Chapter 3
Elliptic Equations

In more than one space dimension, the steady-state equations discussed in Chapter 2 generalize naturally to *elliptic* partial differential equations, as discussed in Section E.1.2. In two space dimensions a constant-coefficient elliptic equation has the form

$$a_1 u_{xx} + a_2 u_{xy} + a_3 u_{yy} + a_4 u_x + a_5 u_y + a_6 u = f, \tag{3.1}$$

where the coefficients a_1, a_2, a_3 satisfy

$$a_2^2 - 4a_1 a_3 < 0. \tag{3.2}$$

This equation must be satisfied for all (x, y) in some region of the plane Ω, together with some boundary conditions on $\partial\Omega$, the boundary of Ω. For example, we may have Dirichlet boundary conditions in which case $u(x, y)$ is given at all points $(x, y) \in \partial\Omega$. If the ellipticity condition (3.2) is satisfied, then this gives a well-posed problem. If the coefficients vary with x and y, then the ellipticity condition must be satisfied at each point in Ω.

3.1 Steady-state heat conduction

Equations of elliptic character often arise as steady-state equations in some region of space, associated with some time-dependent physical problem. For example, the diffusion or heat conduction equation in two space dimensions takes the form

$$u_t = (\kappa u_x)_x + (\kappa u_y)_y + \psi, \tag{3.3}$$

where $\kappa(x, y) > 0$ is a diffusion or heat conduction coefficient that may vary with x and y, and $\psi(x, y, t)$ is a source term. The solution $u(x, y, t)$ generally will vary with time as well as space. We also need initial conditions $u(x, y, 0)$ in Ω and boundary conditions at each point in time at every point on the boundary of Ω. If the boundary conditions and source terms are independent of time, then we expect a steady state to exist, which we can find by solving the elliptic equation

$$(\kappa u_x)_x + (\kappa u_y)_y = f, \tag{3.4}$$

where again we set $f(x, y) = -\psi(x, y)$, together with the boundary conditions. Note that (3.2) is satisfied at each point, provided $\kappa > 0$ everywhere.

We first consider the simplest case where $\kappa \equiv 1$. We then have the *Poisson problem*

$$u_{xx} + u_{yy} = f. \tag{3.5}$$

In the special case $f \equiv 0$, this reduces to *Laplace's equation*,

$$u_{xx} + u_{yy} = 0. \tag{3.6}$$

We also need to specify boundary conditions all around the boundary of the region Ω. These could be Dirichlet conditions, where the temperature $u(x, y)$ is specified at each point on the boundary, or Neumann conditions, where the normal derivative (the heat flux) is specified. We may have Dirichlet conditions specified at some points on the boundary and Neumann conditions at other points.

In one space dimension the corresponding Laplace's equation $u''(x) = 0$ is trivial: the solution is a linear function connecting the two boundary values. In two dimensions even this simple equation in nontrivial to solve, since boundary values can now be specified at every point along the curve defining the boundary. Solutions to Laplace's equation are called *harmonic functions*. You may recall from complex analysis that if $g(z)$ is any complex analytic function of $z = x + iy$, then the real and imaginary parts of this function are harmonic. For example, $g(z) = z^2 = (x^2 - y^2) + 2ixy$ is analytic and the functions $x^2 - y^2$ and $2xy$ are both harmonic.

The operator ∇^2 defined by

$$\nabla^2 u = u_{xx} + u_{yy}$$

is called the *Laplacian*. The notation ∇^2 comes from the fact that, more generally,

$$(\kappa u_x)_x + (\kappa u_y)_y = \nabla \cdot (\kappa \nabla u),$$

where ∇u is the gradient of u,

$$\nabla u = \begin{bmatrix} u_x \\ u_y \end{bmatrix}, \tag{3.7}$$

and $\nabla\cdot$ is the divergence operator,

$$\nabla \cdot \begin{bmatrix} u \\ v \end{bmatrix} = u_x + v_y. \tag{3.8}$$

The symbol Δ is also often used for the Laplacian but would lead to confusion in numerical work where Δx and Δy are often used for grid spacing.

3.2 The 5-point stencil for the Laplacian

To discuss discretizations, first consider the Poisson problem (3.5) on the unit square $0 \le x \le 1$, $0 \le y \le 1$ and suppose we have Dirichlet boundary conditions. We will use a uniform Cartesian grid consisting of grid points (x_i, y_j), where $x_i = i\Delta x$ and $y_j = j\Delta y$. A section of such a grid is shown in Figure 3.1.

