IMM
DEPARTMENT OF MATHEMATICAL MODELLING

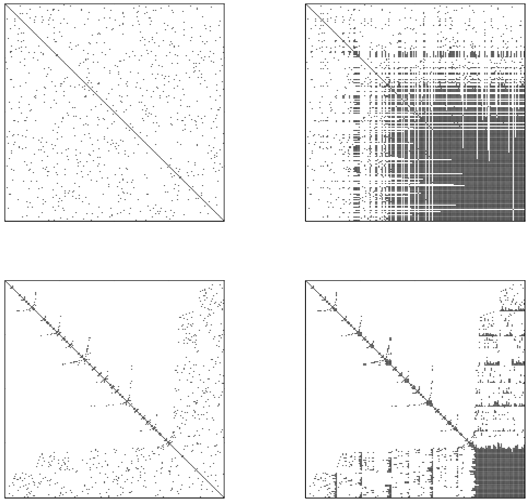
Technical University of Denmark
DK-2800 Lyngby – Denmark

J. No. H65
16.1.1997
HBN/ms

DIRECT METHODS FOR SPARSE MATRICES

An Introduction

Hans Bruun Nielsen



CONTENTS

1. Introduction

A sparse matrix is characterized by being large and having most of its elements equal to zero. We shall see how this can be exploited when storing the matrix and in some operations where the matrix participates. We only look at the matrix–vector product

$$\boldsymbol{y} = \boldsymbol{A}\mathbf{x} \ , \tag{1.1}$$

and the solution of a linear system of equations

$$\boldsymbol{A}\mathbf{x} \ = \boldsymbol{b} \ . \tag{1.2}$$

Here, $\boldsymbol{A} = \{a_{ij}\}$, $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ is a sparse matrix (i.e. n is large and most $a_{ij} = 0$). We ignore the possibility of many zero elements in the vectors $\boldsymbol{b}, \mathbf{x}, \boldsymbol{y} \in \mathbb{R}^n$.

1.1. Basic Operations

The following two basic operations with $\mathbf{x}, \boldsymbol{y} \in \mathbb{R}^n$ play an important role,

DOT: The *dot* (or *inner*) product

$$<\mathbf{x}, \boldsymbol{y}> = \mathbf{x}^T\boldsymbol{y} = x_1y_1 + \cdots + x_ny_n \ . \tag{1.3}$$

SAXPY: Linear combination of two vectors (mnemonic for scalar alpha $\mathbf{x}$ plus $\boldsymbol{y}$),

$$\boldsymbol{z} = \alpha\mathbf{x} + \boldsymbol{y} \ , \tag{1.4}$$

where $\alpha \in \mathbb{R}$, $\boldsymbol{z} \in \mathbb{R}^n$.

The matrix–vector product (1.1) can be expressed in terms of DOT:

$$\begin{array}{l} \texttt{for } i = 1, \ldots, n \\ \quad y_i = <\boldsymbol{A}_{i,:}, \mathbf{x}> \ , \end{array} \tag{1.5a}$$

where $\boldsymbol{A}_{i,:} = [\, a_{i1} \ \ \ldots \ \ a_{in} \,]$ is the ith row of $\boldsymbol{A}$.

Alternatively, we can express $\boldsymbol{y}$ as a linear combination of the columns of $\boldsymbol{A}$,

$$\boldsymbol{y} = x_1\boldsymbol{A}_{:,1} + \cdots + x_n\boldsymbol{A}_{:,n} \ , \tag{1.5b}$$

where $\boldsymbol{A}_{:,j} = [\, a_{1j} \ \ \ldots \ \ a_{nj} \,]^T$ is the jth column of $\boldsymbol{A}$. With this interpretation $\boldsymbol{y}$ can be evaluated by a series of SAXPYs:

$$\begin{array}{l} \boldsymbol{y} := \boldsymbol{0} \\ \texttt{for } j = 1, \ldots, n \\ \quad \boldsymbol{y} := x_j\boldsymbol{A}_{:,j} + \boldsymbol{y} \ . \end{array} \tag{1.5c}$$

1.2. Solution of a linear system of equations

We only discuss *direct methods*[1] for solving the linear system of equations (1.2). All direct methods are variants of Gaussian elimination, and have two major steps,

1° Transform $\boldsymbol{A}\mathbf{x} = \boldsymbol{b}$ to $\boldsymbol{U}\boldsymbol{z} = \boldsymbol{w}$, where $\boldsymbol{z}$ is a permutation of $\mathbf{x}$ and $\boldsymbol{U}$ is an upper triangular matrix[2].

2° Use back substitution to solve $\boldsymbol{U}\boldsymbol{z} = \boldsymbol{w}$ and permute $\boldsymbol{z}$ to $\mathbf{x}$.

In Gaussian elimination with pivoting the transformation from $\boldsymbol{A}$ to $\boldsymbol{U}$ is expressed in the following algorithm

$$\begin{array}{lr} \texttt{for } k = 1, \ldots, n & \\ \quad \text{Find pivot } a_{p_k,q_k} \qquad \{\text{See page 7}\} & (1.6\text{a}) \\ \quad \texttt{if } |a_{p_k,q_k}| \text{ is too small } \texttt{then } \text{STOP} & (1.6\text{b}) \\ \quad \text{Interchange cols. } \boldsymbol{A}_{:,k} \text{ and } \boldsymbol{A}_{:,q_k} & (1.6\text{c}) \\ \quad \text{Interchange rows } \boldsymbol{A}_{k,:} \text{ and } \boldsymbol{A}_{p_k,:} & (1.6\text{d}) \\ \quad \texttt{for } i = k{+}1, \ldots, n & \\ \quad\quad \ell_{ik} := a_{ik}/a_{kk} & (1.6\text{e}) \\ \quad\quad \boldsymbol{A}_{i,:} := \boldsymbol{A}_{i,:} - \ell_{ik} * \boldsymbol{A}_{k,:} & (1.6\text{f}) \\ \quad \texttt{end} & \\ \texttt{end} & \end{array}$$

1) The alternative is *iterative methods;* see e.g. Barker (1992).

2) i.e. $u_{ij} = 0$ for all $j < i$.

The result of this algorithm is that $\boldsymbol{A}$ contains $\boldsymbol{U}$ (all $a_{ik}=0$ for $i>k$). Note, that the row operation (1.6f) is a SAXPY.

We say that the algorithm gives a *factorization* of the matrix,

$$\boldsymbol{PAQ} = \boldsymbol{LU} \ , \tag{1.7}$$

where $\boldsymbol{P}$ and $\boldsymbol{Q}$ are *permutaion matrices* representing the row and column interchanges, respectively, and $\boldsymbol{L}$ is a *unit lower triangular matrix,* i.e.

$$(\boldsymbol{L})_{kk} = 1, \quad (\boldsymbol{L})_{ik} = \ell_{ik} \quad \text{for } i>k \ . \tag{1.8}$$

As indicated, $\boldsymbol{U}$ successively overwrites the upper triangle of $\boldsymbol{A}$. The subdiagonal elements of $\boldsymbol{L}$ can also be stored in $\boldsymbol{A}$: In (1.6f) the first $k-1$ elements in both $\boldsymbol{A}_{k,:}$ and the right-hand side verison of $\boldsymbol{A}_{i,:}$ have been made zero, and we can replace (1.6e–f) by

$$\begin{aligned} &a_{ik} := a_{ik}/a_{kk} \\ &\boldsymbol{A}_{i,k+1:n} := \boldsymbol{A}_{i,k+1:n} - a_{ik} * \boldsymbol{A}_{i,k+1:n} \ . \end{aligned} \tag{1.9}$$

The permutation matrices can be represented by vectors $\boldsymbol{p},\boldsymbol{q} \in \mathbb{Z}^n$ with the kth elements p_k, q_k as given by (1.6a).

It is useful to think in terms of the following splitting of $\boldsymbol{A}$ during the elimination process,

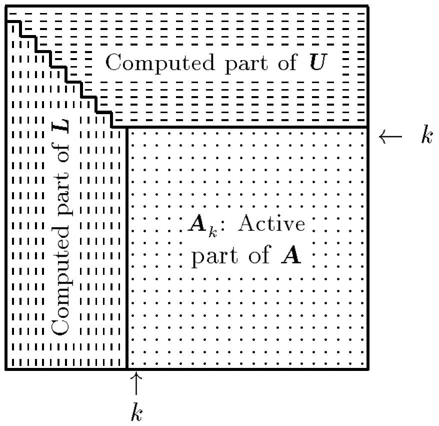


Figure 1.1

Once the factorization (1.7) has been found, it is easy to solve (1.2): By inserting the factorization, we see that the problem is equivalent with

$$(\boldsymbol{PAQ})\boldsymbol{Q}^{-1}\mathbf{x} = \boldsymbol{LU}(\boldsymbol{Q}^{-1}\mathbf{x}) = \boldsymbol{Pb} \ ,$$

which is solved in three steps,

$$1^\circ \quad \text{Solve} \quad \boldsymbol{Lw} = \boldsymbol{Pb} \ , \tag{1.10a}$$

$$2^\circ \quad \text{Solve} \quad \boldsymbol{Uz} = \boldsymbol{w} \ , \tag{1.10b}$$

$$3^\circ \quad \mathbf{x} = \boldsymbol{Qz} \ . \tag{1.10c}$$

In (1.10a–b) the matrices are triangular, and the solution is performed simply by forward and back substitution, respectively. The computation in (1.10c) is just an interchange of elements in the vector $\boldsymbol{z}$.

The solution of (1.10b) can be implemented as

$$\begin{aligned} &\texttt{for } i = n, \ldots, 2, 1 \\ &\quad z_i := (w_i - <\boldsymbol{U}_{i,i+1:n}, z_{i+1:n}>)/u_{ii} \ , \end{aligned} \tag{1.11a}$$

i.e. by means of DOT operations. Alternatively, we can exploit the interpretation (1.5b) of the matrix–vector product $\boldsymbol{Uz}$. The only contribution to the last element is $u_{nn}z_n = w_n$, and it is easily seen that the solution can be found by a series of SAXPY operations:

$$\begin{aligned} &\mathbf{z} := \mathbf{w}; \quad z_n := w_n/u_{nn} \\ &\texttt{for } i = n-1, \ldots, 2, 1 \\ &\quad \mathbf{z}_{1:i} := \mathbf{z}_{1:i} - z_{i+1}\boldsymbol{U}_{1:i,i+1} \\ &\quad z_i := z_i/u_{ii} \\ &\texttt{end} \end{aligned} \tag{1.11b}$$

Similarly, we can choose between a DOT or a SAXPY version for solving the lower triangular system (1.10a). Here the loops are reversed, and the divisors $\ell_{ii} = 1$.

For a *dense matrix*[3)] the solution of each of the systems (1.10a–b) is an $O(n^2)$ proces, and (1.10c) is an $O(n)$ proces. The computationally dominant part is the factorization. More specifically, the accumulated cost of (1.6f), which is $O(n^3)$.

