Sparse matrices and substructures : with a novel implementation of finite element algorithms

SPARSE MATRICES AND SUBSTRUCTURES

FRANS PETERS

SPARSE MATRICES AND SUBSTRUCTURES

WITH A NOVEL IMPLEMENTATION OF FINITE ELEMENT ALGORITHMS

SPARSE MATRICES AND SUBSTRUCTURES

WITH A NOVEL IMPLEMENTATION OF FINITE ELEMENT ALGORITHMS

PROEFSCHRIFT

TER VERKRIJGING VAN DE GRAAD VAN DOCTOR IN DE TECHNISCHE WETENSCHAPPEN AAN DE TECHNISCHE HOGESCHOOL EINDHOVEN, OP GEZAG VAN DE RECTOR MAGNIFICUS, PROF.IR. J. ERKELENS, VOOR EEN COMMISSIE AANGEWEZEN DOOR HET COLLEGE VAN DEKANEN IN HET OPENBAAR TE VERDEDIGEN OP VRIJDAG 7 DECEMBER 1979 TE 16.00 UUR

DOOR

FRANCISCUS JOHANNES PETERS

GEBOREN TE EMMEN

1979

MATHEMATISCH CENTRUM, AMSTERDAM

Dit proefschrift is goedgekeurd
door de promotoren

Prof.dr. R.J. Lunbeck
en
Prof.dr. G.W. Veltkamp

Aan Thea,

Inge en Yvonne

CONTENTS

0. Introduction 1

0.1. Linear equations and partitions 1

0.2. A novel implementation of finite element algorithms 2

1. Linear equations 5

1.1. LU-decomposition 5

1.2. Profile and envelope algorithms 7

1.3. Graph-theoretic notation 10

1.4. Connection and decomposition graphs 12

1.5. Consistent orderings 14

1.6. Preserving palms 17

2. Partitions 19

2.1. Preserving partitions 20

2.2. Perfect preserving partitions 23

2.3. Proper pp-partitions 25

2.4. Construction of a proper pp-partition 28

2.5. Example 29

2.6. Non-symmetric equations 33

2.7. Nested dissection 36

3. Finite element equations 39

3.1. Outline of the finite element method 39

3.2. Traditional organization 42

3.3. Substructuring 43

4. A novel finite element algorithm for n × m grids 49

4.1. Procedure *ur* 49

4.1.1. Element specification and storage of results 51

4.1.2. Dissection of the rectangle and representation of the element matrices 52

4.1.3. *Decompose* 53

4.1.4. *Assemble* 55

4.1.5. Removal of reduced structure matrices 56

4.2. Procedures *fur* and *bur* 57

4.2.1. Computation of reduced structure vector 57

4.2.2. Computation of solution vector and derived results 59

5. Efficiency of *ur* 61
5.1. Storage and operation counts 61
5.2. Reduction of storage requirements 66

6. Adaptation to more complicated structures 73
6.1. Frame structures 73
6.2. Solid quadrilateral structures 74
6.2.1. Rectangles 74
6.2.2. Quadrilaterals 75
6.2.2.1. Quadrilateral given by points 76
6.2.2.2. Quadrilateral given by parametric functions 78
6.3. Local mesh refinements 79
6.3.1. Procedure *lm* 79
6.3.2. Storage and operation counts of *lm* 81
6.4. General plane and curved surfaces 82

7. Closing remarks 85
7.1. Other implementations 85
7.2. Data retrieval 86
7.3. Triangular dissections 87
7.4. One- and three-dimensional problems 87
7.5. Structure with more than one structure vector 88
7.6. Iterative methods 88
7.7. Data structuring facilities of PASCAL 89
7.8. Generalized element method, element merge tree 89
7.9. Parallel computation 91

Index 93

References 95

Samenvatting 99

Curriculum vitae 101

CHAPTER 0

INTRODUCTION

In this thesis we will prove that no intricate sparse matrix algorithms are required for an efficient solution of large, sparse sets of linear equations. Such equations occur in finite element calculations, for which a novel organization has been developed. Thus will be indicated in the second part of this introduction.

0.1. Linear equations and partitions

Large sets of linear equations are usually sparse, that is, nearly all coefficients of the associated matrix are zero. Let $Qw = f$ be such a set of equations, with Q a non-singular $n \times n$ matrix, w and f vectors of unknown and known values. Suppose this set can be solved (without permutation of rows and columns) by LU-decomposition [Wilkinson '65], i.e. there exist a lower triangular matrix L and an upper triangular matrix U such that $Q = LU$. The matrices L and U usually contain many more non-zero coefficients than Q (i.e. "fill-in" appears), but even so they are often sparse. To obtain L and U efficiently, one has to avoid, as much as possible, arithmetical operations with zero coefficients. Optimum efficiency in this sense is achieved by so called sparse matrix algorithms. For a survey see [Duff '77], which contains 604 references. Due to the fill-in these algorithms and the associated data structures are rather intricate. Envelope algorithms are much more simple. However, in general these are not so efficient, because zeros within the envelopes are not taken into account.

It is well-known [Duff '77, Tarjan '76] that the order of the equations and variables influences the sparsity of the associated triangular factors and the number of arithmetical operations with non-zero coefficients. Therefore one may try to find suitable permutation matrices P_1 and P_2 so as to solve the permuted set $(P_1 Q P_2^t)(P_2 w) = P_1 f$. The determination of permutation matrices, which achieve some minimum operation or storage count, is an NP-complete problem in some circumstances and is conjectured to be one in others [Tarjan '76]. Thus to obtain a good ordering of variables and

equations, the best way seems to be an heuristic approach to the original problem, which gives rise to the equations. In this thesis we shall not be concerned with finding optimal or good orderings, but with finding a method to avoid, given an ordering, arithmetical operations with zeros.

Using graph-theoretic terminology it is demonstrated that for a given matrix Q (irrespective of how well the equations are ordered) it is always possible to partition the set of variables in such a way that the triangular factors L and U can be obtained from the (partial) decompositions of smaller matrices determined by the partitioning. The decomposition of the smaller matrices will require the same arithmetical operations with non-zero matrix coefficients as the decomposition of Q. Because no zeros occur within the envelopes of the triangular factors associated with those smaller matrices, no intricate sparse matrix algorithms are required to avoid operations with zero coefficients; simpler (viz. envelope) algorithms suffice to obtain the triangular factors of Q with the least number of arithmetical operations. An algorithm will be presented which determines such a proper perfect preserving partition, as it is called.

0.2. A novel implementation of finite element algorithms

Large, sparse sets of equations are encountered in the finite element method, this being a widely used method to solve certain types of partial differential equations. Already before its invention, the mathematical soundness of the finite element method was shown in [Courant '43]. The method was independently developed in the fifties by aeronautical engineers concerned with stress and structural analysis [Turner e.a. '56]. The term "finite element" was used for the first time in [Clough '60].

The method was well received; it is applied to a variety of problems of the non-structural type such as occur in fluid mechanics, heat conduction, seepage flow, electric and magnetic potential. Its acceptance among engineers was assured at an early stage (the first edition of Zienkiewicz' book dates from 1967). Later, applied mathematicians [Zlamal '68, Strang and Fix '73] became interested. The popularity of the finite element method is due to the fact that it is highly suitable for computers. The Linköping survey [Fredriksson '76] already contains the description of 450 different computer programs for structural mechanics applications only. A recent

comprehensive bibliography [Norrie and De Vries '76] mentions over 7000 references up to the end of 1975.

The later part of this thesis deals with a novel efficient organization of finite element calculations. It does not aim to discuss possible specific applications of the method or when and under which circumstances the finite element method is to be favoured. Finite element computations include the assembly of a large sparse matrix from so called element matrices and the solution of an associated set of linear equations. The results described above for computing the triangular factors of a sparse matrix justify a novel organization of finite element computations. Instead of one large set of equations, a hierarchical series of smaller ones is set up or, in finite element terminology, instead of one large structure a hierarchy of smaller substructures is analysed. In particular we will show that if the nodes of the finite element structure are ordered according to the nested dissection strategy [George '73], then the associated proper perfect preserving partition is obtained immediately.

For (not necessarily homogeneous) $n \times m$ grids we describe algorithms in PASCAL, which find a suitable hierarchy of substructures and perform the decomposition of the associated matrices. These newly developed recursive algorithms differ from existing ones in that a number of traditionally consecutive steps (mesh generation, assembly, decomposition and forward substitution) are carried out here in an interleaved way. Moreover, the only data stored explicitly are (non-zero) matrix coefficients; no overhead data like pointers, etc. are required. The algorithms are conceptually simple and it is also possible to make them useful for arbitrary two-dimensional solid and frame structures. For three-dimensional structures similar algorithms could be developed.

CHAPTER 1

LINEAR EQUATIONS

By its very nature, it is convenient to view a sparse matrix as a graph [Parter '61, Rose '71, Tarjan '76]. In this chapter we will formulate LU-decomposition of a matrix in graph-theoretic terminology. Because reordering of the rows and columns of a matrix leads to different triangular factors, we will have to consider ordered graphs, i.e. graphs in which the vertices are ordered. Some new results will be derived concerning orderings which lead to the same triangular factors. The chapter will be concluded with a discussion of the new concept of a preserving palm, which is a graph for which the fill-in is restricted to certain edges.

1.1. LU-decomposition

Let

$$Qw = f \tag{1.1}$$

be a set of n linear equations in n unknowns, with Q a non-singular matrix, for which

$$Q = LU ,$$

where L is a lower triangular and U an upper triangular matrix. The coefficients of Q, L and U will be denoted by q_{ij}, ℓ_{ij} and u_{ij}, respectively ($1 \le i,j \le n$). The following relations hold [Wilkinson '65]:

$$\begin{cases} \ell_{jj}u_{jj} = q_{jj} - \sum_{t=1}^{j-1} \ell_{jt}u_{tj} & (1 \le j \le n) \\ \ell_{ij} = \left(q_{ij} - \sum_{t=1}^{j-1} \ell_{it}u_{tj}\right) \cdot u_{jj}^{-1} & \\ & (1 \le j < i \le n) \\ u_{ji} = \ell_{jj}^{-1} \cdot \left(q_{ji} - \sum_{t=1}^{j-1} \ell_{jt}u_{ti}\right) & \end{cases} \tag{1.2}$$

The coefficients of L and U must obviously be computed in a certain (partially prescribed) order.

The solution w of (1.1) can be obtained by a forward substitution, i.e. solving w' from

$$Lw' = f$$

followed by a backward substitution, i.e. solving w from

$$Uw = w' \ .$$

If Q is a symmetric matrix, then it is not necessary to compute all coefficients of both L and U. If Q is moreover positive definite then one may, for instance, use the formulae for Cholesky decomposition: $Q = LL^t$ [Wilkinson '65]

$$(1.2') \quad \begin{cases} \ell_{jj} = \left(q_{jj} - \sum_{t=1}^{j-1} \ell_{jt}^2\right)^{\frac{1}{2}} \\ \\ \ell_{ij} = \left(q_{ij} - \sum_{t=1}^{j-1} \ell_{it}\ell_{jt}\right) / \ell_{jj} \end{cases} \qquad (1 \le j < i \le n)$$

If I is a set of row indices and J a set of column indices, then Q_{IJ} denotes the matrix obtained from Q by deleting all coefficients with a row index not belonging to I or a column index not belonging to J. Let for a certain k $(1 \le k \le n)$ $I = \{1,\ldots,k\}$ and $J = \{k+1,\ldots,n\}$, then

$$Q = \begin{pmatrix} Q_{II} & Q_{IJ} \\ Q_{JI} & Q_{JJ} \end{pmatrix} .$$

Partial decomposition of Q with Q_{II} als block-pivot or, equivalently, partial decomposition of Q with its first k pivots is defined to be the following decomposition of Q [Bunch and Rose '74]:

$$Q = L_I U_I + \begin{pmatrix} 0_{II} & 0_{IJ} \\ 0_{JI} & Q_I^r \end{pmatrix}$$

with 0_{II}, 0_{IJ} and 0_{JI} zero matrices,

$$L_I = \begin{pmatrix} L_{II} \\ L_{JI} \end{pmatrix} , \quad U_I = (U_{II}, U_{IJ}) \ ,$$

where L_{II} is lower triangular and U_{II} upper triangular such that

$$L_{II}U_{II} = Q_{II} \qquad L_{II}U_{IJ} = Q_{IJ}$$

$$L_{JI}U_{II} = Q_{JI} \qquad Q_I^r = Q_{JJ} - L_{JI}U_{IJ} \ .$$

This partial decomposition is only defined if Q_{II} is non-singular and has itself an LU-decomposition.

If Q is symmetric and positive definite, then partial Cholesky decomposition of Q with Q_{II} as block-pivot results in:

$$Q = L_I U_I' + \begin{pmatrix} 0_{II} & 0_{IJ} \\ 0_{JI} & Q_I^r \end{pmatrix}$$

where

$$L_I^t = U_I' = (L_{II}^t \ , L_{JI}^t)$$

with

$$L_{II}L_{II}^t = Q_{II}$$

$$L_{JI}L_{II}^t = Q_{JI}$$

$$Q_I^r = Q_{JJ} - L_{JI}L_{JI}^t \ .$$

To compute the coefficients of L_I and Q_I^r for partial Cholesky decomposition the following formulae will be used:

$$(1.2^*) \quad \begin{cases} \ell_{jj} = \left(q_{jj} - \sum_{t=1}^{j-1} \ell_{jt}^2\right)^{1/2} & (1 \le j \le k) \\ \ell_{ij} = \left(q_{ij} - \sum_{t=1}^{j-1} \ell_{it}\ell_{jt}\right) / \ell_{jj} & (1 \le i \le n, \ 1 \le j \le \min(k, i-1)) \\ q^r_{i-k,j-k} = q_{ij} - \sum_{t=1}^{k} \ell_{it}\ell_{jt} & (k < j \le i \le n) \ . \end{cases}$$

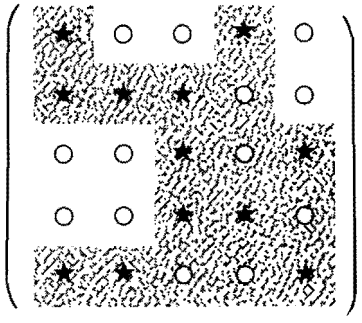
1.2. Profile and envelope algorithms

If the matrix Q is sparse, then usually its triangular factors L and U contain more non-zero coefficients than Q. Nevertheless L and U are often sparse matrices as well. A coefficient which is zero in Q but non-zero in

L+U is said to belong to the fill-in of Q. (A formal definition of fill-in will be given in Section 1.4.) The fill-in is always restricted to the so called envelope of Q, defined as follows. Let the column index of the first non-zero coefficient in the lower triangular part of row i be denoted by $r_Q(i)$. If no such coefficient exists in row i, then $r_Q(i)$ is defined to be equal to i. In the same way $c_Q(j)$ denotes the row index of the first non-zero coefficient in the upper triangular part of column j. Again, if no such coefficient exists, then $c_Q(j) = j$. (The subscript Q will be dropped if no confusion is likely.) If $b_\ell = \max\{j - r(j) \mid 1 \le j \le n\}$ and $b_u = \max\{j - c(j) \mid 1 \le j \le n\}$, then the *bandwidth* of Q is defined to be the maximum of b_ℓ and b_u. Next env(Q) (the *envelope* of Q) is defined as:

$$env(Q) = \{(i,j) \mid j \ge r(i) \text{ and } i \ge c(j)\}$$

which implies: if $(i,j) \notin env(Q)$ then $q_{ij} = 0$. For an example see figure 1.



★ denotes non-zero coefficient

○ denotes zero coefficient

envelope

bandwidth is 4

Figure 1

If Q has an LU-decomposition Q = LU, what do we know of env(L) and env(U) ? As usual we neglect numerical cancellation: ℓ_{ij} (i > j) is considered to be a zero coefficient only if $q_{ij} = 0$ and all relevant products $\ell_{it}\ell_{jt}$ or $\ell_{it}u_{tj}$ in formulae 1.2 or 1.2' are zero; analogously for u_{ij} (i < j). It is well known [George and Liu '75] that with this convention

$$\text{env}(L+U) = \text{env}(Q) \ .$$

Only those coefficients of L and U whose indices belong to env(L + U) need to be computed, the others being zero. If Q is symmetric, we use instead of the envelope the *profile* defined by:

$$\text{pr}(Q) = \{(i,j) \mid i \geq j \text{ and } j \geq r(i)\} \ .$$

We then have:

$$\text{pr}(L) = \text{pr}(Q) \ .$$

If $(i,j) \in \text{env}(Q)$ and $q_{ij} = 0$ $(i \neq j)$, then (i,j) will be called a zero element of env(Q). An envelope without zero elements will be called *dense*. In an analogous way a dense profile is defined. By definition (i,j) is a first zero element of env(Q) if it is a zero element and if for all other zero elements (i',j') of env(Q) it yields:

$$i' > i \quad \text{or} \quad j' > j \ .$$

In the next chapter we will use the following lemma, which is a generalization of a theorem in [George and Liu '75]:

LEMMA 1: If (i,j) is a first zero element of env(L + U), then
$c_Q(j) = c_U(j) = j$ (if $i > j$) or $r_Q(i) = c_L(i) = i$ (if $i < j$).

Proof: Suppose $i > j$.
From (1.2) and $\ell_{ij} = 0$ it follows

$$\ell_{it} u_{tj} = 0 \ , \quad 1 \leq t < j \ . \tag{1.3}$$

(i,j) is a first zero element of env(L + U) implies

$$\ell_{i,j-1} \neq 0 \ . \tag{1.4}$$

From (1.3) and (1.4) we deduce

$$u_{j-1,j} = 0 \ . \tag{1.5}$$

Because (i,j) is a first zero element of env(L + U), (1.5) implies:

$$(j-1,j) \notin \text{env}(L+U) \ .$$

From this we conclude $c_Q(j) = c_U(j) = j$.
In the same way it can be shown that $i < j$ implies $r_Q(i) = r_L(i) = i$. □

Remark: Let L_I and U_I be the factors obtained from a partial decomposition of Q with its first k pivots ($1 \leq k \leq n$). From Lemma 1 it follows: if $r_Q(i) < i$ for $1 < i \leq k$ and $c_Q(j) < j$ for $1 < j \leq k$, then both $env(L_I)$ and $env(U_I)$ are dense.

Corollary: If Q is a symmetric matrix and (i,j) is a first zero element of pr(L), then $r_Q(j) = c_Q(j) = j$.

Algorithms which use the property that only the coefficients whose indices belong to pr(L) or env(L+U) are non-zero, will be called profile and envelope algorithms respectively.

The profile and envelope algorithms may be overly inefficient in that they may process many matrix coefficients which are in fact zero. This will happen with sparse matrices having large envelopes or profiles. To handle those matrices with optimum efficiency, i.e. to avoid arithmetical operations with zeros, so called *sparse matrix algorithms* have been developed; these are algorithms in which for every coefficient is recorded whether it is zero or not. Obviously these algorithms require considerable organizational overhead, the more so as it is not a priori clear which coefficients of L and U are non-zero. (The so called fill-in consists of all the coefficients which are zero in Q, but non-zero in L or U. This fill-in will be discussed into more detail in the sequel.) For a description see [Gustavson '72]. Usually a sparse matrix algorithm contains the following steps:

- symbolic decomposition: to determine the location of the non-zeros in L and U;
- numeric decomposition: to determine the values of the non-zero coefficients.

It will be clear that sparse matrix algorithms are rather intricate.

1.3. Graph-theoretic notation

Following [Rose '71, George '77] we will introduce in this section a graph-theoretic notation and nomenclature to be used later on.

A *directed graph* G = (V,E) consists of a finite set V of *vertices* and a finite set $E \subseteq \{(v,w) \mid v,w \in V,\ v \neq w\}$ of ordered vertex pairs called *edges*. An *undirected graph* G = (V,E) consists of a finite set V of vertices

and a finite set E of unordered vertex pairs, i.e. (v,w) is considered to be the same as (w,v). Whenever in the sequel it is left unspecified whether or not the graph G is directed, G may be either.

Let G = (V,E) be a graph. If W $\subset$ V, then the *section graph* G(W) is the subgraph (W,E(W)), where E(W) = {(v,w) $\in$ E | v,w $\in$ W}. For v $\in$ V the *adjacency set* adj(v) is defined by adj(v) = {w | (v,w) $\in$ E}. For distinct vertices v and w a *path* from v to w of length k is defined to be a sequence of distinct vertices $v = v_0, v_1, \ldots, v_k = w$, such that $(v_{i-1}, v_i) \in E$, for $i = 1, \ldots, k$.

An undirected graph is called *connected* and a directed graph is called *strongly connected*, if for every pair of distinct vertices v, w there is a path from v to w. If a graph is not (strongly) connected, then it consists of two or more (strongly) connected components. The set W $\subset$ V is a *separator* of the graph G = (V,E) if the section graph G(V \ W) is not (strongly) connected. A separator W of G = (V,E) is *minimal* if no proper subset W' $\subset$ W is a separator of G.

A *rooted tree* T is an undirected graph with a distinguished vertex r, called the *root*, such that there is a unique path from r to any vertex. If v is on the path from r to w (w $\neq$ v), then v is an *ancestor* of w and w is a *descendant* of v. If moreover (v,w) is a tree edge, then v is the *predecessor* of w and w is a *successor* of v. A vertex without successors is called a *leaf* vertex.

The concepts defined above are rather standard and definitions of them occur rather frequently in literature. The following concept, although not new, is less well-known. It has been introduced (with the name palm tree) in [Tarjan '72] in connection with depth-first searches in graphs. In this thesis we will name it a palm and use it to investigate the LU-decomposition of associated matrices. An undirected graph G = (V,E) is called a *palm*, if the edge set E consists of two disjoint sets E = E1 $\cup$ E2 such that

i) the graph T = (V,E1) is a rooted tree,

ii) if (v,w) $\in$ E2 then v is an ancestor or descendant of w in T.

The edges of E1 and E2 are called tree edges and *fronds* respectively. Hence a palm may be obtained from a rooted tree T by appending a number of (possibly zero) fronds; a frond is always an edge from a vertex to one of its ancestors in T. A *palm forest* is defined to be an undirected graph, whose connected components are palms.

If v is a vertex of a palm or a tree, then $A(v)$ and $D(v)$ denote the sets of ancestor, respectively descendant vertices; $\bar{A}(v)$ denotes $A(v) \cup v$ and $\bar{D}(v)$ denotes $D(v) \cup v$.

For a graph $G = (V,E)$ with $|V| = n$, an *ordering* α of V is a bijection $\alpha: \{1,2,\ldots,n\} \to V$. $G_\alpha = (V,E,\alpha)$ denotes an *ordered graph*.

Given a graph $G = (V,E)$, let P be a *partition* of V, i.e. $P = \{V1,\ldots,Vm\}$, such that $V = \bigcup_{s=1}^{m} Vs$ and $Vs \cap Vt = \emptyset$ for $s \neq t$. The *quotient graph* of G with respect to P, denoted by G/P is the graph $G/P = (P,E)$, where $(Vs,Vt) \in E$ if and only if vertices $v \in Vs$ and $w \in Vt$ exist, so that $(v,w) \in E$. Obviously: G (strongly) connected implies G/P (strongly) connected.

