preconditioner. Since iterative methods are typically used on sparse matrices, we will review here a number of sparse storage formats. Often, the storage scheme used arises naturally from the specific application problem.

In this section we will review some of the more popular sparse matrix formats that are used in numerical software packages such as `ITPACK` [140] and `NSPCG` [165]. After surveying the various formats, we demonstrate how the matrix-vector product and an incomplete factorization solve are formulated using two of the sparse matrix formats.

4.3.1 Survey of Sparse Matrix Storage Formats

If the coefficient matrix A is sparse, large-scale linear systems of the form $Ax = b$ can be most efficiently solved if the zero elements of A are not stored. Sparse storage schemes allocate contiguous storage in memory for the nonzero elements of the matrix, and perhaps a limited number of zeros. This, of course, requires a scheme for knowing where the elements fit into the full matrix.

There are many methods for storing the data (see for instance Saad [186] and Eijkhout [87]). Here we will discuss Compressed Row and Column Storage, Block Compressed Row Storage, Diagonal Storage, Jagged Diagonal Storage, and Skyline Storage.

Compressed Row Storage (CRS)

The Compressed Row and Column (in the next section) Storage formats are the most general: they make absolutely no assumptions about the sparsity structure of the matrix, and they don't store any unnecessary elements. On the other hand, they are not very efficient, needing an indirect addressing step for every single scalar operation in a matrix-vector product or preconditioner solve.

The Compressed Row Storage (CRS) format puts the subsequent nonzeros of the matrix rows in contiguous memory locations. Assuming we have a nonsymmetric sparse matrix A, we create 3 vectors: one for floating-point numbers (`val`), and the other two for integers (`col_ind`, `row_ptr`). The `val` vector stores the values of the nonzero elements of the matrix A, as they are traversed in a row-wise fashion. The `col_ind` vector stores the column indexes of the elements in the `val` vector. That is, if $\texttt{val}(\texttt{k}) = a_{i,j}$ then $\texttt{col\_ind}(\texttt{k}) = j$. The `row_ptr` vector stores the locations in the `val` vector that start a row, that is, if $\texttt{val}(\texttt{k}) = a_{i,j}$ then $\texttt{row\_ptr}(\texttt{i}) \leq k < \texttt{row\_ptr}(\texttt{i}+\texttt{1})$. By convention, we define $\texttt{row\_ptr}(\texttt{n}+\texttt{1}) = nnz + 1$, where nnz is the number of nonzeros in the matrix A. The storage savings for this approach is significant. Instead of storing n^2 elements, we need only $2nnz + n + 1$ storage locations.

As an example, consider the nonsymmetric matrix A defined by

$$A = \begin{pmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{pmatrix} . \tag{4.1}$$

The CRS format for this matrix is then specified by the arrays {`val`, `col_ind`, `row_ptr`} given below

val	10	-2	3	9	3	7	8	7	$3 \cdots 9$	13	4	2	-1
col_ind	1	5	1	2	6	2	3	4	$1 \cdots 5$	6	2	5	6

row_ptr	1	3	6	9	13	17	20

.

If the matrix A is symmetric, we need only store the upper (or lower) triangular portion of the matrix. The trade-off is a more complicated algorithm with a somewhat different pattern of data access.

Compressed Column Storage (CCS)

Analogous to Compressed Row Storage there is Compressed Column Storage (CCS), which is also called the *Harwell-Boeing* sparse matrix format [78]. The CCS format is identical to the CRS format except that the columns of A are stored (traversed) instead of the rows. In other words, the CCS format is the CRS format for A^T.

The CCS format is specified by the 3 arrays {`val`, `row_ind`, `col_ptr`}, where `row_ind` stores the row indices of each nonzero, and `col_ptr` stores the index of the elements in `val` which start a column of A. The CCS format for the matrix A in (4.1) is given by

val	10	3	3	9	7	8	4	8	$8 \cdots 9$	2	3	13	-1
row_ind	1	2	4	2	3	5	6	3	$4 \cdots 5$	6	2	5	6

col_ptr	1	4	8	10	13	17	20

.

Block Compressed Row Storage (BCRS)

If the sparse matrix A is comprised of square dense blocks of nonzeros in some regular pattern, we can modify the CRS (or CCS) format to exploit such block patterns. Block matrices typically arise from the discretization of partial differential equations in which there are several *degrees of freedom* associated with a grid point. We then partition the matrix in small blocks with a size equal to the number of degrees of freedom, and treat each block as a dense matrix, even though it may have some zeros.

If n_b is the dimension of each block and $nnzb$ is the number of nonzero blocks in the $n \times n$ matrix A, then the total storage needed is $nnz = nnzb \times n_b^2$. The block dimension n_d of A is then defined by $n_d = n/n_b$.

