Chapter 5

Matrix Computations

The next item on our agenda is the linear equation problem $Ax = b$. However, before we get into algorithmic details, it is important to study two simpler calculations: matrix-vector multiplication and matrix-matrix multiplication. Both operations are supported in MATLAB so in that sense there is "nothing to do." However, there is much to learn by studying how these computations can be implemented. Matrix-vector multiplications arise during the course of solving $Ax = b$ problems. Moreover, it is good to uplift our ability to think at the matrix-vector level before embarking on a presentation of $Ax = b$ solvers.

The act of setting up a matrix problem also deserves attention. Often, the amount of work that is required to initialize an n-by-n matrix is as much as the work required to solve for x. We pay particular attention to the common setting when each matrix entry a_{ij} is an evaluation of a continuous function $f(x, y)$.

A theme throughout this chapter is the exploitation of structure. It is frequently the case that there is a pattern among the entries in A which can be used to reduce the amount of work. The fast Fourier transform and the fast Strassen matrix multiply algorithm are presented as examples of recursion in the matrix computations. The organization of matrix-matrix multiplication on a ring of processors is also studied and gives us a nice snapshot of what algorithm development is like in a distributed memory environment.

5.1 Setting Up Matrix Problems

Before a matrix problem can be solved, it must be set up. In many applications, the amount of work associated with the set-up phase rivals the amount of work associated with the solution

phase. Therefore, it is in our interest to acquire intuition about this activity. It is also an occasion to see how many of MATLAB's vector capabilities extend to the matrix level.

5.1.1 Simple ij Recipes

If the entries in a matrix $A = (a_{ij})$ are specified by simple recipes, such as

$$a_{ij} = \frac{1}{i+j-1},$$

then a double-loop script can be used for its computation:

```
A = zeros(n,n);
for i=1:n
   for j=1:n
      A(i,j) = 1/(i+j-1);
   end
end
```

Preallocation with `zeros(n,n)` reduces memory management overhead.

Sometimes the matrix defined has patterns that can be exploited. The preceding matrix is *symmetric* since $a_{ij} = a_{ji}$ for all i and j. This means that the (i, j) recipe need only be applied half the time:

```
A = zeros(n,n);
for i=1:n
   for j=i:n
      A(i,j) = 1/1(i+j-1);
      A(j,i) = A(i,j);
   end
end
```

This particular example is a *Hilbert matrix*, and it so happens that there a built-in function `A = hilb(n)` that can be used in lieu of the preceding scripts. Enter the command `type hilb` to see a fully vectorized implementation.

The setting up of a matrix can often be made more efficient by exploiting relationships that exist between the entries. Consider the construction of the lower triangular matrix of binomial coefficients:

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 2 & 1 & 0 \\ 1 & 3 & 3 & 1 \end{bmatrix}.$$

The binomial coefficient "m-choose-k" is defined by

$$\begin{pmatrix} m \\ k \end{pmatrix} = \begin{cases} \dfrac{m!}{k!(m-k)!} & \text{if } 0 \le k \le m \\ \\ 0 & \text{otherwise} \end{cases}.$$

If $k \leq m$, then it specifies the number of ways that k objects can be selected from a set of m objects. The ij entry of the matrix we are setting up is defined by

$$p_{ij} = \begin{pmatrix} i-1 \\ j-1 \end{pmatrix}.$$

If we simply compute each entry using the factorial definition, then $O(n^3)$ flops are involved. On the other hand, from the 5-by-5 case we notice that P is lower triangular with ones on the diagonal and in the first column. An entry not in these locations is the sum of its "north" and "northwest" neighbors. That is,

$$p_{ij} = p_{i-1,j-1} + p_{i-1,j}.$$

This permits the following set-up strategy:

```
P = zeros(n,n);
P(:,1) = ones(n,1);
for i=2:n
   for j=2:i
      P(i,j) = P(i-1,j-1) + P(i-1,j);
   end
end
```

This script involves $O(n^2)$ flops and is therefore an order of magnitude faster than the method that ignores the connections between the p_{ij}.

5.1.2 Matrices Defined by a Vector of Parameters

Many matrices are defined in terms of a vector of parameters. Recall the *Vandermonde* matrices from Chapter 2:

$$V = \begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 \\ 1 & x_2 & x_2^2 & x_2^3 \\ 1 & x_3 & x_3^2 & x_3^3 \\ 1 & x_4 & x_4^2 & x_4^3 \end{bmatrix}.$$

We developed several set-up strategies but settled on the following column-oriented technique:

```
n = length(x);
V(:,1) = ones(n,1);
for j=2:n
   % Set up column j.
   V(:,j) = x.*V(:,j-1);
end
```

The *circulant* matrices are also of this genre. They too are defined by a vector of parameters, for example

$$C = \begin{bmatrix} a_1 & a_2 & a_3 & a_4 \\ a_4 & a_1 & a_2 & a_3 \\ a_3 & a_4 & a_1 & a_2 \\ a_2 & a_3 & a_4 & a_1 \end{bmatrix}.$$

Each row in a circulant is a shifted version of the row above it. Here are two circulant set-up functions:

```
  function C = Circulant1(a)
% C = Circulant1(a) is a circulant matrix with first row equal to a.

n = length(a);
C = zeros(n,n);
for i=1:n
   for j=1:n
     C(i,j) = a(rem(n-i+j,n)+1);
   end
end

  function C = Circulant2(a)
% C = Circulant2(a) is a circulant matrix with first row equal to a.

n = length(a);
C = zeros(n,n);
C(1,:) = a;
for i=2:n
   C(i,:) = [ C(i-1,n) C(i-1,1:n-1) ];
end
```

`Circulant1` exploits the fact that $c_{ij} = a_{((n-i+j) \bmod n)+1}$ and is a scalar-level implementation. `Circulant2` exploits the fact that $C(i,:)$ is a left shift of $C(i-1,:)$ and is a vector-level implementation. The script `CircBench` compares t_1 (the time required by `Circulant1`) with t_2 (the time required by `Circulant2`):

```
   n     t1/t2
-----------------
  100    19.341
  200    20.903
  400    31.369
```

Once again we see that it pays to vectorize in MATLAB.

Circulant matrices are examples of *toeplitz* matrices. Toeplitz matrices arise in many applications and are constant along their diagonals. For example,

$$T = \begin{bmatrix} c_1 & r_2 & r_3 & r_4 \\ c_2 & c_1 & r_2 & r_3 \\ c_3 & c_2 & c_1 & r_2 \\ c_4 & c_3 & c_2 & c_1 \end{bmatrix}.$$

If `c` and `r` are n-vectors, then `T = toeplitz(c,r)` sets up the matrix

$$t_{ij} = \begin{cases} c_{i-j} & i \geq j \\ \\ r_{j-i} & j > i \end{cases}.$$

5.1.3 Band Structure

Many important classes of matrices have lots of zeros. Lower triangular matrices

$$L = \begin{bmatrix} \times & 0 & 0 & 0 & 0 \\ \times & \times & 0 & 0 & 0 \\ \times & \times & \times & 0 & 0 \\ \times & \times & \times & \times & 0 \\ \times & \times & \times & \times & \times \end{bmatrix},$$

upper triangular matrices

$$U = \begin{bmatrix} \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & \times & \times \\ 0 & 0 & 0 & 0 & \times \end{bmatrix},$$

and tridiagonal matrices

$$T = \begin{bmatrix} \times & \times & 0 & 0 & 0 \\ \times & \times & \times & 0 & 0 \\ 0 & \times & \times & \times & 0 \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & \times & \times \end{bmatrix}$$

are the most important special cases. The $\times$-0 notation is a handy way to describe patterns of zeros and nonzeros in a matrix. Each "$\times$" designates a nonzero scalar.

In general, a matrix $A = (a_{ij})$ has *lower bandwidth* p if $a_{ij} = 0$ whenever $i > j + p$. Thus, an upper triangular matrix has lower bandwidth 0 and a tridiagonal matrix has lower bandwidth 1. A matrix $A = (a_{ij})$ has *upper bandwidth* q if $a_{ij} = 0$ whenever $j > i + q$. Thus, a lower triangular matrix has upper bandwidth 0 and a tridiagonal matrix has lower bandwidth 1. Here is a matrix with upper bandwidth 2 and lower bandwidth 3:

$$A = \begin{bmatrix} \times & \times & \times & 0 & 0 & 0 & 0 & 0 \\ \times & \times & \times & \times & 0 & 0 & 0 & 0 \\ \times & \times & \times & \times & \times & 0 & 0 & 0 \\ \times & \times & \times & \times & \times & \times & 0 & 0 \\ 0 & \times & \times & \times & \times & \times & \times & 0 \\ 0 & 0 & \times & \times & \times & \times & \times & \times \\ 0 & 0 & 0 & \times & \times & \times & \times & \times \\ 0 & 0 & 0 & 0 & \times & \times & \times & \times \end{bmatrix}.$$

Diagonal matrices have upper and lower bandwidth zero and can be established using the `diag` function. If `d = [10 20 30 40]` and `D = diag(d)`, then

$$D = \begin{bmatrix} 10 & 0 & 0 & 0 \\ 0 & 20 & 0 & 0 \\ 0 & 0 & 30 & 0 \\ 0 & 0 & 0 & 40 \end{bmatrix}.$$

Two-argument calls to `diag` are also possible and can be used to reference to the "other" diagonals of a matrix. An entry a_{ij} is on the kth diagonal if $j - i = k$. To clarify this, here is a matrix whose entries equal the diagonal values:

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 \\ -3 & -2 & -1 & 0 \\ -4 & -3 & -2 & -1 \end{bmatrix}.$$

If `v` is an m-vector, then `D = diag(v,k)` establishes an $(m+k)$-by-$(m+k)$ matrix that has a kth diagonal equal to v and is zero everywhere else. Thus

$$\texttt{diag([10 20 30],2)} = \begin{bmatrix} 0 & 0 & 10 & 0 & 0 \\ 0 & 0 & 0 & 20 & 0 \\ 0 & 0 & 0 & 0 & 30 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

If `A` is a matrix, then `v = diag(A,k)` extracts the kth diagonal and assigns it (as a column vector) to `v`.

The functions `tril` and `triu` can be used to "punch out" a banded portion of a given matrix. If `B = tril(A,k)`, then

$$b_{ij} = \begin{cases} a_{ij} & j \leq i + k \\ \\ 0 & j > i + k \end{cases}.$$

Analogously, if `B = triu(A,k)`, then

$$b_{ij} = \begin{cases} a_{ij} & i \leq j + k \\ \\ 0 & i > j + k \end{cases}.$$

The command

```
T = -triu(tril(ones(6,6),1),-1) + 3*eye(6,6)
```

sets up the matrix

$$T = \begin{bmatrix} 2 & -1 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 2 & -2 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & -1 & 2 \end{bmatrix}.$$

The commands

```
T =  -diag(ones(5,1),-1) + diag(2*ones(6,1),0) - diag(ones(5,1),1)
T = toeplitz([2; -1; zeros(4,1)],[2 ; -1; zeros(4,1)])
```

do the same thing.

5.1.4 Block Structure

The notation

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$$

means that A is a 2-by-3 matrix with entries a_{ij}. The a_{ij} are understood to be scalars, and MATLAB supports the synthesis of matrices at this level (i.e., `A=[a11 a12 a13; a21 a22 a23]`).