1.4. Connection and decomposition graphs

Let $G = (V,E,\alpha)$ be an ordered graph and define $E' = E \cup \{(v,v) \mid v \in V\}$. Suppose a map $q: E' \to \mathbb{R}$ is associated with G. The numbers $q((v,w))$, associated with the edges (v,w), may be arranged in an $n \times n$ matrix Q (with $n = |V|$):

$$q_{ij} = q((\alpha(i),\alpha(j))) \quad \text{if } (\alpha(i),\alpha(j)) \in E' \qquad 1 \leq i,j \leq n.$$
$$q_{ij} = 0 \quad \text{otherwise}$$

The graph G is then the *connection graph* of Q.
The coefficients q_{ij} with $(\alpha(i),\alpha(j)) \in E$ will be called structurally non-zero, even though their value may happen to be zero. Whenever henceforth a coefficient is said to be non-zero, we will always mean structurally non-zero.

Conversely, with each square matrix a connection graph is associated in the following way. Let Q be an $n \times n$ matrix. Define the vertex set V by $V = \{1,\ldots,n\}$ and the edge set E by:

$$(v,w) \in E \text{ iff } q_{v,w} \neq 0 \quad (v \neq w).$$

Let moreover the ordering α be defined by $\alpha(i) = i$ $(1 \leq i \leq n)$. Then the graph $G = (V,E,\alpha)$ is a connection graph of Q.

Generally, a connection graph $G = (V,E,\alpha)$ is directed. However, if E is such that $(v,w) \in E$ iff $(w,v) \in E$, G will be considered to be undirected. In that case an associated matrix Q is structurally symmetric, i.e. $q_{ij} \neq 0$ iff $q_{ji} \neq 0$. If an undirected graph is not connected, then the associated (structurally symmetric) matrices are *decomposable*. If a directed graph is not strongly connected, then the associated matrices are *reducible*. If the matrix Q is reducible or decomposable, then the associated set of equations may be replaced by a number of smaller sets [Varga '62].

For a vertex v of the graph $G = (V,E,\alpha)$ the *elimination graph* G_v is defined as $G_v = (V \setminus \{v\}, E(V \setminus \{v\}) \cup D(v))$, where $D(v) = \{(x,y) \mid (x,v) \in E, (v,y) \in E, x \neq y, (x,y) \notin E\}$. D(v) is called the *deficiency* of v in G.

Let now $G = (V,E,\alpha)$ be the connection graph of a matrix Q. Partial decomposition of Q with only its first pivot gives:

$$Q_1^r = \begin{bmatrix} q_{22} - \ell_{21}u_{12} & \cdots & q_{2n} - \ell_{21}u_{1n} \\ \vdots & & \vdots \\ q_{n2} - \ell_{n1}u_{12} & \cdots & q_{nn} - \ell_{n1}u_{1n} \end{bmatrix} .$$

Since $q_{ij}^r \neq 0$ iff $(q_{ij} \neq 0$ or $(q_{i1} \neq 0$ and $q_{1j} \neq 0))$, it is easily seen that the elimination graph $G_{\alpha(1)}$ is precisely the connection graph of Q_1^r. The deficiency $D(\alpha(1))$ consists of those vertices $(\alpha(i),\alpha(j))$ with $q_{ij} = 0$ and $q_{ij}^r \neq 0$.

Let $G^\circ = G = (V,E,\alpha)$ and for $i = 1,\ldots,n-1$ let G^i be recursively defined by: G^i is the elimination graph $(G^{i-1})_{\alpha(i)}$. The *fill-in* $D_\alpha(G)$ is defined by

$$D_\alpha(G) = \bigcup_{i=1}^{n-1} D^{i-1}(\alpha(i)) ,$$

where $D^{i-1}(\alpha(i))$ is the deficiency of $\alpha(i)$ in G^{i-1}. The *decomposition graph* G_α^* is defined by: $G_\alpha^* = (V,E \cup D_\alpha(G))$. If $G = (V,E,\alpha)$ is the connection graph of an $n \times n$ matrix Q with LU-decomposition $Q = LU$, then the decomposition graph G_α^* is the connection graph of $L+U$ [Rose and Tarjan '75]. The fill-in $D_\alpha(G)$ consists precisely of those edges $(\alpha(i),\alpha(j))$ which satisfy

$$q_{ij} = 0 \quad \text{and} \quad (\ell_{ij} \neq 0 \text{ or } u_{ij} \neq 0) .$$

For an example see figure 3.

The fill-in is characterized by the following lemma:

LEMMA 2: Let $G = (V,E,\alpha)$ be an ordered graph. Then (v,w) is an edge of $G_\alpha^* = (V, E \cup D_\alpha(G))$ if and only if there exists a path $v = v_0,\ldots,v_k = w$ in G such that $\alpha^{-1}(v_i) < \text{minimum}(\alpha^{-1}(v),\alpha^{-1}(w))$ for $i = 1,\ldots,k-1$.

A proof of this lemma may be found in [Rose and Tarjan '75]. □

In Section 1.2 we have introduced the notation r_Q and c_Q. Let $G = (V,E,\alpha)$ be the connection graph of the matrix Q with LU-decomposition $Q = LU$. Suppose v is a vertex of G with $\alpha(i) = v$. A notation equivalent to $r_Q(i) < i$ is:

$$\exists_{w\in V}\ [\alpha^{-1}(w) < \alpha^{-1}(v) \text{ and } (v,w) \in E] .$$

Similarly $r_Q(j) < j$ is equivalent to:

$$\exists_{w\in V}\ [\alpha^{-1}(w) < \alpha^{-1}(v) \text{ and } (w,v) \in E] .$$

Hence we know from Lemma 1: if for all $v \in V$ with $\alpha^{-1}(v) \neq 1$, there exists a $w \in V$ with $\alpha^{-1}(w) < \alpha^{-1}(v)$ and $(v,w) \in E$ and $(w,v) \in E$, then both env(L) and env(U) are dense; that is to say, the envelopes of L and U do not contain any zero element.

1.5. Consistent orderings

Let $G_\alpha = (V,E,\alpha)$ be the connection graph of an $n \times n$ matrix Q. Let β be also an ordering of V. The graph $G_\beta = (V,E,\beta)$ is the connection graph of the matrix $\tilde{Q}$:

$$\begin{aligned} &\tilde{q}_{ij} = q((\beta(i),\beta(j))) , && \text{if } (\beta(i),\beta(j)) \in E \\ &\tilde{q}_{ij} = 0 && \text{otherwise} \end{aligned} \qquad (1 \le i,j \le n) . \tag{1.6}$$

The orderings α and β together determine a permutation $\pi = \beta^{-1}\alpha$ of $\{1,\ldots,n\}$. A *permutation matrix* associated with π is defined by:

$$(P)_{ij} = \delta_{i,\pi(j)} \qquad (1 \le i,j \le n)$$

where δ is the Dirac delta function. The following relation holds:

$$\tilde{Q} = PQP^t$$

or in other words, G_β is the connection graph of PQP^t.

Proof:

$$(PQP^t)_{ij} = \sum_{k,\ell=1}^{n} \delta_{i,\pi(\ell)}\, q_{\ell k}\, \delta_{j,\pi(k)} = q_{\pi^{-1}(i),\pi^{-1}(j)} .$$

Hence:

$$(1.7) \qquad (PQP^t)_{ij} = q_{\alpha^{-1}\beta(i),\alpha^{-1}\beta(j)} .$$

Since G_α is the connection graph of Q, we know that

$$(1.8) \quad \begin{cases} q_{\alpha^{-1}\beta(i),\alpha^{-1}\beta(j)} = q((\beta(i),\beta(j))) & \text{if } (\beta(i),\beta(j)) \in E \\ q_{\alpha^{-1}\beta(i),\alpha^{-1}\beta(j)} = 0 & \text{otherwise.} \end{cases}$$

Since G_β is the connection graph of $\tilde{Q}$ we know that (1.6) holds. Hence we conclude from (1.7), (1.8) and (1.6):

$$(PQP^t)_{ij} = \tilde{q}_{ij} .$$ □

Suppose the matrix Q has an LU-decomposition: $Q = LU$. The fill-in $D_\alpha(G)$ and the decomposition graph G^*_α depend upon the ordering α of V. Usually, LU-decomposition of the permuted matrix PQP^t results in triangular factors essentially different from L and U. That is to say, the triangular factors of PQP^t can *not* be obtained from a suitable permutation of the rows and columns of L and U. For instance, the number of non-zero coefficients in the triangular factors of PQP^t usually differs from the number of non-zeros in L and U. However, if PLP^t is lower triangular and PUP^t is upper triangular, then

$$PQP^t = (PLP^t)(PUP^t)$$

is an LU-decomposition of $\tilde{Q} = PQP^t$.

An ordering β of V will be called *consistent* with α if all edges (v,w) of the decomposition graph $G^*_\alpha = (V, E \cup D_\alpha(G), \alpha)$ satisfy:

$$\alpha^{-1}(v) > \alpha^{-1}(w) \quad \text{iff} \quad \beta^{-1}(v) > \beta^{-1}(w) \ .$$

We will prove:

LEMMA 3: If the ordering β is consistent with α and P is the permutation matrix associated with $\pi = \beta^{-1}\alpha$, then PLP^t and PUP^t are lower and upper triangular respectively.

Proof: $(PLP^t)_{\pi(i),\pi(j)} = \ell_{ij}$, hence in order to prove that PLP^t is lower triangular, it suffices to show:

$$\ell_{ij} \neq 0 \rightarrow \pi(i) > \pi(j) \ . \tag{1.9}$$

From $\ell_{ij} \neq 0$, assuming $\alpha(i) = v$ and $\alpha(j) = w$, it follows that

$$\alpha^{-1}(v) > \alpha^{-1}(w) \ .$$

Because β is consistent with α we conclude

$$\beta^{-1}(v) > \beta^{-1}(w) \ .$$

Hence $\pi(i) = \beta^{-1}\alpha(i) > \beta^{-1}\alpha(j) = \pi(j)$. Herewith (1.9) is proved. In the same way it is shown that PUP^t is upper triangular. □

LEMMA 4: If PLP^t is lower triangular and PUP^t upper triangular, then the expressions to be evaluated during the decompositions of Q and PQP^t respectively, both contain the same non-zero terms.

Proof: Let $Q' = PQP^t$, $L' = PLP^t$ and $U' = PUP^t$, then

$$q'_{\pi(i),\pi(j)} = q_{ij} \ , \quad \ell'_{\pi(i),\pi(j)} = \ell_{ij} \ , \quad u'_{\pi(i),\pi(j)} = u_{ij} \quad (1 \leq i,j \leq n). \tag{1.10}$$

When decomposing PQP^t, the computation of ℓ'_{ij} requires the evaluation of:

$$\left(q'_{ij} - \sum_{t=1}^{j-1} \ell'_{it}\, u'_{tj}\right) / u'_{jj} = \left(q'_{ij} - \sum_{\substack{t=1 \\ t\neq j}}^{n} \ell'_{it}\, u'_{tj}\right) / u'_{jj} \ . \tag{1.11}$$

From (1.10) we see that the righthand side of (1.11) may be rewritten as

$$\left(q_{\pi^{-1}(i),\pi^{-1}(j)} - \sum_{\substack{t=1 \\ t\neq j}}^{n} \ell_{\pi^{-1}(i),\pi^{-1}(t)}\, u_{\pi^{-1}(t),\pi^{-1}(j)}\right) / u_{\pi^{-1}(j),\pi^{-1}(j)} \ .$$

Substituting k for $\pi^{-1}(t)$ in the above we get

$$\left(q_{\pi^{-1}(i),\pi^{-1}(j)} - \sum_{\substack{k=1 \\ k\neq\pi^{-1}(j)}}^{n} \ell_{\pi^{-1}(i),k}\, u_{k,\pi^{-1}(j)}\right) / u_{\pi^{-1}(j),\pi^{-1}(j)}$$

which is the expression for $\ell_{\pi^{-1}(i),\pi^{-1}(j)}$. Hence the expression to be evaluated to compute ℓ'_{ij} contains (apart from the order) the same non-zero terms as the expression for $\ell_{\pi^{-1}(i),\pi^{-1}(j)}$. In the same manner we can prove that the expressions for u'_{ij} and $u_{\pi^{-1}(i),\pi^{-1}(j)}$ both contain (apart from the order) the same non-zero terms. □

An immediate consequence of Lemmas 3 and 4 is:

Corollary: Let α and β be the orderings associated with the matrices Q and PQP^t respectively (P being a permutation matrix). If β is consistent with α then the LU-decompositions of Q and PQP^t both require the evaluation of the same expressions with non-zero terms.

1.6. Preserving palms

An ordered undirected graph $G = (V,E,\alpha)$ is a *preserving palm*, if it is a palm with the property: $v \in V$, $w \in \mathcal{D}(v) \rightarrow \alpha^{-1}(v) > \alpha^{-1}(w)$. Let $G = (V,E,\alpha)$ be the connection graph associated with the $n \times n$ matrix Q and let Q have an LU-decomposition: $Q = LU$. If the decomposition graph $G^*_\alpha = (V, E \cup D_\alpha(G), \alpha)$ is a preserving palm, then we can show that

$$(1.12)\quad \begin{cases} \ell_{ij} = \left(q_{ij} - \sum_{\alpha(t)\in\mathcal{D}(\alpha(j))} \ell_{it}\, u_{tj}\right) \cdot u_{jj}^{-1} \\ \qquad\qquad 1 \le i \le n,\ \alpha(j) \in \mathcal{D}(\alpha(i)) \\ u_{ji} = \ell_{jj}^{-1}\left(q_{ji} - \sum_{\alpha(t)\in\mathcal{D}(\alpha(j))} \ell_{jt}\, u_{ti}\right) \\ \ell_{jj}\, u_{jj} = q_{jj} - \sum_{\alpha(t)\in\mathcal{D}(\alpha(j))} \ell_{jt}\, u_{tj} \qquad (1 \le j \le n) \end{cases}$$

Since L is lower triangular and G^*_α is the connection graph of L + U, we have

$$(1.13)\quad i \neq j \text{ and } \ell_{ij} \neq 0 \text{ iff } i > j \text{ and } (\alpha(i),\alpha(j)) \in E \cup D_\alpha(G)\ .$$

Since G_α^* is supposedly a palm, we conclude from (1.13)

(1.14) $\quad i \neq j$ and $\ell_{ij} \neq 0$ iff $i > j$ and $[\alpha(i) \in \mathcal{D}(\alpha(j))$ or $\alpha(j) \in \mathcal{D}(\alpha(i))]$.

However, since G_α^* is a preserving palm, it follows from (1.14)

(1.15) $\quad i \neq j$ and $\ell_{ij} \neq 0$ iff $\alpha(j) \in \mathcal{D}(\alpha(i))$.

The matrix Q is structurally symmetric, therefore:

$$\ell_{ij} \neq 0 \text{ iff } u_{ji} \neq 0 .$$

Hence (1.15) may be extended to:

(1.16) $\quad i \neq j$ and $\ell_{ij} \neq 0$ iff $i \neq j$ and $u_{ji} \neq 0$ iff $\alpha(j) \in \mathcal{D}(\alpha(i))$.

Finally, from (1.16) together with (1.2) we conclude that (1.12) holds.

CHAPTER 2

PARTITIONS

In this chapter we will investigate certain partitions of a matrix Q (with LU-decomposition Q = LU) into blocks (which are again matrices). Associated with such a partition is the quotient graph obtained from the corresponding partition of the vertex set of the connection graph of Q.

For undirected graphs, corresponding to structurally symmetric matrices, we introduce preserving partitions and describe how to compute the coefficients of L and U. Perfect preserving partitions will be shown to have the nice property that the corresponding computations to obtain the coefficients of L and U do not require sparse matrix algorithms to avoid arithmetical operations with zeros. It will be proved that with every ordered undirected graph an (even proper) perfect preserving partition is associated. Hence, irrespective of how well the rows and columns of a matrix are ordered, the coefficients of its triangular factors can be obtained with optimum efficiency by using only envelope algorithms. For directed graphs similar, though under certain circumstances less stronger results hold.

The last section of this chapter deals with nested dissection [George '73], a well-known way of finding a suitable ordering for a connection graph. It is shown that if nested dissection is used to construct an ordering, then finding a proper perfect preserving partition is trivial. Hence, in order to implement a nested dissection decomposition of a matrix with optimum efficiency, no sparse matrix codes are needed; envelope algorithms suffice.

In this chapter it is assumed, unless stated otherwise, that $G = (V,E,\alpha)$ is the ordered undirected connection graph of a structurally symmetric $n \times n$ matrix Q with LU-decomposition Q = LU, where L is lower triangular and U is upper triangular.

2.1. Preserving partitions

A *block-matrix* is a matrix whose coefficients are matrices, called blocks. All formulas and results derived thusfar apply to block matrices with square matrices as diagonal blocks. A block is considered to be zero if it is a zero matrix.

Let $P = \{V1,\ldots,Vm\}$ be a partition of V. Let Q_{rs} denote the matrix obtained from Q by deletion of all rows i for which $\alpha(i) \notin Vr$ and all columns j for which $\alpha(j) \notin Vs$. Define:

$$\tilde{Q} = \begin{bmatrix} Q_{11} & \cdots & Q_{1m} \\ \vdots & & \vdots \\ Q_{m1} & \cdots & Q_{mm} \end{bmatrix} . \tag{2.1}$$

($\tilde{Q}$ will be considered to be an $n \times n$ matrix.) Moreover, let β denote the unique ordering $\{1,\ldots,n\} \to V$ determined by

i) $v \in Vr$, $w \in Vs$, $r > s \to \beta^{-1}(v) > \beta^{-1}(w)$

(i.e. β orders first the vertices in V1, next the vertices in V2,... and so on);

ii) $v,w \in Vr$, $\alpha^{-1}(v) > \alpha^{-1}(w)$ iff $\beta^{-1}(v) > \beta^{-1}(w)$

(i.e. within every Vr the vertices are ordered according to α).

Hence the rows and columns of $\tilde{Q}$ are ordered in such a way that $G_\beta = (V,E,\beta)$ is the connection graph of $\tilde{Q}$. Therefore we know from Section 1.5:

$$\tilde{Q} = PQP^t$$

where P is the permutation matrix associated with the permutation $\pi = \beta^{-1}\alpha$.

The ordering $\sigma : \{1,\ldots,m\} \to P$ is defined by $\sigma(r) = Vr$ $(1 \leq r \leq m)$. If P satisfies the following properties:

i) the quotient graph $(G^*_\alpha / P)_\sigma$ is a preserving palm (i.e. $Vs \in \mathcal{D}(Vr) \to r > s$);

ii) $Vs \in \mathcal{D}(Vr) \to \alpha^{-1}(v) > \alpha^{-1}(w)$ for $v \in Vr$, $w \in Vs$,

then P will be called a *preserving partition* or *p-partition* for short.

If P is a p-partition, then β as defined above is consistent with α.

Proof: We must show

$$\forall_{(v,w) \in E \cup D_\alpha(G)} : \alpha^{-1}(v) > \alpha^{-1}(w) \not\Leftrightarrow \beta^{-1}(v) > \beta^{-1}(w) .$$

Assume that $(v,w) \in E \cup D_\alpha(G)$. If $v,w \in Vr$ then property ii) of β gives:

$$\alpha^{-1}(v) > \alpha^{-1}(w) \not\Leftrightarrow \beta^{-1}(v) > \beta^{-1}(w) .$$

If, however, $v \in Vr$, $w \in Vs$ $(r \neq s)$ then (because G_α^* / P is a palm) $Vs \in \mathcal{D}(Vr)$ or $Vr \in \mathcal{D}(Vs)$. Assuming that $Vs \in \mathcal{D}(Vr)$, we know from property ii) of a p-partition: $\alpha^{-1}(v) > \alpha^{-1}(w)$. From property i) of a p-partition we know $r > s$; hence property i) of β gives: $\beta^{-1}(v) > \beta^{-1}(w)$.
Now we have shown:

$$Vs \in \mathcal{D}(Vr) \rightarrow [\alpha^{-1}(v) > \alpha^{-1}(w) \quad \text{and} \quad \beta^{-1}(v) > \beta^{-1}(w)] .$$

In the same way we may show:

$$Vr \in \mathcal{D}(Vs) \rightarrow [\alpha^{-1}(w) > \alpha^{-1}(v) \quad \text{and} \quad \beta^{-1}(w) > \beta^{-1}(v)] .$$

Hence

$$[\alpha^{-1}(v) > \alpha^{-1}(w) \text{ and } \beta^{-1}(v) > \beta^{-1}(w)] \text{ or } [\alpha^{-1}(w) > \alpha^{-1}(v) \text{ and } \beta^{-1}(w) > \beta^{-1}(v)]$$

which is equivalent with:

$$\alpha^{-1}(v) > \alpha^{-1}(w) \not\Leftrightarrow \beta^{-1}(v) > \beta^{-1}(w) . \qquad \square$$

Because β is consistent with α we know from Lemma 3 that $\tilde{L} = PLP^t$ and $\tilde{U} = PUP^t$ are triangular factors of $\tilde{Q} = PQP^t$. The matrix $\tilde{Q}$ is in (2.1) partitioned into blocks; the matrices $\tilde{L}$ and $\tilde{U}$ may be partitioned in the same way:

$$\tilde{L} = PLP^t = \begin{pmatrix} L_{11} & & \\ \vdots & \ddots & \\ L_{m1} & \cdots & L_{mm} \end{pmatrix}$$

$$\tilde{U} = PUP^t = \begin{pmatrix} U_{11} & \cdots & U_{1m} \\ & \ddots & \vdots \\ & & U_{mm} \end{pmatrix}$$

Obviously the matrices L_{rr} and U_{rr} $(1 \le r \le m)$ are lower and upper triangular, respectively.

The connection graph of the block matrix in (2.1) is G/P. If P is a p-partition, then $(G/P)^*_\sigma = (G^*_\alpha/P)_\sigma$ is a preserving palm. Hence we may apply (1.16) to the block matrices $\tilde{Q}$, $\tilde{L}$ and $\tilde{U}$ and the ordering σ:

$$r \neq s \text{ and } L_{rs} \neq 0 \text{ iff } r \neq s \text{ and } U_{rs} \neq 0 \text{ iff } Vs \in \mathcal{D}(Vr) \ .$$

Let us furthermore consider the computation of the matrices:

$$\begin{pmatrix} L_{ss} \\ L_{s+1,s} \\ \vdots \\ L_{ms} \end{pmatrix} \quad \text{and} \quad (U_{ss}, U_{s,s+1}, \ldots, U_{sm}) \ .$$

Defining

$$\bar{Q}_{ss} = Q_{ss} - \sum_{Vt \in \mathcal{D}(Vs)} L_{st} U_{ts}$$

$$\bar{Q}_{rs} = Q_{rs} - \sum_{Vt \in \mathcal{D}(Vs)} L_{rt} U_{ts}$$

$$\bar{Q}_{sr} = Q_{sr} - \sum_{Vt \in \mathcal{D}(Vs)} L_{st} U_{tr}$$

we get from applying (1.12)

$$\begin{aligned} L_{ss} U_{ss} &= \bar{Q}_{ss} \\ L_{rs} U_{ss} &= \bar{Q}_{rs} \qquad 1 \le s \le m, \ Vr \in A(Vs) \ . \\ L_{ss} U_{sr} &= \bar{Q}_{sr} \end{aligned}$$

Writing

$$A(Vs) = \{Vs1, \ldots, Vsk\}$$

it follows from (1.2) that partial decomposition of Q_s defined by

$$Q_s = \begin{pmatrix} \bar{Q}_{ss} & \bar{Q}_{s,s1} & \cdots & \bar{Q}_{s,sk} \\ \bar{Q}_{s1,s} & & & \\ \vdots & & 0 & \\ Q_{sk,s} & & & \end{pmatrix}$$

with $\bar{Q}_{ss}$ a block pivot gives:

$$Q_s = L_s U_s + Q_s^r$$

where

$$L_s = \begin{pmatrix} L_{ss} \\ L_{s1,s} \\ \vdots \\ L_{sk,s} \end{pmatrix} \quad \text{and} \quad U_s = (U_{ss}, U_{s,s1}, \ldots, U_{s,sk}) .$$

Because $L_{rs} = U_{sr} = 0$ for $Vr \notin A(Vs)$, all non-zero coefficients of L and U can be obtained by (partial) decomposition of the matrices Q_s $(1 \leq s \leq m)$. Furthermore, the matrix Q_s and hence L_s and U_s can be computed once L_t and U_t are computed for all t with $Vt \in \mathcal{D}(Vs)$.