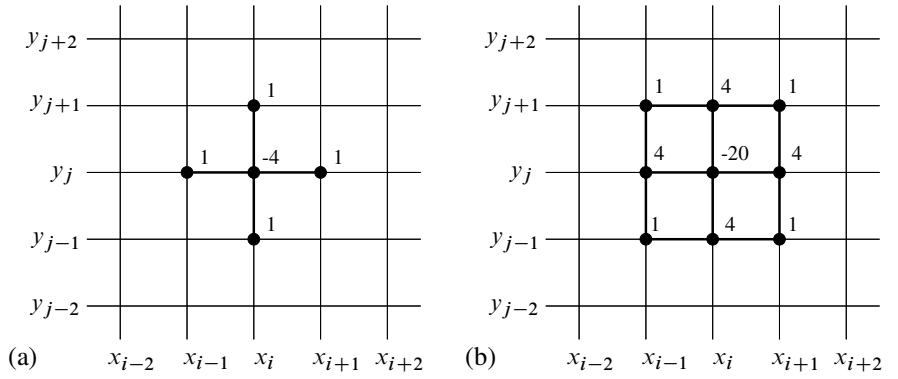

Figure 3.1. *Portion of the computational grid for a two-dimensional elliptic equation. (a) The 5-point stencil for the Laplacian about the point* (i, j) *is also indicated. (b) The 9-point stencil is indicated, which is discussed in Section 3.5.*

Let u_{ij} represent an approximation to $u(x_i, y_j)$. To discretize (3.5) we replace the x- and y-derivatives with centered finite differences, which gives

$$\frac{1}{(\Delta x)^2}(u_{i-1,j} - 2u_{ij} + u_{i+1,j}) + \frac{1}{(\Delta y)^2}(u_{i,j-1} - 2u_{ij} + u_{i,j+1}) = f_{ij}. \tag{3.9}$$

For simplicity of notation we will consider the special case where $\Delta x = \Delta y \equiv h$, although it is easy to handle the general case. We can then rewrite (3.9) as

$$\frac{1}{h^2}(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{ij}) = f_{ij}. \tag{3.10}$$

This finite difference scheme can be represented by the *5-point stencil* shown in Figure 3.1. We have both an unknown u_{ij} and an equation of the form (3.10) at each of m^2 grid points for $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, m$, where $h = 1/(m + 1)$ as in one dimension. We thus have a linear system of m^2 unknowns. The difference equations at points near the boundary will of course involve the known boundary values, just as in the one-dimensional case, which can be moved to the right-hand side.

3.3 Ordering the unknowns and equations

If we collect all these equations together into a matrix equation, we will have an $m^2 \times m^2$ matrix that is very *sparse*, i.e., most of the elements are zero. Since each equation involves at most five unknowns (fewer near the boundary), each row of the matrix has at most five nonzeros and at least $m^2 - 5$ elements that are zero. This is analogous to the tridiagonal matrix (2.9) seen in the one-dimensional case, in which each row has at most three nonzeros.

Recall from Section 2.14 that the structure of the matrix depends on the order we choose to enumerate the unknowns. Unfortunately, in two space dimensions the structure of the matrix is not as compact as in one dimension, no matter how we order the

unknowns, and the nonzeros cannot be as nicely clustered near the main diagonal. One obvious choice is the *natural rowwise ordering*, where we take the unknowns along the bottom row, $u_{11}, u_{21}, u_{31}, \ldots, u_{m1}$, followed by the unknowns in the second row, $u_{12}, u_{22}, \ldots, u_{m2}$, and so on, as illustrated in Figure 3.2(a). The vector of unknowns is partitioned as

$$u = \begin{bmatrix} u^{[1]} \\ u^{[2]} \\ \vdots \\ u^{[m]} \end{bmatrix}, \quad \text{where } u^{[j]} = \begin{bmatrix} u_{1j} \\ u_{2j} \\ \vdots \\ u_{mj} \end{bmatrix}. \tag{3.11}$$

This gives a matrix equation where A has the form

$$A = \frac{1}{h^2} \begin{bmatrix} T & I & & & & \\ I & T & I & & & \\ & I & T & I & & \\ & & \ddots & \ddots & \ddots & \\ & & & I & T \end{bmatrix}, \tag{3.12}$$

which is an $m \times m$ *block tridiagonal matrix* in which each block T or I is itself an $m \times m$ matrix,

$$T = \begin{bmatrix} -4 & 1 & & & & \\ 1 & -4 & 1 & & & \\ & 1 & -4 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -4 \end{bmatrix},$$

and I is the $m \times m$ identity matrix. While this has a nice structure, the 1 values in the I matrices are separated from the diagonal by $m-1$ zeros, since these coefficients correspond to grid points lying above or below the central point in the stencil and hence are in the next or previous row of unknowns.