The notation can be generalized to handle the case when the specified entries are *matrices* themselves. Suppose A_{11}, A_{12}, A_{13}, A_{21}, A_{22}, and A_{23} have the following shapes:

$$A_{11} = \begin{bmatrix} u & u & u \\ u & u & u \\ u & u & u \end{bmatrix} \quad A_{12} = \begin{bmatrix} v \\ v \\ v \end{bmatrix} \quad A_{13} = \begin{bmatrix} w & w \\ w & w \\ w & w \end{bmatrix}$$

$$A_{21} = \begin{bmatrix} x & x & x \\ x & x & x \end{bmatrix} \quad A_{22} = \begin{bmatrix} y \\ y \end{bmatrix} \quad A_{23} = \begin{bmatrix} z & z \\ z & z \end{bmatrix}.$$

We then define the 2-by-3 *block matrix*

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{bmatrix}$$

by

$$A = \left[\begin{array}{ccc|c|cc} u & u & u & v & w & w \\ u & u & u & v & w & w \\ u & u & u & v & w & w \\ \hline x & x & x & y & z & z \\ x & x & x & y & z & z \end{array}\right].$$

The lines delineate the block entries. Of course, A is also a 5-by-6 scalar matrix.

Block matrix manipulations are very important and can be effectively carried out in MATLAB. The script

```
A11 = [ 10 11 12 ; 13 14 15 ; 16 17 18 ];
A12 = [ 20 ; 21 ; 22 ];
A13 = [ 30 31 ; 32 33 ; 34 35 ];
A21 = [ 40 41 42 ; 43 44 45 ];
A22 = [ 50 ; 51 ];
A23 = [ 60 61 ; 62 63 ];
A = [ A11 A12 A13 ; A21 A22 A23 ];
```

results in the formation of

$$A = \begin{bmatrix} 10 & 11 & 12 & 20 & 30 & 31 \\ 13 & 14 & 15 & 21 & 32 & 33 \\ 16 & 17 & 18 & 22 & 34 & 35 \\ 40 & 41 & 42 & 50 & 60 & 61 \\ 43 & 44 & 45 & 51 & 62 & 63 \end{bmatrix}.$$

The block rows of a matrix are separated by semicolons, and it is important to make sure that the dimensions are consistent. The final result must be rectangular at the scalar level. For example,

```
A = [1 zeros(1,6); ...
    zeros(2,1) [10 20;30 40] zeros(2,2); ...
    zeros(2,3) [50 60;70 80]]
```

sets up the matrix

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 10 & 20 & 0 & 0 \\ 0 & 30 & 40 & 0 & 0 \\ 0 & 0 & 0 & 50 & 60 \\ 0 & 0 & 0 & 70 & 80 \end{bmatrix}.$$

The extraction of blocks requires the colon notation. The assignment `C = A(2:4,5:6)` is equivalent to any of the following:

```
C = [A(2:4,5) A(2:4,6)]
C = [A(2,5:6) ; A(3,5:6) ; A(4,5:6)]
C = [A(2,5) A(2,6) ; A(3,5) A(3,6) ; A(4,5) A(4,6)]
```

A block matrix can be conveniently represented as a cell array with matrix entries. Here is a function that does this when the underlying matrix can be expressed as a square block matrix with square blocks:

```
  function A = MakeBlock(A_scalar,p)
% A = MakeBlock(A_scalar,p)
% A_scalar is an n-by-n matrix and p divides n. A is an (n/p)-by-(n/p)
% cell array that represents A_scalar as a block matrix with p-by-p blocks.
[n,n] = size(A_scalar);
m = n/p;
A = cell(m,m);
for i=1:m
   for j=1:m
      A{i,j} = A_scalar(1+(i-1)*p:i*p,1+(j-1)*p:j*p);
   end
end
```

Problems

P5.1.1 For `n=[5 10 20 40]` and for each of the two methods mentioned in §5.1.1, compute the number of flops required to set up the matrix P on pages 169-170.

P5.1.2 Give a MATLAB one-liner using `Toeplitz` that sets up a circulant matrix with first row equal to a given vector `a`.

P5.1.3 Any Toeplitz matrix can be "embedded" in a larger circulant matrix. For example,

$$T = \begin{bmatrix} c & d & e \\ b & c & d \\ a & b & c \end{bmatrix}$$

is the leading 3-by-3 portion of

$$C = \begin{bmatrix} c & d & e & a & b \\ b & c & d & e & a \\ a & b & c & d & e \\ e & a & b & c & d \\ d & e & a & b & c \end{bmatrix}.$$

Write a MATLAB function `C = EmbedToep(col,row)` that sets up a circulant matrix with the property that `C(1:n,1:n) = Toeplitz(col,row)`.

P5.1.4 Write a MATLAB function `A = RandBand(m,n,p,q)` that returns a random m-by-n matrix that has lower bandwidth p and upper bandwidth q.

P5.1.5 Let n be a positive integer. Extrachromosomal DNA elements called *plasmids* are found in many types of bacteria. Assume that in a particular species there is a plasmid P and that exactly n copies of it appear in every cell. Sometimes the plasmid appears in two slightly different forms. These may differ at just a few points in their DNA. For example, one type might have a gene that codes for resistance of the cell to the antibiotic ampicillin, and the other could have a gene that codes for resistance to tetracycline. Let's call the two variations of the plasmid A and B.

Assume that the cells in the population reproduce in unison. Here is what happens at that time. The cell first replicates its DNA matter. Thus, if a cell has one type A plasmid and three type B plasmids, then it now has two type A plasmids and six type B plasmids. After replication, the cell divides. The two daughter cells will each receive four of the eight plasmids. There are several possibilities and to describe them we adopt a handy notation. We say that a cell is (i_A, i_B) if it has i_A type A plasmids and i_B type B plasmids. So if the parent is $(1,3)$, then its daughter will be either $(0,4)$, $(1,3)$, or $(2,2)$.

The probability that a daughter cell is (i'_A, i'_B) given its parent is (i_A, i_B) is specified by

$$P_{i'_A,i_A} = \frac{\binom{2i_A}{i'_A}\binom{2i_B}{i'_B}}{\binom{2n}{n}}.$$

In the numerator you see the number of ways we can partition the parent's replicated DNA so that the daughter is (i'_A, i'_B). The denominator is the total number of ways we can select n plasmids from the replicated set of $2n$ plasmids.

Let $P(n)$ be the $(n+1)$-by-$(n+1)$ matrix whose (i'_A, i_A) entry is given by $P_{i'_A,i_A}$. (Note subscripting from zero.) If $n = 4$, then

$$P = \frac{1}{\binom{8}{4}} \begin{bmatrix} \binom{0}{0}\binom{8}{4} & \binom{2}{0}\binom{6}{4} & \binom{4}{0}\binom{4}{4} & \binom{6}{0}\binom{2}{4} & \binom{8}{0}\binom{0}{4} \\ \binom{0}{1}\binom{8}{3} & \binom{2}{1}\binom{6}{3} & \binom{4}{1}\binom{4}{3} & \binom{6}{1}\binom{2}{3} & \binom{8}{1}\binom{0}{3} \\ \binom{0}{2}\binom{8}{2} & \binom{2}{2}\binom{6}{2} & \binom{4}{2}\binom{4}{2} & \binom{6}{2}\binom{2}{2} & \binom{8}{2}\binom{0}{2} \\ \binom{0}{3}\binom{8}{1} & \binom{2}{3}\binom{6}{1} & \binom{4}{3}\binom{4}{1} & \binom{6}{3}\binom{2}{1} & \binom{8}{3}\binom{0}{1} \\ \binom{0}{4}\binom{8}{0} & \binom{2}{4}\binom{6}{0} & \binom{4}{4}\binom{4}{0} & \binom{6}{4}\binom{2}{0} & \binom{8}{4}\binom{0}{0} \end{bmatrix}.$$

Call this matrix the plasmid transition matrix. Write a MATLAB function `SetUp(n)` that computes the $(n+1)$-by-$(n+1)$ plasmid transition matrix $P(n)$. (You'll have to adapt the preceding discussion to conform to MATLAB's subscripting from one requirement.) Exploit structure. If successful, you should find that the number of flops required is quadratic in n. Print a table that indicates the number of flops required to construct $P(n)$ for n = 5,6,10,11,20,21,40,41. (See F.C. Hoppenstadt and C.Peskin (1992), *Mathematics in Medicine and the Life Sciences*, Springer-Verlag, New York, p.50.)

P5.1.6 Generalize the function `MakeBlock` to `A = MakeBlock(A_scalar,m,n)`, where `m` and `n` are vectors of integers that sum to `mA` and `nA` respectively, and `[mA,nA] = size(A)`. `A` should be a `length(mA)`-by-`length(nA)` cell array, where `A{i,j}` is a matrix of size `mA(i)`-by-`nA(j)`.

5.2 Matrix Operations

Once a matrix is set up, it can participate in matrix-vector and matrix-matrix products. Although these operations are MATLAB one-liners, it is instructive to examine the different ways that they can be implemented.

5.2.1 Matrix-Vector Multiplication

Suppose $A \in \mathbb{R}^{m\times n}$, and we wish to compute the matrix-vector product $y = Ax$, where $x \in \mathbb{R}^n$. The usual way this computation proceeds is to compute the dot products

$$y_i = \sum_{j=1}^{n} a_{ij}x_j$$

one at a time for $i = 1{:}m$. This leads to the following algorithm:

```
[m,n] = size(A);
y = zeros(m,1);
for i = 1:m
   for j = 1:n
      y(i) = y(i) + A(i,j)*x(j);
   end
end
```

The one-line assignment `y = A*x` is equivalent and requires $2mn$ flops.

Even though it is not necessary to hand-code matrix-vector multiplication in MATLAB, it is instructive to reconsider the preceding double loop. In particular, recognizing that the j-loop oversees an inner product of the ith row of A and the x vector, we have

```
  function y = MatVecRO(A,x)
% y = MatVecRO(A,x)
% Computes the matrix-vector product y = A*x (via saxpys) where
% A is an m-by-n matrix and x is a columnn-vector.
[m,n] = size(A);
y = zeros(m,1);
for i=1:m
   y(i) = A(i,:)*x;
end
```

The colon notation has the effect of highlighting the dot products that make up Az. The procedure is *row oriented* because A is accessed by row.

A *column-oriented* algorithm for matrix-vector products can also be developed. We start with a 3-by-2 observation:

$$y = Ax = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \begin{bmatrix} 7 \\ 8 \end{bmatrix} = \begin{bmatrix} 1\cdot 7 + 2\cdot 8 \\ 3\cdot 7 + 4\cdot 8 \\ 5\cdot 7 + 6\cdot 8 \end{bmatrix} = 7\begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix} + 8\begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix} = \begin{bmatrix} 23 \\ 53 \\ 83 \end{bmatrix}.$$

In other words, y is a linear combination of A's columns with the x_j being the coefficients. This leads us to the following reorganization of `MatVecRO`:

```
  function y = MatVecCO(A,x)
% y = MatVecCO(A,x)
% This computes the matrix-vector product y = A*x (via saxpys) where
% A is an m-by-n matrix and x is a columnn-vector.
[m,n] = size(A);
y = zeros(m,1);
for j=1:n
   y = y + A(:,j)*x(j);
end
```

In terms of program transformation, this function is just `MatVecRO` with the i and j loops swapped. The inner loop now oversees an operation of the form

$$\text{vector} \leftarrow \text{scalar}\cdot\text{vector} \ + \ \text{vector}.$$

This is known as the *saxpy* operation. Along with the dot product, it is a key player in matrix computations. Here is an expanded view of the saxpy operation in `MatVecCO`:

$$\begin{bmatrix} \texttt{y(1)} \\ \texttt{y(2)} \\ \vdots \\ \texttt{y(m)} \end{bmatrix} = \begin{bmatrix} \texttt{A(1,j)} \\ \texttt{A(2,j)} \\ \vdots \\ \texttt{A(m,j)} \end{bmatrix} \texttt{x(j)} \ + \ \begin{bmatrix} \texttt{y(1)} \\ \texttt{y(2)} \\ \vdots \\ \texttt{y(m)} \end{bmatrix}.$$

`MatVecCO` requires $2mn$ flops just like `MatVecRO`. However, to stress once again the limitations of flop counting, we point out that in certain powerful computing environments our two matrix-vector product algorithms may execute at radically different rates. For example, if the matrix entries a_{ij} are stored column by column in memory, then the saxpy version accesses A-entries that are contiguous in memory. In contrast, the row-oriented algorithm accesses non-contiguous a_{ij}. As a result of that inconvenience, it may require much more time to execute.