For the (i,j)th element (with $i,j>k$) the transformation (1.6f) is equivalent with

[3)] In contrast to a sparse matrix.

$$a_{ij} := a_{ij} - \ell_{ik} a_{kj} = a_{ij} - \frac{a_{ik} a_{kj}}{a_{kk}} \; . \tag{1.12}$$

For dense matrices pivoting is used to enhance *numerical stability*; i.e. to reduce the effect of rounding errors. With *partial pivoting* the kth pivot is found as the numerically largest element of the active part (cf. figure 1.1) of the kth column:

$$q_k = k, \quad p_k = \underset{i=k,\ldots,n}{\operatorname{argmax}} \left\{|a_{ik}|\right\} \; . \tag{1.13a}$$

From (1.6) it follows, that with this choice $|\ell_{ik}| \leq 1$, and (1.6f) shows that this reduces the risk of the "old" ith equation "drowning" in the contribution from the kth equation.

Similarly, the pivot can be determined by

$$p_k = k, \quad q_k = \underset{j=k,\ldots,n}{\operatorname{argmax}} \left\{|a_{kj}|\right\} \; . \tag{1.13b}$$

After the column interchange (1.6c) we have $|a_{kj}/a_{kk}| \leq 1$, and (1.12) shows that this reduces the risk of the "old" jth column "drowns" in the contribution from the kth column.

Finally, with *full pivoting* the pivotal element is chosen as the numerically largest element in the active part of the matrix. In this case the determination of $\left\{(p_k, q_k)\right\}_{k=1}^{n}$ is an $O(n^3)$ process. Experience shows, however, that this strategy rarely gives more accurate results than (1.13a) or (1.13b), both of which accumulate to an $O(n^2)$ process.

When $\boldsymbol{A}$ is sparse, many of the elements $a_{ik} = 0$, and therefore $\ell_{ik} = 0$. For such rows (1.6f) can be skipped, indicating the possibility of a large saving in the number of arithmetic operations as compared with the $O(n^3)$ for a dense matrix. From (1.12), however, it follows that if $a_{ij}^{(\mathrm{old})} = 0$ and both $a_{ik} \neq 0$ and $a_{kj} \neq 0$, then $a_{ij}^{(\mathrm{new})} \neq 0$. Thus, a new *nonzero*[4] – a so-called *fill-in* – is created. Pivoting may help to reduce the number of fill-ins, as demonstrated in the first example.

Example 1.1. Below we show the structure of a "sparse" matrix of order $n = 6$. The position of the original nonzeros is indicated by $\star$ and a fill-ins are shown with $\diamond$.

[4] Short for nonzero element.

$$\boldsymbol{A} = \begin{bmatrix} \star & & & \star & & \\ \star & \star & & & \star & \\ & \star & \star & & \star & \\ \star & & \star & & & \star \\ & & \star & & \star & \\ & \star & & & & \end{bmatrix} \sim \begin{bmatrix} \star & & & \star & & \\ & \star & & \diamond & \star & \\ & \star & \star & & \star & \\ & & \star & \diamond & & \star \\ & & \star & & \star & \\ & \star & & & & \end{bmatrix} \sim \begin{bmatrix} \star & & & \star & & \\ & \star & & \diamond & \star & \\ & & \star & \diamond & \star & \\ & & \star & \diamond & & \star \\ & & \star & & \star & \\ & & & \diamond & \diamond & \end{bmatrix} \sim$$

$$\begin{bmatrix} \star & & & \star & & \\ & \star & & \diamond & \star & \\ & & \star & \diamond & \star & \\ & & & \diamond & \diamond & \star \\ & & & \diamond & \star & \\ & & & \diamond & \diamond & \end{bmatrix} \sim \begin{bmatrix} \star & & & \star & & \\ & \star & & \diamond & \star & \\ & & \star & \diamond & \star & \\ & & & \diamond & \diamond & \star \\ & & & & \star & \diamond \\ & & & & \diamond & \diamond \end{bmatrix} \sim \begin{bmatrix} \star & & & \star & & \\ & \star & & \diamond & \star & \\ & & \star & \diamond & \star & \\ & & & \diamond & \diamond & \star \\ & & & & \star & \diamond \\ & & & & & \diamond \end{bmatrix} = \boldsymbol{U} \; .$$

The structure of the corresponding $\boldsymbol{L}$ in (1.7) is

$$\boldsymbol{L} = \begin{bmatrix} 1 & & & & & \\ \star & 1 & & & & \\ & \star & 1 & & & \\ \star & & \star & 1 & & \\ & & \star & \diamond & 1 & \\ & \star & & \diamond & \diamond & 1 \end{bmatrix} .$$

Apart from the ones on the diagonal of $\boldsymbol{L}$, the total number of nonzeros in $\boldsymbol{L}$ and $\boldsymbol{U}$ is 23, while $\boldsymbol{A}$ has 14 nonzeros.

Now, we perform the elimination with the pivoting sequence given by

k	1	2	3	4	5	6
p_k	1	6	4	6	5	6
q_k	4	2	6	4	5	6

The corresponding permutation matrices (omitting the zeros) are

$$\boldsymbol{P} = \begin{bmatrix} 1 & & & & & \\ & & & & & 1 \\ & & & 1 & & \\ & 1 & & & & \\ & & & & 1 & \\ & & 1 & & & \end{bmatrix}, \quad \boldsymbol{Q} = \begin{bmatrix} & & & 1 & & \\ & 1 & & & & \\ & & & & & 1 \\ 1 & & & & & \\ & & & & 1 & \\ & & 1 & & & \end{bmatrix} .$$

and we get no fill-in:

$$\boldsymbol{PAQ} = \begin{bmatrix} \star & & & \star & & \\ & \star & & & & \\ & & \star & \star & & \star \\ & \star & & \star & \star & \\ & & & & \star & \star \\ & \star & & & \star & \star \end{bmatrix} = \begin{bmatrix} 1 & & & & & \\ & 1 & & & & \\ & & 1 & & & \\ & \star & & 1 & & \\ & & & & 1 & \\ & \star & & & \star & 1 \end{bmatrix} \begin{bmatrix} \star & & & \star & & \\ & \star & & & & \\ & & \star & \star & & \star \\ & & & \star & \star & \\ & & & & \star & \star \\ & & & & & \star \end{bmatrix} .$$

2. Storage Schemes

In a computer program a dense matrix $\boldsymbol{A}$ is normally represented by (stored in) an $n \times n$ array A with the element a_{ij} found as $\mathsf{A}(i,j)$. For a sparse matrix a large amount of storage is wasted if it is stored in an $n \times n$ array. We shall discuss a number of ways to store the nonzeros only.

2.1. Coordinate Representation

This is the simplest scheme. A real array Elm is used to store the element values, and in parallel two integer arrays Rnr and Cnr hold the row and column numbers,[5] respectively.

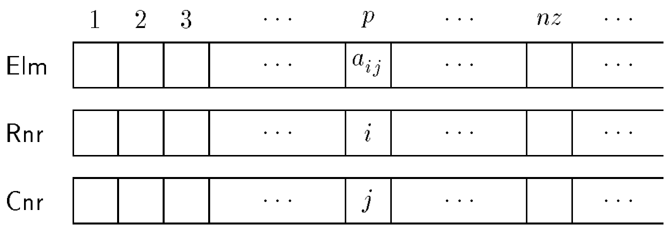


Figure 2.1. *Coordinate representation*

As indicated, these arrays must have length at least nz, the number of nonzeros in the matrix.

Example 2.1. Consider the matrix with $n=6$, $nz=14$,

$$\boldsymbol{A} = \begin{bmatrix} 1.1 & & & 1.4 & & \\ 2.1 & 2.2 & & & 2.5 & \\ & 3.2 & 3.3 & & 3.5 & \\ 4.1 & & 4.3 & & & 4.6 \\ & & 5.3 & & 5.5 & \\ & 6.2 & & & & \end{bmatrix} .$$

[5] The indices can be thought of as "coordinates" of a_{ij} in the matrix.

A possible coordinate representation is

p	1	2	3	4	5	6	7	8	9	10	11	12	13	14	⋯
Elm	1.1	2.1	2.2	4.1	3.2	1.4	3.3	4.3	2.5	6.2	5.3	3.5	5.5	4.6	
Rnr	1	2	2	4	3	1	3	4	2	6	5	3	5	4	
Cnr	1	1	2	1	2	4	3	3	5	2	3	5	5	6	

The sparse matrix representation shall enable the desired task without too much overhead for searching etc. The coordinate representation is well suited for the first task mentioned in chapter 1, $\boldsymbol{y} = \boldsymbol{A}\mathbf{x}$.

Let $\mathbf{x}$ be stored in an array x with $\mathsf{x}(i) = x_i$. A similar representation of $\boldsymbol{y}$ is found by

```
for i = 1,...,n
   y(i) := 0
for p = 1,...,nz
   y(Rnr(p)) := y(Rnr(p)) + Elm(p) * x(Cnr(p))
```
(2.1)

In this chapter we also consider the implementation of the SAXPY operation with two rows of the matrix,

$$\boldsymbol{w} = c\boldsymbol{A}_{k,:} + \boldsymbol{A}_{i,:} \ . \tag{2.2}$$

Here, the coordinate representation is less suitable, but the operation can be implemented as

```
for i = 1,...,n
   w(i) := 0
for p = 1,...,nz
   if Rnr(p) = i then
      w(Cnr(p)) := w(Cnr(p)) + Elm(p)
   else if Rnr(p) = k then
      w(Cnr(p)) := w(Cnr(p)) + c * Elm(p)
end
```
(2.3)

Note, that we only use the elements from the ith and kth row, but have search all the nonzeros. In connection with the solution of $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ the row operation (1.6f) is performed many times, and coordinate representation implies an unacceptably large overhead from these searches.

2.2. Row Ordered Storage

In connection with a SAXPY operation it is better to use a row ordered storage of $\boldsymbol{A}$: The nonzeros from the kth row are stored in consecutive slots of Elm, and their column numbers in parallel in Cnr. We do not need Rnr, but two integer arrays Rst and Rlg, so that the elements from the kth row are found at the addresses $\mathsf{Rst}(k)+s$ for $s=0,1,\ldots,\mathsf{Rlg}(k)$. Thus, the number of nonzeros in the kth row is $\{Rlg(k)+1$.

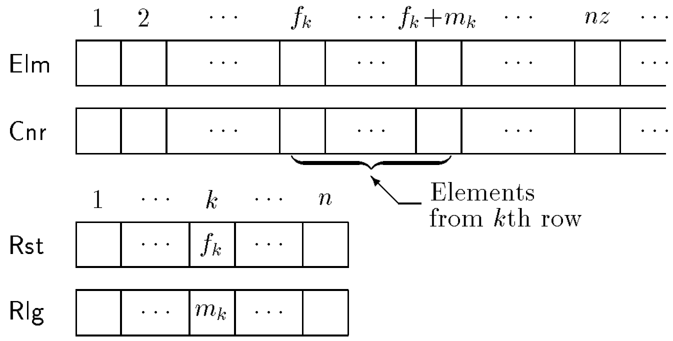


Figure 2.2. *Row ordered storage*

Example 2.2. The matrix from example 2.1 can be stored in the following arrays. Note, that within a row the elements need not be ordered by columns.

p	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Elm	1.4	1.1	6.2	4.3	4.1	4.6	2.1	2.5	2.2	3.2	3.3	3.5	5.5	5.3
Cnr	4	1	2	3	1	6	1	5	2	2	3	5	5	3
Rst	1	7	10	4	13	3								
Rlg	1	2	2	2	1	0								

With this representation the product $\boldsymbol{y}=\boldsymbol{A}\boldsymbol{x}$ can be implemented as

```
for i = 1,...,n
  y(i) := 0
  for p = Rst(i),...,Rst(i)+Rlg(i)                    (2.4)
    y(i) := y(i) + Elm(p) * x(Cnr(p))
end
```

The SAXPY operation (2.2) can be implemented as


```
for i = 1,...,n
  w(i) := 0
for p = Rst(i),...,Rst(i)+Rlg(i)
  w(Cnr(p)) := Elm(p)                                 (2.5)
for p = Rst(k),...,Rst(k)+Rlg(k)
  w(Cnr(p)) := w(Cnr(p)) + c * Elm(p)
```

In chapter 3 we discuss further aspects of the use of row ordered storage in connection with Gaussian elimination. This includes the treatment of fill-ins: $\boldsymbol{w}$ given by (2.2) with $c=-\ell_{ik}$ shall replace the current $\boldsymbol{A}_{i,:}$ but has more nonzeros. Inserting a new nonzero is simple when we use the coordinate representation, but not straightforward with the row ordered storage – see the discussion in connection with figure 3.1 and the next section.