Summarizing: The matrices L and U may be computed as follows. Take a vertex Vs for which all matrices L_t and U_t with $Vt \in \mathcal{D}(Vs)$ have been computed and compute L_s and U_s by partial decomposition of Q_s. The computations must start with a leaf vertex, for instance (but not necessarily) V1; the computations end with the decomposition of Q_m. All non-zero coefficients of L and U are contained in the matrices L_s and U_s $(1 \leq s \leq m)$. The above process will be called decomposition of Q based on its p-partition P.

In the following sections we will show that sparse matrix algorithms to compute L and U are always equivalent with a decomposition based on a p-partition satisfying special properties.

2.2. Perfect preserving partitions

A sufficient condition for the graph G that env(L) and env(U) are dense is formulated at the end of Section 1.4. We will now formulate a condition under which the envelopes of L_s and U_s (encountered during a decomposition of Q based on a p-partition) are dense. First we will introduce the following definition. A partition $P = \{V1,\ldots,Vm\}$ of V is called *perfect* if for all $v \in Vs$ $(1 \leq s \leq m)$ with $\alpha^{-1}(v) \neq \min\{\alpha^{-1}(w) \mid w \in Vs\}$ there is a $w \in Vs$ such that

i) $\alpha^{-1}(w) < \alpha^{-1}(v)$,

ii) there is a path $v = v_0,\ldots,v_k = w$ in G with $\alpha^{-1}(v_h) < \alpha^{-1}(w)$ for $h = 1,\ldots,k-1$.

Perfect preserving partition will be abbreviated to *pp-partition*. We will now prove:

LEMMA 5: Let $P = \{V1,\ldots,Vm\}$ be a pp-partition of V. Let L_s and U_s $(1 \le s \le m)$ be obtained from a decomposition of Q based on P. The envelopes of L_s and U_s are dense.

Proof: L_s and U_s $(1 \le s \le m)$ are obtained from a partial decomposition of Q_s. Let the ordering $\beta : \{1,\ldots,|Vs|\} \to Vs$ be induced by α, i.e. $\alpha^{-1}(v) < \alpha^{-1}(w)$ iff $\beta^{-1}(v) < \beta^{-1}(w)$ for $v,w \in Vs$. To show that L_s and U_s have dense envelopes, use is made of the remark immediately after Lemma 1; it suffices to show

$$c_{Q_s}(j) < j \quad \text{and} \quad r_{Q_s}(j) < j \quad \text{for } 1 < j \le |Vs| .$$

Because Q_s is structurally symmetric, we know

$$c_{Q_s}(j) = r_{Q_s}(j) ,$$

hence it suffices to show:

(2.2) $$c_{Q_s}(j) < j \quad \text{for } 1 < j \le |Vs| .$$

Let $v \in Vs$, such that $\alpha^{-1}(v) \neq \min\{\alpha^{-1}(w) \mid w \in Vs\}$. From the definition of perfect partition, we know there exists a $w \in Vs$ such that

$$\alpha^{-1}(w) < \alpha^{-1}(v)$$

and there exists a path $v = v_0,\ldots,v_k = w$ in G with $\alpha^{-1}(v_h) < \alpha^{-1}(w)$ for $h = 1,\ldots,k-1$. From Lemma 2 we conclude $(v,w) \in E \cup D_\alpha(G)$. Herewith we have shown

(2.3) $$\forall_{v\in Vs,\, \alpha^{-1}(v)\neq\min\{\alpha^{-1}(w)\,|\,w\in Vs\}} \; \exists_{w\in Vs} \; [\alpha^{-1}(w) < \alpha^{-1}(v) \text{ and } \ell_{\alpha^{-1}(v),\alpha^{-1}(w)} \neq 0] .$$

Because the coefficients of L_s are ordered according to α, (2.3) is equivalent to:

(2.4) $\quad c_{L_s}(j) < j \quad \text{for } 1 < j \leq |Vs| \ .$

But from Section 1.2 we know that

(2.5) $\quad c_{L_s}(j) = c_{Q_s}(j)$

and from (2.4) and (2.5) we infer (2.2). □

Suppose $P = \{V1,\ldots,Vm\}$ is a pp-partition of V. The non-zero coefficients of L and U may be obtained by calculating the matrices L_s and U_s $(1 \leq s \leq m)$, where L_s and U_s are the results of a partial decomposition of Q_s. From Lemma 5 we know that all coefficients whose indices belong to $env(L_s)$ and $env(U_s)$ are non-zero. Hence a sparse matrix algorithm (in which it is explicitly determined whether a coefficient is non-zero) is not necessary to avoid arithmetical operations with non-zero coefficients during the decomposition of Q_s; an envelope algorithm suffices. To compute therefore the non-zero coefficients of L and U we only need to apply envelope algorithms to obtain the matrices L_s and U_s $(1 \leq s \leq m)$. But the decomposition of Q based on a p-partition results in the evaluation of expressions with the same non-zero terms as any sparse matrix algorithm. Hence the decomposition of Q based on a perfect p-partition using envelope algorithms results also in the evaluation of expressions with precisely the same non-zero terms as any sparse matrix algorithm to compute L and U.

Next we will show that for every undirected graph, for every structurally symmetric matrix a pp-partition exists. The trivial partition $T = \{\{\alpha(1)\},\{\alpha(2)\},\ldots,\{\alpha(n)\}\}$ could be a pp-partition with the property that every vertex of the palm G^*_α / T (except $\{\alpha(1)\}$) has precisely one successor. To administer in that case all the envelopes of Q_s $(1 \leq s \leq n)$, is essentially the same as to record for every coefficient of Q whether it is zero or not. There is then no essential difference between a sparse matrix algorithm and a decomposition based on T. Hence we will introduce proper p-partitions.

2.3. Proper pp-partitions

First we introduce the following definitions. A *palm* will be called *proper* if each vertex has either zero or more than one tree-successor. A *p-parti-*

tion P of V will be called *proper* if the palm G^*_α / P is proper.

THEOREM 1: For every non-decomposable, structurally symmetric matrix there exists a proper pp-partition.

To prove this theorem, we will first prove:

LEMMA 6: If $G = (V,E,\alpha)$ is an undirected (not necessarily connected) graph, then there exists a perfect partition $P = \{V1,\ldots,Vm\}$ of V, with the property that G^*_α / P is a forest of proper palms consistent with α, i.e. $Vs \in \mathcal{D}(Vr) \rightarrow \alpha^{-1}(v) > \alpha^{-1}(w)$ for $v \in Vr$, $w \in Vs$.

Proof: We will prove this lemma by induction with respect to n, the number of vertices in V. The lemma obviously holds for $n = 1$, because a graph consisting of only one vertex is a proper palm.
Now assume that $n > 1$. Let $x = \alpha(n)$ be the vertex with the highest number in G. Let $A(x)$ and $A^*(x)$ denote the adjacency set of $x = \alpha(n)$ in G and G^*_α respectively; let moreover H be $G(V \setminus \{x\})$, i.e. the graph obtained from G by removing $x = \alpha(n)$ and all edges connected to x. The ordering of H induced by the ordering of G will be called α again. From the induction hypothesis applied to H, the existence follows of a perfect partition $P' = \{V1,\ldots,Vm\}$ of $V \setminus \{x\}$, with the property that H^*_α / P' is a forest of proper palms consistent with α: P1,...,Pz. Let C denote the collection of palms Pj, such that the vertex set of Pj has at least one vertex in common with $A(x)$. We will distinguish three cases: $|C| = 0$, $|C| > 1$ and $|C| = 1$.

i) $|C| = 0$. Define $P = P' \cup \{x\}$. Obviously P is a perfect partition of V. The graph Pt consisting of only one vertex, viz. x, is a proper palm consistent with α. From $|C| = 0$ it follows that $A(x) = \emptyset$. Hence, from Lemma 2, $A^*(x) = \emptyset$. Therefore G^*_α / P is a forest of proper palms P1,...,Pz,Pt, consistent with α.

ii) $|C| > 1$. Define $P = P' \cup \{x\}$. Obviously P is a perfect partition of V. The collection of palms C may be considered as a (disconnected) graph. The graph Pt is obtained from C by adding $\{x\}$ to the vertex set of C; the edge set of Pt is obtained by adding $(\{x\},Vs)$ to the edge set of C for those Vs with a vertex w such that $w \in A^*(x)$. Pt is a proper palm with root $\{x\}$ and consistent with α. Obviously G^*_α / P consists of the palms Pt and Pj ($1 \leq j \leq z$, $Pj \notin C$). Hence, G^*_α / P is a forest of proper palms consistent with α.

iii) $|C| = 1$. The partition $P = P' \cup \{x\}$ does not meet the requirements, because the palm Pt as defined in the preceding case is not proper. Therefore we have to distinguish this case $|C| = 1$ from the case $|C| > 1$.

C consists of only one palm, say Ps. Let Vt be the root of Ps. Define the following partition of V: $P = \{V1,\ldots,Vt \cup \{x\},\ldots,Vm\}$. G_α^* / P consists of the palms Pj ($1 \le j \le z$, $j \neq s$) and $P\dot{s}$, where $P\dot{s}$ is obtained from Ps by joining $x = \alpha(n)$ to its root Vt. $P\dot{s}$ obviously is a proper palm consistent with α. To prove that P is a perfect partition of V, it suffices (because P' is a perfect partition of $V \setminus \{x\}$) to show that there exists a vertex $w \in Vt$ and a path $x = v_0,\ldots,v_k = y$ in G with $\alpha^{-1}(v_h) < \alpha^{-1}(y)$ for $h = 1,\ldots,k-1$. Vt is the root of the palm Ps, which is consistent with α. Let y be the highest numbered vertex of Ps, then $y \in Vt$. From the definition of C we know that the vertex set of Ps has a vertex in common with A(x). Let v_1 be such a vertex. Because v_1 and y both belong to the same connected component of H_α^* / P', there is a path $v_1,\ldots,v_k = y$ with $\alpha^{-1}(v_h) < \alpha^{-1}(y)$ for $h = 1,\ldots,k-1$ in H. Hence $x = v_0,\ldots,v_k = y$ is a path in G. □

Proof of Theorem 1: The connection graph $G = (V,E,\alpha)$ associated with a structurally symmetric, non-decomposable matrix is a connected undirected graph. From Lemma 6 the existence follows of a perfect partition $P = \{V1,\ldots,Vm\}$ of V, such that G_α^* / P is a forest of proper palms, with the property

$$Vs \in \mathcal{D}(Vr) \Rightarrow \alpha^{-1}(v) > \alpha^{-1}(w) \quad \text{for } v \in Vr,\ w \in Vs\ .$$

Because G is connected, G_α^* / P is connected; hence G_α^* / P is a palm. Let the partition elements V1,...,Vm be ordered according to a post-order traversal of the palm G_α^* / P (i.e. of every subtree in the palm, the root is visited last), and let σ denote the ordering $\sigma(s) = Vs$, then $(G_\alpha^* / P)_\sigma$ is even a preserving palm.

From the above we conclude: P is a proper perfect preserving partition. □

2.4. Construction of a proper pp-partition

The proof of Lemma 6 is constructive. A (worst case) $O(n^2)$ algorithm to construct a proper pp-partition could be derived from that proof. However, such a partition may also be obtained in still another way.

Let $P = \{V1,\ldots,Vm\}$ $(m > 1)$ be a proper pp-partition of the graph $G = (V,E,\alpha)$. It is easily verified that Vm, the root of G^*_α / P, is a separator of G, containing the vertices with the highest numbers. On the other hand, if there is no separator $S \subset V$ of G with the property:

$$\exists_{1<k\leq n} [S = \{w \in V \mid \alpha^{-1}(w) > k\}] ,$$

then $P' = \{V\}$ is a proper pp-partition of G.

Proof: G / P' is a graph consisting of one vertex only, therefore it is a proper preserving palm. That P' is moreover perfect follows from the observation that, if v would be a vertex for which no path $v = v_0,\ldots,v_k = w$ exists with $\alpha^{-1}(v_h) < \alpha^{-1}(w) < \alpha^{-1}(v)$ for $h = 1,\ldots,k-1$, then $S = \{w \in V \mid \alpha^{-1}(w) > \alpha^{-1}(v)\}$ would be a separator of G. □

The following construction results in a proper pp-partition. Let initially P be empty. Determine a minimal separator S of G with the property

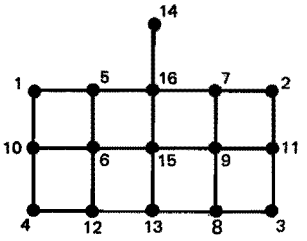
$$(2.6) \qquad \exists_{k > \min_{v\in V}\{\alpha^{-1}(v)\}} [S = \{w \in V \mid \alpha^{-1}(w) > k\}];$$

S is minimal if no other separator with the same property is contained in S. If no separator with property (2.6) exists, then set S = V. Next add S to P and apply the above recursively to each of the connected components of $G(V \setminus S)$.

With induction to the depth of G / P, it may be verified that P obtained in this way is a proper pp-partition. Using an $O(m)$ algorithm of [Tarjan '72], to determine the connected components of a graph consisting of m vertices, the above construction may be implemented ultimately resulting in an $O(n^2)$ algorithm.

2.5. Example

As an example we will use an undirected graph G, with vertex set $V = \{1,2,\ldots,16\}$ and ordering α, with $\alpha(i) = i$ $(i = 1,\ldots,16)$. The edge set of G is represented in Figure 2; an edge is drawn from vertex v to vertex w if and only if (v,w) belongs to the edge set of G.



Connection graph

Figure 2

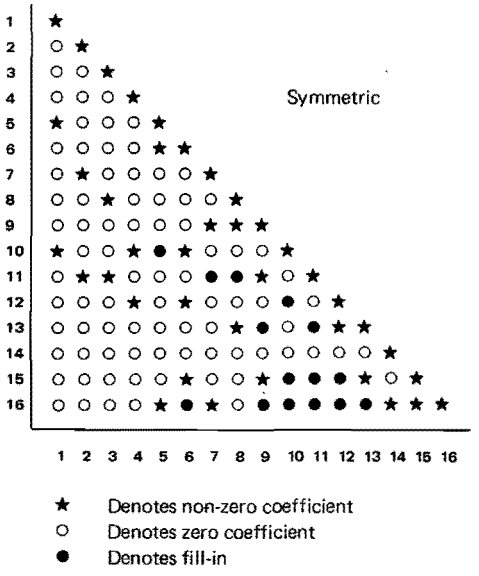


Figure 3

Figure 3 shows the zero/non-zero structure of an associated matrix. Both constructions as outlined in the preceding section result in a partition $P = \{V1,\ldots,V9\}$ with $V1 = \{2,7\}$, $V2 = \{3,8\}$, $V3 = \{9,11\}$, $V4 = \{4\}$, $V5 = \{1,5,6\}$, $V6 = \{10,12\}$, $V7 = \{13,15\}$, $V8 = \{14\}$, $V9 = \{16\}$.

The graph G_α^* / P is depicted in Figure 4; Figure 5 shows the according to P reordered matrix. The decomposition based on P proceeds as follows:

1) decompose

$$Q_1 = \begin{pmatrix} q_{2,2} & & & & \\ q_{7,2} & q_{7,7} & & & \\ 0 & q_{9,7} & 0 & & \\ q_{11,2} & 0 & 0 & 0 & \\ 0 & q_{16,7} & 0 & 0 & 0 \end{pmatrix}$$

with the first two pivots, giving

$$L_1 = \begin{pmatrix} \ell_{2,2} & \\ \ell_{7,2} & \ell_{7,7} \\ 0 & \ell_{9,7} \\ \ell_{11,2} & \ell_{11,7} \\ 0 & \ell_{16,7} \end{pmatrix} \quad \text{and} \quad Q_1^r = \begin{pmatrix} q^1_{9,9} & & \\ q^1_{11,9} & q^1_{11,11} & \\ q^1_{16,9} & q^1_{16,11} & q^1_{16,16} \end{pmatrix}$$

2) decompose, in the same way, Q_2 with the first two pivots giving L_2 and Q_2^r.

3) decompose

$$Q_3 = \begin{pmatrix} q_{9,9} + q^1_{9,9} + q^2_{9,9} & & & & \\ q_{11,9} + q^1_{11,9} + q^2_{11,9} & q_{11,11} + q^1_{11,11} + q^2_{11,11} & & & \\ q^2_{13,9} & q^2_{13,11} & 0 & & \\ q_{15,9} & 0 & 0 & 0 & \\ q^1_{16,9} & q^1_{16,11} & 0 & 0 & 0 \end{pmatrix}$$

with the first two pivots, giving

$$L_3 = \begin{pmatrix} \ell_{9,9} & \\ \ell_{11,9} & \ell_{11,11} \\ \ell_{13,9} & \ell_{13,11} \\ \ell_{15,9} & \ell_{15,11} \\ \ell_{16,9} & \ell_{16,11} \end{pmatrix} \quad \text{and} \quad Q_3^r = \begin{pmatrix} q^3_{13,13} & & \\ q^3_{15,13} & q^3_{15,15} & \\ q^3_{16,13} & q^3_{16,15} & q^3_{16,16} \end{pmatrix}$$

Note: the superscripts of the matrix coefficients refer to the step in which they were computed.

4) decompose Q_4 with its first pivot only;

5) decompose Q_5 with the first three pivots;

6) decompose Q_6 with the first two pivots;

7) decompose

$$Q_7 = \begin{pmatrix} q_{13,13} + q^2_{13,13} + q^3_{13,13} + q^6_{13,13} & & \\ q_{15,13} + q^3_{15,13} + q^6_{15,13} & q_{15,15} + q^3_{15,15} + q^5_{15,15} + q^6_{15,15} & \\ q^3_{16,13} + q^6_{16,13} & q_{16,15} + q^3_{16,15} + q^5_{16,15} + q^6_{16,15} & 0 \end{pmatrix}$$

with the first two pivots, giving

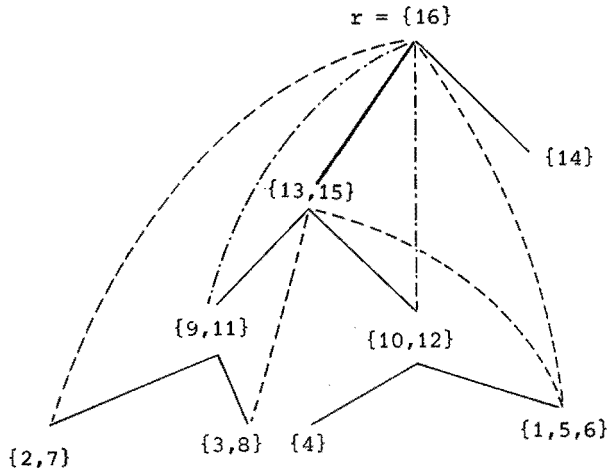
$$L_7 = \begin{pmatrix} \ell_{13,13} & \\ \ell_{15,13} & \ell_{15,15} \\ \ell_{16,13} & \ell_{16,15} \end{pmatrix} \quad \text{and} \quad Q_7^r = \begin{pmatrix} q^7_{16,16} \end{pmatrix}$$

8) decompose

$$Q_8 = \begin{pmatrix} q_{14,14} & \\ q_{16,14} & 0 \end{pmatrix}$$

with its first pivot, giving

$$L_8 = \begin{pmatrix} \ell_{14,14} \\ \ell_{16,14} \end{pmatrix} \quad \text{and} \quad Q_8^r = \begin{pmatrix} q^8_{16,16} \end{pmatrix}$$



Quotient graph of a pp-partition

r : root

——————: tree-edge

- - - - - - - - - : frond

.-.-.-.-.-.-: fill-in

Figure 4

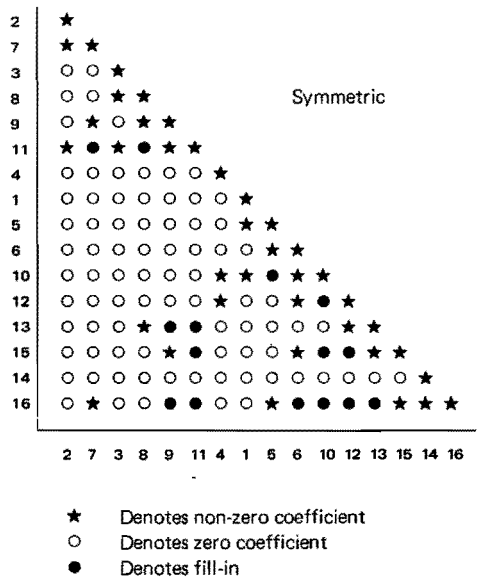


Figure 5

9) decompose

$$Q_9 = \left\{ q_{16,16} + q^1_{16,16} + q^3_{16,16} + q^5_{16,16} + q^6_{16,16} + q^7_{16,16} + q^8_{16,16} \right\}$$

with its first pivot, giving

$$L_9 = (\ell_{16,16}) .$$

The coefficients of L which are *not* computed in the above steps, for instance $\ell_{14,11}$, are zero. Note that steps 1, 2, 4, 5 and 8 are independent of each other and may be done in any order or even simultaneously. Another order of the steps corresponds with another post-order traversal through G^*_α / P.

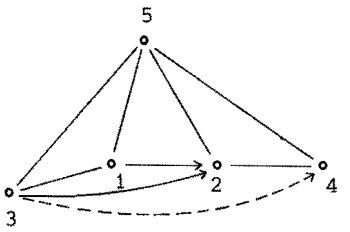
2.6. Non-symmetric equations

The preceding sections dealt with structurally symmetric matrices. If the set of equations to be solved is not structurally symmetric, then the associated connection graph is directed. Analogous, though less strong, results will be shown to hold.

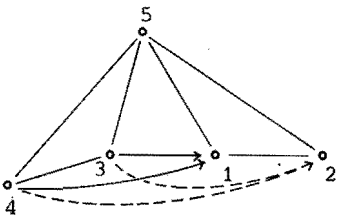
The matrix Q, with triangular decomposition Q = LU, will be assumed to be such that its connection graph G = (V,E,α) is strongly connected (otherwise we would in fact be dealing with a number of sets of equations). As in the symmetric case, we may try to find a separator S ⊂ V with the property:

$$\exists_{1<k\leq n} \left[S = \{ w \in V \mid \alpha^{-1}(w) > k \} \right] .$$

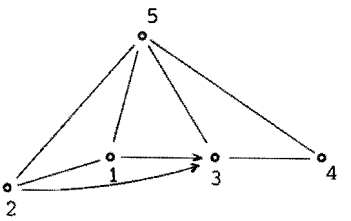
It may be proven (with Lemma 1 and a reasoning as we used in Section 2.4) that if env(L+U) contains zero elements, then such a separator exists. In the symmetric case, the variables associated with the connected components may be eliminated in arbitrary order. This is not allowed in the non-symmetric case as may be seen as follows. Consider the graph of Figure 6a with ordering α defined by: α(i) = i (i = 1,...,5). The separator S = {5} yields the strongly connected components V1 = {1,3} and V2 = {2,4}. Eliminating first the variables associated with V2 leads to an elimination order which results in triangular factors different from L and U; this may be seen in figure 6b, where the vertices are renumbered according to this new ordering. Figure 6c shows that eliminating first the variables associated with V1 also leads to other triangular factors.