5.2.2 Exploiting Structure

In many matrix computations the matrices are structured with lots of zeros. In such a context it may be possible to streamline the computations. As a first example of this, we examine the matrix-vector product problem $y = Az$, where $A \in \mathbb{R}^{n\times n}$ is upper triangular. The product looks like this in the $n = 4$ case:

$$\begin{bmatrix} \times \\ \times \\ \times \\ \times \end{bmatrix} = \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix} \begin{bmatrix} \times \\ \times \\ \times \\ \times \end{bmatrix}.$$

The derivation starts by looking at `MatVecRO`. Observe that the inner products in the loop

```
for i = 1:n
   y(i) = A(i,:)*x
end
```

involve long runs of zeros when `A` is upper triangular. For example, if $n = 7$, then the inner product `A(5,:)*x` looks like

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & \times & \times \end{bmatrix} \begin{bmatrix} \times \\ \times \\ \times \\ \times \\ \times \\ \times \\ \times \end{bmatrix}$$

and requires a reduced number of flops because of all the zeros. Thus, we must "shorten" the inner products so that they only include the nonzero portion of the row.

From the observation that the first i entries in `A(i,:)` are zero, we see that `A(i,i:n)*x(i:n)` is the nonzero portion of the full inner product `A(i,:)*x` that we need. It follows that

```
[n,n] = size(A);
y = zeros(n,1);
for i = 1:n
   y(i) = A(i,i:n)*z(i:n)
end
```

is a structure-exploiting upper triangular version of `MatVecRO`. The assignment to `y(i)` requires $2i$ flops, and so overall

$$\sum_{i=1}^{n} (2i) = 2(1 + 2 + \cdots + n) = n(n+1)$$

flops are required. However, in keeping with the philosophy of flop counting, we do not care about the $O(n)$ term and so we merely state that the algorithm requires n^2 flops. Our streamlining halved the number of floating point operations.

`MatVecCO` can also be abbreviated. Note that `A(:,j)` is zero in components $j+1$ through n, and so the "essential" saxpy to perform in the jth step is

$$\begin{bmatrix} \mathtt{y(1)} \\ \mathtt{y(2)} \\ \vdots \\ \mathtt{y(j)} \end{bmatrix} = \begin{bmatrix} \mathtt{A(1,j)} \\ \mathtt{A(2,j)} \\ \vdots \\ \mathtt{A(j,j)} \end{bmatrix} \mathtt{x(j)} + \begin{bmatrix} \mathtt{y(1)} \\ \mathtt{y(2)} \\ \vdots \\ \mathtt{y(j)} \end{bmatrix},$$

rendering

```
[n,n] = size(A);
y = zeros(n,1);
for j = 1:n
   y(1:j) = A(1:j,j)*x(j) + y(1:j);
end
```

Again, the number of required flops is halved.

5.2.3 Matrix-Matrix Multiplication

If $A \in \mathbb{R}^{m\times p}$ and $B \in \mathbb{R}^{p\times n}$, then the product $C = AB$ is defined by

$$c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}$$

for all i and j that satisfy $1 \le i \le m$ and $1 \le j \le n$. In other words, each entry in C is the inner product of a row in A and a column in B. Thus, the fragment

```
C = zeros(m,n);
for j=1:n
   for i=1:m
      for k=1:p
         C(i,j) = C(i,j) + A(i,k)*B(k,j);
      end
   end
end
```

computes the product AB and assigns the result to `C`. MATLAB supports matrix-matrix multiplication, and so this can be implemented with the one-liner

```
C = A*B
```

However, there are a number of different ways to look at matrix multiplication, and we shall present four distinct versions.

We start with the recognition that the innermost loop in the preceding script oversees the dot product between row i of A and column j of B:

```
  function C = MatMatDot(A,B)
% C = MatMatDot(A,B)
% This computes the matrix-matrix product C =A*B (via dot products) where
% A is an m-by-p matrix, B is a p-by-n matrix.
[m,p] = size(A);
[p,n] = size(B);
C = zeros(m,n);
for j=1:n
   % Compute j-th column of C.
   for i=1:m
      C(i,j) = A(i,:)*B(:,j);
   end
end
```

On other hand, we know that the jth column of C equals A times the jth column of B. If we apply `MatVecCO` to each of these matrix vector products, we obtain

```
  function C = MatMatSax(A,B)
% C = MatMatSax(A,B)
% This computes the matrix-matrix product C = A*B (via saxpys) where
% A is an m-by-p matrix, B is a p-by-n matrix.
[m,p] = size(A);
[p,n] = size(B);
C = zeros(m,n);
for j=1:n
   % Compute j-th column of C.
   for k=1:p
      C(:,j) = C(:,j) + A(:,k)*B(k,j);
   end
end
```

This version of matrix multiplication highlights the saxpy operation. By replacing the inner loop in this with a single matrix-vector product we obtain

```
  function C = MatMatVec(A,B)
% C = MatMatVec(A,B)
% This computes the matrix-matrix product C = A*B (via matrix-vector products)
% where A is an m-by-p matrix, B is a p-by-n matrix.
[m,p] = size(A);
[p,n] = size(B);
C = zeros(m,n);
for j=1:n
   % Compute j-th column of C.
   C(:,j) = C(:,j) + A*B(:,j);
end
```

Finally, we observe that a matrix multiplication is a sum of outer products. The *outer product* between a column m-vector u and a row n-vector v is given by

$$uv^T = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_m \end{bmatrix} \begin{bmatrix} v_1 & v_2 & \cdots & v_n \end{bmatrix} = \begin{bmatrix} u_1v_1 & u_1v_2 & \cdots & u_1v_n \\ u_2v_1 & u_2v_2 & \cdots & u_2v_n \\ \vdots & \vdots & \ddots & \vdots \\ u_mv_1 & u_mv_2 & \cdots & u_mv_n \end{bmatrix}.$$

Appreciate this as just the ordinary matrix multiplication of an m-by-1 matrix and a 1-by-n matrix:

$$\begin{bmatrix} 10 \\ 15 \\ 20 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix} = \begin{bmatrix} 10 & 20 & 30 & 40 \\ 15 & 30 & 45 & 60 \\ 20 & 40 & 60 & 80 \end{bmatrix}.$$

Returning to the matrix multiplication problem,

$$C = AB = \begin{bmatrix} A(:,1) \mid A(:,2) \mid \cdots \mid A{:},p) \end{bmatrix} \begin{bmatrix} B(1,:) \\ B(2,:) \\ \vdots \\ B(p,:) \end{bmatrix} = \sum_{k=1}^{p} A(:,k)B(k,:).$$

Thus,

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \begin{bmatrix} 10 & 20 \\ 30 & 40 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix} \begin{bmatrix} 10 & 20 \end{bmatrix} + \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix} \begin{bmatrix} 30 & 40 \end{bmatrix}$$
$$= \begin{bmatrix} 10 & 20 \\ 30 & 60 \\ 50 & 100 \end{bmatrix} + \begin{bmatrix} 60 & 80 \\ 120 & 160 \\ 180 & 240 \end{bmatrix}$$
$$= \begin{bmatrix} 70 & 100 \\ 150 & 220 \\ 230 & 340 \end{bmatrix}.$$

This leads to the outer product version of matrix multiplication:

```
  function C = MatMatOuter(A,B)
% C = MatMatOuter(A,B)
% This computes the matrix-matrix product C = A*B (via outer products) where
% A is an m-by-p matrix, B is a p-by-n matrix.
[m,p] = size(A);
[p,n] = size(B);
C = zeros(m,n);
for k=1:p
  % Add in k-th outer product
   C = C + A(:,k)*B(k,:);
end
```

The script file `MatBench` benchmarks the four various matrix-multiply functions that we have developed along with the direct, one-liner `C = A*B`.

```
 n      Dot      Saxpy    MatVec   Outer    Direct
-------------------------------------------------
100   0.8850   1.2900   0.0720   0.5930   0.0220
200   4.1630   5.5750   0.3020   5.3610   0.3080
400  22.4810  33.5050   8.4800  49.8840   3.0920
```

The most important thing about the table is not the actual values reported but that it shows the weakness of flop counting. Methods for the same problem that involve the same number of flops can perform very differently. The nature of the kernel operation (saxpy, dot product, matrix-vector product, outer product, etc.) is more important than the amount of arithmetic involved.

5.2.4 Sparse Matrices

For many matrices that arise in practice, the ratio

$$\frac{\text{Number of Nonzero Entries}}{\text{Number of Zero Entries}}$$

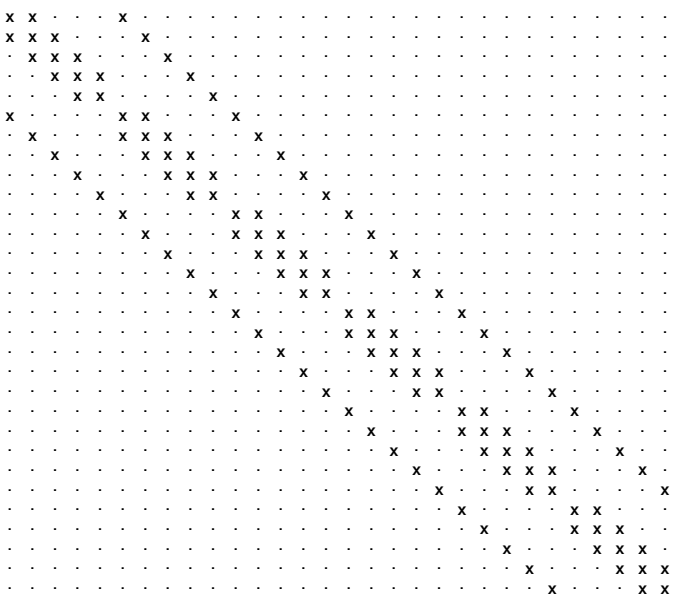


FIGURE 5.1 A Sparse matrix

is often very small. Matrices with this property are said to be *sparse*. An important class of sparse matrices are band matrices, such as the "block" tridiagonal matrix displayed in Figure 5.1. (See §5.1.3 and §5.1.4.) If A is sparse then (a) it can be represented with reduced storage and (b) matrix-vector products that involve A can be carried out with a reduced number of flops. For example, if A is an n-by-n tridiagonal matrix then it can be represented with three n-vectors and when it multiplies a vector only $5n$ flops are involved. However, this would not be the case if A is represented as a full matrix. Thus,

```
A = diag(2*ones(n,1)) - diag(ones(n-1,1),-1) - diag(ones(n-1,1),1);
y = A*rand(n,1);
```

involves $O(n^2)$ storage and $O(n^2)$ flops.

The `sparse` function addresses these issues in MATLAB . If `A` is a matrix then `S_A = sparse(A)` produces a sparse array representation of `A`. The sparse array `S_A` can be engaged in the same matrix operations as `A` and MATLAB will exploit the underlying sparse structure whenever possible. Consider the script

```
A = diag(2*ones(n,1)) - diag(ones(n-1,1),-1) - diag(ones(n-1,1),1);
S_A = sparse(A);
y = S_A*rand(n,1);
```

The representation `S_A` involves $O(n)$ storage and the product `y = S_A*rand(n,1)` $O(n)$ flops. The script `ShowSparse` looks at the flop efficiency in more detail and produces the plot shown in Figure 5.2 (on the next page). There are more sophisticated ways to use `sparse` which the interested reader can pursue via `help`.