Example 2.3. By keeping a *column ordered list* of row numbers we can facilitate the search for other rows with elements in the pivotal column. For the matrix of examples 2.1–2 we could use

p	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Rnr	2	1	4	2	3	5	4	6	2	3	3	4	5	1
Cst	1	8	11	14	4	7								
Clg	2	2	2	0	2	0								

2.3. Linked List

This representation enables an easy insertion of new nonzeros without the large search overhead of the coordinate representation. We shall discuss this scheme in connection with row ordered storage.

Each nonzero is stored in a *node* containing the value of a_{ij}, the value of j, and a *pointer* to the next node with a nonzero from the ith row. In figure 2.3 we illustrate this, including an n-array Rst of pointers to the first elements in each row. The "nil pointer" •—→|| points from the last element in the row.

In some programming languages (e.g. Pascal, C, Fortran90) a node can be realized as a *record*. In that case this scheme can make fully dynamic use of the computer's storage.

Instead of giving a record–implementation in a specific language we

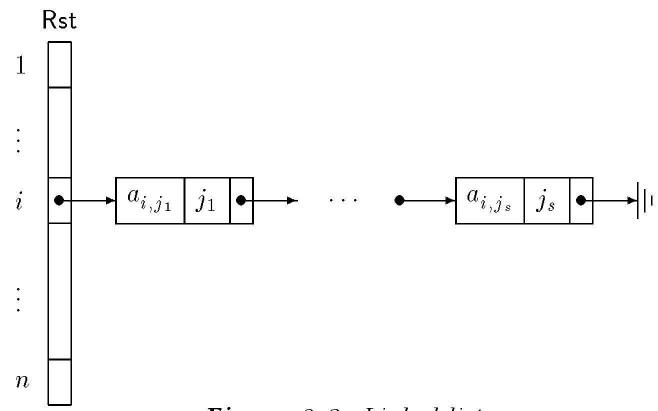


Figure 2.3. *Linked list*

shall discuss an array–implementation, where fixed length arrays are used in a dynamic fashion.

As in section 2.2 Elm(p) and Cnr(p) hold the value of a nonzero and its column number. Ptr(p) is the address of the next element in the row, and Rst(k) is the address of the first node for row k.

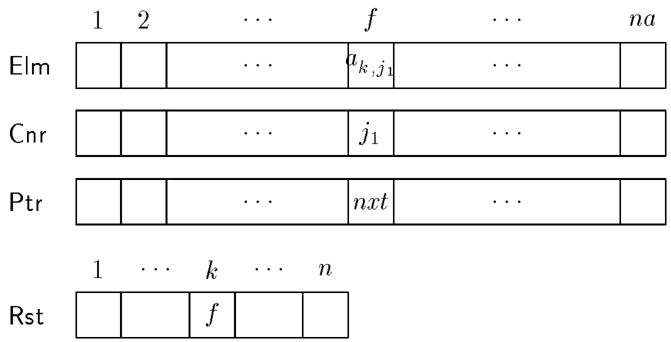


Figure 2.4. *Arrays for linked list*

The length of the arrays Elm, Cnr and Ptr is na with $na \geq nz$. Normally, $na > nz$, and the vacant positions (where there is room to store fill-ins) can be linked via Ptr to a *free list* with a pointer Vac to the first vacant slot. The "nil pointer" must be a non-existing address. Letting the pointer of the last element in the ith row have the value $na+i$, provides a simple means for getting the row number of any matrix element: Just follow the links to the end of the row. The last element in the free list can be marked by the pointer value 0 (zero).

Example 2.4. The matrix from example 2.1 can be represented by the following linked list, where a free slot is marked by a $\star$. We use $na = 17$ and Vac $= 5$.

p	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Elm	$\star$	1.1	2.1	2.2	$\star$	4.1	3.2	1.4	3.3	4.6	2.5	6.2	5.3	3.5	5.5	4.3	$\star$
Cnr	$\star$	1	1	2	$\star$	1	2	4	3	6	5	2	3	5	5	3	$\star$
Ptr	0	8	4	11	17	16	9	18	14	21	19	23	15	20	22	10	1
Rst	2	3	7	6	13	12											

With this scheme the product $\boldsymbol{y} = \boldsymbol{A}\mathbf{x}$ can be implemented as

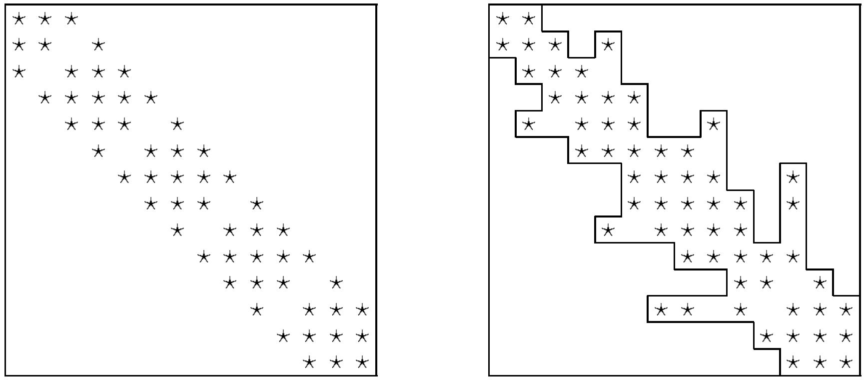
```
for i = 1,...,n
   y(i) := 0;   p := Rst(i)
   while p ≤ na
      y(i) := y(i) + Elm(p) * x(Cnr(p))                    (2.6)
      p := Ptr(p)
   end
end
```

The other standard operation (2.2) is implemented as (2.5) with modifications similar to the changes between (2.4) and (2.6). Rather than giving a code for this, we consider the insertion of a new nonzero a in position (i, j) of the matrix. We assume that before the insertion there is at least one element in the ith row and no element in position (i, j).

```
new := Vac;   Vac := Ptr(Vac)           {Update free list}
Elm(new) := a;   Cnr(new) := j
p := Rst(i)
if j < Cnr(p) then Rst(i) := new
else
   while j > Cnr(p)                                        (2.7)
      po := p;   p := Ptr(p)
   end
   Ptr(po) := new
end
Ptr(new) := p
```

2.4. Band Matrices

This is a special type of sparse matrices, characterized by having the nonzeros lumped close to the diagonal. Figure 2.5 shows two matrices with symmetric sparsity pattern. We say that the matrix in figure 2.5a has *fixed bandwidth*, while figure 2.5b shows a matrix with *variable bandwidth*. In both cases there may be zeros within the band. Such zeros are stored explicitly.



a) Fixed bandwidth *b) Variable bandwidth*

Figure 2.5. *Band matrices*

We only discuss symmetric band matrices, although symmetry is not a necessary condition for exploiting a band structure.

A symmetric matrix with *fixed bandwidth* $2m{+}1$ ($m=2$ in figure 2.5a) is characterized by

$$a_{ij} = a_{ji} = 0 \quad \text{for } j > i{+}m \ . \tag{2.8}$$

such a matrix can be stored in an $n\times(m{+}1)$ array Elm so that

$$a_{i,i+p} = a_{i+p,i} = \mathsf{Elm}(i,p{+}1),\ \ p=0,1,\ldots,\min\{m,n{-}i\}\ . \tag{2.9}$$

This gives a very simple addressing without extra arrays like Cnr and Rst (but we have to reserve $\frac{1}{2}m(m{+}1)$ superfluous array locations).

Example 2.5. Consider the matrix (with $n=7$)

$$\boldsymbol{A} = \begin{bmatrix} 11 & 1 & & & & & \\ 1 & 12 & 2 & 3 & & & \\ & 2 & 13 & & 5 & & \\ & 3 & & 14 & & & \\ & & 5 & & 15 & 6 & 7 \\ & & & & 6 & 16 & 8 \\ & & & & 7 & 8 & 17 \end{bmatrix} .$$

It can be represented as a symmetric band matrix with fixed bandwidth $m=2$. Elements marked with $*$ are "wasted".

$$\mathsf{Elm} = \begin{bmatrix} 11 & 1 & 0 \\ 12 & 2 & 3 \\ 13 & 0 & 5 \\ 14 & 0 & 0 \\ 15 & 6 & 7 \\ 16 & 8 & * \\ 17 & * & * \end{bmatrix} .$$

With this representation the operation $\boldsymbol{y} = \boldsymbol{A}\mathbf{x}$ can be implemented as shown in (2.10) below. We assume $n \geq m$. Note, that (2.10) uses both the DOT version (1.5a) and the SAXPY version (1.5c).

$$\begin{array}{l} \mathsf{y} := \boldsymbol{0} \\ \texttt{for}\ i=1,\ldots,n \\ \quad q := \min\{m, n{-}i\} \\ \quad \mathsf{y}(i) := \mathsf{y}(i) + <\mathsf{Elm}(i,1\!:\!q{+}1), \mathsf{x}(i\!:\!i{+}q)> \\ \quad \mathsf{y}(i{+}1\!:\!i{+}q) := \mathsf{y}(i{+}1\!:\!i{+}q) + \mathsf{x}(i) * \mathsf{Elm}(i,2\!:\!q{+}1)^T \\ \texttt{end} \end{array} \tag{2.10}$$

A symmetric matrix with *variable bandwidth* is normally stored in a one dimensional array Elm. In chapter 4 we shall see that it is advantageous to store the diagonal elements in the first n locations and then the elements in the strictly lower part of $\boldsymbol{A}$, taken rowwise. We also need an integer array `Rnd` with n elements so that the subdiagonal elements of the kth row are found as $\mathsf{Elm}(\mathsf{Rnd}(k{-}1){+}1),\ldots,\mathsf{Elm}(\mathsf{Rnd}(k))$. Thus, $\mathsf{Rnd}(1)=n$ and $\mathsf{Rnd}(n)=nz$, the number of nonzeros within the band in the lower triangle of $\boldsymbol{A}$.[6)]

[6)] This number is often called the *profile* of $\boldsymbol{A}$.

The first nonzero in the kth row is a_{k,j_1} with $j_1 = k + \mathsf{Rnd}(k{-}1) - \mathsf{Rnd}(k)$.