Another possibility, which has some advantages in the context of certain iterative methods, is to use the *red-black ordering* (or checkerboard ordering) shown in Figure 3.2. This is the two-dimensional analogue of the odd-even ordering that leads to the matrix (2.63) in one dimension. This ordering is significant because all four neighbors of a red grid point are black points, and vice versa, and it leads to a matrix equation with the structure

$$\begin{bmatrix} D & H \\ H^T & D \end{bmatrix} \begin{bmatrix} u_{\text{red}} \\ u_{\text{black}} \end{bmatrix} = \begin{bmatrix} f_{\text{red}} \\ -f_{\text{black}} \end{bmatrix}, \tag{3.13}$$

where $D = -\frac{4}{h^2} I$ is a diagonal matrix of dimension $m^2/2$ and H is a banded matrix of the same dimension with four nonzero diagonals.

When direct methods such as Gaussian elimination are used to solve the system, one typically wants to order the equations and unknowns so as to reduce the amount of fill-in during the elimination procedure as much as possible. This is done automatically if the backslash operator in MATLAB is used to solve the system, provided it is set up using sparse storage; see Section 3.7.1.

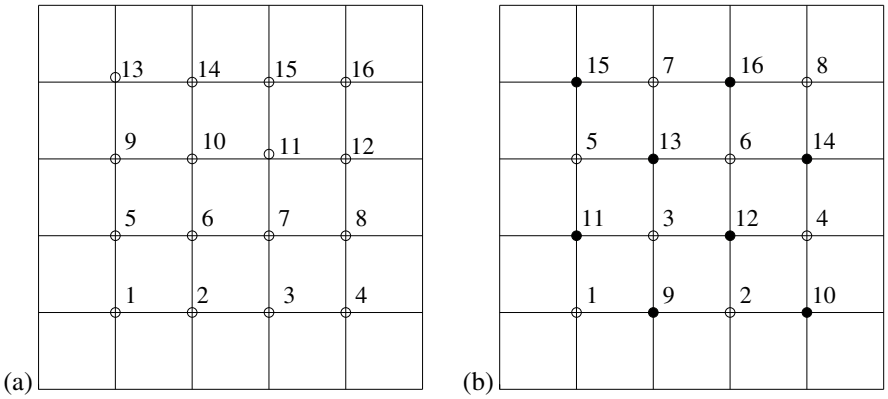

Figure 3.2. (a) *The natural rowwise order of unknowns and equations on a* 4×4 *grid.* (b) *The red-black ordering.*

3.4 Accuracy and stability

The discretization of the two-dimensional Poisson problem can be analyzed using exactly the same approach as we used for the one-dimensional boundary value problem. The local truncation error τ_{ij} at the (i, j) grid point is defined in the obvious way,

$$\tau_{ij} = \frac{1}{h^2}(u(x_{i-1}, y_j) + u(x_{i+1}, y_j) + u(x_i, y_{j-1}) + u(x_i, y_{j+1}) - 4u(x_i, y_j)) - f(x_i, y_j),$$

and by splitting this into the second order difference in the x- and y-directions it is clear from previous results that

$$\tau_{ij} = \frac{1}{12}h^2(u_{xxxx} + u_{yyyy}) + O(h^4).$$

For this linear system of equations the global error $E_{ij} = u_{ij} - u(x_i, y_j)$ then solves the linear system

$$A^h E^h = -\tau^h$$

just as in one dimension, where A^h is now the discretization matrix with mesh spacing h, e.g., the matrix (3.12) if the rowwise ordering is used. The method will be globally second order accurate in some norm provided that it is stable, i.e., that $\|(A^h)^{-1}\|$ is uniformly bounded as $h \to 0$.