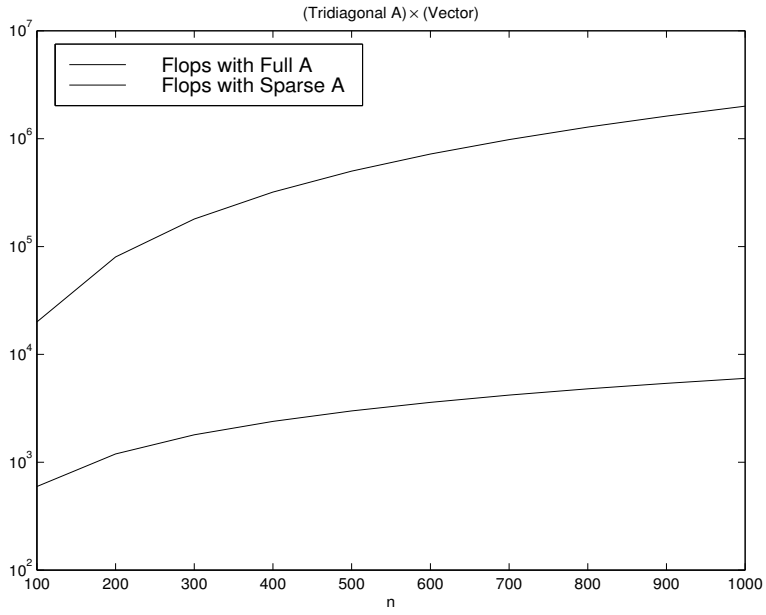


FIGURE 5.2 Exploiting sparsity

5.2.5 Error and Norms

We conclude this section with a brief look at how errors are quantified in the matrix computation area. The concept of a *norm* is required. Norms are a vehicle for measuring distance in a vector space. For vectors $x \in \mathbb{R}^n$, the 1, 2, and infinity norms are of particular importance:

$$\begin{aligned} \| x \|_1 &= |x_1| + \cdots + |x_n| \\ \| x \|_2 &= \sqrt{x_1^2 + \cdots + x_n^2} \\ \| x \|_\infty &= \max\{|x_1|, \ldots, |x_n|\} \end{aligned}$$

A norm is just a generalization of absolute value. Whenever we think about vectors of errors in an order-of-magnitude sense, then the choice of norm is generally not important. It is possible to show that

$$\begin{aligned} \| x \|_\infty &\leq \| x \|_1 \leq n \| x \|_\infty \\ \| x \|_\infty &\leq \| x \|_2 \leq \sqrt{n} \| x \|_\infty . \end{aligned}$$

Thus, the 1-norm cannot be particularly small without the others following suit.

In MATLAB, if `x` is a vector, `norm(x,1)`, `norm(x,2)`, and `norm(x,inf)` can be used to ascertain these quantities. A single-argument call to `norm` returns the 2-norm (e.g., `norm(x)`). The script

`AveNorms` tabulates the ratios $\| x \|_1 / \| x \|_\infty$ and $\| x \|_2 / \| x \|_\infty$ for large collections of random n-vectors.

The idea of a norm extends to matrices and, as in the vector case, there are number of important special cases. If $A \in \mathbb{R}^{m \times n}$, then

$$\| A \|_1 = \max_{1 \le j \le n} \sum_{i=1}^{m} |a_{ij}| \qquad \| A \|_2 = \max_{\| x \|_2 = 1} \| Ax \|_2$$

$$\| A \|_\infty = \max_{1 \le i \le m} \sum_{j=1}^{n} |a_{ij}| \qquad \| A \|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2}$$

In MATLAB if `A` is a matrix, then `norm(A,1)`, `norm(A,2)`, `norm(A,inf)`, and `norm(A,'fro')` can be used to compute these values. As a simple illustration of how matrix norms can be used to quantify error at the matrix level, we prove a result about the roundoff errors that arise when an m-by-n matrix is stored.

Theorem 5 *If $\hat{A}$ is the stored version of $A \in \mathbb{R}^{m \times n}$, then $\hat{A} = A + E$ where $E \in \mathbb{R}^{m \times n}$ and*

$$\| E \|_1 \le \texttt{eps} \ \| A \|_1.$$

Proof From Theorem 1, if $\hat{A} = (\hat{a}_{ij})$, then

$$\hat{a}_{ij} = fl(a_{ij}) = a_{ij}(1 + \epsilon_{ij}),$$

where $|\epsilon_{ij}| \le$ `eps` . Thus,

$$\begin{aligned} \| E \|_1 &= \| \hat{A} - A \|_1 = \max_{1 \le j \le n} \sum_{i=1}^{m} |\hat{a}_{ij} - a_{ij}| \\ &\le \max_{1 \le j \le n} \sum_{i=1}^{m} |a_{ij} \epsilon_{ij}| \le \texttt{eps} \max_{1 \le j \le n} \sum_{i=1}^{m} |a_{ij}| = \texttt{eps} \ \| A \|_1. \quad \square \end{aligned}$$

This says that errors of order `eps`$\| A \|_1$ arise when a real matrix A is stored in floating point. There is nothing special about our choice of the 1-norm. Similar results apply for the other norms defined earlier.

When the effect of roundoff error is the issue, we will be content with order-of-magnitude approximation. For example, it can be shown that if A and B are matrices of floating point numbers, then

$$\| \text{fl}(AB) - AB \| \approx \texttt{eps} \| A \| \| B \|.$$

By fl(AB) we mean the computed, floating point product of A and B. The result says that the errors in the computed product are roughly the product of the unit roundoff `eps`, the size of the numbers in A, and the size of the numbers in B. The following script confirms this result:

```
% Script File: ProdBound
% Examines the error in 3-digit matrix multiplication.

  clc
  eps3 = .005;      % 3-digit machine precision
  nRepeat = 10;      % Number of trials per n-value
  disp('   n   1-norm factor ')
  disp('-------------------------')
  for n = 2:10
     s = 0;
     for r=1:nRepeat
        A = randn(n,n);
        B = randn(n,n);
        C = Prod3Digit(A,B);
        E = C - A*B;
        s = s+ norm(E,1)/(eps3*norm(A,1)*norm(B,1));
     end
     disp(sprintf('%4.0f     %8.3f    ',n,s/nRepeat))
  end
```

The function `Prod3Digit(A,B)` returns the product AB computed using simulated three-digit floating point arithmetic developed in §1.6.1. The result is compared to the "exact" product obtained by using the prevailing, full machine precision. Here are some sample results:

```
   n    1-norm factor
--------------------
   2        0.323
   3        0.336
   4        0.318
   5        0.222
   6        0.231
   7        0.227
   8        0.218
   9        0.207
  10        0.218
```

Problems

P5.2.1 Suppose $A \in \mathbb{R}^{m\times n}$ has the property that a_{ij} is zero whenever $i > j+1$. Write an efficient, row-oriented dot product algorithm that computes $y = Az$.

P5.2.2 Suppose $A \in \mathbb{R}^{m\times n}$ is upper triangular and that $x \in \mathbb{R}^n$. Write a MATLAB fragment for computing the product $y = Ax$. (Do not make assumptions like $m \geq n$ or $m \leq n$.)

P5.2.3 Modify `MatMatSax` so that it efficiently handles the case when `B` is upper triangular.

P5.2.4 Modify `MatMatSax` so that it efficiently handles the case when both `A` and `B` are upper triangular and n-by-n.

P5.2.5 Modify `MatMatDot` so that it efficiently handles the case when `A` is lower triangular and `B` are upper triangular and both are n-by-n.

P5.2.6 Modify `MatMatSax` so that it efficiently handles the case when B has upper and lower bandwidth p. Assume that both A and B are n-by-n.

P5.2.7 Write a function `B = MatPower(A,k)` so that computes $B = A^k$, where A is a square matrix and k is a nonnegative integer. Hint: First consider the case when k is a power of 2. Then consider binary expansions (e.g., $A^{29} = A^{16}A^8A^4A$).

P5.2.8 Develop a nested multiplication for the product $y = (c_1I + c_2A + \cdots + c_kA^{k-1})v$, where the c_i are scalars, A is a square matrix, and v is a vector.

P5.2.9 Write a MATLAB function that returns the matrix

$$C = \left[\begin{array}{c|c|c|c|c} B & AB & A^2B & \cdots & A^{p-1}B \end{array} \right],$$

where A is n-by-n, B is n-by-t, and p is a positive integer.

P5.2.10 Write a MATLAB function `ScaleRows(A,d)` that assumes A is m-by-n and d is m-by-1 and multiplies the ith row of A by $d(i)$.

P5.2.11 Let $P(n)$ be the matrix of P5.1.2. If v is a plasmid state vector, then after one reproductive cycle, $P(n)v$ is the state vector for the population of daughters. A vector $v(0{:}n)$ is *symmetric* if $v(n{:}-1{:}0) = v$. Thus [2;5;6;5;2] and [3;1;1;3] are symmetric.) Write a MATLAB function `V = Forward(P,v0,k)` that sets up a matrix V with k rows. The kth row of V should be the transpose of the vector P^kv_0. Assume v_0 is symmetric.

P5.2.12 A matrix M is tridiagonal if $m_{ij} = 0$ whenever $|i - j| > 1$. Write a MATLAB function `C = Prod(A,B)` that computes the product of an n-by-n upper triangular matrix A and an n-by-n tridiagonal matrix B. Your solution should be efficient (no superfluous floating point arithmetic) and vectorized.

P5.2.13 Make the following function efficient from the flop point of view and vectorize.

```
  function C = Cross(A)
% A is n-by-n and with A(i,j) = 0 whenever i>j+1
% C = A^{T}*A
[n,n] = size(A);
C = zeros(n,n);
for i=1:n
   for j=1:n
      for k=1:n
         C(i,j) = C(i,j) + A(k,i)*A(k,j);
      end
   end
end
```

P5.2.14 Assume that A, B, and C are matrices and that the product ABC is defined. Write a flop-efficient MATLAB function `D = ProdThree(A,B,C)` that returns their product.

P5.2.15 (Continuation of P5.1.6.) Write a matrix-vector product function `y = BlockMatVec(A,x)` that computes $y = Ax$ but where `A` is a cell array that represents the underlying matrix as a block matrix.

P5.2.16 Write a function `v = SparseRep(A)` that takes an n-by-n matrix A and returns a length n array `v`, with the property that `v(i).jVals` is a row vector of indices that name the nonzero entries in `A(i,:)` and `v(i).aVals` is the row vector of nonzero entries from `A(i,:)`. Write a matrix-vector product function that works with this representation. (Review the function `find` for this problem.)

5.3 Once Again, Setting Up Matrix Problems

On numerous occasions we have been required to evaluate a continuous function $f(x)$ on a vector of values (e.g., `sqrt(linspace(0,9))`). The analog of this in two dimensions is the evaluation of a function $f(x, y)$ on a pair of vectors x and y.

5.3.1 Two-Dimensional Tables of Function Values

Suppose $f(x, y) = \exp^{-(x^2+3y^2)}$ and that we want to set up an n-by-n matrix F with the property that

$$f_{ij} = e^{-(x_i^2+3y_j^2)},$$

where $x_i = (i-1)/(n-1)$ and $y_j = (j-1)/(n-1)$. We can proceed at the scalar, vector, or matrix level. At the scalar level we evaluate `exp` at each entry:

```
v = linspace(0,1,n);
F = zeros(n,n);
for i=1:n
   for j=1:n
      F(i,j) = exp(-(v(i)^2  + 3*v(j)^2));
   end
end
```

At the vector level we can set `F` up by column:

```
v = linspace(0,1,n)';
F = zeros(n,n);
for j=1:n
   F(:,j) = exp(-( v.^2 + 3*v(j)^2));
end
```

Finally, we can even evaluate `exp` on the matrix of arguments:

```
v = linspace(0,1,n);
A = zeros(n,n);
for i=1:n
   for j=1:n
      A(i,j) = -(v(i)^2  + 3*v(j)^2);
   end
end
F = exp(A);
```

Many of MATLAB's built-in functions, like `exp`, accept matrix arguments. The assignment `F = exp(A)` sets `F` to be a matrix that is the same size as `A` with $f_{ij} = e^{a_{ij}}$ for all i and j.