6a



6b



6c : no fill-in

——————→ directed edge

———————— edge in two directions

- - - - - - - - → fill-in edge

Figure 6

To guarantee obtaining the correct triangular factors we modify the construction of a partition in the following way. A separator S, consisting of vertices with the highest numbers, is looked for with the further property that the vertices in each strongly connected component are consecutively numbered. Applying this rule recursively to each of the strongly connected components, we obtain a partitioning $P = \{V1,\ldots,Vm\}$ of V. The matrix Q may be partitioned into blocks:

$$Q = \begin{pmatrix} Q_{11} & \cdots & Q_{1m} \\ \vdots & & \vdots \\ Q_{m1} & \cdots & Q_{mm} \end{pmatrix}$$

where Q_{rs} denotes the matrix obtained from Q by deletion of all rows i for which $\alpha(i) \notin Vr$ and all columns j for which $\alpha(j) \notin Vs$. (Note that contrary to the symmetric case, the rows and columns of Q are *not* permuted to obtain the partitioned matrix.)

Let the matrices L and U be partitioned in the same way. Defining

$$\bar{Q}_{rs} = Q_{rs} - \sum_{t=1}^{\min(r,s)-1} L_{rt}\, U_{ts}$$

it follows that partial decomposition of

$$Q_s = \begin{pmatrix} \bar{Q}_{ss} & \bar{Q}_{s,s+1} & \cdots & \bar{Q}_{sm} \\ \bar{Q}_{s+1,s} & & & \\ \vdots & & 0 & \\ \bar{Q}_{ms} & & & \end{pmatrix}$$

with $\bar{Q}_{ss}$ as block-pivots gives

$$Q_s = L_s\, U_s + Q_s^r$$

where

$$L_s = \begin{pmatrix} L_{ss} \\ L_{s+1,s} \\ \vdots \\ L_{ms} \end{pmatrix} \quad \text{and} \quad U_s = (U_{ss}, U_{s,s+1}, \ldots, U_{sm})$$

(see (1.2) and formulae for partial decomposition). Hence all coefficients of L and U can be obtained by (partial) decomposition of the matrices Q_s ($1 \leq s \leq m$). The matrix Q_s and hence L_s and U_s can be computed once L_t and U_t are computed for all t with $1 \leq t < s$. Note that contrary to the symmetric case, the order of computing the matrices Q_s is strictly determined.

If the partitioning P is obtained as outlined above, it may happen that $env(L_s + U_s)$, for some s ($1 \leq s \leq m$), contains zero elements. In that case P may be refined. Let $env(L_t + U_t)$ contain a zero element. Then the section graph G(Vt) contains a separator S', with the properties that S' contains vertices with the highest number in Vt and the associated strongly connected components all contain consecutively numbered vertices. By removing Vt from P and adding S' and vertex sets of the associated strongly connected components, we get a refined partition.

Let P^* be a partition which needs not be refined any further. Then the computation of L and U may be based (in the above described way) upon P^* in such a way that envelope algorithms suffice to avoid arithmetical operations with zero coefficients.

Note that contrary to the structurally symmetric case, G^*_α / P^* is not a palm or another type of a nicely structured graph. Therefore non-symmetric matrices exist for which there is hardly any difference between sparse matrix decomposition and decomposition based on a partition constructed in the above way.

2.7. Nested dissection

In Section 2.4 we have pointed out how to construct a proper pp-partition for a given matrix Q. In this section we will show that if a so called nested dissection ordering is used to arrange the rows and columns of a matrix, then a proper pp-partition is obtained trivially. Hence, to implement nested dissection decomposition, no sparse matrix codes are needed for efficiency; envelope algorithms suffice.

The set of equations $Qw = f$ is equivalent with

$$(P_1 Q P_2^t)(P_2 w) = P_1 f$$

where P_1 and P_2 are permutation matrices. If $P_1 = P_2$, symmetry is preserved. Many papers (see [Duff '77]) are devoted to the subject of choosing the permutation matrices in such a way that some minimum or other is obtained. For instance, one may try to minimize the profile, envelope or fill-in of $P_1 Q P_2^t$. Unfortunately, these minimization problems are proven or conjectured to be NP-complete [Papadimitriou '76, Tarjan '76]. We are in agreement with the following quotation from [Tarjan '76]: "In view of the NP-completeness results, we cannot hope to solve the general problem of efficiently implementing sparse Gaussian elimination. We can only try to solve the problem for special cases."

One such special case is the class of symmetric matrices arising in two or three dimensional finite element equations. For these problems the nested dissection ordering has been developed [George and Liu '78 oct, George '73], which has been proven to be a good ordering [Lipton e.a. '79]. We will show that a proper pp-partition is trivially obtained for such an ordering.

A *nested dissection ordering* of a connected undirected graph $G = (V,E)$ is formally defined as follows [George and Liu '78 oct]. First, an algorithmic definition of a *nested dissection partition* $\mathcal{P}$ of V is:

0) Initially, set $\mathcal{P}$ empty.
1) Choose a minimal separator V' of G; if G does not contain a separator then set $V' = V$. Add V' to $\mathcal{P}$.
2) If $V \neq V'$, then apply step 1 recursively to each of the connected components of the section graph $G(V \setminus V')$.

The set $\mathcal{P} = \{V1,\ldots,Vm\}$ thus obtained is a partition of V, a nested dissection partition of V. A *rooted dissection tree* $T = (\mathcal{P},E)$ is associated with $\mathcal{P}$, where E is defined as follows. Let Wr $(1 \leq r \leq m)$ denote the edge set of the section graph separated by Vr; then $(Vr,Vs) \in E$ if and only if Vs has been chosen as the separator of one of the connected components of the section graph $G(Wr \setminus Vr)$; Vr is in that case the predecessor of Vs. An ordering $\alpha: \{1,\ldots,n\} \to V$ is called *nested dissection ordering* if it is consistent with the nested dissection partition $\mathcal{P}$ in the following way:

$$Vs \in \mathcal{D}(Vr) \to \alpha^{-1}(v) > \alpha^{-1}(w) \quad \text{for } v \in Vr,\ w \in Vs\ .$$

Note that in finding a nested dissection ordering, both a partition of V is created and a consistent ordering on it is chosen; whereas in finding a

proper pp-partition an ordering should be given beforehand, then a partition is constructed. We will now prove:

THEOREM 2: Let α be a nested dissection ordering consistent with the nested dissection partition $P = \{V1,\ldots,Vm\}$, then P is a proper pp-partition associated with α.

Proof: Let T be the dissection tree associated with P. Because T is a tree, the elements of P may be arranged in such a way that σ: $\{1,\ldots,m\} \rightarrow P$ defined by $\sigma(r) = Vr$ is an ordering of P with the property:

$$Vs \in \mathcal{D}(Vr) \rightarrow \sigma^{-1}(Vr) = r > \sigma^{-1}(Vs) = s \ .$$

It is easily verified that G_α^* / P is the tree with some fronds added to it; hence G_α^* / P is a palm and $(G_\alpha^* / P)_\sigma$ is a preserving palm. Hence P is a preserving partition. Moreover, every vertex of T has either zero or more than one successor, hence G_α^* / P is a proper palm and therefore P is a proper p-partition. From the construction of P and the connectedness of G it follows that P is a perfect partition. Hence P is a proper pp-partition associated with α. □

Corollary 1: If a given algorithm determines a nested dissection partition and ordering, then no other algorithm to find a proper pp-partition associated with this ordering is required.

Corollary 2: To implement a nested dissection decomposition of a matrix, no sparse matrix codes are needed for efficiency; envelope algorithms suffice.

It is well-known [Hoffman e.a. '73] that nested dissection applied to so called regular n × n grids may lead to an (asymptotically) optimal ordering (in the least-arithmetic or fill-in sense). In view of our Corollary 1 above, we do not support a quotation from [George '74]: "In order to actually benefit from these orderings, it is necessary to use general sparse matrix techniques". A similar remark appears in [George and Liu '78 apr]. All known implementations of nested dissection decomposition use, at least for parts of the matrix involved, sparse matrix techniques, requiring considerable storage overhead for pointers etc. [George '77]. In Chapter 4 we will show how to implement nested dissection decomposition for n × m grids without storage overhead, using the concept of substructures.

CHAPTER 3

FINITE ELEMENT EQUATIONS

In applications of the finite element method (to the solution of partial differential equations) one often encounters large, sparse sets of linear equations. Many papers (see [Duff '77]) deal with the problem of solving those finite element equations efficiently. A way to avoid large sets of equations is the use of the substructuring technique. In this chapter first the finite element method with the drawbacks of its traditional organization is outlined and next we discuss the relation between substructuring and perfect preserving partitions.

3.1. Outline of the finite element method

We will outline the finite element method only briefly, since proofs and details may be found elsewhere [Strang and Fix '73, Zienkiewicz '77].

Suppose the problem to be solved is finding the function $\tilde{w}$ which minimizes a given energy expression:

$$E(\tilde{v}) = (A\tilde{v},\tilde{v}) - 2(\tilde{f},\tilde{v}) \tag{3.1}$$

where A is a differential operator, $\tilde{v}$ an element from a function space H with domain Ω, and (.,.) denotes the inner product in that space. A finite element solution is obtained as follows: the function $\tilde{w}$ is approximated by:

$$\tilde{w}_h = \sum_{i=1}^{n} w_i \varphi_i \tag{3.2}$$

where the φ_i are certain functions in H, called *shape functions*. Next the constant coefficients w_i will be determined so that (3.2) minimizes (3.1). Substitution of (3.2) in (3.1) yields:

$$\left(A \sum_{i=1}^{n} w_i\varphi_i , \sum_{i=1}^{n} w_i\varphi_i\right) - 2\left(\tilde{f} , \sum_{i=1}^{n} w_i\varphi_i\right) =$$

$$\left(\sum_{i,j=1}^{n} w_i(A\varphi_i,\varphi_j)w_j\right) - 2 \sum_{i=1}^{n} w_i(\tilde{f},\varphi_i) \ . \tag{3.3}$$

Defining the matrix Q by: $q_{ij} = (A\varphi_j, \varphi_i)$, w as the vector consisting of the coefficients w_i and the vector f by $f_i = (\tilde{f}, \varphi_i)$, (3.3) may be written in matrix notation as:

(3.4) $\quad w^t Q w - 2 w^t f$.

If Q is a symmetric matrix, then (3.4) obtains its minimum for w satisfying:

(3.5) $\quad Qw = f$.

(If Q is not a symmeyric matrix, we may work with the symmetric matrix $Q' = \frac{1}{2}(Q + Q^t)$.)

(3.5) shows a set of n linear equations with n unknowns, which may be solved for w. In most applications Q is a positive definite matrix.

In a similar way one can even solve differential equations of quite general type: solve $\tilde{w}$, a function in the function space H, from

$$A\tilde{w} = \tilde{f}$$

where A is a differential operator. A way to obtain an approximate solution is the following. Choose a set of n test functions $\psi_i \in H$ $(i = 1,\ldots,n)$; approximate $\tilde{w}$ by a linear combination of shape functions $\varphi_i \in H$:

(3.2') $\quad \tilde{w}_h = \sum_{i=1}^{n} w_i \varphi_i$

and require

$$(A\tilde{w}_h, \psi_i) = (\tilde{f}, \psi_i) \qquad (1 \le i \le n) \ .$$

Substitution of (3.2') in the above gives

(3.3') $\quad \sum_{j=1}^{n} (A\varphi_j, \psi_i) w_j = (\tilde{f}, \psi_i) \qquad (1 \le i \le n)$.

Defining the matrix Q by: $q_{ij} = (A\varphi_j, \psi_i)$, w as the vector consisting of the coefficients w_i and the vector f by $f_i = (\tilde{f}, \psi_i)$, (3.3') may be written in matrix notation as

(3.5') $\quad Qw = f$.

Again (3.5') shows a set of n linear equations in n unknowns. These equations are not necessarily symmetric or positive definite. We will not address the question under which conditions (3.2) or (3.2') is a reliable approximation of $\tilde{w}$.

The domain Ω of H is in finite element terminology usually called *structure*. The matrix Q, respectively vector f, will be called *structure matrix*, respectively *structure vector* (in the literature often called stiffness matrix, respectively load vector).

In order to obtain the shape functions characteristic for the finite element method, Ω is divided into a finite number of *elements*. Each shape function is now chosen in such a way that its support (i.e. that part of Ω where it is different from zero) consists of a small number of elements. The choice of the elements and their shape functions is determined by the problem to be solved.

Let $\Pi e(i)$, for $1 \leq i \leq k_e$, denote the indices of the shape functions whose support includes a certain element e. The $k_e \times k_e$ matrix Q^e is defined by:

$$q^e_{ij} = (A\varphi_{\Pi e(j)}, \varphi_{\Pi e(i)})_e \qquad (1 \leq i,j \leq k_e)$$

where $(.,.)_e$ denotes the inner product restricted to e. Q^e will be called *element matrix*.

If C_e denotes the so called *connection matrix* defined by:

$$(C_e)_{ij} = \delta_{i,\Pi e(j)} \qquad (1 \leq i \leq n,\ 1 \leq j \leq k_e)$$

where δ is the Dirac delta function, then Q is obtained by "assembling" the element matrices Q^e as follows:

$$Q = \sum_e C_e Q^e C_e^t \ .$$

In an analogous way the *element vectors* f^e and structure vector f may be computed:

$$f^e_i = (\tilde{f}, \varphi_i)_e$$

$$f = \sum_e C_e f^e \ .$$

Because the support of each shape function is restricted to only a few elements, many coefficients of Q are zero. In the finite element method the shape functions are usually chosen in such a way that the coefficients w_i in (3.2) have obvious physical interpretations; they may be identified with

the values of $\tilde{w}$ or its derivatives at particular element positions, usually called *nodes*.

3.2. Traditional organization

In the actual computations the following consecutive steps are usually encountered:

1) choice of element types (i.e. shapes of elements and kind of shape functions) to be employed,
2) mesh generation: division of the structure in elements,
3) node numbering: determination of the order in which rows and columns of the structure matrix are arranged,
4 a) computation of element matrices,
   b) assembly of element matrices into the structure matrix,
   c) computation of the structure vector,
5) computation of the solution vector,
6) computation of results determined by the solution vector.

Steps 4a and 4b are usually not carried out strictly consecutively, but as follows: as soon as an element matrix is computed, it is assembled into the already partially formed structure matrix.

As steps 1, 4a and 6 depend upon the specific problem to be solved, we will in this thesis only be concerned with: mesh generation, node numbering, assembly and solution of the equations. First we will discuss some problems that are encountered when the finite element computations are organized in this traditional way.

The partitioning of a structure in elements results in a mesh for that structure. Mesh generation is an important aspect of finite element calculations. The shape and number of elements have to be chosen. Irregular structure boundaries must be approximated by element boundaries. The accuracy of the calculations depends upon certain mesh characteristics, such as the slenderness of the elements [Strang and Fix '73]. Specifying the mesh often involves much work, in particular if the mesh is irregular and contains many nodes. Therefore, so called mesh generators [Zienkiewicz and Phillips '71, Schoofs e.a. '79] have been developed, i.e. programs generating the necessary geometrical input data for finite element programs.

Another difficulty is that the numbering of the nodes, i.e. the ordering of rows and columns of the structure matrix Q affects the efficiency of the solution process [George '71]. A poor numbering results in triangular matrices L and U (defined by Q = LU), many coefficients of which are non-zero, whereas a good numbering results in much sparser L and U, thus reducing the required number of arithmetical operations to solve the associated set of equations. Several algorithms have been developed to minimize the bandwidth of a given matrix by rearranging rows and columns [Cuthill '72]. Unfortunately the determination of the minimum bandwidth turns out to be an inherently hard problem, because it belongs to the class of so called NP-complete problems [Papadimitriou '76]. Instead of minimizing the bandwidth, it is usually better to try to minimize the profile [George '71]. This is however also an NP-complete problem [Garey e.a. '74]. One might also consider the fill-in as a measure of efficiency. However, it has been shown that minimizing the fill-in is again an NP-complete problem for non-symmetric matrices and the same is conjectured to be true for symmetric matrices [Rose and Tarjan '75].

A final difficulty is that in practice special measures are required to assemble Q and f, in particular when, for problems with many nodes, the matrices are too large for integral storage in central memory and the operating system of the computer does not provide a virtual memory (see further [Irons '70]).

3.3. Substructuring

A sensible way to solve a problem in general is by dividing it into a number of simpler problems and then combining the solutions of those simpler problems to get the solution of the overall problem. To split a finite element problem, the substructuring technique [Przemieniecki '68, Williams '73] has been applied. Because our organization of finite element calculations is based upon the substructuring technique, we will describe that technique and prove that it leads to correct results.

Let Q and f be the structure matrix and structure vector, respectively, associated with the structure S. Let S consist of elements with associated element matrices Q^e and element vectors f^e. Then the following relations

hold (see 3.1)

$$Q = \sum_e C_e Q^e C_e^t , \quad f = \sum_e C_e f^e$$

where C_e are the connection matrices associated with the elements. Let w denote the solution vector to be solved from

$$Qw = f .$$

When applying the substructuring technique to compute w one proceeds as follows. The structure S is divided into k *substructures* S1,...,Sk; i.e. every element of S belongs to precisely one substructure (the prefix sub will often be omitted). Assembling the elements of Sj (j = 1,...,k) results in the substructure matrix:

$$Q_{Sj} = \sum_{e \text{ in } Sj} C_e^j Q^e (C_e^j)^t$$

and substructure vector

$$f_{Sj} = \sum_{e \text{ in } Sj} C_e^j f^e$$

where the connection matrices C_e^j are obtained from C_e by deleting the rows which do not correspond with a node in Sj. All nodes of a substructure which belong to only that substructure are called *internal nodes*; other nodes are called *external*. The equations associated with structure Sj

$$Q_{Sj} w_{Sj} = f_{Sj} \tag{3.6}$$

may be arranged in such a way that the external nodes are grouped together:

$$\begin{pmatrix} Q_{Ij\,Ij} & Q_{Ij\,Ej} \\ Q_{Ej\,Ij} & Q_{Ej\,Ej} \end{pmatrix} \begin{pmatrix} w_{Ij} \\ w_{Ej} \end{pmatrix} = \begin{pmatrix} f_{Ij} \\ f_{Ej} \end{pmatrix} \tag{3.7}$$

the subscripts Ij and Ej referring to the internal and external nodes of Sj, respectively. Partial elimination of (3.7) with $Q_{Ij\,Ij}$ as block-pivot results in:

$$(Q_{Ej\,Ej} - L_{Ej\,Ij} U_{Ij\,Ej}) w_{Ej} = f_{Ej} - L_{Ej\,Ij} L_{Ij\,Ij}^{-1} f_{Ij} . \tag{3.8}$$

The matrix

$$Q_{Sj}^r = Q_{Ej\,Ej} - L_{Ej\,Ij} U_{Ij\,Ej} \tag{3.9}$$

will be called *reduced structure matrix*; Q^r_{Sj} is said to be obtained from Q_{Sj} by eliminating the internal nodes. The *reduced structure vector* is defined by

$$f^r_{Sj} = f_{Ej} - L_{Ej\,Ij} L^{-1}_{Ij\,Ij} f_{Ij} \quad . \tag{3.10}$$

Now suppose Ej ($1 \le j \le k$) contains m_j nodes, numbered from 1 to m_j; and suppose $\bigcup_{j=1}^{k} E_j$ contains n' different nodes, numbered from 1 to n'. Let for a node ℓ from Ej ($1 \le \ell \le m_j$) $\Pi j(\ell)$ denote the number of ℓ in $\bigcup_{j=1}^{k} E_j$, hence $1 \le \Pi j(\ell) \le n'$. Let the connection matrix Cj ($1 \le j \le k$) be defined by

$$(C_j)_{h\ell} = \delta_{h,\Pi j(\ell)} \quad , \quad 1 \le h \le n', \; 1 \le \ell \le m_j \quad .$$

The sets (3.6) may not be solved independently of each other; an external node occurring in, say, Sj and Sℓ ($j \neq \ell$) must have the same value in both substructures. More formally: the sets (3.6) must be solved subjected to the condition:

$$\exists_{w'} \, \forall_j \, [w_{Ej} = C^t_j w'] \quad . \tag{3.11}$$

Substitution of (3.9), (3.10) and (3.11) in (3.8) yields:

$$Q^r_{Sj} C^t_j w' = f^r_{Sj} \quad \text{for all j: } 1 \le j \le k \quad .$$

Hence:

$$\left(\sum_{j=1}^{k} C_j Q^r_{Sj} C^t_j \right) w' = \sum_{j=1}^{k} C_j f^r_{Sj} \quad . \tag{3.12}$$

From (3.12) it follows that w' can be solved after the reduced structure matrices and vectors have been assembled. If w' is known, then with (3.11) w_{Ej} may be computed for all j ($1 \le j \le k$). Finally, w_{Ij} can be obtained by solving:

$$U_{Ij\,Ij} w_{Ij} = L^{-1}_{Ij\,Ij} f_{Ij} - U_{Ij\,Ej} w_{Ej} \tag{3.13}$$

(this last equation follows from (3.7) and the definition of partial decomposition).

Substructuring may also be viewed in another way. If instead of the reduced structure matrices, the element matrices were assembled together, the

resulting set of equations could have been arranged in the following way:

$$
\begin{pmatrix}
Q_{I1\,I1} & & & Q_{I1\,E1}\,C_1^t \\
& \ddots & 0 & \vdots \\
0 & & Q_{Ik\,Ik} & Q_{Ik\,Ek}\,C_k^t \\
C_1\,Q_{E1\,I1} & \cdots & C_k\,Q_{Ek\,Ik} & \sum_{j=1}^{k} C_j\,Q_{Ej\,Ej}\,C_j^t
\end{pmatrix}
\begin{pmatrix} w_{I1} \\ \vdots \\ w_{Ik} \\ w' \end{pmatrix}
=
\begin{pmatrix} f_{I1} \\ \vdots \\ f_{Ik} \\ \sum_{j=1}^{k} C_j\,f_{Ej} \end{pmatrix}
\tag{3.14}
$$

Partial elimination of (3.14) with

$$
\begin{pmatrix}
Q_{I1\,I1} & & \\
& \ddots & 0 \\
0 & & Q_{Ik\,Ik}
\end{pmatrix}
$$

as block pivot, results in

$$
\left(\sum_{j=1}^{k} C_j\,Q_{Ej\,Ej}\,C_j^t - \sum_{j=1}^{k} C_j\,L_{Ej\,Ij}\,U_{Ij\,Ej}\,C_j^t\right)w' =
\tag{3.15}
$$

$$
= \sum_{j=1}^{k} C_j\,f_{Ej} - \sum_{j=1}^{k} C_j\,L_{Ej\,Ij}\,L_{Ij\,Ij}^{-1}\,f_{Ij}\ .
$$

From (3.9) and (3.10) it follows that (3.15) are precisely the equations (3.12).

The partitioning of the structure S into substructures implies a certain ordering of the nodes: the internal nodes of the substructures are eliminated first, the nodes belonging to more than one substructure are eliminated last. The order of processing the substructures, i.e. computing the reduced structure matrices and vectors, is irrelevant.

If a substructure, say Sj, consists of a number of elements, it is possible to partition also that set of elements. The substructure Sj is then partitioned into substructures $S_1^j,\ldots,S_\ell^j$. To compute the reduced structure matrix of Sj, instead of element matrices, reduced structure matrices associated with S_h^j $(1 \le h \le \ell)$ are then assembled together, and so on. Thus one may create a hierarchy of substructures.