	1	$\cdots$	n	$\cdots$	$p{+}1$	$\cdots$	q	$\cdots$	nz
Elm	$a_{1,1}$	$\cdots$	$a_{n,n}$	$\cdots$	a_{k,j_1}	$\cdots$	$a_{k,k-1}$	$\cdots$	

	1	$\cdots$	$k{-}1$	k	$\cdots$	n
Rnd	n	$\cdots$	p	q	$\cdots$	nz

Figure 2.6. *Arrays for variable bandwidth*

Example 2.6. The matrix of example 2.5 can be represented by

p	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Elm	11	12	13	14	15	16	17	1	2	3	0	5	0	6	7	8
Rnd	7	8	9	11	13	14	16									

The operation $\boldsymbol{y} = \boldsymbol{Ax}$ can be implemented as shown below. Again we use both (1.5a) and (1.5c).

```
y(1) := Elm(1) * x(1);    p2 := Rnd(1)
for i = 2, ..., n
   p1 := p2+1;    p2 := Rnd(i);    q := p2−p1
   y(i) := Elm(i) * x(i) + <Elm(p1:p2), x(i−q:i−1)>                (2.11)
   y(i−q:i−1) := y(i−q:i−1) + x(i) * Elm(p1:p2)
end
```

3. Solution of General Sparse Systems

In this chapter we give an introduction to direct solution of the linear system of equations

$$\boldsymbol{A}\mathbf{x} = \boldsymbol{b} \ , \tag{3.1}$$

where $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ is general sparse,[7] while $\mathbf{x}, \boldsymbol{b} \in \mathbb{R}^n$ are considered as dense vectors.

As discussed in chapter 1, the solution can be found in two steps

1° Determine permutation matrices $\boldsymbol{P}$ and $\boldsymbol{Q}$, a unit lower triangular matrix $\boldsymbol{L}$ and an upper triangular matrix $\boldsymbol{U}$ so that

$$\boldsymbol{PAQ} = \boldsymbol{LU} \ . \tag{3.2}$$

2° Solve (3.1) exploiting (3.2); i.e. via (1.10a–c).

In connection with sparse matrices it is common practice to denote these two steps "Analyse" and "Solve", respectively. In some connections one has to solve a series of problems

$$\boldsymbol{A}^{(k)}\mathbf{x}^{(k)} = \boldsymbol{b}^{(k)}, \quad k = 0, 1, \ldots \ , \tag{3.3}$$

where all $\boldsymbol{A}^{(k)}$ have the same sparsity pattern but different values of the nonzero elements. Then the knowledge of $\boldsymbol{P}$ and $\boldsymbol{Q}$ (the row and column permutations) and of the position of nonzeros in $\boldsymbol{L}$ and $\boldsymbol{U}$ can be exploited in the socalled "Factorize". For large problems with $nz \ll n^2$ a rule of thumb says that $time_{\text{Analyse}} : time_{\text{Factorize}} : time_{\text{Solve}} \simeq 100 : 10 : 1$.

Example 3.1. Consider the nonlinear system of equations

$$\boldsymbol{f}(\mathbf{x}) = \boldsymbol{0} \ ,$$

where $\boldsymbol{f} : \mathbb{R}^n \mapsto \mathbb{R}^n$. This problem may be solved by the method of Newton–Raphson: From an initial guess $\mathbf{x}_0$ a series $\mathbf{x}_1, \ \mathbf{x}_2, \ldots$ is computed by

[7] By this we mean a sparse matrix, which is neither symmetric nor banded.

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \boldsymbol{h}_k, \quad \text{where} \quad \boldsymbol{J}(\mathbf{x}_k)\boldsymbol{h}_k = -\boldsymbol{f}(\mathbf{x}_k) \ .$$

Here, $\boldsymbol{J}$, is the *Jacobian:* $(\boldsymbol{J})_{ij} = \partial f_i / \partial x_j$. For problems with large n it is often the case that each function f_i only depends on relatively few of the elements of $\mathbf{x}$; i.e. $\boldsymbol{J}$ is a sparse matrix. The elements in $\boldsymbol{J}(\mathbf{x}_k)$ depend on $\mathbf{x}_k$, but the sparsity pattern is fixed.

In sections 3.1 – 3.3 we look at some important aspects of these subproblems. We refer to Østergaard & Zlatev (1983) or Duff et al. (1986) for a more elaborate discussion.

3.1. Analyse

In chapter 1 it was mentioned that *pivoting* is used both to obtain numerical stability and to reduce the number of fill-ins. With the aim of getting a better preservation of sparsity it may be appropriate to slacken the stability criterion (1.13) to

$$|a_{p,q}| \geq v \cdot \max_{i=k,\ldots,n} \{|a_{i,q}|\} \tag{3.4a}$$

or

$$|a_{p,q}| \geq v \cdot \max_{j=k,\ldots,n} \{|a_{p,j}|\} \ , \tag{3.4b}$$

where the *stability parameter* v is chosen in the range $0 < v \leq 1$. The choice of storage scheme decides whether (3.4a) or (3.4b) is used – se the end of this section.

Any nonzero in the active part of $\boldsymbol{A}$ that satisfies (3.4) is a candidate for the post as the kth pivot. If $v = 1$, then (3.4) is identical with (1.13). For $v < 1$ there is a possibility of increasing the number of candidates, i.e. to enhance the sparsity criterion. Based on a large number of experiments (see example 3.4 below) it is generally recommended to use $v = 0.1$.

The ultimate goal is to find $\boldsymbol{P}$ and $\boldsymbol{Q}$ – i.e. the series $\{(p_k, q_k)\}_{k=1}^n$, cf. (1.6) – so that the accumulated number of nonzeros in $\boldsymbol{L}$ and $\boldsymbol{U}$ is minimized. In that case the storage, the work involved in the transformations

$$(1.12) : a_{ij} := a_{ij} - \ell_{ik} a_{kj} = a_{ij} - \frac{a_{ik} a_{kj}}{a_{kk}} \ ,$$

and the work involved in Solve will all be as small as possible. The complexity of this problem is so big, however, that it must be considered as prohibitive. Therefore, a number of strategies have been proposed, each of which aim at getting an approximation to the ultimate goal via a series of (approximate) minimizations of fill-in at the kth step of the elimination.

The most popular strategy is due to Markowitz (1957). Instead of the difficult problem of minimizing the number of fill-ins at the kth stage, Markowitz proposed to minimize the work involved; i.e. the number of elements to be changed by (1.12) at the kth step. Intuitively, if this work is minimal, then also the number of fill-ins is small. This is confirmed by computational experience.

Let $r_i^{(k)}$ and $s_j^{(k)}$ denote the number of nonzeros in the active part of the ith row and jth column, respectively; cf. figure 1.1. If the nonzero a_{pq} in $\boldsymbol{A}_k$ is chosen as the kth pivot, then we have to compute $(s_q^{(k)}-1)$ elimination coefficients ℓ_{ik}, (1.6e), and for each of these (1.12) is used for $(r_p^{(k)}-1)$ different values of j. Taking into account the numerical stability, we can now formulate the *Markowitz pivoting strategy:*

$$(p_k, q_k) = \underset{p,q\,\in\,\mathrm{M}_k}{\operatorname{argmin}} \left\{(r_p^{(k)}-1)(s_q^{(k)}-1)\right\}\ , \tag{3.5a}$$

where

$$\mathrm{M}_k = \left\{p, q \mid k \le p, q \le n \text{ and } a_{pq} \text{ satisfies } (3.4)\right\}\ . \tag{3.5b}$$

Often, the choice of pivot is not unique: More than one nonzero in $\boldsymbol{A}_k$ satisfy the stability criterion and give the same minimal value of the "Markowitz cost" $(r_p^{(k)}-1)(s_q^{(k)}-1)$. Different strategies have been proposed to pick (p_k, q_k) in case of such a tie; see chapter 7 in Duff et al. (1986) and example 3.3 below.

Example 3.2. If $\boldsymbol{A}_k$ contains a row or a column with only one nonzero (a "singleton"), then this can be chosen as pivot: It satisfies (3.4) for any $v \le 1$ and the Markowitz cost is zero. This agrees with the fact, that (1.12) is not activated, so no element value is changed and there is no fill-in. Therefore, if $\boldsymbol{A}_k$ contains more than one singleton, it does not matter in which order they are chosen as pivot.

Example 3.3. For the matrix of example 2.1 we find

i	1	2	3	4	5	6
$r_i^{(1)}$	2	3	3	3	2	1
$s_i^{(1)}$	3	3	3	1	3	1

There are three singletons: in positions (6,2), (1,4) and (4,6). By choosing $(p_1, q_1) = (1,4)$ we only need to interchange columns 1 and 4, and get

$$\boldsymbol{A}_2 = \begin{bmatrix} 2.2 & & 2.1 & 2.5 & \\ 3.2 & 3.3 & & 3.5 & \\ & 4.3 & 4.1 & & 4.6 \\ & 5.3 & & 5.5 & \\ 6.2 & & & & \end{bmatrix},$$

i	2	3	4	5	6
$r_i^{(2)}$	3	3	3	2	1
$s_i^{(2)}$	3	3	2	3	1

Next, we choose the singleton at $(p_2, q_2) = (6,2)$ and a row interchange gives $\boldsymbol{A}_3$ with the singleton 4.6 in the position corresponding to $(p_3, q_3) = (4,6)$. A row and a column interchange results in

$$\boldsymbol{A}_4 = \begin{bmatrix} & 3.5 & 3.3 \\ & 5.5 & 5.3 \\ 2.1 & 2.5 & \end{bmatrix},$$

i	4	5	6
$r_i^{(4)}$	2	2	2
$s_i^{(4)}$	1	3	2

We choose $(p_4, q_4) = (6,4)$ and get $\boldsymbol{A}_5 = \begin{bmatrix} 5.5 & 5.3 \\ 3.5 & 3.3 \end{bmatrix}$. Finally, the transformation to upper triangular form is obtained by $(p_5, q_5) = (5,5)$, one application of (1.12), and $(p_6, q_6) = (6,6)$. (It is always the case that $(p_n, q_n) = (n, n)$).

This is the matrix and pivoting sequence treated in example 1.1.

Example 3.4. We consider the choice of the stability parameter v in (3.4). Below we give some results from Table 7.8.1 in Duff et al. (1986), where the matrix has size $n = 541$ and $nz = 4285$. The accumulated number of nonzeros in $\boldsymbol{L}$ and $\boldsymbol{U}$ is denoted nz_{LU}, and $\mathbf{x}$ and $\widetilde{\mathbf{x}}$ are the exact and computed solutions, respectively. The machine accuracy is $\varepsilon_M = 16^{-13} \simeq 2.22\mathrm{D}{-}16$.

v	nz_{LU}/nz	$\Vert\widetilde{\mathbf{x}}-\mathbf{x}\Vert_2/\Vert bx\Vert_2$
1	3.91	2D–10
0.5	3.29	8D–11
0.25	3.33	8D–11
0.1	3.19	3D–9
1D–2	3.51	4D–6
1D–4	3.78	5D–2
1D–10	3.86	5D+15

For $v < 0.1$ the error grows when v is further reduced. This is as expected, since we give numerical stability less priority, in the hope of reducing the number of fill-ins. This hope is not fulfilled: also nz_{LU} grows. The explanation is, that in the start (k small) smaller pivots are accepted, resulting in less fill-in. However, the numbers computed by (1.12) have larger variations in their order of magnitude, so that (3.4) becomes more restrictive for larger k.