In general, the most efficient approach depends on the structure of the matrix of arguments, the nature of the underlying function $f(x, y)$, and what is already available through M-files. Regardless of these details, it is best to be consistent with MATLAB's vectorizing philosophy designing all functions so that they can accept vector arguments. For example,

```
  function F = SampleF(x,y)
% x is a column n-vector, y is a column m-vector and
% F  is an m-by-n matrix with F(i,j) = exp(-(x(j)^2 + 3y(i)^2)).
n = length(x);
m = length(y);
A = -((2*y.^2)*ones(1,n) + ones(m,1)*(x.^2)')/4;
F = exp(A);
```

Notice that the matrix A is the sum of two outer products and that $a_{ij} = -(x_j^2 + 2y_i^2)/4$. The setting up of this grid of points allows for a single (matrix-valued) call to `exp`.

5.3.2 Contour Plots

While the discussion of tables is still fresh, we introduce the MATLAB's contour plotting capability. If $f(x, y)$ is a function of two real variables, then a curve in the xy-plane of the form $f(x, y) = c$ is a *contour*. The function `contour` can be used to display such curves. Here is a script that displays various contour plots of the function `SampleF`:

```
% Script File: ShowContour
% Illustrates various contour plots.

  close all
% Set up array of function values.
  x = linspace(-2,2,50)';
  y = linspace(-1.5,1.5,50)';
  F = SampleF(x,y);

% Number of contours set to default value:
  figure
  Contour(x,y,F)
  axis equal

% Five contours:
  figure
  contour(x,y,F,5);
  axis equal

% Five contours with specified values:
  figure
  contour(x,y,F,[1 .8  .6  .4  .2])
  axis equal

% Four contours with manual labeling:
  figure
  c = contour(x,y,F,4);
  clabel(c,'manual');
  axis equal
```

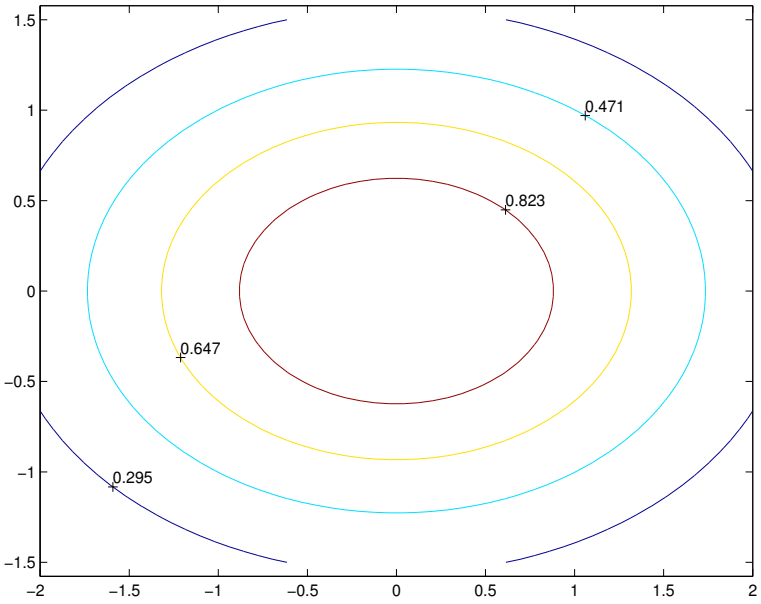


FIGURE 5.3 A `contour` plot

`Contour(x,y,F)` assumes that `F(i,j)` is the value of the underlying function f at (x_j, y_i). Clearly, the length of `x` and the length of `y` must equal the column and row dimension of `F`. The argument after the array is used to supply information about the number of contours and the associated "elevations." `Contour(x,y,F,N)` specifies `N` contours. `Contour(x,y,F,v)`, where `v` is a vector, specifies elevations `v(i)`, where `i=1:length(v)`. The contour elevations can be labeled using the mouse by the command sequence of the form

```
c = contour(x,y,F,...);
clabel(c,'manual');
```

Type `help clabel` for more details. A sample labeled contour plot of the function `SampleF` is shown in Figure 5.3. See the script `ShowContour`.

5.3.3 Spotting Matrix-Vector Products

Let us consider the problem of approximating the double integral

$$I = \int_a^b \int_c^d f(x,y)dy\,dx$$

using a quadrature rule of the form

$$\int_a^b g(x)dx \approx (b-a)\sum_{i=1}^{N_x} \omega_i g(x_i) \;\equiv\; Q_x$$

in the x-direction and a quadrature rule of the form

$$\int_c^d g(y)dy \approx (d-c)\sum_{j=1}^{N_y} \mu_j g(y_j) \;\equiv\; Q_y$$

in the y-direction. Doing this, we obtain

$$\begin{aligned} I &= \int_a^b \left(\int_c^d f(x,y)dy \right) dx \;\approx\; (b-a)\sum_{i=1}^{N_x} \omega_i \left(\int_c^d f(x_i,y)dy \right) \\ &\approx (b-a)\sum_{i=1}^{N_x} \omega_i \left((d-c)\sum_{j=1}^{N_y} \mu_j f(x_i,y_j) \right) \\ &= (b-a)(d-c)\sum_{i=1}^{N_x} \omega_i \left(\sum_{j=1}^{N_y} \mu_j f(x_i,y_j) \right) \;\equiv\; Q. \end{aligned}$$

Observe that the quantity in parentheses is the ith component of the vector $F\mu$, where

$$F = \begin{bmatrix} f(x_1,y_1) & \cdots & f(x_1,y_{N_y}) \\ \vdots & \ddots & \vdots \\ f(x_{N_x},y_1) & \cdots & f(x_{N_x},y_{N_y}) \end{bmatrix} \text{ and } \mu = \begin{bmatrix} \mu_1 \\ \vdots \\ \mu_{N_y} \end{bmatrix}.$$

It follows that

$$Q = (b-a)(d-c)\omega^T(F\mu),$$

where

$$\omega = \begin{bmatrix} \omega_1 \\ \vdots \\ \omega_{N_x} \end{bmatrix}.$$

If Q_x and Q_y are taken to be composite Newton-Cotes rules, then we obtain the following implementation:

```
  function numI2D = CompQNC2D(fname,a,b,c,d,mx,nx,my,ny)
% numI2D = CompQNC2D(fname,a,b,c,d,mx,nx,my,ny)
%
% fname is a string that names a function of the form f(x,y).
% If x and y are vectors, then it should return a matrix F with the
% property that F(i,j) = f(x(i),y(j)), i=1:length(x), j=1:length(y).
%
% a,b,c,d are real scalars.
% mx and my are integers that satisfy 2<=mx<=11, 2<=my<=11.
% nx and ny are positive integers
%
% numI2D  approximation to the integral of f(x,y) over the rectangle [a,b]x[c,d].
% The compQNC(mx,nx) rule is used in the x-direction and the compQNC(my,ny)
```

```
% rule is used in the y-direction.

[omega,x] = CompNCweights(a,b,mx,nx);
[mu,y]    = CompNCweights(c,d,my,ny);
F = feval(fname,x,y);
numI2D = (b-a)*(d-c)*(omega'*F*mu);
```

The function `CompNCweights` uses `NCweights` from Chapter 4 and sets up the vector of weights and the vector of abscissas for the composite m-point Newton-Cotes rule. The script

```
% Script File: Show2DQuad
% Integral of SampleF2 over [0,2]x[0,2] for various 2D composite
% QNC(7) rules.

clc
m = 7;
disp(' Subintervals    Integral            Time')
disp('-----------------------------------------------')
for n = [32 64 128 256]
   tic, numI2D = CompQNC2D('SampleF2',0,2,0,2,m,n,m,n); time = toc;
   disp(sprintf(' %7.0f  %17.15f  %11.4f',n,numI2D,time))
end
```

benchmarks this function when it is applied to

$$I = \int_0^2 \int_0^2 \left(\frac{1}{((x-.3)^2+.1)((y-.4)^2+.1)} + \frac{1}{((x-.7)^2+.1)((y-.3)^2+.1)} \right) dy\, dx.$$

The integrand function is implemented in `SampleF2`. Here are the results:

```
Subintervals      Integral         Relative Time
-----------------------------------------------
     32       46.220349653054726       0.1100
     64       46.220349653055095       0.3300
    128       46.220349653055102       1.7000
    256       46.220349653055109       5.0500
```

For larger values of n we would find that the amount of computation increases by a factor of 4 with a doubling of n reflecting the $O(n^2)$ nature of the calculation.

Problems

P5.3.1 Modify `CompQNC2D` so that it computes and uses `F` row at a time. The implementation should not require a two-dimensional array.

P5.3.2 Suppose we are given an interval $[a,b]$, a *Kernel function* $K(x,y)$ defined on $[a,b] \times [a,b]$, and another function $g(x)$ defined on $[a,b]$. Our goal is to find a function $f(x)$ with the property that

$$\int_a^b K(x,y) f(y) dy = g(x) \qquad a \leq x \leq b.$$

Suppose Q is a quadrature rule of the form

$$Q = (b-a)\sum_{j=1}^{N} \omega_j s(x_j)$$

that approximates integrals of the form

$$I = \int_a^b s(x)dx.$$

(The ω_j and x_j are the weights and abscissas of the rule.) We can then replace the integral in our problem with

$$(b-a)\sum_{j}^{N} \omega_j K(x,x_j)f(x_j) = g(x).$$

If we force this equation to hold at $x = x_1, \ldots, x_N$, then we obtain an N-by-N linear system:

$$(b-a)\sum_{j=1}^{N} \omega_j K(x_i,x_j)f(x_j) = g(x_i) \qquad i = 1{:}N$$

in the N unknowns $f(x_j)$, $j = 1{:}N$.

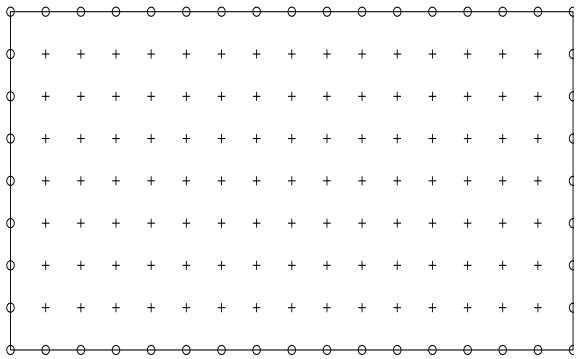
Write a MATLAB function `Kmat = Kernel(a,b,m,n,sigma)` that returns the matrix of coefficients defined by this method, where Q is the composite m-point Newton-Cotes rule with n equal subintervals across $[a,b]$ and $K(x,y) = e^{-(x-y)^2/\sigma}$. Be as efficient as possible, avoiding redundant `exp` evaluations.

Test this solution method with $a = 0$, $b = 5$, and

$$g(x) = \frac{1}{(x-2)^2+.1} + \frac{1}{(x-4)^2+.2}.$$

For $\sigma = .01$, plot the not-a-knot spline interpolant of the computed solution (i.e., the $(x_i, f(x_i))$). Do this for the four cases $(m,n) = (3,5), (3,10), (5,5), (5,10)$. Use `subplot(2,2,*)`. For each subplot, print the time required by your computer to execute `Kernel`. Repeat with $\sigma = .1$.