Some advantages of using the substructuring technique are:

- Instead of assembling all element matrices into one large, usually sparse, structure matrix, a number of smaller matrices is set up.
- Sometimes identical substructures can be distinguished within a structure; obviously the calculations of identical reduced structure matrices need to be done only once, thus saving computations.
- Substructures may be analyzed more or less independently of each other; the effect - on the complete structure - of changes of a substructure may be analyzed without a need to repeat all computations.

As remarked before, substructuring implies a certain ordering of the nodes. The order of the nodes influences the number of arithmetical operations with non-zero matrix coefficients required for the elimination. Hence a possible danger of the substructuring technique is that an inappropriate partitioning may lead to far more than the minimum number of arithmetical operations with non-zero coefficients. With a careful choice of the substructures, however, this danger can be avoided. This may be seen as follows.

Consider a structure S with structure matrix Q and ordering α. Let $G = (V,E,\alpha)$ be the associated connection graph, where V is the set of nodes of S. Suppose $P = V1,\ldots,Vk$ $(k > 1)$ is a proper pp-partition associated with G. Hence Vk is a separator of G; let the connected components of the section graph $G(V \setminus Vk)$ be denoted by $G1,\ldots,Gp$. All coefficients of the element matrices are considered to be structurally non-zero. Hence two nodes (not in Vk) belonging to the same element of S belong to the same connected component Gj. With each component Gj $(1 \leq j \leq p)$ a substructure Sj of S is associated in the following way: all elements of S which have a node in common with Gj together form substructure Sj. The nodes of Sj which belong to Vk are precisely the external nodes of Sj. By applying the above rule we do not necessarily obtain a complete partitioning of all elements; there may be elements, all nodes of which occur in Vk only. Each of these elements may be considered as a separate substructure; they are substructures consisting of one element only and without internal nodes. In the same way as S, its substructures Sj may be partitioned; thus with P a hierarchy of substructures is associated. Computing reduced structure matrices, corresponds with partial decomposition of matrices Q_S as indicated in Section 2.1. From Section 2.3 we know that - irrespective of how well

the equations are ordered - we can find a proper pp-partition. From the above it now follows that, if the ordering of the equations is such that it gives rise to the minimum amount of arithmetical operations and/or storage, substructures can be found which lead to the same minimum. Moreover, to avoid operations with zero coefficients envelope algorithms suffice.

Now, conversely, suppose the structure S, with structure matrix Q and associated unordered connection graph G = (V,E), is partitioned into a hierarchy of substructures. With such a hierarchy a partitioning $P = \{V1,\ldots,Vk\}$ of the nodes V is associated: two nodes belong to the same partition element Vj if and only if they are internal nodes in the same substructure. The internal nodes can be ordered in such a way that the associated $env(L_{IjIj})$ is dense. Let α denote an ordering of V, thus induced by the hierarchy of substructures. It is now easily verified that P is a proper pp-partition for the graph $G = (V,E,\alpha)$. Hence it is not necessary to apply the algorithm of Section 2.4 in order to obtain a proper pp-partition associated with α.

CHAPTER 4

A NOVEL FINITE ELEMENT ALGORITHM FOR N × M GRIDS

In this chapter we will develop an algorithm for finite element calculations on a structure with an n × m mesh, that is to say: the structure consists of n rows and m columns of quadrilateral elements, each element comprising four corner nodes. The most simple example of such a structure is a rectangular plate which, in an obvious way, is partitioned into n × m uniform rectangular elements (i.e. all elements are of the same size). Note, however, that by allowing arbitrary quadrilateral elements, the structure is not necessarily a rectangular plate. Deformations are permitted. It will be assumed that, as far as numerical stability is concerned, the pivot-order of the associated set of equations is not of importance. We will moreover assume that the element matrices are symmetric.

Our approach will be based upon the substructuring technique. The obvious way to dissect a rectangle is to divide it into two rectangles of about equal size. Hence reduced structure matrices and vectors must then be computed, associated with each of the two smaller rectangles. To compute the reduced structure matrices, every substructure will in turn be divided into two (smaller) rectangles, unless the substructure is too small to be dissected, i.e. consists of only one element.

The implementation of this algorithm and the data structures involved will be described with the programming language PASCAL [Jensen and Wirth '78]. A step-wise refinement approach [Wirth '71 comm.] will be employed to clarify the procedures developed.

4.1. Procedure *ur*

The procedure *ur* produces the reduced structure matrix of a structure R provided with an n × m mesh. A corner node of R is a node belonging to precisely one element, hence R has four corner nodes. In an obvious way four sides of R may be distinguished; they will be identified by *left*, *upper*, *right* and *lower*. Only nodes on the sides of R are external. A side

will be called external if it contains external nodes only. Not all sides of R are necessarily external. On a non-external side at most the corner nodes are external.

The procedure heading of *ur* is

```
procedure ur (n, m: integer; ext: set-of-sides);
```

The type *set-of-sides* is defined as follows:

```
type set-of-sides = set of (left, upper, right, lower)
```

The value parameter *ext* refers to the external sides of R. The parameters *n* and *m* refer to the number of elements; their values equal the number of rows and columns, respectively. The body of *ur* is:

```
1.  procedure ur (n, m: integer; ext: set-of-sides);
2.  var n1, m1, n2, m2: integer; e1, e2: set-of sides;
        {ni, mi are the number of elements along the sides of
        substructure i (i = 1,2); ei denotes the external sides}
3.  begin if (n = 1) and (m = 1)
4.        then compute-element
5.        else begin divide R into two rectangles R1 and R2, i.e.
                     compute n1, m1, n2, m2, e1, e2;
6.                   ur (n1, m1, e1);
7.                   ur (n2, m2, e2);
8.                   assemble
                end;
9.        decompose
    end
```

The procedure *assemble* performs the assembly of the two (reduced) structure matrices computed in lines 6 and 7. The procedure *decompose* performs the partial decomposition of either the structure matrix assembled in line 8, or the element matrix computed in line 4. This decomposition results in the reduced structure matrix Q^r and the decomposed matrices L_{II} and L_{EI}, where E denotes the external nodes of R and I stands for the internal nodes eliminated by the last call of *decompose*; hence I are the nodes which are internal in R and external in R1 and R2; I does *not* denote all the internal nodes of R.

The correctness of *ur*, i.e. upon its termination Q^r is correctly computed, is easily verified by induction [Dijkstra '72], taking the correctness of lines 4, 5, 8 and 9 for granted. It is trivial to verify that the execution of *ur* will always terminate.

4.1.1. Element specification and storage of results

In order to compute in line 4 an element matrix, it must be known which element is meant. Therefore we provide *ur* with the further parameters i and j, the row and column number of the lower left element of the structure whose reduced structure matrix must be computed.

Another point where we want to be a bit more specific is the way in which the resulting matrix is recorded. For reasons to be explained in Chapter 7, all data are stored in a global one-dimensional array A of sufficient length. All that is necessary to retrieve data are their indices in A. Therefore we include a variable parameter r in *ur*, whose value upon exit of the procedure is the position in A from where the computed results may be obtained.

With these extensions the declaration of *ur* becomes:

```
 1.  procedure ur (n, m, i, j: integer; ext: set-of-sides;
 2.                    var r: integer);
 3.  var n1, m1, i1, j1, n2, m2, i2, j2: integer;
 4.      e1, e2: set-of-sides;
 5.      r1, r2: integer;
         {ri is the index in A where data of substructure i may be
          retrieved (i = 1,2)}
 6.  begin assign value to r {initialize datastructure};
 7.        if (n = 1) and (m = 1)
 8.        then compute-element (i, j, r)
 9.        else begin compute n1, m1, i1, j1, n2, m2, i2, j2, e1, e2;
10.                   ur (n1, m1, i1, j1, e1, r1);
11.                   ur (n2, m2, i2, j2, e2, r2);
12.                   assemble
                  end;
13.       decompose
      end
```

4.1.2. Dissection of the rectangle and representation of element matrices

In line 9 the rectangular mesh is dissected into two smaller rectangular meshes. The way of dissecting affects the efficiency of the computations. Following intuition, it seems advisable to dissect the rectangle along a line roughly through the middles of the two long sides (that are the sides with the largest number of elements). Therefore line 9 becomes:

```
if n > m
then begin n1 := n div 2;          n2 := n - n1;
           m1 := m;                m2 := m;
           i1 := i;                i2 := i + n1;
           j1 := j;                j2 := j;
           e1 := ext + [upper];    e2 := ext + [lower]
     end
else begin n1 := n;                n2 := n;
           m1 := m div 2;          m2 := m - m1;
           i1 := i;                i2 := i;
           j1 := j;                j2 := j + m1;
           e1 := ext + [right];    e2 := ext + [left]
     end
```

Because the parameters to be passed in lines 10 and 11 are all called by value (except *r1* and *r2*), we may recode lines 10-12 without using the intermediate variables declared in lines 3 and 4. To shorten the code a variable *nm1* will be introduced to avoid repeated evaluation of *n* div 2 or *m* div 2.

As stated before, we will not be concerned with the computation of element matrices. Therefore we assume that the procedure *compute-element* will be supplied by others. An element matrix computed by *compute-element* will subsequently be (partially) decomposed by *decompose*; hence the representation of an element matrix is determined by the specifications of the procedure *decompose* following. In order to free the writer of *compute-element* from the necessity to be aware of the special representation required by *decompose*, we will introduce an auxiliary 2-dimensional array *Q*, to be passed as a parameter to *compute-element*. After a call of *compute-element* *Q* will then represent the element matrix in the usual way: the lower triangular part stored rowwise. Next the contents of *Q* must be

transferred to *A*. Due to restrictions imposed by PASCAL it is not possible to declare *Q* locally (in line 8); therefore *Q* is assumed to be declared globally.

With these modifications the procedure declaration becomes:

```
 1.  procedure ur (n, m, i, j: integer; ext: set-of-sides;
 2.                var r: integer);
 3.  var nm1, r1, r2: integer;
 4.  begin assign value to r;
 5.        if (n = 1) and (m = 1)
 6.        then begin compute-element (i, j, Q);
 7.                   transfer contents of Q to A
                end
 8.        else begin if n > m
 9.                   then begin nm1 := n div 2;
10.                              ur (nm1, m, i, j, ext + [upper], r1);
11.                              ur (n - nm1, m, i + nm1, j, ext + [lower], r2)
                         end
12.                  else begin nm1 := m div 2;
13.                              ur (n, nm1, i, j, ext + [right], r1);
14.                              ur (n, m - nm1, i, j + nm1, ext + [left], r2)
                         end;
15                   assemble
                end;
16.       decompose
    end
```

4.1.3. *Decompose*

Given an assembled structure matrix, the procedure *decompose* computes a reduced structure matrix, whereas the procedure *assemble* has to assemble two given reduced structure matrices into one matrix. Hence, the procedures *assemble* and *decompose* are closely related; there is a trade-off in the share of work that has to be performed by *decompose* and *assemble*. Because *assemble* has a kind of bookkeeping function, we have tried to minimize the work to be done by *decompose*. Therefore the lower triangular part of the matrix to be decomposed is partitioned into two matrices: Q_{EE} (the part

that corresponds with external nodes only) and the rest. (Leaving the internal rows and columns intermixed with the external ones, would, on the other hand, simplify the procedure *assemble*.)

As mentioned before, *decompose* decomposes a given structure matrix

$$Q = \begin{pmatrix} Q_{II} & Q_{EI}^t \\ Q_{EI} & Q_{EE} \end{pmatrix}$$

with Q_{II} as block-pivot into

$$\begin{pmatrix} L_{II} & 0 \\ L_{EI} & Q^r \end{pmatrix} .$$

Let *ni* and *ne* denote the number of internal and external nodes, respectively, and let

q_{ij}	denote	$(Q_{II})_{ij}$	for $1 \le j \le i \le ni$,
ℓ_{ij}	denote	$(L_{II})_{ij}$	for $1 \le j \le i \le ni$,
q_{ij}	denote	$(Q_{EI})_{i-ni,j}$	for $1 \le j \le ni$, $ni+1 \le i \le ni+ne$,
ℓ_{ij}	denote	$(L_{EI})_{ij}$	for $1 \le j \le ni$, $ni+1 \le i \le ni+ne$,
q_{ij}	denote	$(Q_{EE})_{i-ni,j-ni}$	for $ni+1 \le j \le i \le ni+ne$,
q^r_{ij}	denote	$(Q^r)_{ij}$	for $1 \le j \le i \le ne$,

then the procedure body of *decompose* may be coded as follows (see (1.2*)):

```
var i, j, k: integer; h: real;
begin {computation of (lower triangular part of) L_II and L_EI}
      for i := 1 to ni + ne do
      begin for j := 1 to min(i,ni) do
            begin h := q_ij;
                  for k := 1 to j - 1 do h := h - l_ik * l_jk;
                  if j < i
                  then l_ij := h/l_ij
                  else l_ii := sqrt(h)
            end
      end;
      {computation of Q^r = Q_EE - L_EI L_EI^t}
      for i := 1 to ne do
      for j := 1 to i do
      begin h := q_i+ni,j+ni;
            for k := 1 to ni do h := h - l_i+ni,k * l_j+ni,k;
            q^r_ij := h
      end
end
```

Remark: In the actual implementation the space occupied by Q_{II}, Q_{EI} and Q_{EE} will be overwritten by L_{II}, L_{EI} and Q^r, respectively.

4.1.4. *Assemble*

The procedure *assemble* must assemble two matrices, say Q^r_{R1} and Q^r_{R2}, into the structure matrix Q associated with R. A way to do this, is (see 3.12)):

```
procedure assemble;
begin initialize Q with zeros;
      add C_1 Q^r_R1 C_1^t to Q;
      add C_2 Q^r_R2 C_2^t to Q
end
```

where C_i (i = 1,2) is the connection matrix associated with Ri. To compute C_1 and C_2, it is necessary to know how the nodes are numbered. The program as listed in [Peters '79] follows the (rather arbitrary) convention: the external nodes are ordered clockwise, starting with the lower left corner

node; the internal nodes are numbered either from left to right or from top to bottom, depending upon the way R is split into R1 and R2. Of course not the matrices C_1 and C_2 are computed, but instead two vectors *v1* and *v2* (with a length equal to the number of externals of R1 and R2, respectively) defined by:

v1[h] = ℓ iff the h^{th} external node of R1 is the ℓ^{th} external (if ℓ > 0) or internal (if ℓ < 0) node of R.

An analogous definition holds for *v2*. The distinction between internal and external nodes of R is made because Q is partitioned in Q_{II}, Q_{EI} and Q_{EE}.

The values of *n*, *m* and *ext* are needed to compute *v1* and *v2*. The coding of the computation of *v1* and *v2* is straightforward, requiring an extensive case analysis. After assembly the vectors *v1* and *v2* may be deleted. Further details may be found in [Peters '79].

4.1.5. Removal of reduced structure matrices

A reduced structure matrix is, once it is assembled to another matrix, not needed any longer. The space it occupies in the global array *A* may then be used for other data. The array *A* will therefore be used in a stack-like manner. If the structure R, partitioned into all its substructures, is viewed as a tree, then the reduced structure matrices are stored consecutively in *A* in pre-order [Kunth '75], i.e. of every substructure Rj, first its "own" reduced structure matrix is stored, then all the reduced structure matrices associated with its first substructure and next all those matrices of the second substructure.

A global variable *pr* is introduced, indicating from which index in *A* on the next reduced structure matrices may be stored. From the parameters *n*, *m* and *ext* it can be determined precisely how much space is needed to store the reduced structure matrix of R. Hence the value needed to update *pr* is known and in line 4 of the program text in 4.1.2 the old value of *pr* is assigned to *r*. If every reduced structure matrix is removed as soon as it is not needed anymore (and if no other data are stored in *A*), then the pre-order storage implies that the two reduced structure matrices, which are assembled together in line 15 are always the two last ones and removing them is simply achieved by updating *pr*.

Because the decomposed matrices L_{II} and L_{EI} will be used for subsequent computations, they will not be removed; they (and hence Q_{II} and Q_{EI}) are stored in the same pre-order, but separated from the reduced structure matrices in another part of *A*. A second variable *pd* is then required to indicate from which index in *A* on the next decomposed matrices may be stored. Note that an extra parameter in *ur* to indicate where the decomposed matrices are stored is not needed.

4.2. Procedures *fur* and *bur*

4.2.1. Computation of reduced structure vector

The computation of the reduced structure vector with the procedure *fur* is analogous to the computation of the reduced structure matrix with *ur*:

```
 1. procedure fur (n, m, i, j: integer; ext: set-of-sides;
 2.                var r: integer);
 3. var nm1, r1, r2: integer;
 4. begin assign value ro r;
 5.       if (n = 1) and (m = 1)
 6.       then begin compute-element-vector (i, j, F);
 7.                  transfer contents of F to A
               end
 8.       else begin if n > m
 9.                  then begin nm1 := n div 2;
10.                             fur (nm1, m, i, j, ext + [upper], r1);
11.                             fur (n-nm1, m, i+nm1, j, ext + [lower], r2)
                           end
12.                  else begin nm1 := m div 2;
13.                             fur (n, nm1, i, j, ext + [right], r1);
14.                             fur (n, m-nm1, i, j+nm1, ext + [left], r2)
                           end;
15.                  assemble-structure vectors
               end;
16.       forward-substitution
    end
```

All parameters and variables play the same role as the corresponding ones in *ur*. Only the procedure *forward-substitution* and the parameter *r* require some comments.

This procedure *forward-substitution* computes the reduced structure vector (see (3.10)) by calculating $h_I = L_{II}^{-1} f_I$ and next subtracting $L_{EI} h_I$ from f_E. Because h_I is needed again for the computation of the solution vector (see (3.13)), f_I will be overwritten by h_I, called the *substituted structure vector*. The matrices L_{EI} and L_{II} are computed with the procedure *ur*. The execution of *ur* must then precede the first call of *fur* and the array *A* must be global to both procedures. *fur* needs access to both matrices and vectors. This is simply achieved (without having to extend the parameter list) by storing vectors and matrices together: the storage locations in *A* immediately succeeding those of L_{II} will be used to store f_I and later h_I. Prior to the first call of *fur*, the value of the global variable *pd* (associated with *A*) is reset to the value it had immediately before the initial call of *ur*. Hence the meaning of *pd* is slightly changed; it no longer indicates which part of *A* is free, but to which part of *A* the computations are advanced.

As is the case with the reduced structure matrices, also the reduced structure vectors may be removed (overwritten) as soon as they have been used in *assemble-structure-vectors*.

From the observation that the matrices needed in line 16 of *fur* are precisely those computed in line 16 of *ur* it is clear, that both procedures may be combined to form one procedure. If line 4 of *ur* is changed into

```
r := pr; pr := pr + 'expression';
```

where the value of 'expression' is the number of storage locations needed for Q_{EE}^r and f_E^r, and if, moreover, lines 6, 7, 15 and 16 of *fur* are appended to the corresponding lines of *ur*, then the resulting procedure computes correctly in an interleaved way both the reduced structure matrix and the reduced structure vector.

4.2.2. Computation of solution vector and derived results

In the foregoing we have seen how the procedures *ur* and *fur* partition the structure R into substructures and compute the decomposed structure matrices and substituted structure vectors associated with those substructures. Given those decomposed structure matrices and substituted structure vectors (represented in a global array *A* as indicated before), the following procedure *bur* provides the solution vector associated with R. Moreover, *bur* provides (application dependent) quantities for the (structure) elements; these quantities are obtained from the solution vector.

```
1.  procedure bur (n, m, i, j: integer; ext: set-of-sides;
2.                 pwe: integer; var d: integer);
3.  var nml, pwl, pw2: integer;
4.  begin solve; assign value to d;
5.        if (n = 1) and (m = 1)
6.        then begin transfer data from A to F;
7.                   process-solution (i, j, F)
              end
8.        else begin if n > m
9.                   then begin nml := n div 2; separate;
                                {assigns value to pwl and pw2}
10.                             bur (nml, m, i, j, ext + [upper], pwl, d);
11.                             bur (n-nml, m, i+nml, j, ext + [lower], pw2, d)
                           end
12.                  else begin nml := m div 2; separate;
                                {assigns value to pwl and pw2}
13.                             bur (n, nml, i, j, ext + [right], pwl, d);
14.                             bur (n, m-nml, i, j+nml, ext + [left], pw 2, d)
                           end
              end
    end
```

The parameters and other variables of *bur* play the same role as the corresponding ones in *ur* and *fur*. The value parameter *pwe* indicates where the values of w_E (see (3.13)) will be found in the global array *A*. For a structure without external nodes, the value of *pwe* is irrelevant. Upon each entry of the procedure, the value of *d* must indicate where the decomposed

matrices L_{II} and L_{EI} are stored; an adjustment of d with the number of storage locations needed for L_{II}, L_{EI} and h_I, assures, due to the storage sequence of these matrices, that d then indicates the decomposed matrices to be processed next. A simple induction argument shows that in this way d has also the correct value for the recursive call in line 11 or 14.

The procedure *solve* computes w_I (see (3.13)) as follows: first $U_{IE} w_E = L_{EI}^t w_E$ is subtracted from h_I giving h_I' (in an actual implementation h_I may be overwritten by h_I'), next w_I is computed from h_I' by back substitution with L_{II}^t (and again h_I' is overwritten by w_I). Obviously, if there are no external nodes, then $h_I' = h_I$.

In line 6, the solution vector of the element is transferred from A to an auxiliary global array F. The procedure *process-solution* next provides the quantities to be derived from that solution vector (usually geometrical data like node coordinates are required). Just like the procedures *compute-element* and *compute-element-vector* in *ur* and *fur*, also *process-solution* depends upon the specific problem to be solved and has to be written by those working on an application.

The procedures *separate* in lines 9 and 12 perform the opposite of *assemble-structure-vectors* in *fur*. The vectors w_E and w_I, as computed in line 4, are split into two vectors w_{E1} and w_{E2}, which are the solutions for the external nodes of the two substructures of R. The variables *pw1* and *pw2* denote the locations in A where the vectors w_{E1} and w_{E2} may be found.

Although the procedure *bur* has precisely the same control structure as *ur* and *fur*, it can *not* be combined with them to form one procedure, because the substructures are processed in opposite order. If the structure R is viewed as the root of a tree, of which its substructures are the nodes, then *ur* and *fur* process the substructures in post-order, whereas *bur* processes them in pre-order. Moreover, *bur* needs the results of *ur* and *fur*. Hence decomposition and forward substitution cannot be interleaved with backward substitution, they must be carried out consecutively.

CHAPTER 5

EFFICIENCY OF *UR*

In this chapter we present some operation and storage counts for the procedure *ur*, as developed in the preceding chapter. It will be shown how the storage requirements of *ur* may be reduced. If applied to a square $\ell \times \ell$ grid, then in total $2.8\,\ell^2 + O(\ell)$ storage locations are required.

5.1. Storage and operation counts

In this section we will investigate the efficiency of the procedures developed in the preceding chapter. The efficiency can be expressed in terms of the amount of storage required and processor time needed. These two quantities, however, depend upon the specific implementation on a particular computer. Therefore, the - implementation independent - number of arithmetical operations with matrix coefficients and the number of matrix coefficients stored will be considered.

If i and e denote the number of internal, respectively external nodes of the structure to be analyzed with *ur*, then in line 4 of *ur* in Section 4.1.2 space is reserved for:

$$s(i,e) = \tfrac{1}{2}i(i+1) + ie \tag{5.1}$$

coefficients of the decomposed matrices. Space needed for the reduced structure matrices will be dealt with separately. The procedure *decompose* in line 16 requires

$$t(i,e) = \frac{1}{6}i^3 + \frac{1}{2}i^2(e+1) + \frac{1}{2}ie(e+2) + \frac{1}{3}i \tag{5.2}$$

multiplicative operations (i.e. multiplications, divisions and square roots) with matrix coefficients. The number of additive operations is about the same, therefore we will not consider them.

