Czech Technical University in Prague
Faculty of Mechanical Engineering

Ing. Petr Pařík

An Out-of-core Sparse Direct Solver for Very Large Finite Element Problems

Ph.D. Thesis

Prague 2011

Petr Pařík

An Out-of-core Sparse Direct Solver for Very Large Finite Element Problems

Ph.D. Thesis

Czech Technical University in Prague
2011

Contents

Acknowledgements

The work presented in this thesis was carried out at the Department of Impact and Waves in Solids of the Institute of Thermomechanics, Academy of Sciences of the Czech republic.

I would like to express the deep gratitude and thanks to my supervisor Dr. Jiří Plešek for his patience with my work and for all the motivation and guidance endowed upon me. I would also like to thank all members of the department for their willingness to discuss and comment my work at any time, and for their kind and friendly attitude. Last but not least, I would like to thank my parents Hana and Miroslav for their unconditional love and support which enabled me to pursue the highest goals in life.

Abstract

An out-of-core sparse direct solver for very large finite element problems

This thesis is focused on enhancing numerical methods used in the direct solution of large linear equation systems that arise in practical application of the finite element method (FEM) in solid continuum mechanics. A linear equation system forms the basis of every FEM problem, therefore, its fast and efficient solution is always desirable. Large problems are defined as problems for which the requirements on memory storage and computational time make the solution difficult using available computers. Very large problems cannot be solved unless the solution is performed partially out of core, using a disk storage, which unfortunately reduces efficiency.

The first part of this thesis presents a critical overview of fundamental methods used for the solution of linear systems of equations, such as storage methods for coefficient matrices, ordering methods, and solution methods, and also discusses common direct solver implementations. Next part describes in detail the selected K3 sparse matrix storage format, the approximate minimum degree ordering (AMD), and the symmetric LU factorization. In the final part, the enhancements to the aforementioned numerical algorithms are proposed with regard to the out-of-core solution of large sparse symmetric positive-definite linear systems. In particular, an efficient block sparse storage format for the coefficient matrix based on the K3 format is proposed, together with a modified minimum degree algorithm that includes symbolic assembly and works on the block nonzero structure of the matrix, also including a modified left-looking factorization algorithm that has low storage requirements suitable for out-of-core solution.

The proposed numerical algorithms are assessed by means of an out-of-core sparse direct solver implemented into the PMD system, an in-house software package for FEM analysis. Results are presented for both test problems and several large real-world engineering problems. Furthermore, a performance comparison with the existing PMD's linear solver, based on the frontal solution method, is presented, which demonstrates the validity and efficiency of the proposed algorithms.

Notation and symbols

$\mathbf{A}$	coefficient matrix
$\tilde{\mathbf{A}}$	permuted coefficient matrix
$\mathbf{A}_{ij}$	submatrix of the coefficient matrix
$\mathbf{A}_k$	partially reduced coefficient matrix
$\mathcal{A}$	set of variables adjacent to a variable in the quotient graph
a_{ij}	entry of the coefficient matrix
$a_{ij}^{(k)}$	entry of the partially reduced coefficient matrix
$\mathbf{b}$	right-hand side vector
$\mathbf{D}$	diagonal matrix factor
$\mathbf{D}_{ii}$	submatrix of the matrix factor
d	order of block submatrix
d	number of nodal degrees of freedom
d	external degree
$\hat{d}$	approximate degree
$\tilde{d}$	approximate degree
$\overline{d}$	approximate degree
d_{ii}	entry of the matrix factor
E	set of graph edges
E	set of edges between variables in the quotient graph
$\overline{E}$	set of edges between variables and elements in the quotient graph
$\mathcal{E}$	set of elements adjacent to a variable in the quotient graph
G	graph
G	elimination graph
$\mathcal{G}$	quotient graph
k	actual elimination step
$\mathbf{L}$	unit lower triangular matrix factor
$\tilde{\mathbf{L}}$	lower triangular Cholesky matrix factor
$\mathbf{L}_{ji}$	submatrix of the matrix factor
$\mathcal{L}$	set of variables adjacent to an element in the quotient graph
L	length of matrix storage
L_{nz}	length of all nonzero submatrices in the coefficient matrix
l_{ji}	entry of the matrix factor
M	number of matrix block rows
m	number of matrix rows

N	number of mesh nodes
N	number of matrix block columns
n	number of matrix columns
n	order of square matrix
N_{nz}	number of nonzero blocks in matrix
n_{nz}	number of nonzero entries in matrix
$\mathbf{P}$	left permutation matrix
p	actual pivot
$\mathbf{Q}$	right permutation matrix
$\mathbf{Q}$	orthogonal matrix factor
$\mathbf{R}$	upper triangular matrix factor
$\mathcal{S}$	set of variables in supervariable in the quotient graph
t	true degree
$\mathbf{U}$	upper triangular matrix factor
V	set of graph vertices
V	set of variables in the quotient graph
$\overline{V}$	set of elements in the quotient graph
$\mathbf{x}$	solution vector
$\mathbf{y}$	reduced right-hand side vector
$\mathbf{z}$	permuted solution vector

`BUF(LBUF)`	buffer for equations (coefficient matrix)
`ICL`	integer-to-real size factor
`IDCOM`	file *name*.CMN, common problem data
`IDDM1`	file *name*.DM1, auxiliary data (frontal solver)
`IDELM`	file *name*.ELM, element stiffness matrices
`IDEQ1`	file *name*.EQ1, factorized equations (frontal solver)
`IDEQC`	file *name*.EQC, auxiliary data (sparse direct solver)
`IDEQI`	file *name*.EQI, matrix block indices (sparse direct solver)
`IDEQR`	file *name*.EQR, factorized equations (sparse direct solver)
`IDP`	file *name*.P, mesh data
`IDRHS`	file *name*.RHS, right-hand side vectors
`IDSOL`	file *name*.SOL, solution vectors
`IFIXV(NFIXV)`	indices of fixed variables (constrained degrees of freedom)
`INET(LINET)`	indices of nodes in elements
`INT`	integer workspace
`IPNOD(NNOD)`	permutation vector applied to nodes
`IPSOL(LSOL)`	permutation vector applied to degrees of freedom
`KFES`	sparse direct solver status
`LBUF1`	actual size of array `BUF` in assembly ($\leq$ `SBUF1`)
`LBUF2`	actual size of array `BUF` in factorization ($\leq$ `SBUF2`)
`LI`	size of the workspace

LINET	length of array INET
LNNDF	length of array NNDF
LNNET	length of array NNET
LSMTX	maximum length of nodal submatrix (block)
LSOL	length of the solution vector (including reaction forces)
name	problem data name
MBD(NNOD)	number of variables per block
MBP(NBLK2)	pointers to blocks (nodal submatrices)
MCI(NBLK2)	nodal column indices for matrix blocks
MRP(NPIV+1)	nodal row pointers for matrix blocks
NASV	number of load cases (right-hand sides)
NBLK1	number of nonzero blocks in assembled matrix
NBLK2	number of nonzero blocks in factorized matrix
NELEM	number of mesh elements
NFIXV	number of fixed variables
NNDF(LNNDF)	number of degrees of freedom per node
NNET(LNNET)	number of nodes per element
NNOD	number of mesh nodes
NPIV	number of pivot blocks (NNOD minus number of constrained nodes)
NVAR	number of variables (LSOL minus number of reaction forces)
R	real number workspace
RHS(LSOL,NASV)	right-hand side vectors
SBUF1	summed length of blocks in assembled matrix
SBUF2	summed length of blocks in factorized matrix

