its Applications. McGraw-Hill, New York (1999)
6. Brigham, E.: Fast Fourier Transform and Its Applications. Prentice Hall, New York (1988)
7. Canuto, C., Hussaini, M., Quarteroni, A., Zang, T.: Spectral Methods: Fundamentals in Single Domains. Springer, Berlin (2006)
8. Canuto, C., Hussaini, M., Quarteroni, A., Zang, T.: Spectral Methods: Evolution to Complex Geometries and Applications to Fluid Dynamics. Springer, Berlin (2007)
9. Deville, M., Fischer, P., Mund, E.: High Order Methods for Incompressible Fluid Flow. Cambridge University Press, Cambridge (2002)
10. Don, W.S., Solomonoff, A.: Accuracy and speed in computing the Chebyshev collocation derivative. SIAM J. Sci. Comput. **16**, 1253–1268 (1995)
11. Farrashkhalvat, M., Miles, J.P.: Basic Structured Grid Generation: With an Introduction to Unstructured Grid Generation. Butterworth-Heinemann, Stoneham (2003)
12. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proc. IEEE **93**(2), 216–231 (2005)
13. Golub, G.H., Loan, C.F.V.: Matrix Computations. Johns Hopkins University Press, Baltimore (1996)
14. Hesthaven, J.S., Gottlieb, S., Gottlieb, D.: Spectral Methods for Time-Dependent Problems. Cambridge University Press, Cambridge (2007)
15. Knupp, P.M., Steinberg, S.: Fundamentals of Grid Generation. CRC Press, Boca Raton (1993)
16. Knuth, D.E.: The Art of Computer Programming. Addison-Wesley, Reading (1998)
17. Lambert, J.D.: Numerical Methods for Ordinary Differential Systems: The Initial Value Problem. Wiley, New York (1991)
18. Overton, M.L.: Numerical Computing with IEEE Floating Point Arithmetic. SIAM, Philadelphia (2001)
19. Saad, Y.: Iterative Methods for Sparse Linear Systems. SIAM, Philadelphia (2003)
20. Schwab, C.: p- and hp-Finite Element Methods: Theory and Applications in Solid and Fluid Mechanics. Oxford University Press, London (1988)
21. Shen, J., Tang, T.: Spectral and High-Order Methods with Applications. Science Press, Beijing (2006)
22. Swarztrauber, P.: On computing the points and weights for Gauss-Legendre quadrature. SIAM J. Sci. Comput. **24**(3), 945–954 (2002)
23. Temperton, C.: Self-sorting in-place fast Fourier transforms. SIAM J. Sci. Stat. Comput. **12**(4), 808–823 (1991)
24. Toro, E.F.: Riemann Solvers and Numerical Methods for Fluid Dynamics. Springer, Berlin (1999)
25. Williamson, J.: Low storage Runge-Kutta schemes. J. Comput. Phys. **35**, 48–56 (1980)
26. Yakimiw, E.: Accurate computation of weights in classical Gauss-Christoffel quadrature rules. J. Comput. Phys. **129**, 406–430 (1996)

Index of Algorithms

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

Subject Index

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