In the procedure *fur* space is reserved for i coefficients of the substituted structure vector and $\tfrac{1}{2}i(i+1) + ie$ multiplicative operations are performed by *forward-substitution* (see also (3.10)). These numbers are

small compared with (5.1) and (5.2), respectively, therefore we will restrict our attention to *ur*.

To determine the total number of multiplicative operations carried out and matrix coefficients stored by *ur*, we will first restrict ourselves to the case

$$n = m = 2^r = \ell \qquad (r \geq 0) .$$

Let $g_t(\ell)$ denote the number of multiplicative operations for an $\ell \times \ell$ mesh with t adjacent external sides (t = 0,1,2,3,4). Let $g'_t(\ell)$ denote the number of operations for an $\ell \times \frac{1}{2}\ell$ mesh, where g'_1, respectively g'_3, is associated with a mesh, a long respectively one short side of which is external. Similarly, g''_3 is the same function for an $\ell \times \frac{1}{2}\ell$ mesh, all sides of which are external, except for a long one; g''_2 belongs to an $\ell \times \frac{1}{2}\ell$ mesh, the two long sides of which are external. With these definition we have the following set of recursive relations:

$$
\begin{aligned}
g_0(\ell) &= 2g'_1(\ell) && + t(\ell+1,0)\\
g_1(\ell) &= \begin{cases} 2g'_2(\ell) & + t(\ell,\ell+1)\\ g'_1(\ell) + g''_2(\ell) & + t(\ell+1,\ell+1) \end{cases}\\
g'_1(\ell) &= 2g_2(\tfrac{1}{2}\ell) && + t(\tfrac{1}{2}\ell,\ell+1)\\
g_2(\ell) &= g'_2(\ell) + g'_3(\ell) && + t(\ell,2\ell+1)\\
g'_2(\ell) &= g_2(\tfrac{1}{2}\ell) + g_3(\tfrac{1}{2}\ell) && + t(\tfrac{1}{2}\ell,\tfrac{3}{2}\ell+1)\\
g''_2(\ell) &= 2g_3(\tfrac{1}{2}\ell) && + t(\tfrac{1}{2}\ell-1,2\ell+2)\\
g_3(\ell) &= \begin{cases} 2g'_3(\ell) & + t(\ell,3\ell+1)\\ g''_3(\ell) + g'_4(\ell) & + t(\ell-1,3\ell+1) \end{cases}\\
g'_3(\ell) &= g_3(\tfrac{1}{2}\ell) + g_4(\tfrac{1}{2}\ell) && + t(\tfrac{1}{2}\ell-1,\tfrac{5}{2}\ell+1)\\
g''_3(\ell) &= 2g_3(\tfrac{1}{2}\ell) && + t(\tfrac{1}{2}\ell,2\ell+1)\\
g_4(\ell) &= 2g'_4(\ell) && + t(\ell-1,4\ell)\\
g'_4(\ell) &= 2g_4(\tfrac{1}{2}\ell) && + t(\tfrac{1}{2}\ell-1,3\ell)
\end{aligned}
\tag{5.3}
$$

For g_1 there are two possibilities, each corresponds with a different splitting. The first relation corresponds with a splitting from the external to a non-external side, the second one with a splitting parallel to the external side. We will ultimately choose the splitting, which leads to the least number of operations. A similar remark applies to g_3.

Elimination of g_j' and g_j'' $(j = 0,1,2,3,4)$ from the above set of relations, together with (5.2) gives:

$$
\begin{aligned}
g_0(\ell) &= 4g_2(\tfrac{1}{2}\ell) && + \tfrac{23}{24}\ell^3 + O(\ell^2) \\
g_1(\ell) &= \begin{cases} 2g_2(\tfrac{1}{2}\ell) + 2g_3(\tfrac{1}{2}\ell) & + \tfrac{65}{24}\ell^3 + O(\ell^2) \\ 2g_2(\tfrac{1}{2}\ell) + 2g_3(\tfrac{1}{2}\ell) & + \tfrac{68}{24}\ell^3 + O(\ell^2) \end{cases} \\
g_2(\ell) &= g_2(\tfrac{1}{2}\ell) + 2g_3(\tfrac{1}{2}\ell) + g_4(\tfrac{1}{2}\ell) && + \tfrac{35}{6}\ell^3 + O(\ell^2) \\
g_3(\ell) &= \begin{cases} 2g_3(\tfrac{1}{2}\ell) + 2g_4(\tfrac{1}{2}\ell) & + \tfrac{239}{24}\ell^3 + O(\ell^2) \\ 2g_3(\tfrac{1}{2}\ell) + 2g_4(\tfrac{1}{2}\ell) & + \tfrac{121}{12}\ell^3 + O(\ell^2) \end{cases} \\
g_4(\ell) &= 4g_4(\tfrac{1}{2}\ell) && + \tfrac{371}{24}\ell^3 - 17\ell^2 + O(\ell)
\end{aligned}
\tag{5.4}
$$

Now it is clear which relations for g_1 and g_3 are the best ones; they both correspond to a splitting from an external to a non-external side.

The set (5.4) contains recursive equations of the form:

$$g(\ell) = \alpha g(\tfrac{1}{2}\ell) + p(\ell) \ , \quad \alpha = 1,2,4 \tag{5.5}$$

where p is a known polynomial. The following properties of this kind of equations are easily verified:

1) additivity: if $g(\ell)$ satisfies (5.5) and $\bar{g}(\ell)$ satisfies

$$\bar{g}(\ell) = \alpha\bar{g}(\tfrac{1}{2}\ell) + \bar{p}(\ell)$$

then $h(\ell) = g(\ell) + \bar{g}(\ell)$ satisfies:

$$h(\ell) = \alpha h(\ell) + p(\ell) + \bar{p}(\ell) \ ;$$

2) non-uniqueness: if $g(\ell)$ satisfies (5.5), then $g'(\ell) = g(\ell) + c.\ell^{\log^2\alpha}$, with c an arbitrary constant, satisfies:

$$g'(\ell) = \alpha g'(\tfrac{1}{2}\ell) + p(\ell) \ ;$$

3) if $p(\ell) = \beta\ell^s$, the solution is:

$$g(\ell) = \frac{\beta}{1 - \frac{\alpha}{2^s}} \cdot \ell^s + c.\ell^{\log^2\alpha} \ , \quad \text{if } \alpha \neq 2^s \ ,$$

$$g(\ell) = \beta\ell^s \log^2\ell + c.\ell^s \ , \quad \text{if } \alpha = 2^s \ ,$$

with c an arbitrary constant;

4) if $p(\ell) = \beta \ell^s \log^2 \ell$, the solution for $\alpha \neq 2^s$ is:

$$g(\ell) = \frac{\beta}{1 - \frac{\alpha}{2^s}} \cdot \ell^s \left(\log^2 \ell - \frac{\alpha}{2^s - \alpha} \right) + c.\ell^{\log^2 \alpha}$$

with c an arbitrary constant.

From the above properties it may be derived that the solution of (5.4) is given by:

$$\begin{aligned} g_4(\ell) &= \tfrac{371}{12}\, \ell^3 - 17\ell^2 \log^2 \ell + O(\ell^2) \\ g_3(\ell) &= \tfrac{849}{36}\, \ell^3 - 17\ell^2 \log^2 \ell + O(\ell^2) \\ g_2(\ell) &= \tfrac{4491}{252}\, \ell^3 - 17\ell^2 \log^2 \ell + O(\ell^2) \\ g_1(\ell) &= \tfrac{3291}{252}\, \ell^3 - 17\ell^2 \log^2 \ell + O(\ell^2) \\ g_0(\ell) &= \tfrac{2487}{252}\, \ell^3 - 17\ell^2 \log^2 \ell + O(\ell^2) \ . \end{aligned} \tag{5.6}$$

From property 2 it follows that initial values only affect the second and lower order coefficients of the solution polynomials.

In a similar way we may deduce from (5.1) that, if $h_j(\ell)$ denotes the number of matrix coefficients stored for an $\ell \times \ell$ mesh, with j adjacent external sides, then:

$$h_j(\ell) = \tfrac{31}{4}\, \ell^2 \log^2 \ell + O(\ell^2) \ , \quad 0 \leq j \leq 4 \ . \tag{5.7}$$

The procedure *ur* needs space to store not only the decomposed matrices, but also the reduced structure matrices, notwithstanding the fact that these are all ultimately removed. Let $r(e) = \frac{1}{2}e(e+1)$ denote the number of coefficients of a reduced structure matrix belonging to a structure with e external nodes. If $f_j(\ell)$ denotes the maximum number of coefficients of reduced structure matrices stored at any time for an $\ell \times \ell$ mesh, with j (j = 0,1,2,3,4) adjacent external sides, then we may derive the following set of recurrence relations:

$$f_0(\ell) = f_2(\tfrac{1}{2}\ell) + r(\ell+1)$$

$$f_1(\ell) = \max(f_2(\tfrac{1}{2}\ell), f_3(\tfrac{1}{2}\ell)) + r(\ell+1) + r(\tfrac{3}{2}\ell+1)$$

$$f_2(\ell) = \max(f_2(\tfrac{1}{2}\ell), f_3(\tfrac{1}{2}\ell), f_4(\tfrac{1}{2}\ell)) + r(2\ell+1) + r(\tfrac{5}{2}\ell+1)$$

$$f_3(\ell) = \max(f_3(\tfrac{1}{2}\ell), f_4(\tfrac{1}{2}\ell)) + r(3\ell+1) + r(\tfrac{5}{2}\ell+1)$$

$$f_4(\ell) = f_4(\tfrac{1}{2}\ell) + r(4\ell) + r(3\ell) \ .$$

Solution of these equations yields:

$$f_4(\ell) = \frac{50}{3}\ell^2 + O(\ell)$$

$$f_3(\ell) = \frac{283}{24}\ell^2 + O(\ell)$$

(5.8) $$f_2(\ell) = \frac{223}{24}\ell^2 + O(\ell)$$

$$f_1(\ell) = \frac{331}{96}\ell^2 + O(\ell)$$

$$f_0(\ell) = \frac{271}{96}\ell^2 + O(\ell) \ .$$

From

$$t(i+1,e+1) - t(i,e) = O(i^2) + O(ie) + O(e^2)$$

$$s(i+1,e+1) - s(i,e) = O(i) + O(e)$$

$$r(e+1) - r(e) = O(e)$$

together with the properties 1 and 3 above, it follows that (5.6), (5.7) and (5.8) are generally valid, also if ℓ is not a power of 2.

If $n \neq m$ the set of recurrence relations is much more intricate. The solution of an $n \times m$ problem ($n > m$) requires less computations than that of an $n \times n$ problem; hence if $n > m$ then $g_j(n)$ and $h_j(n)$ may serve as upper bounds. If $n >> m$, say $n = 2^r.m$ $(r > 0)$, then the rectangular structure is by *ur* partitioned into 2^r $m \times m$ substructures; solution of the corresponding set of recurrence relations yields:

$$G_0(n,m) \simeq 21\, n m^2 + O(m^2)$$

$$H_0(n,m) \simeq \frac{31}{4} n m^2 + O(m^2)$$

$$F_0(n,m) \simeq (\frac{223}{24} + \tfrac{1}{2}r)m^2 + O(m)$$

where G_0, H_0 and F_0 denote the number of operations, decomposed and reduced matrix coefficients, respectively, for an n × m mesh without external sides.

The recursion depth of *ur* is 1 + entier($\log^2$(n)) + entier($\log^2$(m)). All parameters and local variables of *ur* are simple, except *ext*, which is of fixed size. Hence the stack associated with *ur* requires only $O(\log(n.m))$ storage locations or $O(\log(\ell))$ for n = m = ℓ, which is small compared with (5.7) and (5.8).

All matrices are represented in the usual way, no special measures have been taken to deal with non-zero coefficients only. As we know from Section 3.3, the envelopes of the decomposed matrices are dense; however, an envelope does not necessarily comprise all its matrix coefficients. Assume that two reduced structure or element matrices are dense, i.e. do not contain a zero coefficient. If the two matrices are assembled and (partially) decomposed, then the resulting decomposed and reduced matrices are dense again, unless there are no internal nodes (an internal node would be associated with *both* matrices). When applying *ur*, indeed substructures occur without internal nodes, hence the decomposed matrices of certain substructures may contain zeros. As experience indicates, for an n × n mesh and n not too small (n > 13), the number of zeros stored is less than 2% of the total number of coefficients and of all multiplications less than 2% has a zero multiplicant. Therefore it is not worthwhile to replace the full matrix algorithms in *ur* by envelope or profile algorithms.

5.2. Reduction of storage requirements

In [Eisenstat e.a. '76] it is suggested, that for certain finite element types of equations, it is advantageous to recompute certain data, instead of saving them. In subsection 4.1.5 it has been pointed out that the procedure *ur* saves the decomposed structure matrices requiring $h_j(\ell) = O(\ell^2 \log \ell)$ storage locations (see (5.7)). At the expense of a two to six fold increase of the operations count, the storage requirements may be reduced to $O(\ell^2)$, by not saving the decomposed structure matrices. To that end the procedures are modified as follows:

```
1.  procedure mur (n, m, i, j: integer; ext: set-of-sides;
2.                 var r: integer);
3.  var nm1, r1, r2: integer;
4.  begin assign value to r;
5.        if (n = 1) and (m = 1)
6.        then begin compute-element (i, j, Q);
7.                   compute-element-vector (i, j, F);
8.                   transfer contents of Q and F to A
               end
9.        else begin if n > m
10.                  then begin nm1 := n div 2;
11.                             mur (nm1, m, i, j, ext + [upper], r1);
12                              mur (n-nm1, m, i+nm1, j, ext + [lower], r2)
                           end
13.                  else begin nm1 := m div 2;
14.                             mur (n, nm1, i, j, ext + [right], r1);
15.                             mur (n, m-nm1, i, j+nm1, ext + [left], r2)
                           end;
16                         reserve space in A for L_II, L_EI and h_I;
17.                        assemble; assemble-structure-vectors
               end;
18.       decompose; forward-substitution;
19.       free space reserved for L_II, L_EI and h_I
    end
```

This procedure computes both the reduced structure matrix and reduced structure vector, associated with the structure R, characterized by the parameters *n*, *m*, *i*, *j* and *ext*. The parameter *r* indicates where the computed results may be found in the global array *A*. Essentially *mur* is the procedure *ur* combined with *fur*, modified in such a way that the decomposed structure matrices and substituted structure vectors are not saved.

The next procedure just computes the decomposed structure matrices and substituted structure vector of R:

```
 1.  procedure lur (n, m, i, j: integer; ext: set-of-sides;
 2.                 var d: integer);
 3.  var nm1, r1, r2;
 4.  begin if (n = 1) and (m = 1)
 5.        then begin compute-element (i, j, Q);
 6.                   compute-element-vector (i, j, F);
 7.                   transfer contents of Q and F to A
                  end
 8.        else begin if n > m
 9.                   then begin nm1 := n div  ;
10.                              mur (nm1, m, i, j, ext + [upper], r1);
11.                              mur (n-nm1, m, i+nm1, j, ext + [lower], r2)
                            end
12.                  else begin nm1 := m div 2;
13.                             mur (n, nm1, i, j, ext + [right], r1);
14.                             mur (n, m-nm1, i, j+nm1, ext + [left], r2)
                            end;
15.                  reserve space in A for L_II, L_EI and h_I;
16.                  assemble; assemble-structure-vectors
                  end;
17.       decompose; forward-substitution
    end
```

Note that the procedure *lur* itself is not recursive.

As the number of arithmetical operations and the number of storage locations required for the vectors are of a lower order than needed for the matrices, we may use $g_j(\ell)$ (see (5.6)) also to denote the operations count for *lur*. The storage count $f_j^*(\ell)$ follows from the observation:

$$f_j^*(\ell) = \max(f_j(\ell), s_j(\ell))$$

where $s_j(\ell)$ is the number of storage locations needed for L_{II}, L_{IE}, Q^r_{EE}, $Q^r_{E1,E1}$ and $Q^r_{E2,E2}$, E1 and E2 are the externals of the substructures in which the $\ell \times \ell$ mesh is divided. Hence

$$f_0^*(\ell) = \frac{271}{96}\,\ell^2 + O(\ell)$$

$$f_1^*(\ell) = \frac{17}{4}\,\ell^2 + O(\ell)$$

$$f_2^*(\ell) = \frac{223}{24}\,\ell^2 + O(\ell)$$

$$f_3^*(\ell) = \frac{57}{4}\,\ell^2 + O(\ell)$$

$$f_4^*(\ell) = \frac{43}{2}\,\ell^2 + O(\ell)\ .$$

The next procedure, which computes the solution vector, is a slight modification of *bur*:

```
 1.  procedure tur (n, m, i, j: integer; ext: set-of-sides;
 2.                 pwe: integer);
 3.  var nm1, d, pw1, pw2: integer;
 4.  begin lur (n, m, i, j, ext, d);
 5.        solve;
 6.        if (n = 1) and (m = 1)
 7.        then begin transfer data from A to F;
 8.                   process-solution (i, j, F)
                end
 9.        else begin if n > m
10.                   then begin nm1 := n div 2; separate;
11.                              tur (nm1, m, i, j, ext+[upper], pw1);
12.                              tur (n-nm1, m, i+nm1, j, ext+[lower], pw2)
                            end
13.                   else begin nm1 := m div 2; separate;
14.                              tur (nm1, m, i, j, ext+[right], pw1);
15.                              tur (n, m-nm1, i, j+nm1, ext+[left], pw2)
                            end
                end
    end
```

The storage requirements of *tur* are precisely those of *lur* in line 4. The operation count $\bar{g}_j(\ell)$ may be deduced from:

$$\bar{g}_0(\ell) = 2\bar{g}_1'(\ell) + g_0(\ell)$$

$$\bar{g}_1(\ell) = 2\bar{g}_2'(\ell) + g_1(\ell)$$

$$\bar{g}_1'(\ell) = 2\bar{g}_2(\tfrac{1}{2}\ell) + g_1'(\ell)$$

$$\bar{g}_2(\ell) = \bar{g}_2'(\ell) + \bar{g}_3'(\ell) + g_2(\ell)$$

$$\bar{g}_2'(\ell) = \bar{g}_2(\tfrac{1}{2}\ell) + \bar{g}_3(\tfrac{1}{2}\ell) + g_2'(\ell)$$
$$\bar{g}_3(\ell) = 2\bar{g}_3'(\ell) + g_3(\ell)$$
$$\bar{g}_3'(\ell) = \bar{g}_3(\tfrac{1}{2}\ell) + \bar{g}_4(\tfrac{1}{2}\ell) + g_3'(\ell)$$
$$\bar{g}_4(\ell) = 2\bar{g}_4'(\ell) + g_4(\ell)$$
$$\bar{g}_4'(\ell) = 2\bar{g}_4(\tfrac{1}{2}\ell) + g_4'(\ell)$$

where $\bar{g}_j'$ $(j = 1,\ldots,4)$ is defined analogously to g_j'. Elimination of $\bar{g}_j'$ $(j = 1,\ldots,4)$ from the above gives:

$$\bar{g}_0(\ell) = 4\bar{g}_2(\tfrac{1}{2}\ell) + g_0(\ell) + 2g_1'(\ell)$$
$$\bar{g}_1(\ell) = 2\bar{g}_2(\tfrac{1}{2}\ell) + 2\bar{g}_3(\tfrac{1}{2}\ell) + g_1(\ell) + 2g_2'(\ell)$$
$$\bar{g}_2(\ell) = \bar{g}_2(\tfrac{1}{2}\ell) + 2\bar{g}_3(\tfrac{1}{2}\ell) + \bar{g}_4(\tfrac{1}{2}\ell) + g_2(\ell) + g_2'(\ell) + g_3'(\ell)$$
$$\bar{g}_3(\ell) = 2\bar{g}_3(\tfrac{1}{2}\ell) + 2\bar{g}_4(\tfrac{1}{2}\ell) + g_3(\ell) + 2g_3'(\ell)$$
$$\bar{g}_4(\ell) = 4\bar{g}_4(\tfrac{1}{2}\ell) + g_4(\ell) + 2g_4'(\ell) .$$

Combining the above with (5.3) yields:

$$\bar{g}_0(\ell) = 4\bar{g}_2(\tfrac{1}{2}\ell) + g_0(\ell) + 4g_2(\tfrac{1}{2}\ell) + 2t(\tfrac{1}{2}\ell, \ell+1)$$
$$\bar{g}_1(\ell) = 2\bar{g}_2(\tfrac{1}{2}\ell) + 2\bar{g}_3(\tfrac{1}{2}\ell) + g_1(\ell) + 2g_2(\tfrac{1}{2}\ell) + 2g_3(\tfrac{1}{2}\ell) + 2t(\tfrac{1}{2}\ell, \tfrac{3}{2}\ell+1)$$
$$\bar{g}_2(\ell) = \bar{g}_2(\tfrac{1}{2}\ell) + 2\bar{g}_3(\tfrac{1}{2}\ell) + \bar{g}_4(\tfrac{1}{2}\ell) + g_2(\ell) + g_2(\tfrac{1}{2}\ell) + 2g_3(\tfrac{1}{2}\ell) + g_4(\tfrac{1}{2}\ell) + t(\tfrac{1}{2}\ell, 3\ell+1) + t(\tfrac{1}{2}\ell-1, \tfrac{5}{2}\ell+1)$$
$$\bar{g}_3(\ell) = 2\bar{g}_3(\tfrac{1}{2}\ell) + 2\bar{g}_4(\tfrac{1}{2}\ell) + g_3(\ell) + 2g_3(\tfrac{1}{2}\ell) + 2g_4(\tfrac{1}{2}\ell) + 2t(\tfrac{1}{2}\ell-1, \tfrac{5}{2}\ell+1)$$
$$\bar{g}_4(\ell) = 4\bar{g}_4(\tfrac{1}{2}\ell) + g_4(\ell) + 4g_4(\tfrac{1}{2}\ell) + 2t(\tfrac{1}{2}\ell-1, 3\ell) .$$

It is easily verified that the following may be added to the properties of equations (5.5):

5) if $p(\ell) = \beta\ell^s \log^2\ell$, the solution for $\alpha = 2^s$ is:

$$g(\ell) = \tfrac{1}{2}\beta\,\ell^s(\log^2\ell)^2 + \tfrac{1}{2}\beta\,\ell^s \log^2\ell + c\ell^s$$

(c is an arbitrary constant);

6) if $p(\ell) = \beta\ell^s(\log^2\ell)^2$, the solution for $\alpha \neq 2^s$ is:

$$g(\ell) = \frac{\beta}{1 - \frac{\alpha}{2^s}}\,\ell^s(\log^2\ell)^2 + O(\ell^s \log \ell) .$$

Hence, solution of the above set of recursive equations gives:

$$\bar{g}_4(\ell) = \frac{620}{6}\ell^3 - \frac{17}{2}\ell^2(\log^2\ell)^2 + O(\ell^2 \log \ell)$$

$$\bar{g}_3(\ell) = \frac{802}{9}\ell^3 - \frac{17}{2}\ell^2(\log^2\ell)^2 + O(\ell^2 \log \ell)$$

(5.9) $$\bar{g}_2(\ell) = \frac{34106}{441}\ell^3 - \frac{17}{2}\ell^2(\log^2\ell)^2 + O(\ell^2 \log \ell)$$

$$\bar{g}_1(\ell) = \frac{9785}{147}\ell^3 - \frac{17}{2}\ell^2(\log^2\ell)^2 + O(\ell^2 \log \ell)$$

$$\bar{g}_0(\ell) = \frac{25684}{441}\ell^3 - \frac{17}{2}\ell^2(\log^2\ell)^2 + O(\ell^2 \log \ell) \ .$$

From (5.9) and (5.6) it follows:

$$\bar{g}_4(\ell) \,/\, g_4(\ell) \simeq 3.3$$

$$\bar{g}_0(\ell) \,/\, g_0(\ell) \simeq 5.9 \ .$$

Hence for a structure with four external sides we have, at the expense of a three-fold increase in the number of computations, reduced the storage count from $\frac{31}{4}\ell^2 \log^2\ell$ to $\frac{43}{2}\ell^2$ (there is also a reduction if $\ell < 8$); for structures without external sides the storage count is even reduced to $\frac{271}{96}\ell^2$, requiring a six-fold increase of the operations count. Note that the complete structure matrix associated with a square $\ell \times \ell$ grid contains $5\ell^2 + O(\ell)$ non-zero coefficients. By other, more intricate, modifications it is possible to restrict the increase in the number of computations to a factor 2 yielding a storage count of $\frac{31}{4}\ell^2$ for a structure without external sides.