To avoid having to examine all nonzeros in $\boldsymbol{A}_k$, the data structure is supplemented with two integer n–arrays to hold the values $r_i^{(k)}$ and $s_j^{(k)}$.[8] These arrays are examined for increasing values of $r_i^{(k)}$, and the search stops when $(r_i^{(k)}-1)\cdot\min\{s_j^{(k)}-1\}$ is larger than the currently smallest Markowitz cost $(r_p^{(k)}-1)(s_q^{(k)}-1)$.

Now, we have found the pivot a_{p_k,q_k}. The active part of the p_kth row provides the kth row of $\boldsymbol{U}$, and the active part of the q_kth column provides the kth column of $\boldsymbol{L}$ after division by the pivot.

A suitable data structure for $\boldsymbol{A}$ is a row ordered linked list. Here, we can simply leave the $\boldsymbol{U}$–elements and add fill-ins. The elements of $\boldsymbol{L}$ can share the same arrays (figure 2.4), but ordered and linked columnwise, with an extra integer n–array Cst, whose kth element points to the first subdiagonal element in the kth column.

On some computers you get a speed–up if numbers that are used consecutively are also stored consecutively. After a few fill-ins this will not be satisfied by a linked list representation. On such computers it is better to use a row ordered storage scheme (see section 2.2), modified for better handling of fill-ins, and combined with a column ordered list of rownumbers to give easy access to the other nonzeros in the pivotal column.

We shall sketch some details of one way to use such a storage scheme, where $\boldsymbol{L}$ and $\boldsymbol{U}$ successively overwrite $\boldsymbol{A}$.

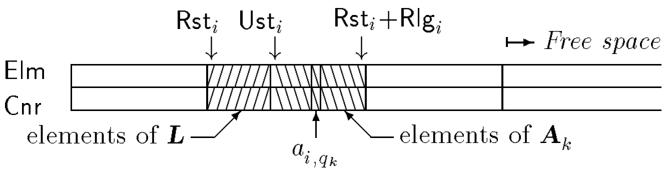


Figure 3.1. *Row ordered list*

The data structure is supplemented with an n–array w, which is initialized to zero, and an integer n–array Ust, that marks the start in each row of the elements in the active part (which eventually land in $\boldsymbol{U}$).

---

[8] If the storage scheme of figure 2.2 is used for $\boldsymbol{A}_k$, then Rlg holds the values $r_i^{(k)}-1$.

The typical row operation from (1.6),

$$\ell_{ik} := a_{ik}/a_{kk}; \quad \boldsymbol{A}_{i,:} := \boldsymbol{A}_{i,:} - \ell_{ik} * \boldsymbol{A}_{k,:}$$

is implemented as follows,

1° Copy $\boldsymbol{L}$–elements to free space and add $\ell_{ik} = a_{i,q_k}/a_{p_k,q_k}$. Update Ust_i.

2° Perform (2.2) in w with $c = -\ell_{ik}$ and with interchanges corresponding to $q_1, \ldots, q_k$.

3° Move nonzeros from w to free space, leaving zeros in w.

4° Mark positions $\mathsf{Rst}_i, \ldots, \mathsf{Rst}_i+\mathsf{Rlg}_i$ as vacant.

5° Update Rst_i and Rlg_i.

The algorithm is efficient only, if Elm and Cnr have enough "elbow room" to hold several copies of $\{\boldsymbol{A}_{i,:}\}$ before the vacant space is recovered by compressing the list – a socalled "garbage collection". This is an expensive (and non productive) operation, and should not take place too often.

3.2. Factorize

This is used when $\boldsymbol{P}$, $\boldsymbol{Q}$ and the sparsity patterns of $\boldsymbol{L}$ and $\boldsymbol{U}$ are known. Initially, the nonzeros of the new $\boldsymbol{A}$ are stored in the locations corresponding to the reordered matrix

$$\boldsymbol{A}' = \boldsymbol{PAQ} \,, \tag{3.6}$$

and zeros are stored in the locations reserved for fill-ins. Next, we have to compute the elements in

$$\boldsymbol{LU} = \boldsymbol{A}' \,. \tag{3.7}$$

With the storage scheme discussed in connection with figure 3.1, where nonzeros in the kth row of $\boldsymbol{L}$ and $\boldsymbol{U}$ are stored in consecutive locations, it is better to compute the elements of $\boldsymbol{L}$ rowwise than columnwise (as sketched in figure 1.1). This can be done by

Doolittle factorization: From (3.7) it follows that

$$a'_{kj} = \sum_{p=1}^{n} \ell_{kp} u_{pj} = \sum_{p=1}^{\min\{k,j\}} \ell_{kp} u_{pj} \,, \tag{3.8}$$

where the rephrasing follows from the triangular form of the matrices. By reordering the terms and exploiting that $\ell_{kk} = 1$ we get

$$\ell_{kj} = \big(a'_{kj} - \sum_{p=1}^{j-1} \ell_{kp} u_{pj}\big)/u_{jj} \ , \quad j = 1, \ldots, k-1 \ , \tag{3.9a}$$

$$u_{kj} = a'_{kj} - \sum_{p=1}^{j-1} \ell_{kp} u_{pj} \ , \quad j = k, \ldots, n \ . \tag{3.9b}$$

When the elements are computed in lexicographical order, all elements on the right-hand sides are known before they are used. We refer to section 9.3 in Duff et al. (1986) for details about sparse implementation.

With the factorization (3.7) we do not heed numerical stability, and there is a risk that the new values of the nonzeros in $\boldsymbol{A}$ can lead to excessive effect of rounding errors. Therefore, it is advisable to watch the values of the nonzeros, e.g. by computing the numbers

$$v_k = |u_{kk}| / \max_{j>k}\{|u_{kj}|\} \tag{3.10}$$

in connection with (3.9b). If any of these numbers is considerably smaller than the stability parameter v used for finding $\boldsymbol{P}$ and $\boldsymbol{Q}$, then the factorization is stopped, and ANALYSE is called with the new $\boldsymbol{A}$.

3.3. Solve

This part of the program implements (1.10):

$$1^\circ \quad \text{Solve} \quad \boldsymbol{Lw} = \boldsymbol{Pb} \ , \tag{3.11a}$$

$$2^\circ \quad \text{Solve} \quad \boldsymbol{Uz} = \boldsymbol{w} \ , \tag{3.11b}$$

$$3^\circ \quad \boldsymbol{x} = \boldsymbol{Qz} \ . \tag{3.11c}$$

In section 1.2 we gave two alternatives for the solution of a triangular system of equations. If $\boldsymbol{L}$ is stored rowwise, then it is best to use the DOT version. For (3.11a) it takes the form

$$\begin{aligned} &\texttt{for } i = 1, \ldots, n \\ &\quad w_i := (\boldsymbol{Pb})_i - \langle \boldsymbol{L}_{i,1:i-1}, w_{1:i-1} \rangle \ . \end{aligned} \tag{3.12a}$$

If $\boldsymbol{L}$ is stored columnwise, then it is more efficient to use the SAXPY version

$$\begin{aligned} &\boldsymbol{w} := \boldsymbol{Pb} \\ &\texttt{for } i = 2, \ldots, n \\ &\quad \mathbf{w}_{i:n} := \mathbf{w}_{i:n} - w_{i-1} \boldsymbol{L}_{i:n,i-1} \ . \end{aligned} \tag{3.12b}$$

Similarlarly, we can choose between (1.11a) and (1.11b) for solving (3.11b).

Sparse implementations of these algorithms depend on the storage schemes, and they are analogous to the implementations of the operations $\boldsymbol{y} = \boldsymbol{Ax}$ and $\boldsymbol{A}_{i,:} + c\boldsymbol{A}_{k,:}$ discussed in chapter 2.

Finally, we assume that $\boldsymbol{Q}$ is represented by $\boldsymbol{q} = [\, q_1 \ \ \ldots \ \ q_n \,]$. Then (3.11c) is implemented as

$$\begin{aligned} &\mathbf{x} := \boldsymbol{z} \\ &\texttt{for } k = n-1, \ldots, 2, 1 \\ &\quad \texttt{if } q_k > k \texttt{ then} \text{ interchange } x_k \text{ and } x_{q_k} \ . \end{aligned} \tag{3.13}$$

If $\boldsymbol{P}$ is represented by $\boldsymbol{p} = [\, p_1 \ \ \ldots \ \ p_n \,]$, then the transformation from $\boldsymbol{b}$ to $\boldsymbol{Pb}$ is done similar to (3.13) with $\boldsymbol{Pb}$ overwriting $\boldsymbol{b}$. Further, $\boldsymbol{w}$ can overwrite $\boldsymbol{Pb}$, $\boldsymbol{z}$ can overwrite $\boldsymbol{w}$, and – as shown in (3.13) – $\mathbf{x}$ overwrites $\boldsymbol{z}$. Thus, the entire process (3.11) can be carried out without extra storage.

3.4. Incomplete Factorization

This heading covers a variety of methods to reduce the number of nonzeros during the factorization. The resulting $\boldsymbol{L}$ and $\boldsymbol{U}$ satisfy[9)]

$$\boldsymbol{LU} = \boldsymbol{PAQ} + \boldsymbol{E} \ . \tag{3.14}$$

The solution found by (3.11) with these $\boldsymbol{L}$, $\boldsymbol{U}$ differs from the true solution to (3.1), but there are ways to improve it – see the end of this section.

A simple way to get an incomplete factorization is to choose the sparsity pattern of $\boldsymbol{L}$ and $\boldsymbol{U}$ before the factorization. As an example one can choose to ignore all fill-ins, so that $\boldsymbol{L}$ and $\boldsymbol{U}$ together have the same sparsity pattern as $\boldsymbol{A}$.

Another way is to "drop" small elements during ANALYSE. More specifically: Let $a_{ij}^{(k)}$ be an element in the active part of $\boldsymbol{A}$. If

9) Actually, because of rounding errors the *computed* $\boldsymbol{L}$, $\boldsymbol{U}$ do not satisfy (3.2), but a relation of the form (3.14) with $\|\boldsymbol{E}\| = O(\varepsilon_M \|\boldsymbol{A}\|)$. Normally, $\|\boldsymbol{E}\|$ increases when the stability parameter v in (3.4) decreases, cf. example 3.4.

$$|a_{ij}^{(k)}| \le \delta \cdot \max_{k \le s \le n} \{|a_{is}^{(k)}|\} , \tag{3.15}$$

then it is replaced by zero. The number δ is the socalled *"relative drop tolerance"*, chosen by the user. For further discussion, see Zlatev et al. (1978) and section 9.9 in Duff et al. (1986).