In the 2-norm this is again easy to check for this simple problem, since we can explicitly compute the spectral radius of the matrix, as we did in one dimension in Section 2.10. The eigenvalues and eigenvectors of A can now be indexed by two parameters p and q corresponding to wave numbers in the x- and y-directions for $p, q = 1, 2, \ldots, m$. The (p, q) eigenvector $u^{p,q}$ has the m^2 elements

$$u_{ij}^{p,q} = \sin(p\pi ih)\sin(q\pi jh). \tag{3.14}$$

The corresponding eigenvalue is

$$\lambda_{p,q} = \frac{2}{h^2}((\cos(p\pi h) - 1) + (\cos(q\pi h) - 1)). \tag{3.15}$$

The eigenvalues are strictly negative (A is negative definite) and the one closest to the origin is

$$\lambda_{1,1} = -2\pi^2 + O(h^2).$$

The spectral radius of $(A^h)^{-1}$, which is also the 2-norm, is thus

$$\rho((A^h)^{-1}) = 1/\lambda_{1,1} \approx -1/2\pi^2.$$

Hence the method is stable in the 2-norm.

While we're at it, let's also compute the condition number of the matrix A^h, since it turns out that this is a critical quantity in determining how rapidly certain iterative methods converge. Recall that the 2-norm condition number is defined by

$$\kappa_2(A) = \|A\|_2 \|A^{-1}\|_2.$$

We've just seen that $\|(A^h)^{-1}\|_2 \approx -1/2\pi^2$ for small h, and the norm of A is given by its spectral radius. The largest eigenvalue of A (in magnitude) is

$$\lambda_{m,m} \approx -\frac{8}{h^2}$$

and so

$$\kappa_2(A) \approx \frac{4}{\pi^2 h^2} = O\left(\frac{1}{h^2}\right) \quad \text{as } h \to 0. \tag{3.16}$$

The fact that the matrix becomes very ill-conditioned as we refine the grid is responsible for the slow-down of iterative methods, as discussed in Chapter 4.

3.5 The 9-point Laplacian

Above we used the 5-point Laplacian, which we will denote by $\nabla_5^2 u_{ij}$, where this denotes the left-hand side of equation (3.10). Another possible approximation is the 9-point Laplacian

$$\begin{aligned}\nabla_9^2 u_{ij} = \frac{1}{6h^2}[&4u_{i-1,j} + 4u_{i+1,j} + 4u_{i,j-1} + 4u_{i,j+1} \\ &+ u_{i-1,j-1} + u_{i-1,j+1} + u_{i+1,j-1} + u_{i+1,j+1} - 20u_{ij}]\end{aligned} \tag{3.17}$$

as indicated in Figure 3.1. If we apply this to the true solution and expand in Taylor series, we find that

$$\nabla_9^2 u(x_i, y_j) = \nabla^2 u + \frac{1}{12}h^2(u_{xxxx} + 2u_{xxyy} + u_{yyyy}) + O(h^4).$$

At first glance this discretization looks no better than the 5-point discretization since the error is still $O(h^2)$. However, the additional terms lead to a very nice form for the dominant error term, since

$$u_{xxxx} + 2u_{xxyy} + u_{yyyy} = \nabla^2(\nabla^2 u) \equiv \nabla^4 u.$$

This is the Laplacian of the Laplacian of u and ∇^4 is called the *biharmonic operator*. If we are solving $\nabla^2 u = f$, then we have

$$u_{xxxx} + 2u_{xxyy} + u_{yyyy} = \nabla^2 f.$$

Hence we can compute the dominant term in the truncation error easily from the known function f without knowing the true solution u to the problem.

In particular, if we are solving Laplace's equation, where $f = 0$, or more generally if f is a harmonic function, then this term in the local truncation error vanishes and the 9-point Laplacian would give a fourth order accurate discretization of the differential equation.

More generally, we can obtain a fourth order accurate method of the form

$$\nabla_9^2 u_{ij} = f_{ij} \tag{3.18}$$

for arbitrary smooth functions $f(x, y)$ by defining

$$f_{ij} = f(x_i, y_j) + \frac{h^2}{12}\nabla^2 f(x_i, y_j). \tag{3.19}$$

We can view this as deliberately introducing an $O(h^2)$ error into the right-hand side of the equation that is chosen to cancel the $O(h^2)$ part of the local truncation error. Taylor series expansion easily shows that the local truncation error of the method (3.18) is now $O(h^4)$. This is the two-dimensional analogue of the modification (2.117) that gives fourth order accuracy for the boundary value problem $u''(x) = f(x)$.