$\div$	integer division operator, used to distinguish integer division (that involves a remainder) from real-number division (operator /)

Chapter 1

Introduction

The finite element method [5] is without doubt one of the most important numerical techniques available in solid continuum mechanics. By definition, it is a variational method for the approximate solution of boundary value problems, where partial differential equations are transformed into a corresponding ordinary differential system, or in the case of steady state problems, into a corresponding algebraic system. The finite element method was implemented on computers uncountable times since its early years in 1950s. Today, many software packages, both commercial and free, are available that provide the capabilities for a comprehensive finite element analysis in many areas of solid and fluid continuum mechanics, beginning with the creation of the finite element mesh (usually automatized) and ending with the evaluation of the numerical results aided by their visual representation, using various contour plots. Robustness, ease of use, and readily available software significantly contributed to the popularity of the finite element method in engineering practice in past decades.

An important advantage of the finite element method is that it is not limited to simple domains or to uniform regions. Meshes can be of any shape, and different elements can have different sizes and shapes. Therefore, quite complicated domains, such as whole machines, can be approximated very closely, provided that the mesh is sufficiently fine. The accuracy of the numerical solution obtained by the finite element method can be easily affected by refining or coarsening the size and shape of the elements and, therefore, theoretically arbitrary precision can be achieved in any part of the mesh. In reality, refining of the finite element mesh is severely limited, since finer meshes require substantially more storage space and computational time in order to be processed. Therefore, finite element meshes are usually constructed to be finer in the parts of interest, and coarser in the other parts. Unfortunately, parts of the mesh that have complex shapes also need to be refined so that they appropriately represent the geometry, thus, the ultimate number of equations that describe the problem can be often relatively high.

Some fifty years ago, the size of the largest problem computable by the finite element method using a state-of-the-art computer was only one or two hundred equations, and the problems were limited to very simple geometries and a few mesh elements. Today, computers are incomparably more powerful and have much larger capacity in storage space, which allows the finite element problems to have about 10^5 to 10^7 equations. Of

course, these sizes are relevant to steady state problems that lead to a system of linear equations.

In the past decades, much attention has been directed towards an efficient implementation of solvers for large finite element problems. Direct methods, based on the Gaussian elimination, could not be overly used due to the limited capacity of available computers, which favored iterative methods that had lower computational demands but were much less robust. With the increasing performance of computers in late 1980s it became possible to implement skyline direct solvers, which exploited the fact that the skyline of the coefficient matrix was retained during the factorization. Today, skyline solvers are still mistakenly considered by some as the ultimate direct solvers for large finite element problems. However, the research conducted in the last decade showed that very large finite element problems could be solved more efficiently by a general sparse direct solver that would work only with nonzero entries of the coefficient matrix.

The work presented in this thesis has been motivated by the need for a new direct linear solver in the PMD finite element system (see Section 2.4.2 and Section 4.4) to allow efficient computations on large finite element problems. The PMD's existing linear solver has been developed in late 1970s and is based on the frontal solution method [24] that is memory-efficient and robust. However, this method is generally unsuitable for large finite element problems, since the demands on the disk (out-of-core) storage and the solution time are impractical in most cases. State-of-the-art preordering techniques that reduce storage requirements and computational costs cannot be applied to the frontal solution method, and therefore a completely new solver code has to be developed. The performance of the existing frontal solver is also a limiting factor in the application of some more robust solution methods for nonlinear and dynamic problems, where the need for solving a large linear equation system occurs repetitively and thus an efficient implementation is crucial.

The basic methods involved in a sparse direct solution are the matrix storage method, the (pre)ordering method, and the solution (factorization) method, which are all interdependent to some degree. The storage of the whole coefficient matrix (or approximately one half in symmetric cases) is obviously never acceptable except the smallest problems that are however not practical. Matrix storage schemes thus try to exploit sparsity, symmetry and other properties of the coefficient matrix to store as few zero entries of the matrix as possible. Although it is perfectly reasonable not to store any nonzero entries, which is clearly the most efficient option and is indeed used for example in iterative methods, direct methods unfortunately spoil the sparsity structure by introducing new nonzero entries during the triangularization (factorization) of the coefficient matrix. This occurence of new nonzeros is called the *fill-in* and presents a major drawback and difficulty of direct methods. The initial nonzero structure of the matrix as well as the final nonzero structure (the amount of fill-in) can be substantially affected by using an ordering method. Storage of the coefficient matrix in the case of large problems requires a careful consideration, since it has a significant impact on the practical implementation of the solution method.

Ordering methods switch rows and columns of the matrix to obtain another, preferably more suitable, order of pivots on the main diagonal. An important consequence is that the resulting nonzero structure of the reordered matrix may allow an efficient storage and/or

factorization using direct methods. Different ordering methods usually imply certain types of matrix storage schemes. For example, profile minimization algorithms move all nonzero entries close to the main diagonal, yielding a band or skyline matrix, therefore, a band or skyline storage format is ideal. The fill-in can occur only within the band or under the skyline of the coefficient matrix, thus it is also effectively reduced. Fill-in minimization algorithms are used specifically to reduce the fill-in, but they result in a more complicated nonzero structure requiring more complex storage schemes. Ordering methods usually work with graphs representing the nonzero structure of the matrix, and since operations on graphs are computationally expensive, the time complexity rises quickly in the case of large problems. However, without a suitable ordering, the direct solution of large problems is generally impossible due to uncontrollable fill-in.

The sparse direct solution is mostly performed using a variant of the Gaussian elimination. In order to be efficient, the method must exploit the sparsity of the coefficient matrix by avoiding unnecessary operations on zero entries. This largely depends on the employed matrix storage scheme and also on the used ordering method. Small and medium problems can be usually fully solved in memory (in core), but in the case of large problems, it may be necessary to store a part of the matrix data temporarily on the disk (out of core). Out-of-core solution of course involves much more complicated algorithms and since the disk storage is much slower than the memory storage, great care must be taken to implement the solution method efficiently.

The thesis is organized in the following way. Overview of the numerical methods used in linear equation solvers is presented in Chapter 2. Based on the state of the art summarized in the overview, the particular aims of the thesis are formulated in Chapter 3. Chapter 4 explains in detail the methods mentioned in the overview which have been selected as the basis for the sparse direct solver implementation. Chapter 5 describes and discusses the proposed modifications to the applied methods and algorithms, deals with the sparse direct solver implementation, and presents applications in the finite element analysis, including a comparison with the frontal solver. Finally, some conclusions are drawn in Chapter 6, summarizing results and pointing to further research.