Similar to the CRS format, we require 3 arrays for the BCRS format: a rectangular array for floating-point numbers (`val`$(1 : nnzb, 1 : n_b, 1 : n_b)$) which stores the nonzero blocks in (block) row-wise fashion, an integer array (`col_ind`$(1 : nnzb)$) which stores the actual column indices in the original matrix A of the $(1, 1)$ elements of the nonzero blocks, and a pointer array (`row_blk`$(1 : n_d + 1)$) whose entries point to the beginning of each block row in `val(:,:,:)` and `col_ind(:)`. The savings in storage locations and reduction in indirect addressing for BCRS over CRS can be significant for matrices with a large n_b.

Compressed Diagonal Storage (CDS)

If the matrix A is banded with bandwidth that is fairly constant from row to row, then it is worthwhile to take advantage of this structure in the storage scheme by storing subdiagonals of the matrix in consecutive locations. Not only can we eliminate

the vector identifying the column and row, we can pack the nonzero elements in such a way as to make the matrix-vector product more efficient. This storage scheme is particularly useful if the matrix arises from a finite element or finite difference discretization on a tensor product grid.

We say that the matrix $A = (a_{i,j})$ is *banded* if there are nonnegative constants p, q, called the left and right *halfbandwidth*, such that $a_{i,j} \neq 0$ only if $i-p \leq j \leq i+q$. In this case, we can allocate for the matrix A an array `val(1:n,-p:q)`. The declaration with reversed dimensions `(-p:q,n)` corresponds to the `LINPACK` band format [73], which unlike CDS, does not allow for an efficiently vectorizable matrix-vector multiplication if `p + q` is small.

Usually, band formats involve storing some zeros. The CDS format may even contain some array elements that do not correspond to matrix elements at all. Consider the nonsymmetric matrix A defined by

$$A = \begin{pmatrix} 10 & -3 & 0 & 0 & 0 & 0 \\ 3 & 9 & 6 & 0 & 0 & 0 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 0 & 0 & 8 & 7 & 5 & 0 \\ 0 & 0 & 0 & 9 & 9 & 13 \\ 0 & 0 & 0 & 0 & 2 & -1 \end{pmatrix} . \tag{4.2}$$

Using the CDS format, we store this matrix A in an array of dimension `(6,-1:1)` using the mapping

$$\texttt{val(i,j)} = a_{i,i+j}. \tag{4.3}$$

Hence, the rows of the `val(:,:)` array are

`val(:,-1)`	0	3	7	8	9	2
`val(:, 0)`	10	9	8	7	9	-1
`val(:,+1)`	-3	6	7	5	13	0

.

Notice the two zeros corresponding to non-existing matrix elements.

A generalization of the CDS format more suitable for manipulating general sparse matrices on vector supercomputers is discussed by Melhem in [154]. This variant of CDS uses a *stripe* data structure to store the matrix A. This structure is more efficient in storage in the case of varying bandwidth, but it makes the matrix-vector product slightly more expensive, as it involves a gather operation.

As defined in [154], a stripe in the $n \times n$ matrix A is a set of positions $S = \{(i, \sigma(i));\ i \in I \subseteq I_n\}$, where $I_n = \{1, \ldots, n\}$ and σ is a strictly increasing function. Specifically, if $(i, \sigma(i))$ and $(j, \sigma(j))$ are in S, then

$$i < j \rightarrow \sigma(i) < \sigma(j).$$

When computing the matrix-vector product $y = Ax$ using stripes, each $(i, \sigma_k(i))$ element of A in stripe S_k is multiplied with both x_i and $x_{\sigma_k(i)}$ and these products are accumulated in $y_{\sigma_k(i)}$ and y_i, respectively. For the nonsymmetric matrix A defined by

$$A = \begin{pmatrix} 10 & -3 & 0 & 1 & 0 & 0 \\ 0 & 9 & 6 & 0 & -2 & 0 \\ 3 & 0 & 8 & 7 & 0 & 0 \\ 0 & 6 & 0 & 7 & 5 & 4 \\ 0 & 0 & 0 & 0 & 9 & 13 \\ 0 & 0 & 0 & 0 & 5 & -1 \end{pmatrix} , \tag{4.4}$$

the 4 stripes of the matrix A stored in the rows of the `val(:,:)` array would be

`val(:,-1)`	0	0	3	6	0	5
`val(:, 0)`	10	9	8	7	9	-1
`val(:,+1)`	0	-3	6	7	5	13
`val(:,+2)`	0	1	-2	0	4	0

.

Jagged Diagonal Storage (JDS)

The Jagged Diagonal Storage format can be useful for the implementation of iterative methods on parallel and vector processors (see Saad [185]). Like the Compressed Diagonal format, it gives a vector length essentially of the size of the matrix. It is more space-efficient than CDS at the cost of a gather/scatter operation.