If we have only data $f(x_i, y_j)$ at the grid points (but we know that the underlying function is sufficiently smooth), then we can still achieve fourth order accuracy by using

$$f_{ij} = f(x_i, y_j) + \frac{h^2}{12}\nabla_5^2 f(x_i, y_j)$$

instead of (3.19).

This is a trick that often can be used in developing numerical methods—introducing an "error" into the equations that is carefully chosen to cancel some other error.

Note that the same trick wouldn't work with the 5-point Laplacian, or at least not as directly. The form of the truncation error in this method depends on $u_{xxxx} + u_{yyyy}$. There is no way to compute this directly from the original equation without knowing u. The extra points in the 9-point stencil convert this into the Laplacian of f, which can be computed if f is sufficiently smooth.

On the other hand, a two-pass approach could be used with the 5-point stencil, in which we first estimate u by solving with the standard 5-point scheme to get a second order accurate estimate of u. We then use this estimate of u to approximate $u_{xxxx} + u_{yyyy}$ and then solve a second time with a right-hand side that is modified to eliminate the dominant term of the local truncation error. This would be more complicated for this particular problem, but this idea can be used much more generally than the above trick, which depends on the special form of the Laplacian. This is the method of *deferred corrections*, already discussed for one dimension in Section 2.20.3.

3.6 Other elliptic equations

In Chapter 2 we started with the simplest boundary value problem for the constant coefficient problem $u''(x) = f(x)$ but then introduced various, more interesting problems, such as variable coefficients, nonlinear problems, singular perturbation problems, and boundary or interior layers.

In the multidimensional case we have discussed only the simplest Poisson problem, which in one dimension reduces to $u''(x) = f(x)$. All the further complications seen in one dimension can also arise in multidimensional problems. For example, heat conduction in a heterogeneous two-dimensional domain gives rise to the equation

$$(\kappa(x, y)u_x(x, y))_x + (\kappa(x, y)u_y(x, y))_y = f(x, y), \tag{3.20}$$

where $\kappa(x, y)$ is the varying heat conduction coefficient. In any number of space dimensions this equation can be written as

$$\nabla \cdot (\kappa \nabla u) = f. \tag{3.21}$$

These problems can be solved by generalizations of the one-dimensional methods. The terms $(\kappa(x, y)u_x(x, y))_x$ and $(\kappa(x, y)u_y(x, y))_y$ can each be discretized as in the one-dimensional case, again resulting in a 5-point stencil in two dimensions.

Nonlinear elliptic equations also arise in multidimensions, in which case a system of nonlinear algebraic equations will result from the discretization. A Newton method can be used as in one dimension, but now in each Newton iteration a large sparse linear system will have to be solved. Typically the Jacobian matrix has a sparsity pattern similar to those seen above for linear elliptic equations. See Section 4.5 for a brief discussion of Newton–Krylov iterative methods for such problems.

In multidimensional problems there is an additional potential complication that is not seen in one dimension: the domain Ω where the boundary value problem is posed may not be a simple rectangle as we have supposed in our discussion so far. When the solution exhibits boundary or interior layers, then we would also like to cluster grid points or adaptively refine the grid in these regions. This often presents a significant challenge that we will not tackle in this book.

3.7 Solving the linear system

Two fundamentally different approaches could be used for solving the large linear systems that arise from discretizing elliptic equations. A *direct method* such as Gaussian elimination produces an exact solution (or at least would in exact arithmetic) in a finite number of operations. An *iterative method* starts with an initial guess for the solution and attempts to improve it through some iterative procedure, halting after a sufficiently good approximation has been obtained.

For problems with large sparse matrices, iterative methods are often the method of choice, and Chapter 4 is devoted to a study of several iterative methods. Here we briefly consider the operation counts for Gaussian elimination to see the potential pitfalls of this approach.

It should be noted, however, that on current computers direct methods can be successfully used for quite large problems, provided appropriate sparse storage and efficient

elimination procedures are used. See Section 3.7.1 for some comments on setting up sparse matrices such as (3.12) in MATLAB.

It is well known (see, e.g., [35], [82], [91]) that for a general $N \times N$ *dense* matrix (one with few elements equal to zero), performing Gaussian elimination requires $O(N^3)$ operations. (There are $N(N-1)/2 = O(N^2)$ elements below the diagonal to eliminate, and eliminating each one requires $O(N)$ operations to take a linear combination of the rows.)