Publications

Papers listed below were published during the course of the work and were compiled into the thesis.

- Pařík P. and Plešek J. (2009). Assessments of the implementation of the minimum degree ordering algorithms. *Pollack Periodica, International Journal for Engineering and Information Sciences*, **4**, 3, pp. 121–128.

- Pařík P. (2009). Performance tests of the minimum degree ordering algorithm. *Engineering Mechanics 2009*, pp. 929–935.

- Pařík P. (2008). Sparse storage schemes. In: Okrouhlík M., editor. *Numerical methods in computational mechanics*, pp. 272–281. Institute of Thermomechanics ASCR, Prague.
- Pařík P. (2008). Implementation of a sparse direct solver for large linear systems. *Výpočty konstrukcí metodou konečných prvků 2008*, pp. 98–101.
- Pařík P. (2007). Sparse direct solver with fill-in optimization. *Engineering Mechanics 2007*.
- Pařík P. (2005). Numerická implementace lineárního řešiče na bázi algoritmu AMD. *Summer Workshop of Applied Mechanics 2005*, pp. 75–84.

Chapter 2

Overview

In this chapter, the state of the art in the numerical techniques for solving linear equation systems is presented, with a special interest in the direct solution of large systems with sparse symmetric positive definite coefficient matrices, that are the most common in the finite element analysis of solids and structures. The first section summarizes the methods available for solving linear equation systems. The second section deals with the ordering methods and their importance in the direct solution of large systems. The third section describes the storage methods for dense and sparse matrices. Finally, the last section discusses direct solvers and their common implementations.

2.1 Solution of linear systems

A set of simultaneous linear equations (linear equation system) can be written conveniently in the matrix form

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{2.1}$$

where $\mathbf{A}$ is the coefficient matrix, $\mathbf{x}$ is the solution vector and $\mathbf{b}$ is the right-hand side vector. The straightforward solution readily obtained from equation (2.1) is

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}, \tag{2.2}$$

where $\mathbf{A}^{-1}$ is the inverse of matrix $\mathbf{A}$. However, computing the inverse is almost never an appropriate nor computationally feasible way for solving system (2.1).

A linear equation system with a handful of unknowns may of course be solved by virtually any method, but as the order of the system (number of equations) increases, the choice of proper numerical techniques quickly becomes crucial. If chosen unwisely, the time and/or storage space required to obtain the solution may be either too large, or the solution may not be possible at all.

Numerical methods for solving the linear equation system (2.1) are divided into two distinct classes, direct methods and iterative methods.

In direct methods, an exact solution is obtained after a finite number of operations. The accuracy of a direct solution is however affected by the employed finite-precision arithmetic.

Year	Order
1970	200
1975	1,000
1980	10,000
1985	100,000
1990	250,000
1995	1,000,000
2005	10,000,000

Table 2.1: Sparse linear equation systems solvable by direct methods in practice as a function of date

In iterative methods, an exact solution would normally be obtained only after an infinite number of operations, hence the accuracy of an iterative solution depends on the chosen stopping criterion.

Traditionally, direct methods are used for small to large banded or skyline systems, while iterative methods are considered most appropriate for very large sparse systems. The definition of *large* or *very large* problem has changed considerably through the past decades, as is illustrated in Table 2.1, which has been adopted from [37]. It can be seen that the application of direct methods to large linear systems has become a practical issue only recently. In the past decade, an efficient numerical implementation of direct methods for very large sparse systems has been a topic of continual research.

2.1.1 Direct methods

Direct methods are based on the factorization of the coefficient matrix $\mathbf{A}$. In a direct method, the matrix is decomposed into a product of two or three factors (hence the term factorization or decomposition), which represent triangular or diagonal systems that can be solved easily by substitution using the right-hand side. Most direct methods employ some variant of the Gaussian elimination to obtain the factors.

The factorization is the most computationally expensive part of the solution process, while the complexity of the substitution part is an order of magnitude less. However, the factorization needs to be carried out only once for a given system, since the factors can then be used to compute the solution for several different right-hand sides with a substantially less effort. This fact can often justify the high costs of the factorization and also presents a significant advantage over iterative methods.

A detailed explanation of direct methods can be found for example in [36], and a more practical approach focused specifically on the finite element method is given for example in [5].

LU factorization

An *LU factorization* of a square matrix $\mathbf{A}$ takes the form

$$\mathbf{A} = \mathbf{LU}, \tag{2.3}$$

where $\mathbf{L}$ is a lower triangular matrix and $\mathbf{U}$ is an upper triangular matrix. The factors are not defined uniquely by equation (2.3) unless further constraints are imposed. To obtain a unique LU factorization, $\mathbf{L}$ is usually limited to a unit lower triangular matrix.

Substituting equation (2.3) into equation (2.1) yields

$$\mathbf{LUx} = \mathbf{b}. \tag{2.4}$$

Using another substitution

$$\mathbf{Ux} = \mathbf{y}, \tag{2.5}$$

the solution of system (2.4) can be divided into two parts.

The first part of the solution, called *forward substitution*, is

$$\mathbf{Ly} = \mathbf{b}, \tag{2.6}$$

from which the reduced right-hand side vector $\mathbf{y}$ is obtained.

The second part of the solution, called *back substitution*, is

$$\mathbf{Ux} = \mathbf{y}, \tag{2.7}$$

from which the solution vector $\mathbf{x}$ and the solution of systems (2.4) and (2.1) is obtained.

The solution of system (2.1) using the LU factorization is easy to carry out since systems (2.6) and (2.7) are triangular. Other direct methods employ a similar procedure for computing the solution using the factors.

Cholesky factorization

Let $\mathbf{A}$ be a *symmetric matrix*. Then

$$\mathbf{A}^{\mathrm{T}} = \mathbf{A}. \tag{2.8}$$

Let $\mathbf{A}$ be a *positive definite matrix*. Then it is symmetric and

$$\mathbf{u}^{\mathrm{T}}\mathbf{Au} > \mathbf{0} \tag{2.9}$$

for any real nonzero vector $\mathbf{u}$.

A *Cholesky factorization* of a symmetric positive definite matrix $\mathbf{A}$ takes the form

$$\mathbf{A} = \tilde{\mathbf{L}}\tilde{\mathbf{L}}^{\mathrm{T}}, \tag{2.10}$$

where $\tilde{\mathbf{L}}$ is a lower triangular matrix. The advantage of this method is that, unlike the other methods, it is always numerically stable, therefore no pivoting is necessary. Cholesky factorization is about twice as efficient as the LU factorization in both the computational costs and the storage requirements. Aside from solving system (2.1) it is also often used to check for the positive definiteness of a matrix.