P5.3.3 The temperature at selected points around the edge of a rectangular plate is known. Our goal is to estimate the temperature $T = T(x,y)$ at selected interior points. In the following figure we depict the points of known temperature by 'o' and the points of unknown temperature by '+':



A reasonable model (whose details we suppress) suggests that the temperature at a '+' point is the average of its four neighbors. (The "north", "east", "south" and "west" neighbors.) Usually the four neighbors are '+' points. However, for a '+' point near the edge, one or two of the neighbors is an 'o' point.

In the figure, the array of '+' points has $m = 7$ rows and $n = 15$ columns. There are thus mn unknown temperatures $t_1, \ldots, t_{mn}$. We associate these unknowns with the '+' points in left-to-right, top-to-bottom order, the order in which we read a page of English text. Let's look at the "averaging" equation at the 37th '+' point. This is the 7th '+' point in the 3rd row (counting rows from the top). First, we figure out who the neighbors are:

- The north neighbor is the 7th '+' point in the 2nd row. (index = 22 = 37-15)
- The west neighbor is the 6th '+' point in the 3rd row. (index = 36 = 37-1)
- The east neighbor is the 8th '+' point in the 3rd row. (index = 38 = 37+1)
- The south neighbor is the 7th '+' point in the 4th row. (index = 52 = 37+15)

Having done that, to say that the temperature at the 37th '+' point is the average of the four neighbor temperatures is to say that

$$-t_{22} - t_{36} + 4t_{37} - t_{38} - t_{52} = 0.$$

This is a linear equation in five unknowns. Equations associated with '+' points that are next to an edge are similar except that known edge temperatures are involved. For example, the equation at the 5th star point is given by

$$-t_4 + 4t_5 - t_6 - t_{20} = north_5,$$

where $north_5$ is the (known) temperature at the 5th 'o' point along the top edge. This is a linear equation that involves four of the unknowns.

Thus, the vector of unknowns t solves an mn-by-mn linear system of the form $At = b$. Write a MATLAB function `[A,b] = Poisson(m,n)` that returns the solution to this system. Assume that the known west edge and east edge temperatures are zero and that the north (top) and south (bottom) edge temperatures are given by

```
x = linspace(0,2,n+2);
fnorth = sin((pi/2)*x)*exp(-x);
north = fnorth(2:n+1);
south = north(n:-1:1);
```

Use \ to solve the linear system. Print A and b for the case $m = 3$, $n = 4$. For the case $m = 7$, $n = 15$, submit the contour plot `cs = contour(Tmatrix,10); clabel(cs)` where `Tmatrix` is the m-by-n matrix obtained by breaking up the solution vector into length n subvectors and stacking them row-by-row. That is, if $t = 1{:}12$, $m = 3$, and $n = 4$, then

$$Tmatrix = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}.$$

Set `axis off` in your contour plot since the xy coordinates are not of particular interest to us in this problem.

5.4 Recursive Matrix Operations

Some of the most interesting algorithmic developments in matrix computations are recursive. Two examples are given in this section. The first is the fast Fourier transform, a super-quick way of computing a special, very important matrix-vector product. The second is a recursive matrix multiplication algorithm that involves markedly fewer flops than the conventional algorithm.

5.4.1 The Fast Fourier Transform

The discrete Fourier transform (DFT) matrix is a complex Vandermonde matrix. Complex numbers have the form $a + i \cdot b$, where $i = \sqrt{-1}$. If we define

$$\omega_4 = \exp(-2\pi i/4) = \cos(2\pi/4) - i \cdot \sin(2\pi/4) = -i,$$

then the 4-by-4 DFT matrix is given by

$$F_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega_4 & \omega_4^2 & \omega_4^3 \\ 1 & \omega_4^2 & \omega_4^4 & \omega_4^6 \\ 1 & \omega_4^3 & \omega_4^6 & \omega_4^9 \end{bmatrix}.$$

The parameter ω_4 is a fourth root of unity, meaning that $\omega_4^4 = 1$. It follows that

$$F_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix}.$$

MATLAB supports complex matrix manipulation. The commands

```
i = sqrt(-1);
F = [1 1 1 1; 1 -i -1 i; 1 -1 1 -1; 1 i -1 -i]
```

assign the 4-by-4 DFT matrix to `F`.

For general n, the DFT matrix is defined in terms of

$$\omega_n = \exp(-2\pi i/n) = \cos(2\pi/n) - i \cdot \sin(2\pi/n).$$

In particular, the n-by-n DFT matrix is defined by

$$F_n = (f_{pq}), \qquad f_{pq} = \omega_n^{(p-1)(q-1)}.$$

Setting up the DFT matrix gives us an opportunity to sample MATLAB's complex arithmetic capabilities:

```
F = ones(n,n);
F(:,2) = exp(-2*pi*sqrt(-1)/n).^(0:n-1)';
for k=3:n
   F(:,k) = F(:,2).*F(:,k-1);
end
```

There is really nothing new here except that the generated matrix entries are complex with the involvement of $\sqrt{-1}$. The real and imaginary parts of a matrix can be extracted using the functions `real` and `imag`. Thus, if $F = F_R + i \cdot F_I$ and $x = x_R + i \cdot x_I$, where F_R, F_I, x_R, and x_I are real, then

```
y = F*x
```

is equivalent to

```
FR = real(F); FI = imag(F);
xR = real(x); xI = imag(x);
y = (FR*xR - FI*xI) + sqrt(-1)*(FR*xI + FI*xR);
```

because

$$y = (F_R x_R - F_I x_I) + i \cdot (F_R x_I + F_I * x_R).$$

Many of MATLAB's built-in functions like `exp`, accept complex matrix arguments. When complex computations are involved, `flops` counts *real* flops. Note that complex addition requires two real flops and that complex multiplication requires six real flops.

Returning to the DFT, it is possible to compute $y = F_n x$ without explicitly forming the DFT matrix F_n:

```
n = length(x);
y = x(1)*ones(n,1);
for k=2:n
   y = y + exp(-2*pi*sqrt(-1)*(k-1)*(0:n-1)') *x(k);
end
```

The update carries out the saxpy computation

$$y = \begin{bmatrix} 1 \\ \omega_n^{k-1} \\ \omega_n^{2(k-1)} \\ \vdots \\ \omega_n^{(n-1)(k-1)} \end{bmatrix} x_k.$$

Notice that since $\omega_n^n = 1$, all powers of ω_n are in the set $\{1, \omega_n, \omega_n^2, \ldots, \omega_n^{n-1}\}$. In particular, $\omega_n^m = \omega_n^{m \bmod n}$. Thus, if

```
v = exp(-2*pi*sqrt(-1)/n)^(0:n-1)';
z = rem((k-1)*(0:n-1)',n ) +1;
```

then `v(z)` equals the kth column of F_n and we obtain

```
  function y = DFT(x)
% y = DFT(x)
% y is the discrete Fourier transform of a column n-vector x.
n = length(x);
y = x(1)*ones(n,1);
if n > 1
   v = exp(-2*pi*sqrt(-1)/n).^(0:n-1)';
   for k=2:n
      z = rem((k-1)*(0:n-1)',n )+1;
      y = y + v(z)*x(k);
   end
end
```

This is an $O(n^2)$ algorithm. We now show how to obtain an $O(n \log_2 n)$ implementation by exploiting the structure of F_n.

The starting point is to look at an even order DFT matrix when we permute its columns so that the odd-indexed columns come first. Consider the case $n = 8$. Noting that $\omega_8^m = \omega_n^{m \bmod 8}$, we have

$$F_8 = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \\ 1 & \omega^2 & \omega^4 & \omega^6 & 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega & \omega^4 & \omega^7 & \omega^2 & \omega^5 \\ 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^5 & \omega^2 & \omega^7 & \omega^4 & \omega & \omega^6 & \omega^3 \\ 1 & \omega^6 & \omega^4 & \omega^2 & 1 & \omega^6 & \omega^4 & \omega^2 \\ 1 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega \end{bmatrix},$$

where $\omega = \omega_8$. If $cols = [1\ 3\ 5\ 7\ 2\ 4\ 6\ 8]$, then

$$F_8(:, cols) = \left[\begin{array}{cccc|cccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega^2 & \omega^4 & \omega^6 & \omega & \omega^3 & \omega^5 & \omega^7 \\ 1 & \omega^4 & 1 & \omega^4 & \omega^2 & \omega^6 & \omega^2 & \omega^6 \\ 1 & \omega^6 & \omega^4 & \omega^2 & \omega^3 & \omega & \omega^7 & \omega^5 \\ \hline 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & \omega^2 & \omega^4 & \omega^6 & -\omega & -\omega^3 & -\omega^5 & -\omega^7 \\ 1 & \omega^4 & 1 & \omega^4 & -\omega^2 & -\omega^6 & -\omega^2 & -\omega^6 \\ 1 & \omega^6 & \omega^4 & \omega^2 & -\omega^3 & -\omega & -\omega^7 & -\omega^5 \end{array}\right].$$

The lines through the matrix help us think of the matrix as a 2-by-2 matrix with 4-by-4 "blocks." Noting that $\omega^2 = \omega_8^2 = \omega_4$ we see that

$$F_8(:, cols) = \left[\begin{array}{c|c} F_4 & DF_4 \\ \hline F_4 & -DF_4 \end{array}\right],$$

where

$$D = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \omega & 0 & 0 \\ 0 & 0 & \omega^2 & 0 \\ 0 & 0 & 0 & \omega^3 \end{bmatrix}.$$

It follows that if x is an 8-vector, then

$$F_8 x = F(:, cols)x(cols) = \left[\begin{array}{c|c} F_4 & DF_4 \\ \hline F_4 & -DF_4 \end{array}\right] \left[\begin{array}{c} x(1{:}2{:}8) \\ \hline x(2{:}2{:}8) \end{array}\right] = \left[\begin{array}{c|c} I & D \\ \hline I & -D \end{array}\right] \left[\begin{array}{c} F_4 x(1{:}2{:}8) \\ \hline F_4 x(2{:}2{:}8) \end{array}\right].$$

Thus, by simple scalings we can obtain the eight-point DFT $y = F_8 x$ from the four-point DFTs $y_T = F_4 x(1{:}2{:}8)$ and $y_B = F_4 x(2{:}2{:}8)$:

$$\begin{aligned} y(1{:}4) &= y_T + d .* y_B \\ y(5{:}8) &= y_T - d .* y_B \end{aligned}$$

where d is the following "vector of weights" $d = \begin{bmatrix} 1\ \omega\ \omega^2\ \omega^3 \end{bmatrix}^T$. In general, if $n = 2m$, then $y = F_n x$ is given by

$$\begin{aligned} y(1{:}m) &= y_T + d .* y_B \\ y(m+1{:}n) &= y_B - d .* y_B, \end{aligned}$$

where $y_T = F_m x(1{:}2{:}n)$, $y_B = F_m x(2{:}2{:}n)$, and

$$d = \begin{bmatrix} 1 \\ \omega \\ \vdots \\ \omega_n^{m-1} \end{bmatrix}.$$

For $n = 2^t$ we can recur on this process until $n = 1$. (The one-point DFT of a one-vector is itself). This gives

```
  function y = FFTRecur(x)
% y = FFTRecur(x)
% y is the discrete Fourier transform of a column n-vector x where
% n is a power of two.

  n = length(x);
  if n ==1
     y = x;
  else
     m = n/2;
     yT = FFTRecur(x(1:2:n));
     yB = FFTRecur(x(2:2:n));
     d = exp(-2*pi*sqrt(-1)/n).^(0:m-1)';
     z = d.*yB;
     y = [ yT+z ; yT-z ];
  end
```

This is a member of the *fast Fourier transform* (FFT) family of algorithms. They involve $O(n \log_2 n)$ flops. We have illustrated a radix-2 FFT. It requires n to be a power of 2. Other radices are possible. MATLAB includes a radix-2 fast Fourier transform `FFT`. (See also `DFTdirect`.) The script `FFTflops` tabulates the number of flops required by `DFT`, `FFTrecur`, and `FFT`:

n	DFT Flops	FFTrecur Flops	FFT Flops
2	87	40	16
4	269	148	61
8	945	420	171
16	3545	1076	422
32	13737	2612	986
64	54089	6132	2247
128	214665	14068	5053
256	855305	31732	11260
512	3414537	70644	24900
1024	13644809	155636	54677

The reason that `FFTrecur` involves more flops than `FFT` concerns the computation of the "weight vector" `d`. As it stands, there is a considerable amount of redundant computation with respect

to the exponential values that are required. This can be avoided by precomputing the weights, storing them in a vector, and then merely "looking up" the values as they are needed during the recursion. With care, the amount of work required by a radix-2 FFT is $5n \log_2 n$ flops.