Applying a general Gaussian elimination program blindly to the matrices we are now dealing with would be disastrous, or at best extremely wasteful of computer resources. Suppose we are solving the three-dimensional Poisson problem on a $100 \times 100 \times 100$ grid—a modest problem these days. Then $N = m^3 = 10^6$ and $N^3 = 10^{18}$. On a reasonably fast desktop that can do on the order of 10^{10} floating point operations per second (10 gigaflops), this would take on the order of 10^8 seconds, which is more than 3 years. More sophisticated methods can solve this problem in seconds.

Moreover, even if speed were not an issue, memory would be. Storing the full matrix A in order to modify the elements and produce L and U would require N^2 memory locations. In 8-byte arithmetic this requires $8\,N^2$ bytes. For the problem mentioned above, this would be 8×10^{12} bytes, or eight terabytes. One advantage of iterative methods is that they do not store the matrix at all and at most need to store the nonzero elements.

Of course with Gaussian elimination it would be foolish to store all the elements of a sparse matrix, since the vast majority are zero, or to apply the procedure blindly without taking advantage of the fact that so many elements are already zero and hence do not need to be eliminated.

As an extreme example, consider the one-dimensional case where we have a tridiagonal matrix as in (2.9). Applying Gaussian elimination requires eliminating only the nonzeros along the subdiagonal, only $N-1$ values instead of $N(N-1)/2$. Moreover, when we take linear combinations of rows in the course of eliminating these values, in most columns we will be taking linear combinations of zeros, producing zero again. If we do not do pivoting, then only the diagonal elements are modified. Even with partial pivoting, at most we will introduce one extra superdiagonal of nonzeros in the upper triangular U that were not present in A. As a result, it is easy to see that applying Gaussian elimination to an $m \times m$ tridiagonal system requires only $O(m)$ operations, not $O(m^3)$, and that the storage required is $O(m)$ rather than $O(m^2)$.

Note that this is the best we could hope for in one dimension, at least in terms of the order of magnitude. There are m unknowns and even if we had exact formulas for these values, it would require $O(m)$ work to evaluate them and $O(m)$ storage to save them.

In two space dimensions we can also take advantage of the sparsity and structure of the matrix to greatly reduce the storage and work required with Gaussian elimination, although not to the minimum that one might hope to attain. On an $m \times m$ grid there are $N = m^2$ unknowns, so the best one could hope for is an algorithm that computes the solution in $O(N) = O(m^2)$ work using $O(m^2)$ storage. Unfortunately, this cannot be achieved with a direct method.

One approach that is better than working with the full matrix is to observe that the A is a banded matrix with bandwidth m both above and below the diagonal. Since a general $N \times N$ banded matrix with a nonzero bands above the diagonal and b below the diagonal

can be factored in $O(Nab)$ operations, this results in an operation count of $O(m^4)$ for the two-dimensional Poisson problem.

A more sophisticated approach that takes more advantage of the special structure (and the fact that there are already many zeros within the bandwidth) is the *nested dissection* algorithm [34]. This algorithm requires $O(m^3)$ operations in two dimensions. It turns out this is the best that can be achieved with a direct method based on Gaussian elimination. George proved (see [34]) that any elimination method for solving this problem requires at least $O(m^3)$ operations.

For certain special problems, very fast direct methods can be used, which are much better than standard Gaussian elimination. In particular, for the Poisson problem on a rectangular domain there are *fast Poisson solvers* based on the fast Fourier transform that can solve on an $m \times m$ grid in two dimensions in $O(m^2 \log m)$ operations, which is nearly optimal. See [87] for a review of this approach.

3.7.1 Sparse storage in MATLAB

If you are going to work in MATLAB with sparse matrices arising from finite difference methods, it is important to understand and use the sparse matrix commands that set up matrices using sparse storage, so that only the nonzeros are stored. Type `help sparse` to get started.

As one example, the matrix of (3.12) can be formed in MATLAB by the commands

```
I = eye(m);
e = ones(m,1);
T = spdiags([e -4*e e],[-1 0 1],m,m);
S = spdiags([e e],[-1 1],m,m);
A = (kron(I,T) + kron(S,I))/h^2;
```

The `spy(A)` command is also useful for looking at the nonzero structure of a matrix.

The backslash command in MATLAB can be used to solve systems using sparse storage, and it implements highly efficient direct methods using sophisticated algorithms for dynamically ordering the equations to minimize fill-in, as described by Davis [24].