If $\boldsymbol{E}$ is sufficiently small (relative to $\boldsymbol{A}$), then the solution $\mathbf{x}$ can be recovered by

Iterative refinement: Let $\mathbf{x}_0$ denote the solution found by (3.11) with $\boldsymbol{L}$, $\boldsymbol{U}$ satisfying (3.14). Compute $\mathbf{x}_1, \mathbf{x}_2, \ldots$ by[10)]

$$\begin{aligned} \mathbf{r}_k &= \boldsymbol{b} - \boldsymbol{A}\mathbf{x}_k , \\ \mathbf{x}_{k+1} &= \mathbf{x}_k + \boldsymbol{Q}(\boldsymbol{LU})^{-1}\boldsymbol{P}\mathbf{r}_k . \end{aligned} \tag{3.16}$$

Assuming that we can neglect effects of rounding errors in the execution of (3.16) and using $\boldsymbol{b} = \boldsymbol{A}\mathbf{x}$ with $\boldsymbol{A} = \boldsymbol{P}^{-1}(\boldsymbol{LU} - \boldsymbol{E})\boldsymbol{Q}^{-1}$, we find that the errors satisfy

$$\begin{aligned} \mathbf{x}_{k+1} - \mathbf{x} &= \mathbf{x}_k - \mathbf{x} + \boldsymbol{Q}(\boldsymbol{LU})^{-1}\boldsymbol{PA}(\mathbf{x} - \mathbf{x}_k) \\ &= \boldsymbol{QFQ}^{-1}(\mathbf{x}_k - \mathbf{x}) , \end{aligned} \tag{3.17a}$$

where

$$\boldsymbol{F} = (\boldsymbol{LU})^{-1}\boldsymbol{E} . \tag{3.17b}$$

Let $\boldsymbol{F}$ have the eigenvalues $\{\lambda_j\}_{j=1}^n$. If $\max\{|\lambda_j|\} < 1$, then $\mathbf{x}_k - \mathbf{x} \to \boldsymbol{0}$ for $k \to \infty$; i.e. $\{\mathbf{x}_k\}$ converges to $\mathbf{x}$.

Incomplete factorization is often used to get a *preconditioning* of the matrix in order to get a speed-up of an iterative method. See Barker (1992) for the general theory and Zlatev & Nielsen (1988) for a discussion of the choice of drop tolerance in this connection.

[10)] The notation $\boldsymbol{z} = \boldsymbol{H}^{-1}\boldsymbol{c}$ is shorthand for "solve the system $\boldsymbol{Hz} = \boldsymbol{c}$".

4. Symmetric, Positive Definite Systems

4.1. Introduction

When the matrix $\boldsymbol{A} \in I\!R^{n\times n}$ is symmetric (i.e. $a_{ij} = a_{ji},\ i,j=1,\dots,n$), then we have full information about it by storing only the elements from the lower triangle ($j \le i$) or the upper triangle ($j \ge i$). If we perform simple[11] Gaussian elimination, then it follows from (1.12) that the symmetry is preserved in the active part of the matrix. Therefore, in the factorization

$$\boldsymbol{A} = \boldsymbol{LU}$$

we have $u_{ij} = u_{ii}\cdot\ell_{ji}$. This implies that the factorization can be written in symmetrical form

$$\boldsymbol{A} = \boldsymbol{LDL}^T \quad \text{with} \quad \boldsymbol{D} = \mathrm{diag}(d_1,\dots,d_n)\ , \tag{4.1}$$

(where $d_i = u_{ii}$).

Simple Gaussian elimination is generally not numerically stable, and pivoting will normally destroy symmetry. Therefore, it is advisable to use either a general equation solver (as described in chapter 3) or a special algorithm, which is stable and exploits the symmetry, see Duff et al. (1979).

In many applications the matrix is not only symmetric, but also *positive definite*, i.e.

$$\forall \mathbf{x} \in I\!R^n,\ \mathbf{x} \neq \boldsymbol{0}\ :\quad \mathbf{x}^T\boldsymbol{A}\mathbf{x} > 0\ . \tag{4.2}$$

For such matrices it can be shown that simple Gaussian elimination is a stable process. This implies that we can use (4.1), and thereby about halve both the storage and the computational work during factorization.

It is easy to show that

$$\{\boldsymbol{A} \text{ is positive definite}\} \ \Leftrightarrow\ \{d_i > 0,\ i=1,\dots,n\} \tag{4.3}$$

[11] i.e. without pivoting.

This can be used during the factorization of a symmetric matrix to check whether it is positive definite, see (4.6) below. This property also implies that instead of (4.1) we can write

$$\boldsymbol{A} = \widetilde{\boldsymbol{L}}\widetilde{\boldsymbol{L}}^T\ , \tag{4.4a}$$

where $\widetilde{\boldsymbol{L}} \in I\!R^{n\times n}$ is the lower triangular matrix given by

$$\widetilde{\boldsymbol{L}} = \boldsymbol{LD}^{\frac{1}{2}} \quad \text{with} \quad \boldsymbol{D}^{\frac{1}{2}} = \mathrm{diag}(\sqrt{d_1},\dots,\sqrt{d_n})\ . \tag{4.4b}$$

This is the socalled *Cholesky factorization* of $\boldsymbol{A}$. Rather than using (4.4b) it is normally computed directly from (4.4a) similar to (4.5) below. We shall not, however, discuss this further, but show how to compute the elements of $\boldsymbol{L}$ and $\boldsymbol{D}$ in (4.1). Throughout we assume that $\boldsymbol{A}$ is given by its lower triangle.

As in the derivation of (3.8), the relation (4.1) gives

$$a_{kj} = \sum_{p=1}^{j} (\boldsymbol{L})_{kp}(\boldsymbol{D})_{pp}(\boldsymbol{L}^T)_{pj} = \sum_{p=1}^{j} \ell_{kp} d_p \ell_{jp},\ j \le k\ ,$$

and (since $\ell_{kk} = 1$) this can be reordered to

$$\ell_{kj} = \left(a_{kj} - \ell_{k1}d_1\ell_{j1} - \cdots - \ell_{k,j-1}d_{j-1}\ell_{j,j-1}\right)/d_j,\ j=1,\dots,k-1\ , \tag{4.5a}$$

$$d_k = a_{kk} - d_1\ell_{k1}^2 - \cdots - d_{k-1}\ell_{k,k-1}^2\ . \tag{4.5b}$$

In figure 4.1 we indicate the elements that appear in the computation of ℓ_{kj}. When the elements are computed in lexicographic order, they are known before they appear in the right hand side of (4.5).

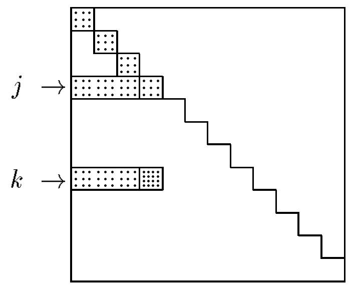


Figure 4.1.

In connection with the computation of the diagonal element it should be checked that

$$d_k > \epsilon|a_{kk}| \quad \text{with} \quad \epsilon = m\varepsilon_M\ , \tag{4.6}$$

where m is the number of nonzero terms in the righ hand side of (4.5b). If this condition is not satisfied, then the matrix is not significantly positive definite, and the factorization should stop. In that case the system $\boldsymbol{A}\mathbf{x} = \boldsymbol{b}$ should be solved with one of the methods mentioned after (4.1).

The formulation (4.5) involves two multiplications for each subtrahend. This can be halved by using an auxiliary vector $\boldsymbol{h}$, so that first we compute

$$d_j \ell_{kj} \equiv h_j = a_{kj} - h_1 \ell_{j1} - \cdots - h_{j-1}\ell_{j,j-1}, \quad j = 1, \ldots, k-1 \; , \qquad (4.7\text{a})$$

and perform the division by d_j in connection with the computation of the diagonal element:

$$\begin{array}{l} d_k := a_{kk} \\ \texttt{for } j = 1, \ldots, k-1 \\ \quad \ell_{kj} := h_j / d_j; \quad d_k := d_k - h_j * \ell_{kj} \\ \texttt{end} \end{array} \qquad (4.7\text{b})$$

With $\boldsymbol{A} = \boldsymbol{LDL}^T$, the solution to

$$\boldsymbol{Ax} = \boldsymbol{b}$$

is found in three steps,

$$1^\circ \quad \text{Solve} \quad \boldsymbol{Lw} = \boldsymbol{b} \; , \qquad (4.8\text{a})$$

$$2^\circ \quad \text{Solve} \quad \boldsymbol{Dz} = \boldsymbol{w} \; , \qquad (4.8\text{b})$$

$$3^\circ \quad \text{Solve} \quad \boldsymbol{L}^T \boldsymbol{x} = \boldsymbol{z} \; . \qquad (4.8\text{c})$$

Step 2° is often combined with step 1° or step 3°.

If $\boldsymbol{L}$ is stored rowwise, then it is preferable to solve (4.8a) by a DOT version as (3.12a) and use a SAXPY version to solve (4.8b), since the elements of the ith column of $\boldsymbol{L}^T$ are found in the ith row of $\boldsymbol{L}$. If $\boldsymbol{L}$ is stored columnwise, then the two versions are interchanged.

The above discussion covers both dense and sparse matrices. In the following sections we give some details about sparse implementation.

4.2. Band Matrices

We only treat matrices with variable bandwidth, as shown in figure 2.5b. It is seen that we cannot get fill-ins outside the indicated *"skyline"*, but zeros within the skyline will normally be filled during the factorization. Therefore, such elements are explicitly initialized to zero.

The data structure of figure 2.6 is well suited for implementation of algorithms (4.7). This is shown in (4.9) below, where we include the test (4.6) and overwrite a_{kj} first with h_j and then with ℓ_{kj}. Note that $d_1 = a_{11}$, and in each row the first $\boldsymbol{h}$-element is equal to the corresponding a-value.


```
k := 1;    posdef := (Elm(1) > 0)
while (k < n) and posdef
   k := k+1;    q0 := Rnd(k−1);    q1 := Rnd(k)
   j := k + q0 − q1                                        {1st col. no.}
   for q = q0+2, ..., q1
      j := j+1;    p := Rnd(j)
      l := min{q−q0, p−Rnd(j−1)} − 1
      Elm(q) := Elm(q) − <Elm(p−l : p), Elm(q−l : q)>
   end                                                      (4.9)
   j := k + q0 − q1 − 1;    d := Elm(k)
   for q = q0+1, ..., q1
      j := j+1;    h := Elm(q)
      Elm(q) := h/Elm(j);    d := d − h * Elm(q)
   end
   posdef := (d > ε * |Elm(k)|)
   Elm(k) := d
end
```

4.3. General Sparse Matrices

We still assume that the matrix is symmetric and positive definite, but now the nonzeros are scattered so that it is not advantageous to use band matrix techniques.

Instead, we can use the techniques of chapter 3, taylored to exploit the symmetry. Pivoting can be used to reduce the number of fill-ins, but in order to preserve symmetry, we only consider candidates from the diagonal. This is equivalent with $q_k = p_k$ in (1.6a), and (1.6b–c) show that the interchange of columns k and p_k is followed by interchange of rows k and q_k, so that the symmetry is restored.

The accumulated effect of the pivoting is that $\boldsymbol{A}$ is permuted to $\boldsymbol{PAP}^T$, which is symmetric when $\boldsymbol{A} = \boldsymbol{A}^T$. Further, it is easy to see from (4.2) that if $\boldsymbol{A}$ is positive definite, then so is $\boldsymbol{PAP}^T$. Thus, we can write

$$\boldsymbol{PAP}^T = \boldsymbol{LDL}^T, \quad \boldsymbol{D} = \operatorname{diag}(d_1, \ldots, d_n) \quad \text{with} \quad d_i > 0 \; . \qquad (4.10)$$

In the choice of pivots we can use a simplified version of the Markowitz strategy (3.5): Numerical stability is guaranteed by the choice of a diagonal element, and the active part of $\boldsymbol{A}$ is symmetric. Therefore, $s_p^{(k)} = r_p^{(k)}$, where $s_p^{(k)}$ ($r_p^{(k)}$) is the number of nonzeros in the active part of the pth column (row), and (3.5) reduces to the socalled *"minimum degree algorithm"*

$$p_k = \underset{k \le p \le n}{\operatorname{argmin}} \{r_p^{(k)}\} \ . \qquad (4.11)$$

If there is more than one row in $\boldsymbol{A}_k$ with the same minimal number of nonzeros, choose the first of these.