$\mathbf{LDL}^{\mathrm{T}}$ factorization

An $\mathbf{LDL}^{\mathrm{T}}$ *factorization* of a symmetric matrix $\mathbf{A}$ takes the form

$$\mathbf{A} = \mathbf{LDL}^{\mathrm{T}}, \tag{2.11}$$

where $\mathbf{L}$ is an unit lower triangular matrix and $\mathbf{D}$ is a diagonal matrix. The solution is computed in the same way as in the LU factorization, substituting $\mathbf{U} \equiv \mathbf{DL}^{\mathrm{T}}$. This method is of particular interest in numerical analysis since it is comparable to the Cholesky factorization in efficiency, but is applicable to both positive definite and indefinite matrices.

QR factorization

Let $\mathbf{Q}$ be an *orthogonal matrix*. Then

$$\mathbf{Q}^{\mathrm{T}} = \mathbf{Q}^{-1}. \tag{2.12}$$

A *QR factorization* of a matrix $\mathbf{A}$ takes the form

$$\mathbf{A} = \mathbf{QR}, \tag{2.13}$$

where $\mathbf{Q}$ is an orthogonal matrix and $\mathbf{R}$ is an upper triangular matrix. The factors are computed using orthogonalization algorithms such as the Gram-Schmidt process, Householder transformations or Givens rotations. QR factorization can be used to solve system (2.1), but its computational costs are high over the LU factorization. However, it is valuable for solving overdetermined systems, i.e., when the coefficient matrix $\mathbf{A}$ in equation (2.1) is rectangular, which is a common case in the least squares problems.

Numerical stability and pivoting

Unless the matrix is symmetric and positive definite, the factorization can run into difficulties when any diagonal entry (pivot) is zero or very small relative to other row entries. To prevent a numerical breakdown of the factorization, partial or full pivoting must be applied to the matrix.

In *partial pivoting*, the entry with the largest absolute value in the pivot column is chosen as the next pivot, switching the correspoding rows. Unlike full pivoting, partial pivoting does not change the order of unknowns.

In *full pivoting*, the entry with the largest absolute value in the remaining uneliminated rows is chosen as the next pivot, switching the correspoding rows and columns.

Partial pivoting is sufficient to ensure the numerical stability of the factorization in most cases. Generally, it is ill-advised to perform the factorization on indefinite matrices without some form of pivoting.

Feature	Direct methods	Iterative methods
Accuracy	not susceptible	can be chosen
Computational costs	predictable	mostly unpredictable, but often low
New right-hand sides	fast	no saving of time
Storage requirements	more	less
Initial guess	not required	mostly advantageous
Black box usage	possible	often not feasible
Robustness	yes	no

Table 2.2: Comparison of direct and iterative solution methods

2.1.2 Iterative methods

The principal advantage of iterative methods is that they require considerably less storage space than direct methods. Only nonzero entries of the coefficient matrix and a few vectors need to be stored. An iterative solution involves only matrix and vector multiplications, therefore it is much less complex than a direct solution, and also often does not require the coefficient matrix to be assembled explicitly.

Iterative methods have been traditionally used for the solution of large sparse systems, where direct methods could not be used due to the lack of sufficient storage space or unreasonable computational costs. They are also used extensively in combined solution techniques such as domain decomposition methods, discussed for example in [27] or [30].

The choice of a particular iterative method largely depends on the properties of the solved system. Great attention must be paid to the preconditioning to ensure a reasonable convergence of the selected method, see for example [3] or [15].

Well known iterative methods include the Jacobi method, the Gauss-Seidel method, the Successive Over-Relaxation (SOR) method, the Conjugate Gradient (CG) method, the Generalized Minimal Residual (GMRES) method and the Multigrid method. Iterative methods will not be discussed in detail, since this work is focused on the direct solution of system (2.1), but are mentioned to complement the overview of available solution methods. Description of common iterative methods can be found for example in [30], and a comprehensive survey is given for example in [36].

The advantages and disadvantages of both direct and iterative methods are summarized in Table 2.2, which has been adopted from [36].

2.2 Ordering methods

If the coefficient matrix $\mathbf{A}$ in equation (2.1) is sparse, i.e., has a large percentage of zero entries, the storage requirements and computational costs necessary to obtain the solution can be substantially reduced by modifying the solution algorithm to work only with the nonzero entries of the matrix. Such solution algorithm can be very efficient, and if implemented properly, quite large systems can be solved.

As already mentioned in Subsection 2.1.1, direct methods generally require pivoting to ensure the numerical stability of the factorization. In the sparse case, pivoting (called *ordering*) is also necessary to preserve the sparsity in the factors. Unfortunately, pivoting for numerical stability and pivoting for sparsity preservation are contradictory goals, and have been a topic of research.

For example, the factorization of an arrowhead matrix

$$\mathbf{A} = \begin{pmatrix} * & * & * & \cdots & * \\ * & * & & & \\ * & & * & & \\ \vdots & & & \ddots & \\ * & & & & * \end{pmatrix} \tag{2.14}$$

leads to dense triangular factors. By switching the first pivot with the last pivot, the obtained matrix

$$\mathbf{B} = \begin{pmatrix} * & & & & * \\ & \ddots & & & \vdots \\ & & * & & * \\ & & & * & * \\ * & \cdots & * & * & * \end{pmatrix} \tag{2.15}$$

exhibits no *fill-in*, i.e., there will not be any nonzero in the entries that were zero in the original matrix. Since the switching of rows and columns are elementary matrix operations that do not change the solution of the associated linear system, matrices (2.14) and (2.15) are equivalent.

A more practical example of the effect of the ordering is illustrated in Figure 2.1. The sample matrix is shown in original ordering (top), reordered using the reverse Cuthill-McKee algorithm (middle), and reordered using the minimum degree algorithm (bottom), each time with the corresponding Cholesky factor. It can be seen that the number of nonzero entries in the factor can be substantially reduced when a suitable ordering is applied.

The implication for large sparse systems is that while the coefficient matrix usually can be stored, its factors cannot if there is too much fill-in. Consequently, the reordering of the coefficient matrix prior to the factorization is essential to make the direct solution computationally feasible. The reordered coefficient matrix used in the factorization is

$$\tilde{\mathbf{A}} = \mathbf{PAQ} \tag{2.16}$$

in the unsymmetric case or

$$\tilde{\mathbf{A}} = \mathbf{PAP}^{\mathrm{T}} \tag{2.17}$$

in the symmetric case, where $\mathbf{P}$ and $\mathbf{Q}$ are left and right permutation matrix, respectively.

The problem of finding the optimal ordering is NP-complete, therefore all ordering algorithms are actually heuristics. The most common ordering methods are summarized in the rest of this section. More about the ordering methods used in the direct solution of sparse linear systems can be found for example in [12], [15] or [18].