CHAPTER 6

ADAPTATION TO MORE COMPLICATED STRUCTURES

Applications of the algorithms presented in the preceding chapters are not restricted to simple rectangular plates with a uniform mesh. We will indicate in this chapter how the algorithms may be adopted for more general two-dimensional solid as well as frame structures.

The algorithms (or variants of them) are in particular useful if (parts of) the structure to be analyzed can be provided with a "topologically" regular mesh. Irregular element partitionings are often encountered if triangular elements are used to refine grids locally. It will be shown how locally refined grids are most efficiently analyzed.

6.1. Frame structures

A structure will be called *frame structure* if it consists of elements with two nodes only. Let the elements of the frame structure F be arranged to form a rectangular $n \times m$ grid, such that the $(n+1) \times (m+1)$ nodes are the grid points and each element is a horizontal or vertical edge joining two neighbouring grid points. It is not immediately obvious how to decompose F into two similar substructures (how to partition the elements into two sets). One may want to dissect F along a line roughly through the middles of the two sides with the most elements, but then the dissection line passes through some elements and for every element along the dissection line a choice must be made to which substructure that element belongs. Two possibilities for a sensible decision are:

i) every element along the dissection line belongs to the same, say first, substructure;
ii) starting at one end of the dissection line, the elements alternatingly belong to the first or the second substructure.

In both cases the number of different kinds of substructures increases. In the first case we obtain substructures, an external side of which may or may not be "notched". In the second case all external sides are "dashed", but the first two nodes of an external side may or may not belong to a same

element. Hence in both cases there are two types of external sides. This can be accounted for by extending the procedures *ur*, *fur* and *bur* with an extra parameter (or by extending the parameter *ext*) to indicate the type of each external side. Of the procedure *ur* only the procedures *assemble* and *compute-element* need be adapted to these extensions, leading to an increase of the amount of code required. The efficiency of the computations is not adversely affected; only the length of the program text increases due to a more extensive case analysis. The number of arithmetic operations and matrix coefficients stored is precisely the same as in the quadrilateral element case. However, the matrices associated with frame structures contain more zero coefficients. This may be seen as follows.

An element matrix associated with a quadrilateral four node element is a 4×4 matrix without zeros, whereas in the frame structure case, each call of *compute-element* yields a 4×4 matrix M, which is assembled from at most four 2×2 matrices and hence the envelope of M does not contain all the coefficients of M. As may be estimated (see also [Duff e.a. '76]) the savings may amount to about 25% of storage and to about 30% of arithmetical operations by storing only the envelopes of the matrices. In these percentages, the overhead in using envelopes only is *not* included.

6.2. Solid quadrilateral structures

The procedures as developed in Chapter 4 apply to finite element calculations for structures with an $n \times m$ mesh. To derive the element matrices, one usually needs geometrical data, like node coordinates. In this section we will describe how these geometrical data are obtained from the row and column number of the element concerned.

6.2.1. Rectangles

Let the rectangle R be determined by the following X, Y coordinates of its four corner nodes: (0,0), (a,0), (a,b), (0,b), where the nodes are listed clockwise, starting with the lower left node ($a,b \geq 0$). Suppose the mesh is uniform, i.e. all elements are rectangular and of the same size. If there are N and M elements along the sides parallel to the X- and Y-axis, respectively, then the coordinates of the lower left node of the element in row i and column j are:

$$(\frac{a}{N}\, i,\ \frac{b}{M}\, j) \qquad (0 \le i < N,\ 0 \le j < M)\ .$$

The coordinates of the other element nodes may be obtained from similar expressions. Because $\frac{a}{N}$ and $\frac{b}{M}$ may be considered as global constants for the procedures *ur*, *fur* etc., each coordinate requires in fact only one multiplication.

In order to ensure that the results of the finite element computations are sufficiently accurate, the size of the elements must be small enough [Strang and Fix '73]. The smaller the size of the elements, the larger the number of nodes, hence the more computations. For several finite element problems, it is not necessary that all elements have the same size. By allowing elements with different sizes, one may achieve sufficient accuracy with only a modest amount of computations. Hence, it is useful to provide a facility to "grade" the mesh. For the rectangle R, to be divided into rectangular elements, grading is simply achieved by choosing two monotone functions φ: {0,...,N-1} → [0,a) and ψ: {0,...,M-1} → [0,b) with $\varphi(0) = \psi(0) = 0$ and defining the coordinates of the lower left node of the elements in row i and column j to be

$$(\varphi(i),\psi(j))\ .$$

For example:

$$\varphi(i) = \frac{a}{p^N - 1}\,(p^i - 1) \qquad (i = 0,\ldots,N-1)$$

with p a suitably chosen constant, is such a monotone function.

As is well known, the accuracy of the results of the finite element computation depends also upon the shape of the elements: the more a rectangular element deviates from a square, the less accurate the results of the computation are [Strang and Fix '73]. Grading the mesh in the above way may lead to elements which are too slender. To avoid those too slender elements the grading technique as outlined in Section 3 is more appropriate.

6.2.2. Quadrilaterals

In this subsection we will considet a (curvilinear) quadrilateral structure C with a mesh consisting of rows and columns of (not necessarily rectangular) elements. We will describe how, in this general case, the node

coordinates can be obtained from the row and column number of the elements concerned. A transformation Φ will be constructed, which is a one-to-one map from a suitably chosen rectangle R onto the quadrilateral C. (In the sequel x, y will denote coordinates of a point in R and ξ, η will be used for points in C.) The map Φ transforms a mesh of R with rectangular elements into a mesh of C with quadrilateral elements. The coordinates of the element nodes of C are obtained by applying Φ to the coordinates of the corresponding element nodes of R. In which way the node coordinates of R are derived from the row and column number of the element concerned, has been shown in the preceding subsection.

6.2.2.1. Quadrilateral given by points

If the quadrilateral C has straight sides and is given by the coordinates of its corner nodes $c_i = (\xi_i, \eta_i)$ $(i = 1,2,3,4)$, bilinear shape functions [Zienkiewicz '77] may be used to construct Φ. If for R the unit square with corner nodes $r_1 = (-1,-1)$, $r_2 = (-1,1)$, $r_3 = (1,1)$ and $r_4 = (1,-1)$ is taken, then

$$\Phi(x,y) = \left(\sum_{i=1}^{4} N_i \xi_i \, , \sum_{i=1}^{4} N_i \eta_i \right) \tag{6.1}$$

with

$$\begin{aligned} N_1 &= \tfrac{1}{4}(1-x)(1-y) \\ N_2 &= \tfrac{1}{4}(1-x)(1+y) \\ N_3 &= \tfrac{1}{4}(1+x)(1+y) \\ N_4 &= \tfrac{1}{4}(1+x)(1-y) \; . \end{aligned} \tag{6.2}$$

It is easy to verify that

$$\Phi(r_i) = c_i \qquad (i = 1,2,3,4) \; .$$

A sufficient condition that such a transformation, using bilinear shape functions, is a one-to-one map from R onto C is that no internal angle of C be larger than π [Strang and Fix '73]. (This implies that the transformation may even be used if C degenerates into a triangle.)

Φ as defined by (6.1) and (6.2) is bilinear; hence straight boundary and inter-element line segments are transformed into straight segments. There-

fore the image of a rectangular element in R is a quadrilateral with straight boundaries.

Let now the quadrilateral C be given by m successive points, to be denoted (in clock-wise order) by $c_1 = (\xi_1,\eta_1), \ldots, c_m = (\xi_m,\eta_m)$. Let $c_1 = c_{b1}$, c_{b2}, c_{b3} and c_{b4} be (in clock-wise order) the corner points of C. For R we choose a rectangle with sides parallel to the coordinate axes. On the sides of R we choose (in clock-wise order) m points $r_1 = (x_1,y_1), \ldots, r_m = (x_m,y_m)$ in such a way that r_{b1}, r_{b2}, r_{b3} and r_{b4} are the corner points of R. To construct Φ we choose the shape functions of the "serendipity family" of finite elements [Zienkiewicz '77]. The idea is the following. Let Φ be defined by

$$\Phi(x,y) = \left(\sum_{r=1}^{m} N_i \xi_i , \sum_{i=1}^{m} N_i \eta_i \right)$$

where N_i are functions still to be specified. If we manage to choose those functions in such a way that they satisfy:

$$N_i(r_j) = \delta_{ij} , \qquad 1 \le i,j \le m , \tag{6.3}$$

then obviously the following relations hold:

$$\Phi(r_i) = c_i , \qquad 1 \le i \le m .$$

Lagrangean interpolation is now used to obtain the functions N_i satisfying (6.3):

1) Let r_i (not a corner point) lie on a side s parallel to the Y-axis and let r_k be a point on the side parallel to s, then

$$N_i(x,y) = \frac{y - y_k}{y_i - y_k} \cdot \prod_{r_j \text{ on } s} \frac{x - x_j}{x_i - x_j} .$$

If s is parallel to the X-axis, then x and y are interchanged:

$$N_i(x,y) = \frac{x - x_k}{x_i - x_k} \cdot \prod_{r_j \text{ on } s} \frac{y - y_j}{y_i - y_j} .$$

2) If r_i is a corner point lying on the sides s and t, parallel to the X- and Y-axis respectively, and r_k is a corner point not on s and t, then

$$N_i(x,y) = \frac{x - x_k}{x_i - x_k} \frac{y - y_k}{y_i - y_k} - \sum_{\substack{r_j \text{ on } s \\ \text{except corners}}} \frac{x_j - x_k}{x_i - x_k} N_j(x,y) -$$

$$- \sum_{\substack{r_j \text{ on } t \\ \text{except corners}}} \frac{y_j - y_k}{y_i - y_k} N_j(x,y) \ .$$

If Φ as defined in this way (depending amongst others upon the choices of $r_1,\ldots,r_m$) is a one-to-one map from R on C, then Φ transforms a mesh of R into a mesh of C. Whether, however, Φ is one-to-one and whether the mesh obtained is appropriate, cannot be stated in general. Intuition and graphics facilities must be resorted to.

6.2.2.2. Quadrilateral given by parametric functions

Assume that the four sides of the quadrilateral C are given by the parametric functions:

$$f_j : [0,1] \to \mathbb{R}^2, \quad j = 1,2,3,4$$

with the corner points of C given by

$$f_1(0) = f_3(0) \ , \quad f_3(1) = f_2(0) \ ,$$

$$f_1(1) = f_4(0) \ , \quad f_4(1) = f_2(1) \ .$$

Blending function interpolation [Gordon and Hall '73] can be used to construct a transformation from the rectangle R determined by the corner points (0,0), (1,0), (1,1) and (0,1) to C:

$$\Phi(x,y) = (1-x)f_1(y) + xf_2(y) + (1-y)f_3(x) + yf_4(x)$$
$$- x(1-y)f_2(0) - xyf_2(1)$$
$$- (1-x)(1-y)f_1(0) - (1-x)yf_1(1) \ .$$

Φ transforms the straight line segment $\{(0,t) \mid 0 \le t \le 1\}$ in the curved line segment $\{f_1(t) \mid 0 \le t \le 1\}$ and the line $\{(t,0) \mid 0 \le t \le 1\}$ into $\{f_3(t) \mid 0 \le t \le 1\}$, etcetera. Hence the four sides of R are transformed in the four sides of C.

Again it is difficult to state in general, whether the parametric functions f_j $(j = 1,2,3,4)$ lead to an appropriate mesh for C. So far one is best

guided by experience, geometric intuition and inspection. The aid of computer graphics facilities seems indispensable.

6.3. Local mesh refinements

In Section 6.2.1 we have indicated how grading of a mesh may be achieved. For problems which allow very different element sizes in different parts of the structure, grading in that way leads to elements which are too slender. Therefore, in practice triangular elements are popular, because they are more suited to achieve local mesh refinements. A serious drawback of using triangular elements for that purpose is that they may give rise to irregular meshes. The merits of the procedures as outlined in Chapter 4 are due to exploiting regularity. To obtain local mesh refinements, however, one does not need to resort to irregular meshes. For the ease of presentation we will in the following apply rectangular elements, more specifically blended elements [Cavendish '75]. Blended elements differ from standard elements in that node to node connection for two adjacent elements is not required and thus that two or more smaller elements are allowed to abut against the edge of a larger element. In an obvious way the meshes described can also be obtained by applying (standard) triangular elements.

6.3.1. Procedure $\mathcal{L}m$

Let us consider, to start with, a rectangular structure R with a mesh, locally refined around the lower left corner of R. Consider a partitioning of R into rectangular elements obtained in the following way:

first partition R into four similar subrectangles;
next perform n times
    partition the left, lower subrectangle into four similar subrectangles.

This partitioning leads to a mesh with 2 elements along the upper and right side of R and with $n+2$ elements along the lower and left side. Altogether there are $3n+4$ elements and $5n+9$ nodes.

The elements of R will be identified with two integers, a "row" and a "column" number, in the following way: the three largest elements are (in clock-wise order, starting with the left most one) identified by (n+1,0), (n+1,n+1) and (0,n+1), respectively; of the remaining elements the three

largest ones are, in the same way, identified with (n,0), (n,n) and (0,n); and so on. The lower left element is identified by (0,0).

Let R_m ($0 \leq m \leq n$) denote the substructure containing the elements of which the row and column number do not exceed $m+1$. We then have $R = R_n$. Moreover the elements (m+1,0), (m+1,m+1) and (0,m+1) assembled with R_{m-1} yield precisely R_m. Hence assembling the three element matrices to the reduced structure matrix of R_{m-1} gives a structure matrix consisting of five internal and five external nodes. Elimination of the five internal nodes results in the reduced structure matrix of R_m.

A procedure computing in the same vein as in Chapter 4 the reduced structure matrix associated with R if the five nodes along the upper and right side are external, is:

```
 1.  procedure lm (n: integer; var r: integer);
 2.  var h, r1: integer;
 3.  begin ur (2, 2, 0, 0, [upper, right], r1);
 4.        h := 0;
 5.        while h < n do
 6.        begin h := h+1;
 7.              compute-element (h+1, 0);
 8.              compute-element (0, h+1);
 9.              compute-element (h+1, h+1);
10.              assemble;
11.              decompose; update r1
12.        end;  r := r1
    end
```

The parameter *n* indicates the number of times a lower, left subrectangle of R was partitioned in order to obtain the complete partitioning of R. The value of the parameter *r* upon exit of the procedure is the address of the reduced structure matrix of R in the global array *A*.

The values of the auxiliary variables *h* and *r1* are such that always at the beginning and the end of the repeated compound statement the following relation holds:

r1 is the address of the reduced structure matrix of R_h

The procedure *assemble* assembles the three element matrices, computed in the preceding lines, with the reduced structure matrix of R_{h-1}; elimination of the five internal nodes with the procedure *decompose* gives the reduced structure matrix of R_h. When execution of the while-statement is completed, it yields $h = n$ and hence *rl* is the address of the reduced structure matrix of $R_h = R_n = R$.

It is straightforward to develop in a similar fashion procedures to compute the (reduced) structure and solution vectors.

As is the case with *ur*, *lm* may be applied to rectangles as well as to general quadrilateral structures.

The procedure *lm* resembles *ur* in that decomposition of the structure matrix is interleaved with partitioning the structure, computing the element matrices, assembling and ordering the equations. The traditionally consecutive steps are carried out interleaved.

6.3.2. Storage and operation counts of *lm*

If the matrices are represented as full matrices, then the procedure *decompose* in line 11 requires (see 5.2)

$$\frac{1}{6}\,i^3 + \frac{1}{2}\,i^2(e+1) + \frac{1}{2}\,ie(e+2) + \frac{1}{3}\,i$$

multiplicative operations with i = e = 5. Hence, the while-statement requires 185 n multiplicative operations with matrix coefficients. The procedure *ur* requires 130 such operations, therefore the total number is

$$185\,n + 130\,.$$

By applying profile algorithms that number may be reduced to

(6.4) $$119\,n + 67\,.$$

To store the matrix, assembled in line 10, as a full matrix requires 55 storage locations; for the reduced structure matrix 15 locations are needed. Hence, if the decomposed and reduced structure matrices are overwritten once they are not needed any longer, *lm* requires in total 55 + 15 is

$$70$$

locations (*ur* in line 3 requires less).

If the decomposed matrices are retained, then at most

(6.5) $40\,n + 100$

storage locations are occupied.

It should be noted that (6.4) shows also the operation count associated with a more usual organization of the calculations, i.e. first assembling the complete structure matrix associated with R and next applying a profile algorithm to compute the reduced structure matrix.

Let N denote the total number of nodes of R, then $N = 5n+9$. It follows from (6.4) and (6.5) that the operation and storage counts are about $24N - 147$ and $8N + 28$, respectively. Hence both time and space required are $O(N)$, i.e. linear in the number of nodes. An asymptotically faster algorithm does not exist, because writing down the solution alone requires $O(N)$ space and time. Remember that the corresponding counts for *ur* are $O(N\sqrt{N})$ and $O(N \log N)$, respectively, with N the total number of nodes of the structure concerned.

If the rectangle R would have been partitioned uniformly with all elements in size equal to the smallest element of the locally refined grid, then the total number of multiplicative operations with matrix coefficients would have been $O(8^n)$. Comparing this with (6.4) shows the computational advantages of local mesh refinement.

6.4. General plane and curved surfaces

For more general surfaces, which for instance may contain appendages or holes, an approach as outlined in [Zienkiewicz and Phillips '71] may be followed.

The surface, say S, is divided into a number of quadrilaterals Vi ($i = 1,\ldots,r$). With S a so called key diagram is associated. A key diagram is a rectangular configuration of (possibly empty) rectangles. There is a one-one correspondence between the non-empty rectangles Ri ($i = 1,\ldots,r$) and the quadrilaterals Vi. Moreover, Ri and Rj ($i \neq j$) are adjacent in the key diagram only if Vi and Vj are adjacent in S. For every pair Ri, Vi a transformation Φi is constructed, as outlined in Section 6.2, which transforms a partitioning of Ri into a partitioning of Vi. Of course, if Ri and

Rj have a common boundary b, then the nodes of Ri and Rj on b must coincide; moreover Φi and Φj applied to b must be the same transformation.

To complete the versatility of the scheme it is important to include one further feature, namely to express that two seemingly different boundaries in the key diagram are identical. It is of course necessary to ensure that the transformation conditions just described for common boundaries, are satisfied for such identical boundaries. By including this feature it is even possible to deal with such three-dimensional surfaces like tori and ball surfaces. Key diagrams are in [Zienkiewicz and Phillips '71] used to generate an element partitioning for S only. However, it is just as well possible to apply procedures as described in Chapters 4 and 5 to compute immediately the reduced structure matrices Q^r_{Vi} and vectors f^r_{Vi} associated with the quadrilaterals Vi (i = 1,...,r).

To assemble the matrices Q^r_{Vi} and vectors f^r_{Vi} and to compute the associated parts of the total solution vector, one may proceed in the traditional way. The finite element system FEMSYS [Peters '76] is very well suited not only to perform such matrix calculations, but also to handle the necessary book-keeping. FEMSYS is well suited because of its facilities for specifying structures consisting of arbitrary substructures; moreover its possibility to identify nodes with different numbers is necessary in this case.

CHAPTER 7

CLOSING REMARKS

7.1. Other implementations

A discussion of various strategies to solve a set of linear finite element equations efficiently by direct methods may be found in [George '77, George and McIntyre '78]. Those papers deal only with ordering of the equations, LU-decomposition (or factorization as it is called) and forward and backward substitution; they are not concerned with the assembly of the structure matrix. Comparing figures from those papers with Table 1 indicates (taking into account the different processor speeds and the efficiency of the code produced by the respective compilers) that our solution (by applying the procedures from Chapter 4) is far more efficient (by about a factor 4). This may be due to the fact that reordering as well as overhead storage and bookkeeping are avoided. For a better appraisal we must compare our program with one that executes *all* the relevant steps.

For another comparison we have taken a popular finite element program for structural analysis in use on several computer installations all over the world. From Table 2 it is obvious that our program saves a factor greater that 50 of the processor time. Moreover, our program uses only central memory, whereas the other needs auxiliary disk space. The structural analysis program is intended to be a general purpose one, suited for all kinds of meshes. Therefore the comparison may not be quite fair; still it indicates very clearly which gains in efficiency may be achieved.

7.2. Data retrieval

One of the reasons why the procedures as developed in this thesis are so efficient is undoubtedly that no other data than coefficients of structure matrices and vectors are stored. Whenever the value of a coordinate is required, it is computed. This is easily done, because the regularity of the problem is fully exploited. For instance, if the structure is a rectangle with a uniform mesh, the computation of a coordinate takes only one addition and one multiplication with simple variables as operands, which is

n	number of unknowns	number of multiplications	decomposition time*	total time*	number of coefficients of decomposed matrices
5	36	817	0.018	0.06	220
10	121	6829	0.10	0.30	1170
15	256	24848	0.31	0.83	3200
20	441	62744	0.72	1.73	6561
25	676	123429	1.43	3.12	11230
30	961	216323	2.40	4.77	17314
35	1296	350184	3.81	7.07	25065
40	1681	544868	6.00	10.5	35189
45	2116	772081	8.48	14.3	46350
50	2601	1057805	10.7	17.7	59142

storage and operations counts
decomposition and total time
of
procedure *ur* applied to n × n grids

* times are seconds on IBM 370/165 with double length reals

TABLE 1

number of unknowns	STRUDL	*ur*
450	224	2
882	377	5
3200	1247	23

total processing times
in seconds on IBM 370/165

TABLE 2

no more expensive than the evaluation of a subscripted variable. Hence, it is cheaper to compute than to retrieve the coordinates every time they are needed. For less regular structures the topologies of (parts) of the element meshes are the same. To compute the coordinates of the nodes, transformations as described in Chapter 6 are applied. These transformations are simple, easy to compute functions, if the structure does not deviate too much from that of a uniform rectangle.

7.3. Triangular dissections

The computational steps of *ur*, *fur* and *bur* resemble the traversal of a binary tree. Going from a tree vertex to its successor, all information relevant for the successor is easily derived from the information concerning its predecessor.

Such information includes not only the number of internal and external nodes, but also the ordering of them. Procedures analogous to *ur*, *fur* and *bur* may be developed for other kinds of structures. For instance, a triangular structure T can be dissected into four similar triangles, each of which can be dissected into four ..., and so on. The associated tree of substructures is then a quaternary tree.

7.4. One- and three-dimensional problems

It is straightforward to develop procedures analogous to *ur*, *fur* and *bur* for one- or three-dimensional structures: a line segment can be dissected into two line segments, a brick into two bricks. The adaptations as described in Chapter 6 may be extended to the one- or three-dimensional case as well.