The pivoting sequence can be found without knowing the actual values of the nonzeros – only their positions, the sparsity pattern, is important. This calls for a very simple version of ANALYSE, cf. section 3.1, a socalled textit"symbolic ANALYSE". The computation of $\boldsymbol{L}$ and $\boldsymbol{D}$ defined by (4.10) is done by a FACTORIZE with a sparse implementation of (4.7) (with a_{kj} denoting the (k,j)th element in $\boldsymbol{PAP}^T$). Finally, SOLVE handles (3.11) with $\boldsymbol{U} = \boldsymbol{DL}^T$ and $\boldsymbol{Q} = \boldsymbol{P}^T$. See George & Liu (1981) for details.

Example 4.1. We consider the network (or girder construction) depicted below.

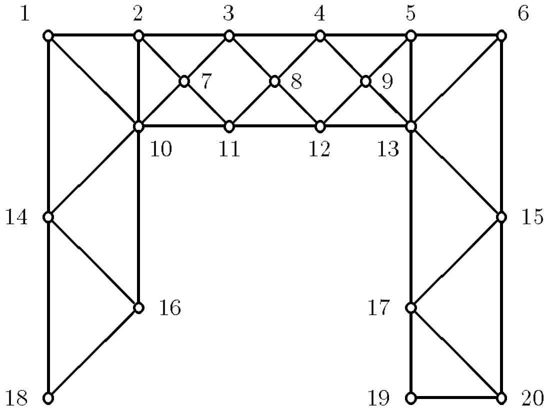


Figure 4.2. *Network*

The network has $n = 20$ *nodes*, and we let x_i be some property of node i that satisfies a linear system of equations

$$\sum a_{ij} x_{ij} = b_i \ , \quad i = 1, \ldots n \ , \qquad (4.12)$$

where $a_{ij} \neq 0$ only if nodes i and j are *neighbours*, i.e. if they are directly connected. We can write (4.12) as $\boldsymbol{Ax} = \boldsymbol{b}$, where $\boldsymbol{A}$ has the symmetric sparsity pattern shown in figure 4.3. We assume that also the values of the nonzeros are symmetric: $a_{ij} = a_{ji}$, and further, that the matrix is positive definite.

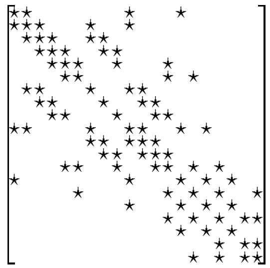


Figure 4.3. *Sparsity pattern for network in figure 4.2*

The number of nonzeros in the lower triangle of $\boldsymbol{A}$ is 58. If we use (4.1), then the total number of nonzeros in $\boldsymbol{D}$ and the strictly lower part of $\boldsymbol{L}$ is $nz_{\text{LD}} = 108$, i.e. the simple factorization gives almost a doubling of the storage demand.

The minimum degree algorithm (4.11) results in the following ordering of the equations and unknowns,

$$18,\ 16,\ 14,\ 1,\ 19,\ 20,\ 17,\ 15,\ 6,\ 10,\ 13,\ 2,\ 7,\ 3,\ 8,\ 11,\ 12,\ 4,\ 5,\ 9 \ . \qquad (4.13)$$

The matrix $\boldsymbol{PAP}^T$ is identical with the matrix $\boldsymbol{A}'$ obtained directly from the renumbered network

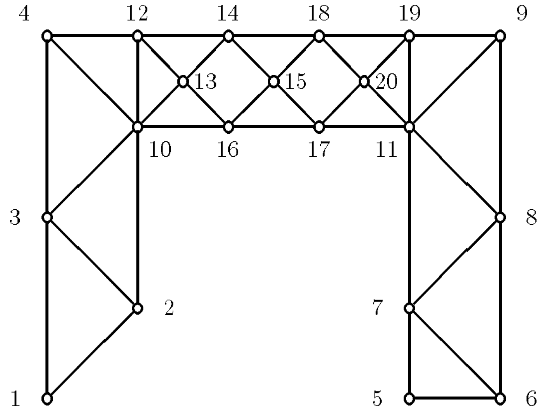


Figure 4.4. *Renumbered network*

The factorization of $\boldsymbol{A}'$ has the following structure, where fill-ins are marked with $\diamond$. There are 5 fill-ins in the lower triangle. This is a drastic reduction compared with the 50 fill-ins with the original ordering.

See the front page and example 4.5 for a larger illustration.

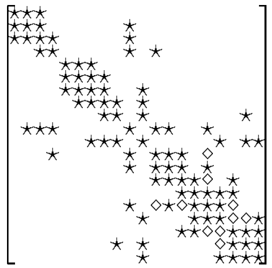


Figure 4.5. *Sparsity pattern of* $\boldsymbol{L}$, $\boldsymbol{D}$, $\boldsymbol{L}^T$ *for network in figure 4.4*

The above example can be generalized: For a given matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ we can construct a *graph* with n *nodes* and an *edge* from the ith to the jth node whenever $a_{ij} \neq 0$. A matrix with the structure shown in figure 4.3 has the graph shown in figure 4.2. The purpose of Analyse is to find a suitable renumbering of the nodes.

4.4. Ordering to Band Form

In chapter 2 and sections 4.1–2 we have indicated that it is simpler to work with band matrices than with general sparse matrices. The overhead is that we cannot exploit zeros within the band.

We want to renumber the nodes in the matrix graph so that the neighbours of the ith node have numbers as close to i as possible. This is equivalent with the nonzeros being as close to the diagonal as possible.

The kernel of such an ordering algorithm is as follows: Choose the first node, number i_1, *level 0*. The matrix has nonzeros in the positions corresponding to the neighbours of this node, and by choosing them next (level 1), the corresponding nonzeros are as close to the diagonal as possible. The general step is: After the nodes in level N put their "unused" neighbours in level $N+1$.

The number of nodes is fixed (n), and intuitively, the more levels we can split them between, the closer to the diagonal we get the nonzeros.

This is the background for the most widely used ordering algorithms: Cutthill–McKee (1969) with variations by e.g. Gibbs et al. (1976): Choose a starting node i_1 and proceed as indicated above. Then, choose a new starting node among the nodes in the last level and recompute. Repeat as long as the number of levels increases.

Example 4.2. For the graph in figure 4.2 we have used the ordering algorithm with starting guess $i_1 = 8$. In the later iterations we choose a node from the last level with minimal number of neighbours. In case of a tie we take the first. Note the fast convergence in spite of choosing the worst possible starting guess.

level	1st it. nodes	2nd it. nodes	3rd it. nodes
0	8	18	19
1	3, 4, 11, 12	14, 16	17, 20
2	2, 7, 5, 9, 10, 13	1, 10	13, 15
3	1, 6, 14, 16, 15, 17	2, 7, 11	5, 6, 9, 12
4	18, 20, 19	3, 8, 12	4, 8, 11
5		4, 9, 13	3, 7, 10
6		5, 6, 15, 17	2, 1, 14, 16
7		20, 29	18

When the iteration has converged, the renumbering is given by the numbers from the last iteration taken in reverse order. This is the socalled *"Reverse Cuthill–McKee"* (RCM) ordering.

Example 4.3. For the matrix of example 4.1 the results in example 4.2 give the RCM ordering

$$18,\ 16,\ 14,\ 1,\ 2,\ 10,\ 7,\ 3,\ 11,\ 8,\ 4,\ 12,\ 9,\ 6,\ 5,\ 15,\ 13,\ 20,\ 17,\ 19\ . \tag{4.14}$$

The corresponding structure of the matrix after factorization is shown in figure 4.6. The profile is 65 – slightly more than the 63 nonzeros in $\boldsymbol{L}$, $\boldsymbol{D}$ from the minimum degree algorithm.

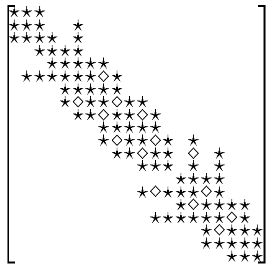


Figure 4.6. *Sparsity pattern of* $\boldsymbol{L}$, $\boldsymbol{D}$, $\boldsymbol{L}^T$ *for network in figure 4.2, renumbered by (4.14)*

The ordering algorithm is a special version of ANALYSE. It is normally implemented by means of an *adjacency list*, which gives the indeces of the neighbours of each node.

Example 4.4. The adjacency list can e.g. be implemented as an $n \times N$ integer array, where N is the maximal number of neighbours of any node. For the problem of example 4.1 $N=6$, and the adjacency array is

$$\texttt{adj} = \begin{array}{l} \left[\begin{array}{rrrrrr} 2 & 10 & 14 & 0 & 0 & 0 \\ 1 & 3 & 7 & 10 & 0 & 0 \\ 2 & 4 & 7 & 8 & 0 & 0 \\ \cdots & & & & & \\ 1 & 2 & 7 & 11 & 14 & 16 \\ \cdots & & & & & \\ 15 & 17 & 19 & 0 & 0 & 0 \end{array}\right] \end{array} \begin{array}{l} \\ \\ \\ \\ \leftarrow\ i=10 \\ \\ \leftarrow\ i=20 \end{array}$$

Experience shows that these ordering algorithms are fast and give (almost) optimal orderings. There are, however, many problems, where it is better to treat the matrix as general sparse, since even the optimal ordering to band form has too many zeros within the skyline.

Example 4.5. The picture on the front page shows (top half) the sparsity pattern of a 200×200 symmetric matrix and the structure of its $\boldsymbol{LDL}^T$ factorization. The number of nonzeros in the lower triangle grows from $nz_A = 599$ to $nz_{LD} = 7045$. The bottom half shows the matrix reordered by the minimum degree algorithm. Here, $nz_{LD} = 2087$, about $\frac{1}{3.5}$ of the number of nonzeros in the simple factorization.

In figure 4.7 we show the corresponding results from RCM (as provided by MATLAB®). Here, $nz_{LD} = 7239$ – more than the simple factorization.

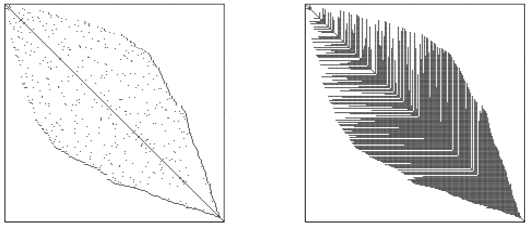


Figure 4.7. *Front page matrix reordered by RCM*

For a band matrix of order n with profile nz_B the storage scheme of figure

2.6 needs nz_B `real` + n `integer` array elements. For a "general symmetric" matrix with $nz_G = nz_{LD}$ elements we can use a row ordered storage scheme as in figure 2.2, supplemented with a column ordered list of row numbers. This demands nz_G `real` + $(2nz_G + 4n)$ `integer` array elements.

Assume that each `real` can hold two `integers`, and that $nz_B \lesssim 2nz_G$. Then it saves storage to represent the matrix as a band matrix.