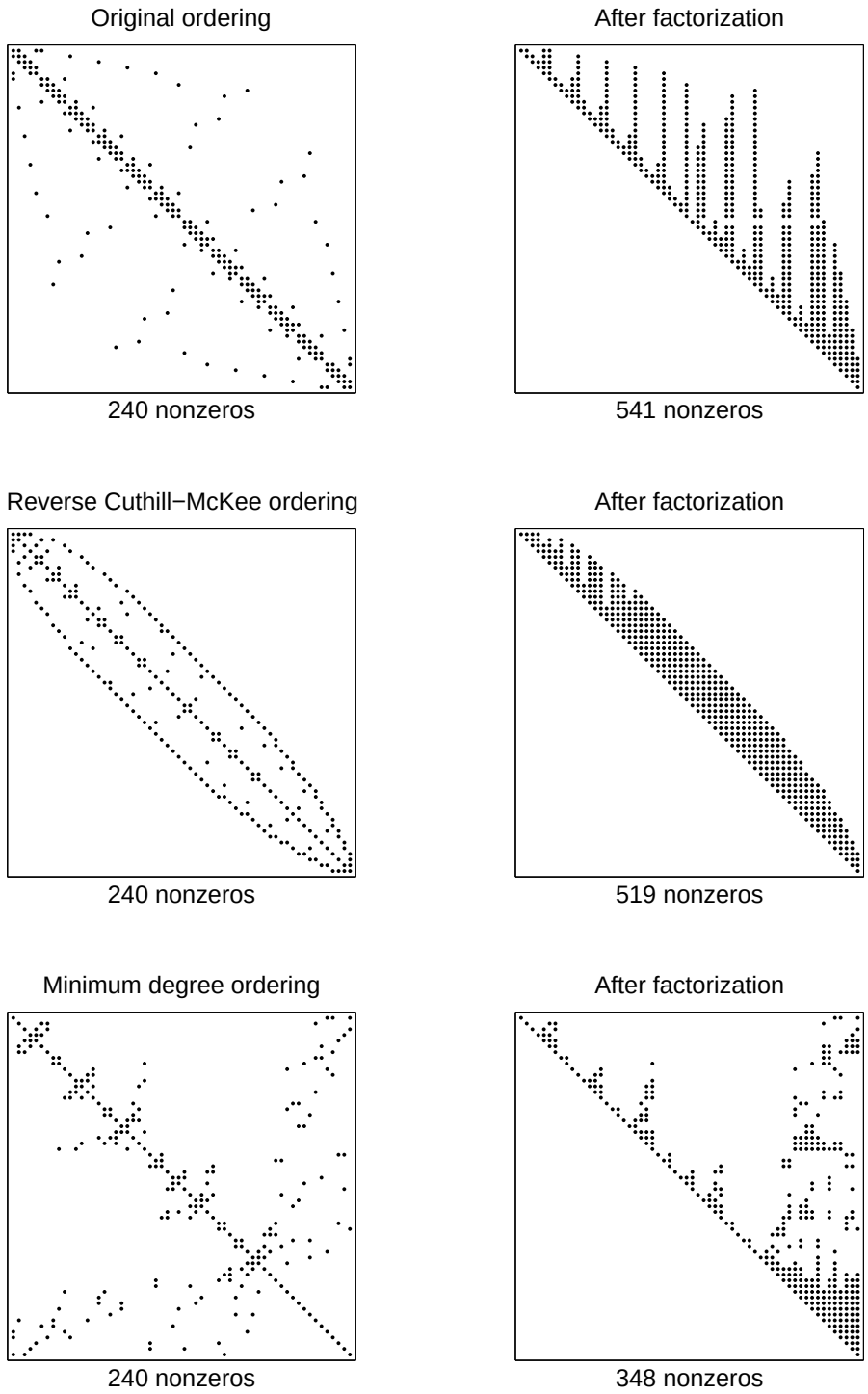


Figure 2.1: Effect of ordering in factorization of a symmetric sparse matrix

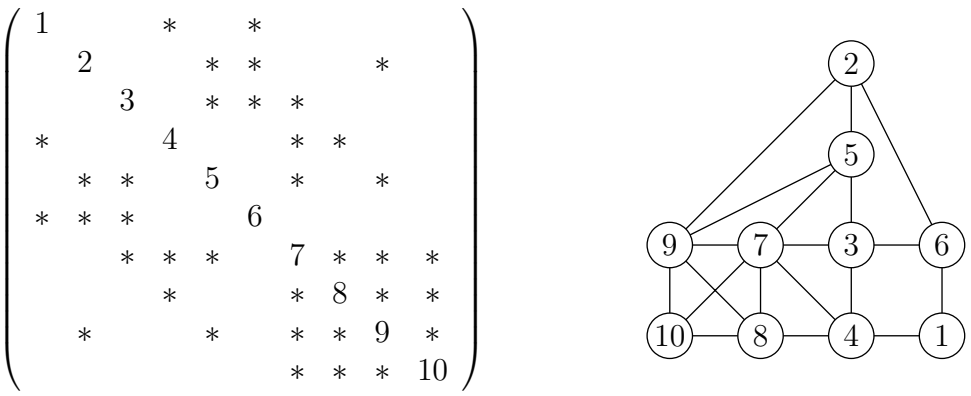


Figure 2.2: Example of symmetric sparse matrix and its corresponding graph

Iterative methods do not exhibit fill-in since they use only matrix-vector and matrix-matrix multiplications. However, an ordering can be useful for preconditioning with an incomplete LU factorization, see for example [15].

Graph theory basics

Ordering methods usually utilize graphs as a convenient and relatively simple way to represent, analyze and manipulate the nonzero structure of sparse matrices.

- The *graph* $G = (V, E)$ of a sparse matrix $\mathbf{A}$ consists of *vertices* V and *edges* E. Edge (i, j) that connects vertices i and j exists in the graph if and only if $a_{ij} \neq 0$. For symmetric matrices, the graph is undirected, i.e., for every edge (i, j) there is also edge (j, i).
- Vertex j is *adjacent* to vertex i if and only if edge (i, j) exists in the graph.
- The *degree* of vertex i is the number of vertices adjacent to vertex i.

An example of a matrix and its graph is presented in Figure 2.2. A comprehensive explanation of the graph theory can be found for example in [11].

2.2.1 Profile minimization

Profile minimization ordering methods are used to reduce the matrix profile (also called *envelope*) in order to store the matrix effectively in a band or skyline format (see Subsection 2.3.2).

Reverse Cuthill-McKee algorithm

The *reverse Cuthill-McKee algorithm* [9] is a simple ordering method for the reduction of the matrix bandwidth.

The algorithm works as follows:

1. Create a graph according to the nonzero structure of the matrix, and an empty list.
2. Choose a starting vertex, number it 1, and add it to the list.
3. Remove the first vertex from the list, number all unnumbered vertices adjacent to the vertex in order of their degree, and append them to the end of the list.
4. Unless all vertices are numbered, repeat from step 3.
5. The permutation vector is given by the sequence in which the vertices were numbered.

Choosing a good starting vertex is the most important part of the algorithm, since the obtained ordering is highly dependent on its choice. Several strategies for finding the (pseudo-) peripheral vertices, which are good candidates for the starting vertex, are used for this purpose.

The described algorithm is actually the original Cuthill-McKee algorithm, but normally the reverse sequence of vertices is used, since it results in even lower bandwidth (hence the reverse Cuthill-McKee algorithm).

Spectral algorithm

The *spectral algorithm for envelope reduction* [4] is an ordering method for minimizing the matrix profile.