5.4.2 Strassen Multiplication

Ordinarily, 2-by-2 matrix multiplication requires eight multiplications and four additions:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}b_{11} + A_{22}B_{21} & A_{21}b_{12} + A_{22}B_{22} \end{bmatrix}.$$

In the *Strassen* multiplication scheme, the computations are rearranged so that they involve seven multiplications and 18 additions:

$$\begin{aligned} P_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ P_2 &= (A_{21} + A_{22})B_{11} \\ P_3 &= A_{11}(B_{12} - B_{22}) \\ P_4 &= A_{22}(B_{21} - B_{11}) \\ P_5 &= (A_{11} + A_{12})B_{22} \\ P_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ P_7 &= (A_{12} - A_{22})(B_{21} + B_{22}) \\ C_{11} &= P_1 + P_4 - P_5 + P_7 \\ C_{12} &= P_3 + P_5 \\ C_{21} &= P_2 + P_4 \\ C_{22} &= P_1 + P_3 - P_2 + P_6 \end{aligned}$$

It is easy to verify that these recipes correctly define the product AB. However, why go through these convoluted formulas when ordinary 2-by-2 multiplication involves just eight multiplies and four additions? To answer this question, we first observe that the Strassen specification holds when the A_{ij} and B_{ij} are square matrices themselves. In this case, it amounts to a special method for computing block 2-by-2 matrix products. The seven multiplications are now m-by-m matrix multiplications and require $2(7m^3)$ flops. The 18 additions are matrix additions and they involve $18m^2$ flops. Thus, for this block size the Strassen multiplication requires

$$2(7m^3) + 18m^2 = \frac{7}{8}(2n^3) + \frac{9}{2}n^2$$

flops while the corresponding figure for the conventional algorithm is given by $2n^3 - n^2$. We see that for large enough n, the Strassen approach involves less arithmetic.

The idea can obviously be applied recursively. In particular, we can apply the Strassen algorithm to each of the half-sized block multiplications associated with the P_i. Thus, if the original A and B are n-by-n and $n = 2^q$, then we can recursively apply the Strassen multiplication algorithm all the way to the 1-by-1 level. However, for small n the Strassen approach involves more flops than the ordinary matrix multiplication algorithm, Therefore, for some $n_{min} \geq 1$ it makes sense to "switch over" to the standard algorithm. In the following implementation, $n_{min} = 16$:

```
  function C = Strass(A,B,nmin)
% C = Strass(A,B,nmin)
% This computes the matrix-matrix product C = A*B (via the Strassen Method) where
% A is an n-by-n matrix, B is a n-by-n matrix and n is a power of two. Conventional
% matrix multiplication is used if n<nmin where nmin is a positive integer.

[n,n] = size(A);
if n < nmin
   C = A*B;
else
   m = n/2; u = 1:m; v = m+1:n;
   P1 = Strass(A(u,u)+A(v,v),B(u,u)+B(v,v),nmin);
   P2 = Strass(A(v,u)+A(v,v),B(u,u),nmin);
   P3 = Strass(A(u,u),B(u,v)-B(v,v),nmin);
   P4 = Strass(A(v,v),B(v,u)-B(u,u),nmin);
   P5 = Strass(A(u,u)+A(u,v),B(v,v),nmin);
   P6 = Strass(A(v,u)-A(u,u),B(u,u) + B(u,v),nmin);
   P7 = Strass(A(u,v)-A(v,v),B(v,u)+B(v,v),nmin);
   C = [ P1+P4-P5+P7   P3+P5; P2+P4 P1+P3-P2+P6];
end
```

The script `StrassFlops` tabulates the number of flops required by this function for various values of n and with `nmin = 32`:

```
   n   Strass Flops/Ordinary Flops
-------------------------------
  32           0.945
  64           0.862
 128           0.772
 256           0.684
 512           0.603
1024           0.530
```

Problems

P5.4.1 Modify `FFTrecur` so that it does not involve redundant weight computation.

P5.4.2 This problem is about a fast solution to the trigonometric interpolation problem posed in §2.4.4 on page 100. Suppose f is a real n-vector with $n = 2m$. Suppose $y = (2/n)F_n f = \tilde{a} - i\tilde{b}$ where $\tilde{a}$ and $\tilde{b}$ are real n-vectors. It can be shown that

$$a_j = \begin{cases} \tilde{a}_j/2 & j = 1, m+1 \\ \tilde{a}_j & j = 2{:}m \end{cases}$$

and

$$b_j = \tilde{b}_{j+1} \qquad j = 1{:}m-1$$

prescribe the vectors `F.a` and `F.b` returned by `CSinterp(f)`. Using these facts, write a fast implementation of that function that relies on the FFT. Confirm that $O(n \log n)$ flops are required.

P5.4.3 Modify `Strass` so that it can handle general n. Hint: You will have to figure out how to partition the matrix multiplication if n is odd.

P5.4.4 Let $f(q,r)$ be the number of flops required by `Strass` if $n = 2^q$ and $n_{min} = 2^r$. Assume that $q \geq r$. Develop an analytical expression for $f(q,r)$. For $q = 1{:}20$, compare $\min_{r \leq s} f(q,r)$ and $2 \cdot (2^q)^3$.

5.5 Distributed Memory Matrix Multiplication

In a shared memory machine, individual processors are able to read and write data to a (typically large) global memory. A snapshot of what it is like to compute in a shared memory environment is given at the end of Chapter 4, where we used the quadrature problem as an example.

In contrast to the shared memory paradigm is the *distributed memory* paradigm. In a distributed memory computer, there is an *interconnection network* that links the processors and is the basis for communication. In this section we discuss the parallel implementation of matrix multiplication, a computation that is highly parallelizable and provides a nice opportunity to contrast the shared and distributed memory approaches.[1]

5.5.1 Networks and Communication

We start by discussing the general set-up in a distributed memory environment. Popular interconnection networks include the ring and the mesh. See Figure 5.4 below and 5.5 on the next page.

The individual processors come equipped with their own processing units, local memory, and input/output ports. The act of designing a parallel algorithm for a distributed memory system is the act of designing a *node program* for each of the participating processors. As we shall see, these programs look like ordinary programs with occasional `send` and `recv` commands that are used for the sending and receiving of messages. For us, a message is a matrix. Since vectors and scalars are special matrices, a message may consist of a vector or a scalar.

In the following we suppress very important details such as (1) how data and programs are downloaded into the nodes, (2) the subscript computations associated with *local* array access,

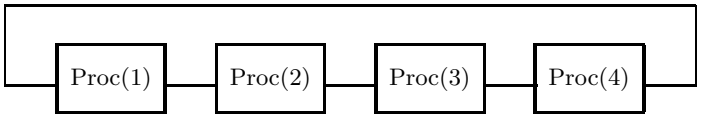


FIGURE 5.4 A ring multiprocessor

[1] The distinction between shared memory multiprocessors and distributed memory multiprocessors is fuzzy. A shared memory can be physically distributed. In such a case, the programmer has all the convenience of being able to write node programs that read and write directly into the shared memory. However, the physical distribution of the memory means that either the programmer or the compiler must strive to access "nearby" memory as much as possible.

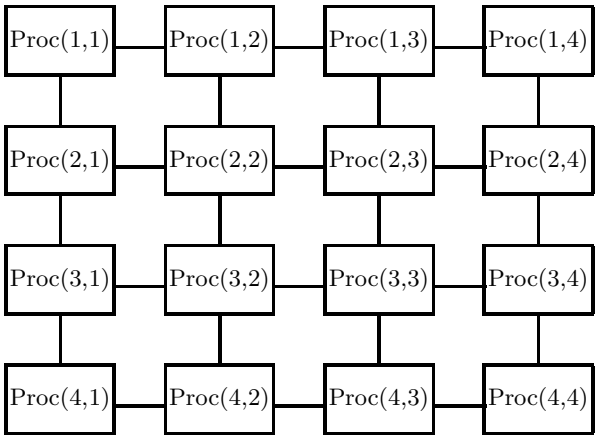


FIGURE 5.5 A mesh multiprocessor

and (3) the formatting of messages. We designate the μth processor by Proc(μ). The μth node program is usually a function of μ.

The distributed memory paradigm is quite general. At the one extreme we can have networks of workstations. On the other hand, the processors may be housed on small boards that are stacked in a single cabinet sitting on a desk.

The kinds of algorithms that can be efficiently solved on a distributed memory multiprocessor are defined by a few key properties:

- *Local memory size.* An interesting aspect of distributed memory computing is that the amount of data for a problem may be so large that it cannot fit inside a single processor. For example, a 10,000-by-10,000 matrix of floating point numbers requires almost a thousand megabytes of storage. The storage of such an array may require the local memories from hundreds of processors. Local memory often has a hierarchy, as does the memory of conventional computers (e.g., disks $\rightarrow$ slow random access memory $\rightarrow$ fast random access memory, etc.).

- *Processor speed.* The speed at which the individual processing units execute is of obvious importance. In sophisticated systems, the individual nodes may have vector capabilities, meaning that the extraction of peak performance from the system requires algorithms that vectorize at the node level. Another complicating factor may be that system is made up of different kinds of processors. For example, maybe every fourth node has a vector processing accelerator.

- *Message passing overhead.* The time it takes for one processor to send another processor a message determines how often a node program will want to break away from productive

calculation to receive and send data. It is typical to model the time it takes to send or receive an n-byte message by

$$T(n) = \alpha + \beta n. \tag{5.1}$$

Here, α is the *latency* and β is the *bandwidth*. The former measures how long it takes to "get ready" for a send/receive while the latter is a reflection of the "size" of the wires that connect the nodes. This model of communication provides some insight into performance, but it is seriously flawed in at least two regards. First, the proximity of the receiver to the sender is usually an issue. Clearly, if the sender and receiver are neighbors in the network, then the system software that routes the message will not have so much to do. Second, the message passing software or hardware may require the breaking up of large messages into smaller packets. This takes time and magnifies the effect of latency.

The examples that follow will clarify some of these issues.