4.5. Least–Squares Problems

Let $\boldsymbol{F} \in \mathbb{R}^{m \times n}$, $\boldsymbol{y} \in \mathbb{R}^m$ (with $m \geq n$) be given and define the *residual*

$$\mathbf{r}(\mathbf{x}) = \boldsymbol{y} - \boldsymbol{F}\mathbf{x} \ . \tag{4.15a}$$

The *least–squares solution* $\mathbf{x}^*$ is defined by

$$\mathbf{x}^* = \underset{\mathbf{x} \in \mathbb{R}^n}{\operatorname{argmin}} \{ r_1^2(\mathbf{x}) + \cdots + r_n^2(\mathbf{x}) \} \ . \tag{4.15b}$$

For a detailed discussion of least–squares problems we refer to chapter 6 in Nielsen (1996) (in Danish) or chapter 5 in Golub & Van Loan (1989) (in English). Here, we only give the background needed to discuss some sparse aspects.

We assume that $\boldsymbol{F}$ has full rank ($rank\boldsymbol{F}) = n$, i.e. the columns of $\boldsymbol{F}$ are linearly independent). Then $\mathbf{x}^*$ is unique, and may be found by *orthogonal transformation:* Determine an orthogonal matrix[12] $\boldsymbol{Q}$ so that

$$\boldsymbol{Q}\mathbf{r}(\mathbf{x}) = \begin{bmatrix} \tilde{\boldsymbol{y}}_1 \\ \tilde{\boldsymbol{y}}_2 \end{bmatrix} - \begin{bmatrix} \boldsymbol{R} \\ \boldsymbol{0} \end{bmatrix} \mathbf{x} \quad , \tag{4.16a}$$

where $\boldsymbol{R} \in \mathbb{R}^{n \times n}$ is upper triangular. The least–squares solution is found by back substitution in

$$\boldsymbol{R}\mathbf{x}^* = \tilde{\boldsymbol{y}}_1 \ . \tag{4.16b}$$

Alternatively, $\mathbf{x}^*$ can be computed as the solution to the *normal equations*

$$\boldsymbol{A}\mathbf{x}^* = \boldsymbol{b} \quad \text{with} \quad \begin{cases} \boldsymbol{A} = \boldsymbol{F}^T\boldsymbol{F} \\ \boldsymbol{b} = \boldsymbol{F}^T\boldsymbol{y} \end{cases} \ . \tag{4.17}$$

[12] i.e $\boldsymbol{Q} \in \boldsymbol{R}^{m \times m}$ and $\boldsymbol{Q}^T\boldsymbol{Q} = \boldsymbol{Q}\boldsymbol{Q}^T = \boldsymbol{I}$.

It is easily verified that $\boldsymbol{A}$ is symmetric and (when $\boldsymbol{F}$ has full rank) positive definite. This means that we can use the methods of sections 4.1–4 to find the factorization

$$\boldsymbol{PAP}^T = \boldsymbol{LDL}^T = \widetilde{\boldsymbol{L}}\widetilde{\boldsymbol{L}}^T \ . \tag{4.18}$$

If we apply orthogonal transformation to

$$\mathbf{r}(\mathbf{x}) = \boldsymbol{y} - (\boldsymbol{FP}^T)\boldsymbol{Px} \ ,$$

then the resulting upper triangular $\boldsymbol{R}$ is identical with $\widetilde{\boldsymbol{L}}^T$, except for rowwise sign.[13] This means that the sparsity pattern for $\boldsymbol{R}$ is the same as for $\widetilde{\boldsymbol{L}}^T$ – and for $\boldsymbol{L}^T$. This property can be used to find a good permutation of the columns in $\boldsymbol{F}$ in connection with orthogonal transformation:

As mentioned p. 33, the ordering corresponding to $\boldsymbol{P}$ can be found by "symbolic factorization", i.e. without the actual values of the nonzeros. This ordering can then be used to permute the columns in $\boldsymbol{F}$ so that the resulting $\boldsymbol{R}$ has (almost) as few nonzeros as possible. See Zlatev & Nielsen (1980) for an application with incomplete factorization and iterative refinement, and Matstoms (1994) for a recent overview.

Returning to the normal equations formulation, we can write

$$\boldsymbol{A} = \boldsymbol{F}^T\boldsymbol{F} = \sum_{i=1}^{m} \boldsymbol{F}_{i,:}^T \boldsymbol{F}_{i,:} \ , \tag{4.19}$$

where $\boldsymbol{F}_{i,:}$ is the row vector with the elements from the ith row in $\boldsymbol{F}$. It follows, that if $\boldsymbol{F}_{i,:}$ has nonzeros in columns $j_1, \ldots, j_{q_i}$, then $\boldsymbol{F}_{i,:}^T \boldsymbol{F}_{i,:}$ contributes to nonzeros in $\boldsymbol{A}$ in rows and columns $j_1, \ldots, j_{q_i}$. This shows, that normally the relative number of nonzeros in $\boldsymbol{A}$ is larger than in $\boldsymbol{F}$. Further, if $\boldsymbol{F}$ contains one (or more) dense row(s), then $\boldsymbol{A}$ is dense. There is, however, a way to come round this problem by "splitting" the matrix:

Let

$$\boldsymbol{F} = \begin{array}{c} \begin{bmatrix} \boldsymbol{F}_1 \\ \boldsymbol{F}_2 \end{bmatrix} \end{array} \begin{array}{l} m-r \\ r \end{array} \ ,$$

where $\boldsymbol{F}_2 \in I\!R^{r\times n}$ is (almost) dense, while $\boldsymbol{F}_1 \in I\!R^{(m-r)\times n}$ is (very) sparse, and we assume that it has full rank (so $m-r \geq n$ is a necessary condition). Then we can write

[13] i.e. $r_{ij} = s_i(\widetilde{\boldsymbol{L}}^T)_{ij}$, where $s_i = 1$ or $s_i = -1$.

$$\boldsymbol{A} = \boldsymbol{F}_1^T\boldsymbol{F}_1 + \boldsymbol{F}_2^T\boldsymbol{F}_2 = \boldsymbol{A}_1 + \boldsymbol{F}_2^T\boldsymbol{F}_2 \ . \tag{4.20}$$

The matrix $\boldsymbol{A}_1 = \boldsymbol{F}_1^T\boldsymbol{F}_1$ is positive definite (because $\boldsymbol{F}_1$ has full rank). This implies that $\boldsymbol{A}_1$ is non singular, and we can use the *Sherman-Morrison-Woodbury formula*[14]

$$(\boldsymbol{H} + \boldsymbol{UV}^T)^{-1} = \boldsymbol{H}^{-1} - \boldsymbol{H}^{-1}\boldsymbol{U}(\boldsymbol{I} + \boldsymbol{V}^T\boldsymbol{H}^{-1}\boldsymbol{U})^{-1}\boldsymbol{V}^T\boldsymbol{H}^{-1} \ . \tag{4.21}$$

In our application $\boldsymbol{H} = \boldsymbol{A}_1$ and $\boldsymbol{U} = \boldsymbol{V} = \boldsymbol{F}_2^T$. The formula looks scary, but remember the footnote on page 27 and that we have to solve the system $\boldsymbol{Ax} = \boldsymbol{b}$ with $\boldsymbol{A}$ given by (4.20): Let $\boldsymbol{z} \in I\!R^n$ and $\boldsymbol{W} \in I\!R^{n\times r}$ be the solutions to

$$\boldsymbol{A}_1\boldsymbol{z} = \boldsymbol{b} \quad \text{and} \quad \boldsymbol{A}_1\boldsymbol{W} = \boldsymbol{F}_2^T \ . \tag{4.22a}$$

With the help of (4.21) we see that $\mathbf{x}$ can be found as

$$\boldsymbol{x} = \boldsymbol{z} - \boldsymbol{Wu} \ , \tag{4.22b}$$

where $\boldsymbol{u}$ is the solution to

$$(\boldsymbol{I} + \boldsymbol{F}_2\boldsymbol{W})\boldsymbol{u} = \boldsymbol{F}_2\boldsymbol{z} \ . \tag{4.22c}$$

The matrix

$$\boldsymbol{I} + \boldsymbol{F}_2\boldsymbol{W} = \boldsymbol{I} + \boldsymbol{F}_2\boldsymbol{A}^{-1}\boldsymbol{F}_2^T$$

is symmetric and positive definite. It is dense but only of order r, which normally is much smaller than n. The solution to (4.22a) is found via a sparse factorization of $\boldsymbol{A}_1$ followed by $(r+1)$ calls of Solve with the right hand sides $\boldsymbol{b}$ and the r columns in $\boldsymbol{F}_2^T$.

This strategy is part of some of the algorithms for solving large linear programming problems by means of interior point methods, see Andersen et al. (1996).

[14] See section 2.1.3 in Golub & Van Loan (1989).

References

E.D. Andersen, J. Gondzio, C. Mészáros and X. Xu (1996): *Implementation of Interior Point Methods for Large Scale Linear Programming.* Publications from Department of Management No 1/96. Odense University, Odense, Denmark.

V.A. Barker (1992): *Iterative Methods for Sparse Systems of Linear Equations.* Lecture Notes #62, Institute for Numerical Analysis (now part of IMM), Technical University of Denmark.

E. Cuthill and J. McKee (1969): *Reducing the Bandwidth of Sparse Symmetric Matrices.* Proceedings 24th National Conference of the ACM. Brandon Press, New Jersey, 157–172.

I.S. Duff, A.M. Erisman and J.K. Reid (1986): *Direct Methods for Sparse Matrices.* Clarendon Press, Oxford.

I.S. Duff, J.K. Reid, N. Munksgaard and H.B. Nielsen (1979): *Direct Solution of sets of Linear Equations whose Matrix is Sparse, Symmetric and Indefinite.* J. Inst. Maths. Applics. **23**, 235–250.

A. George and J.W.H. Liu (1981): *Computer Solution of Large Sparse Positive Definite Systems.* Prentice–Hall, New Jersey.

N.E. Gibbs, W.G. Poole Jr. and P.K. Stockmeyer (1976): *An Algorithm for Reducing the Bandwidth and Profile of a Sparse Matrix.* SIAM J. Numer. Anal. **13**, 236–250.

G.H. Golub and C.F. Van Loan (1989): *Matrix Computations, 2nd edition.* John Hopkins University Press, Baltimore.

H.M. Markowitz (1957): *The Elimination Form of the Inverse and its Application to Linear Programming.* Management Sci. **3**, 255–269.

P. Matstoms (1994): *Sparse QR Factorization with Applications to Linear Least Squares Problems.* Ph.D. dissertation, Department of Mathematics, Linköping University, S-581 83 Linköping, Sweden.

Z. Zlatev, V.A. Barker and P.G. Thomsen (1978): *SSLEST: A FortranIV Subroutine for Solving Sparse Systems of Linear Equations. User's Guide.* Report NI-78-01, Institute for Numerical Analysis (now part of IMM), Technical University of Denmark.

Z. Zlatev and H.B. Nielsen (1980): *Least Squares Solution of Large LinearProblems.* Proceedings *Symposium i Anvendt Statistik 1980.* NEUCC, Lyngby, Denmark, 17–52.

Z. Zlatev and H.B. Nielsen (1988): *Solving Large and Sparse Linear Least Squares Problems by Conjugate Gradient Algorithms.* Comput. Math. Applic. **15**, 185–202.

O. Østerby and Z. Zlatev (1983): *Direct Methods for Sparse Matrices.* Lecture Notes in Computer Science 157, Springer, Berlin.

INDEX