The algorithm works as follows:

1. Form a Laplacian matrix according to the nonzero structure of the matrix.
2. Compute the second eigenvector of the Laplacian matrix.
3. Sort the components of the eigenvector in nondecreasing order, and reorder the matrix using the corresponding permutation vector Also sort the components in nonincreasing order, and compute the corresponding reordering of the matrix. Choose the permutation that leads to smaller profile.

The most computationally difficult part of the algorithm is the computation of the second eigenvector, which is carried out using multilevel approach based on the Lanczos method [3]. Numerical results show that the spectral algorithm usually yields better ordering and is faster than the reverse Cuthill-McKee algorithm.

2.2.2 Fill-in minimization

Fill-in minimization ordering methods are used to reduce the fill-in introduced in the factorization, and are especially useful in the direct solution of large sparse systems. To store the reordered matrix efficiently, a general sparse scheme has to be used (see Subsection 2.3.2) since the matrix profile may be rather large and the nonzero structure of the matrix usually does not follow any exploitable pattern.

Minimum degree algorithm

The *minimum degree algorithm* [34] is one of the most widely used ordering methods since it produces factors with relatively low fill-in on a wide range of matrices.

The algorithm works as follows:

1. Create a graph according to the nonzero structure of the matrix.

2. Remove the vertex with minimum degree from the graph. Add edges to the graph so that all vertices adjacent to the removed vertex form a clique (i.e., are all interconnected with each other).

3. Unless the graph is empty, repeat from step 2.

4. The permutation vector is given by the sequence in which the vertices were removed from the graph.

The most computationally difficult part of the algorithm is the computation of vertex degrees. Since the introduction of the basic algorithm stated above, numerous refinements has been devised to improve its efficiency.

The state of the art implementation is the approximate minimum degree algorithm by Amestoy, Davis and Duff [1, 2].

Nested dissection algorithm

The *nested dissection algorithm* [21, 26] is a recursive ordering method based on graph partitioning. It is known to be theoretically superior to the minimum degree algorithm for sparse symmetric definite matrices, but only very recently the implementations have been shown that are more efficient than the implementations of the minimum degree algorithm.

The algorithm works as follows:

1. Create a graph according to the nonzero structure of the matrix.

2. Partition the graph into two subgraphs of roughly same size using a small vertex separator.

3. Repeat step 2 recursively for every subgraph, until the subgraphs are fairly small.

4. The permutation vector is given by the reverse sequence of the vertex separators. The vertices in the top level separator are ordered last, the vertices in the second-to-top level separator are ordered before them, etc.

Finding a good vertex separator is the most important part of the algorithm. Coordinate-based and coordinate-free methods are used for this purpose.

The state of the art implementation is the METIS software package by Karypis and Kumar [25].

2.3 Matrix storage methods

In the actual numerical implementation, matrix entries have to be stored in the computer memory in some efficient way. The choice of a *storage scheme* (also called *storage format* or *storage mode*) for a particular problem depends on various factors including the structure of the matrix and the solution method employed. Obviously, all storage schemes designed for symmetric matrices can be used for triangular matrices as well.

Direct methods operate primarily on dense matrices, but implementations with sparse matrices became quite popular in recent decades. If the coefficient matrix is stored in a general sparse scheme, the occurence of fill-in must be carefully taken into consideration.

Iterative methods operate primarily on sparse matrices and since they do not modify matrix entries, the implementation of any storage scheme is relatively straightforward. A particular storage scheme can be chosen purely based on the properties of the coefficient matrix (symmetry, profile, block structure, etc.).

The most common matrix storage schemes are summarized in this section. For a more detailed survey, see [36], [3] and [30].

Fill-in consideration

In dense, band and skyline matrix storage schemes (to be explained), the fill-in does not constitute a problem, since all storage locations for matrix entries where the fill-in might occur are present implicitly. In general sparse storage schemes, such as the compressed row format (and other similar schemes), storage locations for the possible fill-in may be missing. This problem can be addressed in three ways:

1. Store the coefficient matrix in a band or skyline format. This option is not feasible for large systems, although a profile minimization ordering might help in some cases.

2. Find the exact locations of the fill-in and include the corresponding matrix entries in the storage scheme. This option requires some suitable method for fill-in analysis and must be performed usually prior to creating the storage scheme. A fill-in minimization ordering, such as the minimum degree, is very useful in that the fill-in locations can be obtained as a by-product of the ordering algorithm with relatively little additional effort.

3. Add the fill-in as a new matrix entry by expanding the storage scheme when needed. The expansion should be done in blocks to achieve high efficiency. This option requires the most sophisticated implementation of both the storage method and the solution method. General sparse storage schemes can be extended to support the addition of fill-in, at the cost of extra overhead information and a more complicated way of accessing the matrix entries in the correct order. Linked lists are usually employed to facilitate adding of matrix entries.

   Alternatively, the storage may be fixed but of a sufficiently large size to accomodate for fill-in additions.

2.3.1 Dense matrices

Dense matrices have all or almost all of the entries nonzero and therefore all entries need to be stored. In practical applications, only very small matrices ($n \leq 10$) are stored in the dense format, because the storage requirements increase quadratically with the matrix order n. Dense storage schemes are however a very useful part of block sparse storage schemes, where individual submatrices are treated as dense.

Rectangular storage

The *rectangular storage* format is the simplest scheme for general matrices. Matrix entries are stored consecutively in a real array s. Storage requirements are mn real numbers.

For example, the general matrix

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \end{pmatrix}$$

has $m = 4$ and $n = 3$, and can be stored either by rows

k	1	2	3	4	5	6	7	8	9	10	11	12
s_k	a_{11}	a_{12}	a_{13}	a_{21}	a_{22}	a_{23}	a_{31}	a_{32}	a_{33}	a_{41}	a_{42}	a_{43}

or by columns

k	1	2	3	4	5	6	7	8	9	10	11	12
s_k	a_{11}	a_{21}	a_{31}	a_{41}	a_{12}	a_{22}	a_{32}	a_{42}	a_{13}	a_{23}	a_{33}	a_{43}

Triangular storage

The *triangular storage* format is useful for symmetric matrices since only the upper (or lower) triangle of the matrix needs to be stored. Corresponding matrix entries are stored consecutively in a real array s, usually starting (or ending) with a diagonal entry. Storage requirements are $n(n+1)/2$ real numbers.

For example, the symmetric matrix

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{12} & a_{22} & a_{23} & a_{24} \\ a_{13} & a_{23} & a_{33} & a_{34} \\ a_{14} & a_{24} & a_{34} & a_{44} \end{pmatrix}$$

has $n = 4$ and can be stored either by rows

k	1	2	3	4	5	6	7	8	9	10
s_k	a_{11}	a_{12}	a_{13}	a_{14}	a_{22}	a_{23}	a_{24}	a_{33}	a_{34}	a_{44}

or by columns

k	1	2	3	4	5	6	7	8	9	10
s_k	a_{11}	a_{12}	a_{22}	a_{13}	a_{23}	a_{33}	a_{14}	a_{24}	a_{34}	a_{44}

2.3.2 Sparse matrices

Sparse matrices have most of the entries zero and therefore by storing only the nonzero entries, a considerable reduction of the storage requirements can be achieved, although storing some zero entries is usually inevitable.