A simplified form of JDS, called `ITPACK` storage or Purdue storage, can be described as follows. In the matrix from (4.4) all elements are shifted left:

$$\begin{pmatrix} 10 & -3 & 0 & 1 & 0 & 0 \\ 0 & 9 & 6 & 0 & -2 & 0 \\ 3 & 0 & 8 & 7 & 0 & 0 \\ 0 & 6 & 0 & 7 & 5 & 4 \\ 0 & 0 & 0 & 0 & 9 & 13 \\ 0 & 0 & 0 & 0 & 5 & -1 \end{pmatrix} \longrightarrow \begin{pmatrix} 10 & -3 & 1 & \\ 9 & 6 & -2 & \\ 3 & 8 & 7 & \\ 6 & 7 & 5 & 4 \\ 9 & 13 & & \\ 5 & -1 & & \end{pmatrix}$$

after which the columns are stored consecutively. All rows are padded with zeros on the right to give them equal length. Corresponding to the array of matrix elements `val(:,:)`, an array of column indices, `col_ind(:,:)` is also stored:

`val`$(:,1)$	10	9	3	6	9	5
`val`$(:,2)$	-3	6	8	7	13	-1
`val`$(:,3)$	1	-2	7	5	0	0
`val`$(:,4)$	0	0	0	4	0	0

,

`col_ind`$(:,1)$	1	2	1	2	5	5
`col_ind`$(:,2)$	2	3	3	4	6	6
`col_ind`$(:,3)$	4	5	4	5	0	0
`col_ind`$(:,4)$	0	0	0	6	0	0

.

It is clear that the padding zeros in this structure may be a disadvantage, especially if the bandwidth of the matrix varies strongly. Therefore, in the CRS format, we reorder the rows of the matrix decreasingly according to the number of nonzeros per row. The compressed and permuted diagonals are then stored in a linear array. The new data structure is called *jagged diagonals*.

The number of jagged diagonals is equal to the number of nonzeros in the first row, *i.e.*, the largest number of nonzeros in any row of A. The data structure to represent the $n \times n$ matrix A therefore consists of a permutation array (`perm(1:n)`) which reorders the rows, a floating-point array (`jdiag(:)`) containing the jagged diagonals in succession, an integer array (`col_ind(:)`) containing the corresponding column indices, and finally a pointer array (`jd_ptr(:)`) whose elements point to the

beginning of each jagged diagonal. The advantages of JDS for matrix multiplications are discussed by Saad in [185].

The JDS format for the above matrix A in using the linear arrays {`perm`, `jdiag`, `col_ind`, `jd_ptr`} is given below (jagged diagonals are separated by semicolons)

`jdiag`	6	9	3	10	9	5;	7	6	8	-3	13	-1;	5	-2	7	1;	4;
`col_ind`	2	2	1	1	5	5;	4	3	3	2	6	6;	5	5	4	4;	6;

`perm`	4	2	3	1	5	6	`jd_ptr`	1	7	13	17

.

Skyline Storage (SKS)

The final storage scheme we consider is for skyline matrices, which are also called variable band or profile matrices (see Duff, Erisman and Reid [80]). It is mostly of importance in direct solution methods, but it can be used for handling the diagonal blocks in block matrix factorization methods. A major advantage of solving linear systems having skyline coefficient matrices is that when pivoting is not necessary, the skyline structure is preserved during Gaussian elimination. If the matrix is symmetric, we only store its lower triangular part. A straightforward approach in storing the elements of a skyline matrix is to place all the rows (in order) into a floating-point array (`val(:)`), and then keep an integer array (`row_ptr(:)`) whose elements point to the beginning of each row. The column indices of the nonzeros stored in `val(:)` are easily derived and are not stored.

For a nonsymmetric skyline matrix such as the one illustrated in Figure 4.1, we store the lower triangular elements in SKS format, and store the upper triangular elements in a column-oriented SKS format (transpose stored in row-wise SKS format). These two separated *substructures* can be linked in a variety of ways. One approach, discussed by Saad in [186], is to store each row of the lower triangular part and each column of the upper triangular part contiguously into the floating-point array (`val(:)`). An additional pointer is then needed to determine where the diagonal elements, which separate the lower triangular elements from the upper triangular elements, are located.

4.3.2 Matrix vector products

In many of the iterative methods discussed earlier, both the product of a matrix and that of its transpose times a vector are needed, that is, given an input vector x we want to compute products

$$y = Ax \qquad \text{and} \qquad y = A^T x.$$

We will present these algorithms for two of the storage formats from §4.3: CRS and CDS.

CRS Matrix-Vector Product

The matrix vector product $y = Ax$ using CRS format can be expressed in the usual way:

$$y_i = \sum_j a_{i,j} x_j,$$