5.5.2 A Two-processor Matrix Multiplication

Consider the matrix-matrix multiplication problem

$$C \leftarrow C + AB,$$

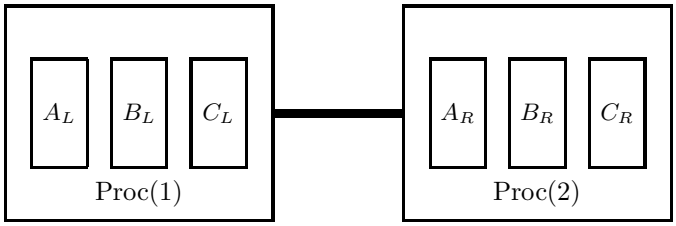
where the three matrices $A, B, C \in \mathbb{R}^{n\times n}$ are distributed in a two-processor distributed memory network. To be specific, assume that $n = 2m$ and that Proc(1) houses the "left halves"

$$A_L = A(:, 1{:}m), \quad B_L = B(:, 1{:}m), \quad C_L = C(:, 1{:}m),$$

and that Proc(2) houses the "right halves"

$$A_R = A(:, m+1{:}n), \quad B_R = B(:, m+1{:}n) \quad C_R = C(:, m+1{:}n).$$

Pictorially we have



We assign to Proc(1) and Proc(2) the computation of the new C_L and C_R, respectively. Let's see where the data for these calculations come from. We start with a complete specification of the overall computation:

```
for j=1:n
   C(:,j) = C(:,j) + A*B(:,j);
end
```

Note that the jth column of the updated C is the jth column of the original C plus A times the jth column of B. The matrix-vector product `A*B(:,j)` can be expressed as a linear combination of A-columns:

$$\texttt{A} * \texttt{B}(:, \texttt{j}) = \texttt{A}(:, 1) * \texttt{B}(1, \texttt{j}) + \texttt{A}(:, 2) * \texttt{B}(2, \texttt{j}) + \cdots + \texttt{A}(:, \texttt{n}) * \texttt{B}(\texttt{n}, \texttt{j}).$$

Thus the preceding fragment expands to

```
for j=1:n
   for k=1:n
      C(:,j) = C(:,j) + A(:,k)*B(k,j);
   end
end
```

Note that Proc(1) is in charge of

```
for j=1:m
   for k=1:n
      C(:,j) = C(:,j) + A(:,k)*B(k,j);
   end
end
```

while Proc(2) must carry out

```
for j=m+1:n
   for k=1:n
      C(:,j) = C(:,j) + A(:,k)*B(k,j);
   end
end
```

From the standpoint of communication, there is both good news and bad news. The good news is that the B-data and C-data required are local. Proc(1) requires (and has) the left portions of C and B. Likewise, Proc(2) requires (and has) the right portions of these same matrices. The bad news concerns A. Both processors must "see" all of A during the course of execution. Thus, for each processor exactly one half of the A-columns are nonlocal, and these columns must somehow be acquired during the calculation. To highlight the local and nonlocal A-data, we pair off the left and the right A-columns:

```
Proc(1) does this:

for j=1:m
   for k=1:m
      C(:,j) = C(:,j) + A(:,k)*B(k,j);
      C(:,j) = C(:,j) + A(:,k+m)*B(k+m,j);
   end
end
```

```
Proc(2) does this:

for j=m+1:n
   for k=1:m
      C(:,j) = C(:,j) + A(:,k+m)*B(k+m,j);
      C(:,j) = C(:,j) + A(:,k)*B(k,j);
   end
end
```

In each case, the second update of `C(:,j)` requires a nonlocal A-column. Somehow, Proc(1) has to get hold of $A(:, k+m)$ from Proc(2). Likewise, Proc(2) must get hold of $A(:, k)$ from Proc(1).

The primitives `send` and `recv` are to be used for this purpose. They have the following syntax:

`send`({*matrix*} , {*destination node*}) `recv`({*matrix*} , {*source node*})

If a processor invokes a `send`, then we assume that execution resumes immediately after the message is sent. If a `recv` is encountered, then we assume that execution of the node program is suspended until the requested message arrives. We also assume that messages arrive in the same order that they are sent.[2]

With `send` and `recv`, we can solve the nonlocal data problem in our two-processor matrix multiply:

```
Proc(1) does this:

for j=1:m
   for k=1:m
      send(A(:,k),2);
      C(:,j) = C(:,j) + A(:,k)*B(k,j);
      recv(v,2);
      C(:,j) = C(:,j) + v*B(k+m,j);
   end
end
```

```
Proc(2) does this:

for j=m+1:n
   for k=1:m
      send(A(:,k+m),1);
      C(:,j) = C(:,j) + A(:,k+m)*B(k+m,j);
      recv(v,1);
      C(:,j) = C(:,j) + v*B(k,j);
   end
end
```

Each processor has a local work vector v that is used to hold an A-column solicited from its neighbor. The correctness of the overall process follows from the assumption that the messages arrive in the order that they are sent. This ensures that each processor "knows" the index of the incoming columns. Of course, this is crucial because incoming A-columns have to scaled by the appropriate B entries.

Although the algorithm is correct and involves the expected amount of floating point arithmetic, it is inefficient from the standpoint of communication. Each processor sends a given local A-column to its neighbor m times. A better plan is to use each incoming A-column in all the $C(:, j)$ updates "at once." To do this, we merely reorganize the order in which things are done in the node programs:

```
Proc(1) does this:

for k=1:m
   send(A(:,k),2);
   for j=1:m
      C(:,j) = C(:,j) + A(:,k)*B(k,j);
   end;
   recv(v,2);
   for j=1:m
      C(:,j) = C(:,j) + v*B(k+m,j);
   end
end
```

```
Proc(2) does this:

for k=1:m
   send(A(:,k+m),1);
   for j=m+1:n
      C(:,j) = C(:,j) + A(:,k+m)*B(k+m,j);
   end
   recv(v,1);
   for j=m+1:n
      C(:,j) = C(:,j) + v*B(k,j);
   end
end
```

Although there is "perfect symmetry" between the two node programs, we cannot assume that they proceed in lock-step fashion. However, the program is *load balanced* because each processor has roughly the same amount of arithmetic and communication.

[2]This need not be true in practice. However, a system of tagging messages can be incorporated that can be used to remove ambiguities.

In the preceding, the matrices A, B, and C are accessed as if they were situated in a single memory. Of course, this will not be the case in practice. For example, Proc(2) will have a local n-by-m array to house its portion of C. Let's call this array `C.loc` and assume that `C.loc(i,j)` houses matrix element $C(i, j + m)$. Similarly, there will be local array `A.loc` and `B.loc` that house their share of A and B, respectively. If we rewrite Proc(2)'s node program in the true "language" of its local arrays, then it becomes

```
for k=1:m
   send(A.loc(:,k),1);
   for j=1:m
      C.loc(:,j) = C.loc(:,j) + A.loc(:,k)*B.loc(k,j);
   end
   recv(v,1);
   for j=1:m
      C.loc(:,j) = C(:,j) + v*B.loc(k+m,j);
   end
end
```

We merely mention this to stress that what may be called "subscript reasoning" undergoes a change when working in distributed memory environments.

5.5.3 Performance Analysis

Let's try to anticipate the time required to execute the communication-efficient version of the two-processor matrix multiply. There are two aspects to consider: computation and communication. With respect to computation, we note first that overall calculation involves $2n^3$ flops. This is because there are n^2 entries in C to update and each update requires the execution of a length n inner product (e.g., $C(i, j) = C(i, j) + A(i, :) * B(:, j)$). Since an inner product of that length involves n adds and n multiplies, $n^2(2n)$ flops are required in total. These flops are distributed equally between the two processors. If computation proceeds at a uniform rate of R flops per second, then

$$T_{comp} = n^3/R$$

seconds are required to take care of the arithmetic.[3]

With respect to communication costs, we use the model (5.1) on page 203 and conclude that each processor spends

$$T_{comm} = n(\alpha + 8\beta n)$$

seconds sending and receiving messages. (We assume that each floating point number has an 8-byte representation.) Note that this is not the whole communication overhead story because we are not taking into account the idle wait times associated with the receives. (The required vector may not have arrived at time of the `recv`.) Another factor that complicates performance evaluation is that each node may have a special input/output processor that more or less handles communication making it possible to overlap computation and communication.

[3]Recall the earlier observation that with many advanced architectures, the execution rate varies with the operation performed and whether or not it is vectorized.

Ignoring these possibly significant details leads us to predict an overall execution time of $T = T_{comp} + T_{comm}$ seconds. It is instructive to compare this time with what would be required by a single-processor program. Look at the ratio

$$S \;=\; \frac{2n^3/R}{(n^3/R) + n(\alpha + 8\beta n)} = \frac{2}{1 \;+\; R\left(\frac{\alpha}{n^2} + \frac{8\beta}{n}\right)}.$$

S represents the *speed-up* of the parallel program. We make two observations: (1) Communication overheads are suppressed as n increases, and (2) if α and β are fixed, then speed-up decreases as the rate of computation R improves.

In general, the speed-up of a parallel program executing on p processors is a ratio:

$$\text{Speed-up} \;=\; \frac{\text{Time required by the best single-processor program}}{\text{Time required for the } p\text{-processor implementation}}.$$

In this definition we do not just set the numerator to be the $p = 1$ version of the parallel code because the best uniprocessor algorithm may not parallelize. Ideally, one would like the speed-up for an algorithm to equal p.

Problems

P5.5.1 Assume that $n = 3m$ and that the n-by-n matrices A, B, C are distributed around a three-processor ring. In particular, assume that processors 1, 2, and 3 house the left third, middle third, and right third of these matrices. For example, $B(:, m+1{:}2m)$ would be housed in Proc(2). Write a parallel program for the computation $C \leftarrow C + AB$.

P5.5.2 Suppose we have a two-processor distributed memory system in which floating point arithmetic proceeds at R flops per second. Assume that when one processor sends or receives a message of k floating point numbers, then $\alpha + \beta k$ seconds are required. Proc(1) houses an n-by-n matrix A, and each processor houses a copy of an n-vector x. The goal is to store the vector $y = Ax$ in Proc(1)'s local memory. You may assume that n is even. **(a)** How long would this take if Proc(1) handles the entire computation itself? **(b)** Describe how the two processors can share the computation. Indicate the data that must flow between the two processors and what they must each calculate. You do not have to write formal node programs. Clear concise English will do. **(c)** Does it follow that if n is large enough, then it is more efficient to distribute the computation? Justify your answer.

M-Files and References

Script Files

`CircBench`	Benchmarks Circulant1 and Circulant2.
`MatBench`	Benchmarks various matrix-multiply methods.
`AveNorms`	Compares various norms on random vectors.
`ProdBound`	Examines the error in three-digit matrix multiplication.
`ShowContour`	Displays various contour plots of SampleF.
`Show2DQuad`	Illustrates CompQNC2D.
`FFTflops`	Compares FFT and DFT flops.
`StrassFlops`	Examines Strassen multiply.

Function Files

`Circulant1`	Scalar-level circulant matrix set-up.
`Circulant2`	Vector-level circulant matrix set-up.
`MatVecRO`	Row-oriented matrix-vector product.
`MatVecCO`	Column-Oriented matrix-vector product.
`MatMatDot`	Dot-product matrix-matrix product.
`MatMatSax`	Saxpy matrix-matrix product.
`MatMatVec`	Matrix-vector matrix-matrix product.
`MatMatOuter`	Outer product matrix-matrix product.
`MakeBlock`	Makes a cell array representation of a block matrix.
`ShowSparse`	Illustrates `sparse`.
`ShowNonZeros`	Displays the sparsity pattern of a matrix.
`Prod3Digit`	Three-digit matrix-matrix product.
`SampleF`	A Gaussian type function of two variables.
`CompQNC2D`	Two-dimensional Newton-Cotes rules.
`wCompNC`	Weight vector for composite Newton-Cotes rules.
`SampleF2`	A function of two variables with strong local maxima.
`DFT`	Discrete Fourier transform.
`FFTRecur`	A recursive radix-2 FFT.
`Strass`	Recursive Strassen matrix multiply.

References

T.F. Coleman and C.F. Van Loan (1988). *Handbook for Matrix Computations*, SIAM Publications, Philadelphia, PA.

G.H. Golub and C.F. Van Loan (1996). *Matrix Computations, Third Edition*, Johns Hopkins University Press, Baltimore, MD.

C.F. Van Loan (1992). *Computational Frameworks for the Fast Fourier Transform*, SIAM Publications, Philadelphia, PA.