Some sparse storage schemes require additional algorithmic and storage overhead to facilitate the access to stored matrix entries. This overhead is however well justified since the storage requirements of a complex sparse storage scheme can be much lower than that of a simple storage scheme, especially for large matrices.

Compressed diagonal storage

The *compressed diagonal storage* format is useful for band matrices since only the entries within the band of the matrix are stored, see Figure 2.3. The entries are stored consecutively by diagonals in a two-dimensional real array s, starting with the pth leftmost nonzero diagonal and ending with the qth rightmost nonzero diagonal.

The compressed diagonal storage format is quite efficient if all nonzero entries are packed tightly around the main diagonal, and requires no overhead for accessing the entries. However, some storage locations (entries of s) do not correspond to actual matrix entries and are always wasted. The band of the matrix may also contain many zero entries, if the matrix is not properly ordered. Storage requirements are mb real numbers, where $b = p + q + 1$ is the bandwidth of the matrix.

For example, the band matrix

$$\mathbf{A} = \begin{pmatrix} a_{11} & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 \\ 0 & a_{32} & a_{33} & 0 & 0 \\ 0 & a_{42} & 0 & a_{44} & a_{45} \\ 0 & 0 & a_{53} & 0 & a_{55} \\ 0 & 0 & 0 & 0 & a_{65} \end{pmatrix}$$

has $m = 6$, $n = 5$, $p = 2$ and $q = 1$, and can be stored as

k	1	2	3	4	5	6
$s_{k,-2}$	$\times$	$\times$	0	a_{42}	a_{53}	0
$s_{k,-1}$	$\times$	a_{21}	a_{32}	0	0	a_{65}
$s_{k,0}$	a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	$\times$
$s_{k,1}$	0	a_{23}	0	a_{45}	$\times$	$\times$

Note: Storage entries marked $\times$ do not correspond to actual matrix entries.

Skyline storage

The *skyline storage* format is useful for skyline (variable band) matrices since only the entries under the skyline of the matrix are stored, see Figure 2.4. It is normally used for symmetric matrices and the entries are stored by columns in a real array s, each column

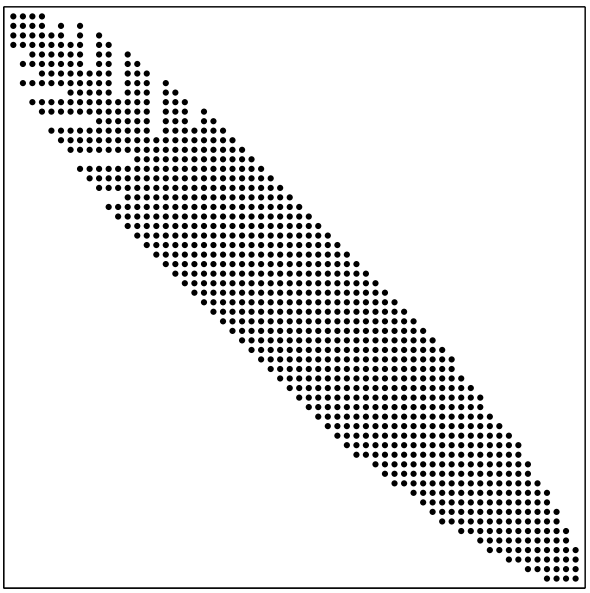


Figure 2.3: Typical nonzero structure of a symmetric band matrix

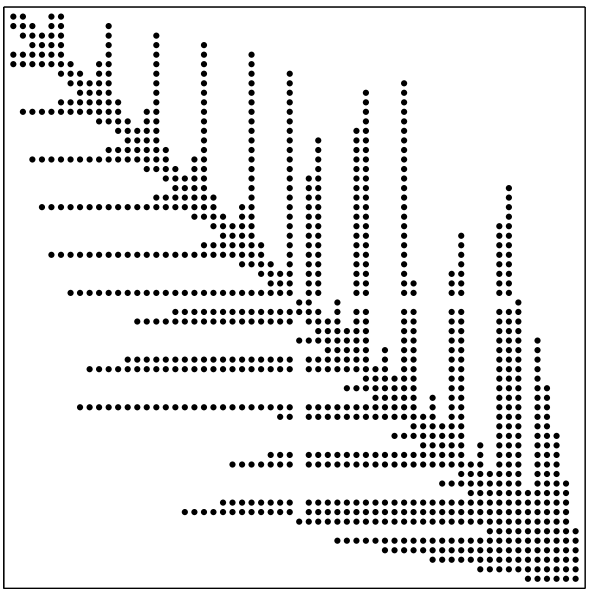


Figure 2.4: Typical nonzero structure of a symmetric skyline matrix

starting with the first nonzero entry and ending with the diagonal entry. An additional integer array c is needed to store the index of the first entry for each column.

The skyline storage format is more efficient than the compressed diagonal storage format in that it does not store any zero entries above the skyline and therefore the bandwith of the matrix may be relatively large. However, there still may be many zero entries under the skyline. Storage requirements are n_{nz} real numbers and $n+1$ integers, where $n_{nz} = c_{n+1} - 1$ is the number of nonzero entries in the matrix. The extra $(n+1)$th entry in c is used to easily obtain the length of any column $l_j = c_{j+1} - c_j$.

For example, the symmetric matrix

$$\mathbf{A} = \begin{pmatrix} a_{11} & 0 & 0 & a_{14} & 0 \\ 0 & a_{22} & a_{23} & 0 & 0 \\ 0 & a_{23} & a_{33} & 0 & a_{35} \\ a_{14} & 0 & 0 & a_{44} & 0 \\ 0 & 0 & a_{35} & 0 & a_{55} \end{pmatrix}$$

has $n = 5$ and $n_{nz} = 11$, and can be stored as

j	1	2	3	4	5	6
c_j	1	2	3	5	9	12

k	1	2	3	4	5	6	7	8	9	10	11
s_k	a_{11}	a_{22}	a_{23}	a_{33}	a_{14}	0	0	a_{44}	a_{35}	0	a_{55}

Unsymmetric matrices require a combined storage of two skyline formats, one for the lower triangle and one for the upper triangle of the matrix.

Compressed row (column) storage

The *compressed row storage* format is useful for general sparse matrices since only the nonzero entries of the matrix are stored. The entries are stored by rows in a real array s, and additional integer arrays r and c are needed to store the index of the first entry for each row and the column index for each entry, respectively.

The compressed row storage format carry a considerable overhead (in the form of the r and c arrays) but is quite efficient for large sparse matrices. Storage requirements are n_{nz} real numbers and $m+1+n_{nz}$ integers, where $n_{nz} = r_{m+1} - 1$ is the number of nonzero entries in the matrix. The extra $(n+1)$th entry in r is used to easily obtain the length of any compressed row $l_i = r_{i+1} - r_i$.