However, for one-dimensional structures it proves to be cheaper both in storage and in number of arithmetical operations not to dissect the line into two lines of about equal size, but to split off just one element at one of the ends. In this way the nodes are successively eliminated from one end of the structure to the other.

The above remark is also valid for two-dimensional structures with an $n \times m$ mesh if $n >> m$. For those structures it is advantageous, as far as efficiency is concerned, to consider them as one-dimensional strings of substruc-

tures with m × m meshes. To compute the matrices and vectors associated with those substructures, the procedures *ur*, *fur* and *bur* may be applied.

7.5. Structures with more than one structure vector

In some applications of the finite element method, many sets of equations having the same coefficient matrix must be solved. If the procedures from Chapter 4 are used, then the decomposed structure matrices associated with the structure, say R, need to be computed only once by a call of *ur* and each structure and solution vector for R requires the execution of the procedures *fur* and *bur*. We have seen that the amount of time needed to execute *fur* and *bur* is small compared with *ur*.

Also the storage space saving procedures of Chapter 5 may be applied if a number of structure vectors is presented simultaneously. A facility to handle simultaneously more than one structure vector instead of only one at a time must be added to *lur*. In the same way also *tur* must be accommodated to handle more than one solution vector. The facilities are easily implemented. If the structure vectors are presented consecutively, then for each structure vector the procedure *lur* must be executed, which implies that for each structure vector the assembly and decomposition of the matrices associated with R are repeated.

7.6. Iterative methods

Direct and iterative methods to solve partial differential equation problems are compared in [Axelsson '77]. Second and fourth order problems with two-dimensional n × n and three-dimensional n × n × n grids are considered. A comparison is made between the asymptotic number of arithmetical operations required by the "SSOR preconditioned conjugate gradient method" and the nested dissection method. It turns out that for a three-dimensional second order problem the iterative methods are asymptotically faster than the backward substitution phase in the direct method. For a fourth order two-dimensional problem the direct method is superior. In other problems the superiority of one method over another is not clear. The size of the problem and the number of structure vectors may influence the choice of the method. It should be noted, however, that direct methods are more generally applicable than iterative ones.

7.7. Data structuring facilities of PASCAL

The data structuring facilities of PASCAL are judged to be among its more attractive features [Wirth '71 acta]. Nevertheless, to represent all the structure matrices and vectors associated with an n × m grid, we have used only the most simple data structure (apart from a simple variable), viz. the one-dimensional array. The reason is the following.

A call of the procedure *ur* results in a hierarchy of substructures with corresponding matrices. Therefore, a tree like data organization with in every vertex the matrices associated with a substructure could be very appropriate. Such dynamic data structures could then be used to represent all matrices. However, PASCAL requires that the sizes of the arrays contained in the vertices of dynamic trees are declared statically, thus requiring setting of fixed limits. This is inefficient for the storage of matrices of various sizes, as generated by the procedure *ur*. These matrices can neither be stored in local arrays, because they are needed outside the block where they are computed, nor in as many global arrays as there are matrices or substructures, because their number and sizes would then have to be declared statically. The only remaining possibility is to store all matrices together into one or two global arrays of sufficient length.

If it were possible to define dynamically the sizes of the arrays in the tree vertices, then tree structures could be considered. However, for the procedure *ur*, trees would result in a less efficient data organization, because pointer variables would be required as well as information concerning the sizes. On the other hand in our data representation as set up for *ur* only matrix coefficients are stored, nothing else.

In nearly every programming language one-dimensional arrays occur; moreover the iterative counterparts of the recursive procedures in Chapter 4 are easily obtained [Peters '78]. Hence the procedures as described in this thesis may be coded in nearly all programming languages.

7.8. Generalized element method, element merge tree

Although one may be tempted to do so, our method for finite element computations must not be confounded with the generalized element method [Speelpenning '78] (a generalization and improvement of the frontal solution

method [Irons '70]). Nor should a proper preserving palm be confused with an element merge tree [Eisenstat e.a. '78].

The generalized element method is different in the following aspects:

- it is motivated by an efficient use of "backing store";
- nodes of a structure are eliminated one at a time;
- the elimination order of the nodes is assumed to be determined in advance;
- the way in which a structure is dissected into substructures is completely controlled by the elimination order of the nodes;
- it requires overhead storage and bookkeeping;
- element matrices are assumed to have been computed in advance.

Similarities between the method and our way of organizing finite element computations are that assembly and decomposition of the structure matrix are interleaved (substructures may be distinguished) and that only full matrices are manipulated.

The generalized element method is only applicable to so called "network equations", which are equations whose associated matrix can be considered to be assembled from (smaller) element matrices. In this sense the MSSE method [Eisenstat e.a. '78] is a generalization, it applies to arbitrary symmetric positive definite matrices. However, this method deals only with LU-decomposition and forward and backward substitution; assembly of the matrix is not considered. The principal advantages of MSSE are described to be "the ability to solve problems in significantly less core and to trade off an increase in execution time for a decrease in core". To achieve this a so called "element merge tree" is constructed. Such an element merge tree depends on the ordering of the equations and variables. In an obvious way a pp-partition can be associated with such an element merge tree. Hence the results of Chapter 2 apply also to those element merge trees. The partition associated with an element merge tree is not always a proper one, however. As a consequence, if the MSSE method is applied to a dense band matrix, it results in highly inefficient moving around of data. In all cases the method requires extensive bookkeeping. The claim "for a nine-point problem with the nested dissection ordering on an $n \times n$ grid fewer than $\frac{7}{2} n^2$ non-zeros must be saved versus $\frac{93}{12} n^2 \log^2 n$ for sparse elimination, while the work required at most doubles" is incorrect.

It should be noted that Williams was the first to point out the equivalence of substructuring and sparse matrix algorithms [Williams '73]. He showed that it is always possible to choose the substructures so that the substructure method will lead to precisely the same computations as any sparse matrix solution, with the minor difference that some additions are performed at different stages of the solution. He showed this, however, in such a way that he was led to the conclusion: "It appears that a sparse matrix method will always be preferable to a substructure method ...". We have shown in this thesis that this conclusion is not valid any longer.

7.9. Parallel computation

Let us consider the procedure *ur* in 4.1.2. Execution of the recursive call in line 11 (or line 14) does not depend upon the execution of the immediately preceding call in line 10 (or line 13), except for the determination of the position to store the computed matrices. Hence both recursive calls may be done in either order or even simultaneously by two different processors. In the last case, each processor in its turn may set to work two other processors. The processors do not need access to common data. The procedure *ur* is therefore well suited for implementation on a computer with many processors. The way in which the processors communicate with each other is independent of the finite element problem being solved. The processors may be linked as a binary tree.

The same remark applies to the procedures *fur*, *bur*, *mur* and *tur*.

INDEX

$\bar{A}(v)$	12	elimination graph	13
$\bar{\bar{A}}(v)$	12	envelope	8
adjacency set	11	external node	44
ancestor vertex	11	fill-in	13
assemble	55	forward substitution	6
backward substitution	6	frame structure	73
bandwidth	8	frond	11
blended element	79	*fur*	57
blending interpolation	78	G^*_α	13
block-matrix	20	graph	11
block-pivot	6	internal node	44
bur	59	key diagram	82
Cholesky decomposition	6	leaf vertex	11
connected component	11	*lm*	80
connected graph	11	LU-decomposition	1
connection graph	12	*lur*	68
connection matrix	41	mesh generation	42
consistent ordering	15	*mur*	67
c_Q	8	nested dissection ordering	37
$\bar{D}(v)$	12	nested dissection partition	37
$\bar{\bar{D}}(v)$	12	node	42
$D(v)$	13	ordered graph	12
decomposable matrix	13	palm	11
decompose	55	palm forest	11
decomposed (structure) matrices	57	partial decomposition	6
decomposition graph	13	partition	12
deficiency	13	path	11
dense	9	perfect partition	23
descendant vertex	11	p-partition	20
directed graph	10	pp-partition	24
dissection tree	37	predecessor vertex	11
edge	10	preserving palm	17
element	41	preserving partition	20
element matrix	41	profile	9
element vector	41	proper palm	25

proper p-partition 25
quotient graph 12
reduced structure matrix 45
reduced structure vector 45
reducible matrix 13
root 11
rooted tree 11
r_Q 8
section graph 11
separator 11
serendipity element 77
shape function 39
sparse matrix algorithm 10
strongly connected graph 11
structure 41
structure matrix 41
structure vector 41
substituted structure vector 58
substructure 44
successor vertex 11
tree 11
tur 69
undirected graph 10
ur 53
vertex 10

REFERENCES

Axelsson, O., Solution of linear systems of equations: iterative methods, in: Sparse matrix techniques, Copenhagen 1976 (V.A. Barker ed.), Berlin, Springer, 1977.

Bunch, J.R. and D.J. Rose, Partitioning, tearing and modification of sparse linear systems. J. Math. Anal. Appl. 48 (1974), 574-593.

Cavendish, J.C., Local mesh refinement using rectangular blended finite elements. J. Comp. Phys. 19 (1975), 211-228.

Clough, R.W., The finite element in plane stress analysis, in: Proceedings second ASCE conference on electronic computation, Pittsburgh, ASCE, 1960.

Courant, R., Variational methods for the solution of problems of equilibrium and vibrations. Bull. Amer. Math. Soc. 49 (1943), 1-23.

Cuthill, E., Several strategies for reducing the bandwidth of matrices, in: Sparse matrices and their applications (D.J. Rose and R.A. Willoughby eds.), New York, Plenum Press, 1972.

Dijkstra, E.W., Notes on structured programming, in: Structured programming (O.J. Dahl, E.W. Dijkstra and C.A.R. Hoare), London, Academic Press, 1972.

Duff, I.S., A survey of sparse matrix research. Proc. IEEE 65 (1977), 500-535.

Duff, I.S., A.M. Erisman and J.K. Reid, On George's nested dissection method. SIAM J. Numer. Anal. 13 (1976), 686-695.

Eisenstat, S.C., M.H. Schultz and A.H. Sherman, Applications of an element model for Gaussian elimination, in: Sparse matrix computations (D.J. Rose and J.R. Bunch eds.), New York, Academic Press, 1976.

Eisenstat, S.C., M.H. Schultz and A.H. Sherman, Software for sparse Gaussian elimination with limited core storage. Yale University, Department of Computer Science, 1978.

Frederisson, B. and J. Mackerle, Structural mechanics finite element computer programs - surveys and availability. Linköping, Linköping

Institute of Technology, 1976.

Garey, M.R., D.S. Johnson and P.K. Stockmeyer, Some simplified NP-complete problems. Proceedings sixth annual ACM symposium on theory of computing, 1974.

George, J.A., Computer implementation of the finite element method. Stanford, Stanford University, Department of Computer Science, 1971.

George, J.A., Nested dissection of a regular finite element mesh. SIAM J. Numer. Anal. 10 (1973), 345-363.

George, J.A., On block elimination for sparse linear systems. SIAM J. Numer. Anal. 11 (1974), 585-603.

George, J.A., Solution of linear systems of equations: direct methods for finite element problems, in: Sparse matrix techniques, Copenhagen 1976 (V.A. Barker ed.), Berlin, Springer, 1977.

George, J.A. and J.W.H. Liu, Some results on fill for sparse matrices. SIAM J. Numer. Anal. 12 (1975), 452-455.

George, J.A. and J.W.H. Liu, An automatic nested dissection algorithm for irregular finite element problems. SIAM J. Numer. Anal. 15 (october 1978), 1053-1069.

George, J.A. and J.W.H. Liu, Algorithms for matrix partitionings and the numerical solution of finite element systems. SIAM J. Numer. Anal. 15 (april 1978), 297-327.

George, J.A. and D.R. McIntyre, On the application of the minimum degree algorithm to finite element systems. SIAM J. Numer. Anal. 15 (1978), 90-112.

Gordon, W.J. and C.A. Hall, Transfinite element methods: blending function interpolation over arbitrary curved element domains. Numer. Math. 21 (1973), 109-129.

Gustavson, F.G., Some basic techniques for solving sparse systems of linear equations, in: Sparse matrices and their applications (D.J. Rose and R.A. Willoughby eds.), New York, Plenum Press, 1972.

Hoffman, A.J., M.S. Martin and D.J. Rose, Complexity bounds for regular finite difference and finite element grids. SIAM J. Numer. Anal. 10 (1973), 364-369.

Irons, B.M., A frontal solution program. Int. J. Numer. Meth. Eng. 2 (1970), 5-32.

Jensen, K. and N. Wirth, PASCAL user manual and report, second edition. New York, Springer, 1978.

Kunth, D.E., The art of computer programming, second edition, volume 1: Fundamental algorithms. Reading, Addison-Wesley, 1975.

Lipton, R.J., D.J. Rose and R.E. Tarjan, Generalized nested dissection. SIAM J. Numer. Anal. 16 (1979), 346-358.

Norrie, D.H. and G. de Vries, Finite element bibliography. New York, Plenum Press, 1976.

Papadimitriou, C.H., The NP-completeness of the bandwidth minimization problem. Computing 16 (1976), 263-270.

Parter, S.V., The use of linear graphs in Gauss elimination. SIAM Rev. 3 (1961), 119-130.

Peters, F.J., A novel implementation of finite element algorithms. Eindhoven, Eindhoven University of Technology, Department of Mathematics, 1979.

Peters, F.J., FEMSYS, een systeem voor op de eindige-elementenmethode gebaseerde berekeningen. Eindhoven, Eindhoven University of Technology, Department of Mathematics, 1976.

Peters, F.J., Another organization of finite element calculations and its implementation with ICETRAN, in: Proceedings XX ICES users group worldwide conference (A.N. Natali ed.), Padua, Padua University, 1978.

Przemieniecki, J.S., Theory of matrix structural analysis. New York, Mc Graw-Hill, 1968.

Rose, D.J., A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations, in: Graph theory and computing (R. Read ed.), New York, Academic Press, 1971.

Rose, D.J. and R.E. Tarjan, Algorithmic aspects of vertex elimination, in: Proceedings Seventh Annual ACM Symposium on Theory of Computing, 1975.

Schoofs, A.J.G., L.H.T.M. van de Beukering and M.L.C. Sluiter, General purpose 2-dimensional mesh generator. Eindhoven, Eindhoven University of Technology, Department of Mechanical Engineering, 1979.

Speelpenning, B., The generalized element method. Urbana, University of Illinois at Urbana-Champaign, Department of Computer Science, 1978.

Strang, G. and G.J. Fix, An analysis of the finite element method. Englewood Cliffs, Prentice-Hall, 1973.

Tarjan, R.E., Depth-first search and linear graph algorithms. SIAM J. Comput. 1 (1972), 146-160.

Tarjan, R.E., Graph theory and Gaussian elimination, in: Sparse matrix computations (J.R. Bunch and D.J. Rose eds.), New York, Academic Press, 1976.

Turner, M.J., R.W. Clough, H.C. Martin and L.J. Topp, Stiffness and deflection analysis of complex structures. J. Aeron. Sci. 23 (1956), 805-823.

Varga, R.S., Matrix iterative analysis. Englewood Cliffs, Prentice-Hall, 1962.

Wilkinson, J.H., The algebraic eigenvalue problem. Oxford, Oxford University Press, 1965.

Williams, F.W., Comparison between sparse stiffness matrix and substructure methods. Int. J. Numer. Meth. Eng. 5 (1973), 383-394.

Wirth, N., Program development by step-wise refinement. Comm. ACM 14 (1971), 221-227.

Wirth, N., The programming language PASCAL. Acta Inf. 1 (1971), 35-63.

Zienkiewicz, O.C., The finite element method, third edition. London, Mc Graw-Hill, 1977.

Zienkiewicz, O.C. and D.V. Phillips, An automatic mesh generation scheme for plane and curved surfaces by isoparametric coordinates. Int. J. Numer. Meth. Eng. 3 (1971), 519-528.

Zlamal, M., On the finite element method. Numer. Math. 12 (1968), 394-409.

SAMENVATTING

De oplossing van een stelsel van n lineaire vergelijkingen in n onbekenden kan (onder zekere voorwaarden) worden verkregen met behulp van Gaussische eliminatie of van (wat in feite op hetzelfde neerkomt) LU-decompositie. Hierbij worden een benedendriehoeksmatrix L en een bovendriehoeksmatrix U bepaald, zodanig dat LU = Q, waarbij Q de bij het stelsel behorende matrix is. Grote stelsels vergelijkingen zijn doorgaans ijl, dat wil zeggen dat slechts een gering aantal coëfficiënten van de matrix Q ongelijk aan nul is. Bovendien zijn de driehoeksmatrices L en U meestal eveneens ijl. In de loop der tijden zijn er ingewikkelde zogenaamde ijle matrix algorithmen ontwikkeld, teneinde die driehoeksmatrices op een efficiënte wijze te verkrijgen, waarbij efficiëntie inhoudt dat arithmetische operaties met coëfficiënten die nul zijn, worden vermeden. Het is bekend dat de volgorde van de variabelen en vergelijkingen niet alleen van invloed is op de ijlheid van de driehoeksmatrices, maar ook op het vereiste aantal arithmetische operaties met coëfficiënten die ongelijk nul zijn.

In dit proefschrift tonen we met behulp van grafentheoretische terminologie aan dat het (bij de gegeven ordening van de vergelijkingen en onbekenden) altijd mogelijk is om de verzameling variabelen zo te partitioneren dat de driehoeksmatrices L en U verkregen kunnen worden door het decomponeren van kleinere matrices, die bepaald zijn door de partitie. Omdat er geen nullen voorkomen in de zogenaamde enveloppen van de bij de kleinere matrices behorende driehoeksmatrices, zijn geen ingewikkelde ijle matrix algorithmen nodig om arithmetische operaties met nullen te vermijden. We kunnen volstaan met eenvoudiger (namelijk envelop-) algorithmen en daarmee toch de coëfficiënten van L en U met het geringste aantal arithmetische bewerkingen verkrijgen.

Grote, ijle stelsels vergelijkingen komt men onder andere tegen in de eindige-elementenmethode, een wijdverbreide methode om zekere typen partiele differentiaalvergelijkingen op te lossen. De hierboven beschreven resultaten voor het oplossen van lineaire stelsels geven aanleiding tot een nieuwe organisatie van eindige-elementenberekeningen. In plaats van één groot stelsel wordt een aantal kleinere opgesteld, of, in eindige-elemen-

tenterminologie, in plaats van aan één grote structuur wordt aan een aantal kleinere substructuren gerekend.

Voor (niet noodzakelijkerwijs homogene) n × m netwerken worden algorithmen in PASCAL beschreven. Deze nieuw ontwikkelde recursieve algorithmen onderscheiden zich van bestaande doordat een aantal traditioneel opeenvolgende stappen verstrengeld zijn. Bovendien is het onnodig andere gegevens dan matrixcoëfficiënten expliciet op te slaan. Met name worden geen administratieve gegevens zoals verwijzingen, tellers enz. gebruikt. De algorithmen zijn conceptueel eenvoudig. Een implementatie van deze algorithmen bleek meer dan een factor 50 sneller dan een veel gebruikt programma pakket voor technische berekeningen. Besparingen in geheugengebruik zijn eveneens aanzienlijk. De algorithmen zijn bruikbaar voor willekeurige tweedimensionale structuren. Op overeenkomstige wijze kunnen algorithmen voor driedimensionale structuren echter ook ontwikkeld worden.

CURRICULUM VITAE

De schrijver van dit proefschrift werd op 28 september 1945 in Emmen geboren. Aan het Gemeentelijk Lyceum te Emmen volgde hij een gymnasiumopleiding tot en met klas 4. In 1964 verkreeg hij op het Twents Carmellyceum te Oldenzaal het gymnasium-β diploma. Van 1 augustus 1968 tot 1 augustus 1969 was hij aan dat lyceum verbonden als leraar wiskunde. In 1970 behaalde hij het doctoraal examen wiskunde met hoofdvak algebra aan de R.K. Universiteit te Nijmegen. Sinds 16 maart 1970 is hij eerst als wetenschappelijk medewerker, daarna als wetenschappelijk hoofdmedewerker verbonden aan de Onderafdeling der Wiskunde van de Technische Hogeschool te Eindhoven. Hij is daar werkzaam in de vakgroep Informatica bij Prof.dr. R.J. Lunbeck. Van 1 januari 1978 tot 1 januari 1979 verbleef hij te Ispra (Italië) om onderzoek te verrichten in het Gemeenschappelijk Onderzoekcentrum van de Europese Gemeenschappen.

STELLINGEN

1.

De opmerking van George en Liu dat de pseudo-diameter van een ongerichte graaf weinig afwijkt van de diameter is niet waar.

A. George en J.W.H. Liu, An automatic nested dissection algorithm for irregular finite element problems. SIAM J. Numer.Anal.15 (1978), 1053 - 1069.

2.

Zij a(M) het aantal arithmetische bewerkingen met niet-nul matrixcoëfficiënten dat nodig is voor de LU-decompositie van een matrix M. Als Q een volle bandmatrix is, dan geldt voor iedere permutatiematrix P:

$$a(Q) \leq a(PQP^t)$$

3.

Zij B een $n \times n$ matrix met volle band en een bandbreedte die klein is ten opzichte van n. Laat T de tijd zijn die nodig is voor LU-decompositie van B met één processor. Als er p ($1 < p << n$) onafhankelijke processoren, ieder met een eigen geheugen, zijn, dan kan voor bepaalde permutatiematrices P de LU-decompositie van PBP^t bepaald worden in een tijd ongeveer T/p. Alleen als $p = 2$ kan die reductie in tijd bereikt worden zonder dat het totale aantal arithmetische operaties groter is dan met één processor.

4.

Het gebruik van met een random generator voortgebrachte testmatrices bij het onderzoek van ijle matrix algorithmen is zinloos.

I.S. Duff, A survey of sparse matrix research.
Proc.IEEE 65 (1977), 500 - 535.

5.

Dat Rose en Whitten's algorithme, in tegenstelling tot het algorithme van Duff e.a., de vier hoekpunten van een rechthoekig netwerk het eerst nummert, is geen verklaring voor de geconstateerde verschillen in het aantal arithmetische operaties.

I.S. Duff, A.M. Erisman en J.K. Reid, On George's nested dissection method. SIAM J. Numer. Anal. 13 (1976), 686 - 695.

6.

Ondanks het belang dat Zienkiewicz hecht aan "blending processes" zijn in diens leerboek de formules (8.40) tot en met (8.43) die daarop betrekking hebben, fout.

O.C. Zienkiewicz, The finite element method.
Third edition. London, Mc Graw-Hill, 1977.

7.

Naast een analogon van de procedure *ur* uit dit proefschrift voor driehoekige structuren bestaat er ook een analogon voor L-vormige structuren.

8.

De procedures uit hoofdstuk 4 van dit proefschrift zijn geschikt als leidraad bij het ontwerp van een eindige-elementenautomaat bestaande uit een groot aantal samenwerkende, onderling identieke machines.

9.

Het is onvoldoende bekend dat een computer geen informatie scheppende, doch integendeel een doorgaans uiterst effectieve informatie vernietigende machine is.

10.

Als men bij de aangifte voor de inkomstenbelasting giften aan instellingen aftrekt, dan komt dat in feite neer op subsidiëring van die instellingen met geld van een ander, namelijk de overheid. Het fiscaal aftrekbaar stellen van giften is derhalve verwerpelijk.

11.

De uitspraak "parapsychologische verschijnselen zijn hersenspinsels" wordt niet weerlegd in de literatuur die wordt aanbevolen bij de cyclus Parapsychologie van het Studium Generale.

Eindhoven, 7 december 1979 F.J. Peters