

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 $\{\text{perm}, \text{jdiag}, \text{col_ind}, \text{jd_ptr}\}$ 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 ($\text{val}(:)$), and then keep an integer array ($\text{row_ptr}(:)$) whose elements point to the beginning of each row. The column indices of the nonzeros stored in $\text{val}(:)$ 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 ($\text{val}(:)$). 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

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

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:

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

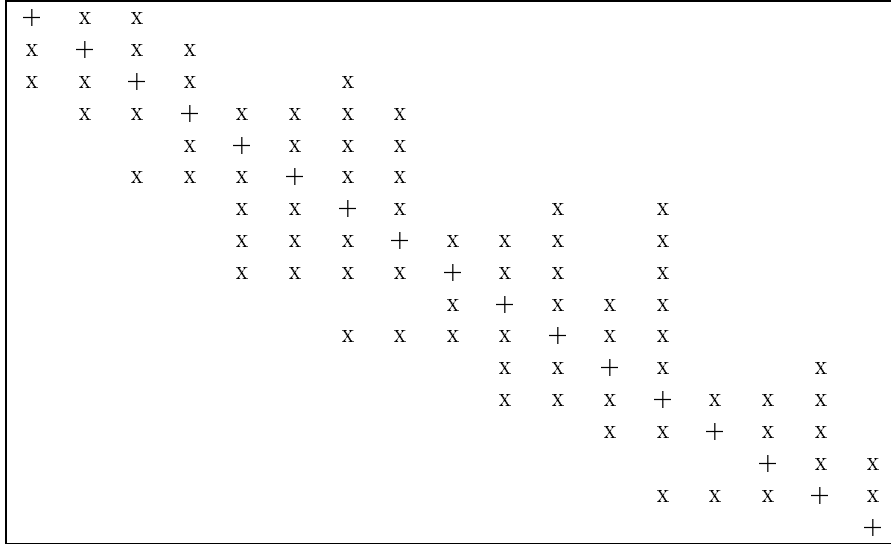


Figure 4.1: Profile of a nonsymmetric skyline or variable-band matrix.

since this traverses the rows of the matrix A . For an $n \times n$ matrix A , the matrix-vector multiplication is given by

```

for i = 1, n
  y(i) = 0
  for j = row_ptr(i), row_ptr(i+1) - 1
    y(i) = y(i) + val(j) * x(col_ind(j))
  end;
end;

```

Since this method only multiplies nonzero matrix entries, the operation count is 2 times the number of nonzero elements in A , which is a significant savings over the dense operation requirement of $2n^2$.

For the transpose product $y = A^T x$ we cannot use the equation

$$y_i = \sum_j (A^T)_{i,j} x_j = \sum_j a_{j,i} x_j,$$

since this implies traversing columns of the matrix, an extremely inefficient operation for matrices stored in CRS format. Hence, we switch indices to

$$\text{for all } j, \text{ do for all } i: \quad y_i \leftarrow y_i + a_{j,i} x_j.$$

The matrix-vector multiplication involving A^T is then given by

```

for i = 1, n
  y(i) = 0
end;
for j = 1, n
  for i = row_ptr(j), row_ptr(j+1)-1
    y(col_ind(i)) = y(col_ind(i)) + val(i) * x(j)
  end;
end;

```

```

    end;
end;

```

Both matrix-vector products above have largely the same structure, and both use indirect addressing. Hence, their vectorizability properties are the same on any given computer. However, the first product ($y = Ax$) has a more favorable memory access pattern in that (per iteration of the outer loop) it reads two vectors of data (a row of matrix A and the input vector x) and writes one scalar. The transpose product ($y = A^T x$) on the other hand reads one element of the input vector, one row of matrix A , and both reads and writes the result vector y . Unless the machine on which these methods are implemented has three separate memory paths (*e.g.*, Cray Y-MP), the memory traffic will then limit the performance. This is an important consideration for RISC-based architectures.

CDS Matrix-Vector Product

If the $n \times n$ matrix A is stored in CDS format, it is still possible to perform a matrix-vector product $y = Ax$ by either rows or columns, but this does not take advantage of the CDS format. The idea is to make a change in coordinates in the doubly-nested loop. Replacing $j \rightarrow i + j$ we get

$$y_i \leftarrow y_i + a_{i,j} x_j \quad \Rightarrow \quad y_i \leftarrow y_i + a_{i,i+j} x_{i+j} .$$

With the index i in the inner loop we see that the expression $a_{i,i+j}$ accesses the j th diagonal of the matrix (where the main diagonal has number 0).

The algorithm will now have a doubly-nested loop with the outer loop enumerating the diagonals **diag=-p,q** with p and q the (nonnegative) numbers of diagonals to the left and right of the main diagonal. The bounds for the inner loop follow from the requirement that

$$1 \leq i, i + j \leq n .$$

The algorithm becomes

```

for i = 1, n
  y(i) = 0
end;
for diag = -diag_left, diag_right
  for loc = max(1,1-diag), min(n,n-diag)
    y(loc) = y(loc) + val(loc,diag) * x(loc+diag)
  end;
end;

```

The transpose matrix-vector product $y = A^T x$ is a minor variation of the algorithm above. Using the update formula

$$\begin{aligned} y_i &\leftarrow y_i + a_{i+j,i} x_j \\ &= y_i + a_{i+j,i+j-j} x_{i+j} \end{aligned}$$

we obtain

```

for i = 1, n
    y(i) = 0
end;
for diag = -diag_right, diag_left
    for loc = max(1,1-diag), min(n,n-diag)
        y(loc) = y(loc) + val(loc+diag, -diag) * x(loc+diag)
    end;
end;

```

The memory access for the CDS-based matrix-vector product $y = Ax$ (or $y = A^T x$) is three vectors per inner iteration. On the other hand, there is no indirect addressing, and the algorithm is vectorizable with vector lengths of essentially the matrix order n . Because of the regular data access, most machines can perform this algorithm efficiently by keeping three base registers and using simple offset addressing.

4.3.3 Sparse Incomplete Factorizations

Efficient preconditioners for iterative methods can be found by performing an incomplete factorization of the coefficient matrix. In this section, we discuss the incomplete factorization of an $n \times n$ matrix A stored in the CRS format, and routines to solve a system with such a factorization. At first we only consider a factorization of the *D-ILU* type, that is, the simplest type of factorization in which no “fill” is allowed, even if the matrix has a nonzero in the fill position (see section 3.4.2). Later we will consider factorizations that allow higher levels of fill. Such factorizations considerably more complicated to code, but they are essential for complicated differential equations. The solution routines are applicable in both cases.

For iterative methods, such as *QMR*, that involve a transpose matrix vector product we need to consider solving a system with the transpose of as factorization as well.

Generating a CRS-based *D-ILU* Incomplete Factorization

In this subsection we will consider a matrix split as $A = D_A + L_A + U_A$ in diagonal, lower and upper triangular part, and an incomplete factorization preconditioner of the form $(D_A + L_A)D_A^{-1}(D_A + U_A)$. In this way, we only need to store a diagonal matrix D containing the pivots of the factorization.

Hence, it suffices to allocate for the preconditioner only a pivot array of length n (**pivots(1:n)**). In fact, we will store the inverses of the pivots rather than the pivots themselves. This implies that during the system solution no divisions have to be performed.

Additionally, we assume that an extra integer array **diag_ptr(1:n)** has been allocated that contains the column (or row) indices of the diagonal elements in each row, that is, **val(diag_ptr(i))** = $a_{i,i}$.

The factorization begins by copying the matrix diagonal

```

for i = 1, n
    pivots(i) = val(diag_ptr(i))
end;

```

Each elimination step starts by inverting the pivot