For example, the matrix

$$\mathbf{A} = \begin{pmatrix} a_{11} & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 \\ 0 & a_{32} & a_{33} & 0 & 0 \\ 0 & a_{42} & 0 & a_{44} & a_{45} \\ 0 & 0 & a_{53} & 0 & a_{55} \\ 0 & 0 & 0 & 0 & a_{65} \end{pmatrix}$$

has $m = 6$, $n = 5$ and $n_{nz} = 12$, and can be stored as

i	1	2	3	4	5	6	7
r_i	1	2	5	7	10	12	13

k	1	2	3	4	5	6	7	8	9	10	11	12
c_k	1	1	2	3	2	3	2	4	5	3	5	5
s_k	a_{11}	a_{21}	a_{22}	a_{23}	a_{32}	a_{33}	a_{42}	a_{44}	a_{45}	a_{53}	a_{55}	a_{65}

The *compressed column storage* format is a column-oriented analogue of the compressed row storage format. The entries of the matrix are stored by columns in array s and arrays c and r store the index of the first entry for each column and the row index for each entry, respectively. Storage requirements will differ from the column variant unless the matrix is square.

The matrix from the above example can be stored as

j	1	2	3	4	5	6
c_j	1	3	6	9	10	13

k	1	2	3	4	5	6	7	8	9	10	11	12
r_k	1	1	2	3	2	3	2	4	5	3	5	5
s_k	a_{11}	a_{21}	a_{22}	a_{32}	a_{42}	a_{23}	a_{33}	a_{53}	a_{44}	a_{45}	a_{55}	a_{65}

In the two examples above, the matrix entries are stored in the order they occur in the matrix. This is not strictly necessary, since any order can be imposed on columns (rows) as long as the indices c_k (r_k) correspond to the entries s_k. However, working with arbitrarily ordered entries complicates the algorithm for accessing the entries in the correct order.

Block compressed row (column) storage

The *block compressed row storage* format is useful for large sparse matrices with a block nonzero structure. The matrix is partitioned evenly into submatrices (blocks) of size $d \times d$, which are stored in a three-dimensional array s as dense, even though they may contain some zero entries. Completely zero submatrices are obviously not stored. Additional integer arrays r and c are needed to store the index of the first block for each row and the column index of (1,1) entry for each block, respectively. Only one index in c is required per one block which can be a very efficient strategy in the case the blocks have order $d \geq 2$. For $d = 1$ the block compressed row format degenerates into the compressed row format described earlier. Storage requirements are $N_{nz}d^2$ real numbers and $M + 1 + N_{nz}$ integers, where $M = m/d$ is the number of block rows and N_{nz} is the number of nonzero blocks in the matrix.

The *block compressed column storage* format can be defined analogically to the above-mentioned format.

All storage schemes described earlier are *static*, meaning that once the data structure for the matrix storage is created, only the entries included in the storage can be changed

and no other entries can be added. It is however possible to define *dynamic* block compressed storage format (row or column) that allows new entries to be added to the storage data structure almost arbitrarily, which is very useful in direct methods not only for the matrix factorization, but also when the matrix is assembled submatrix-by-submatrix, for example in the finite element method.

One of the possible approaches for symmetric matrices is to replace array r with array p that stores the index of the next block on the same row (column), and reserve the first N entries in arrays p, c and s for diagonal blocks. For any block k, entry p_k holds the index of the next block in the same row or 0 if the block is the last block on the row, and entry p_i holds the index of the first nondiagonal block in the row i. If the blocks are inserted into the linked lists in the ascending order of their column indices, the access algorithm can be much simpler (but it is not mandatory). Storage requirements are $N_{nz}d^2$ real numbers and $2N_{nz}$ integers, but in this case N_{nz} must be chosen suitably large to accomodate all blocks to be added. The number of nonzero blocks in any row can be obtained either by scanning the corresponding linked list, or can be stored explicitly in a separate array, which however increases the storage requirements by another N integers.

The dynamic block compressed row storage format of a symmetric block sparse matrix is illustrated in Figure 2.5.

There are several other possible approaches how to define a block sparse storage scheme. For example, a block sparse storage scheme called *K3*, which is designed specifically for the finite element analysis of solids and structures and implemented in C, is described in [38].

2.4 Direct solvers

Direct solvers can be divided according to the used algorithms and storage schemes into dense direct solvers and sparse direct solvers. Dense direct solvers will not be discussed, since they are not suitable for the solution of finite element problems of a practical size due to $O(n^2)$ storage and $O(n^3)$ complexity. Furthermore, the implementation of dense direct solvers is relatively simple and straightforward.

Sparse direct solvers exploit the structure of the sparse coefficient matrix to reduce the storage as well as the number of arithmetic operations needed for factorization and substitution, and thus involve much more complicated algorithms than dense direct solvers. A typical sparse direct solver consists of four steps as opposed to two in the dense case:

1. Ordering, where the rows and columns of the coefficient matrix are reordered to obtain a suitable sparse structure.

2. Analysis, where the coefficient matrix is analyzed to produce suitable data structures for the factorization.

3. Numerical factorization, where the factors are computed.

4. Substitution, where the solution is computed using the forward and back substitution.

The following example symmetric block sparse matrix has $m = n = 10$, $n_{nz} = 24$, $d = 2$, $M = N = 5$ and $N_{nz} = 9$.

$$\mathbf{A} = \left(\begin{array}{cc|cc|cc|cc|cc} a_{11} & a_{12} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ a_{12} & a_{22} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & a_{33} & 0 & 0 & 0 & a_{37} & a_{38} & a_{39} & a_{30} \\ 0 & 0 & 0 & a_{44} & a_{45} & 0 & a_{47} & a_{48} & 0 & a_{40} \\ \hline 0 & 0 & 0 & a_{45} & a_{55} & a_{56} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & a_{56} & a_{66} & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & a_{37} & a_{47} & 0 & 0 & a_{77} & a_{78} & a_{79} & 0 \\ 0 & 0 & a_{38} & a_{48} & 0 & 0 & a_{78} & a_{88} & a_{89} & 0 \\ \hline 0 & 0 & a_{39} & 0 & 0 & 0 & a_{79} & a_{89} & a_{99} & a_{90} \\ 0 & 0 & a_{30} & a_{40} & 0 & 0 & 0 & 0 & a_{90} & a_{00} \end{array}\right)$$

The storage data structure before and after adding submatrix $\begin{pmatrix} 0 & 0 \\ a_{45} & 0 \end{pmatrix}$ is shown below ($q = 1, \ldots, d$).

k	1	2	3	4	5	6	7	8	...
p_k	0	6	0	8	0	7	0	0	...
c_k	1	3	5	7	9	7	9	9	...
s_{kqq}	a_{11} a_{12} 0 a_{22}	a_{33} 0 0 a_{44}	a_{55} a_{56} 0 a_{66}	a_{77} a_{78} 0 a_{88}	a_{99} a_{90} 0 a_{00}	a_{37} a_{38} a_{47} a_{48}	a_{39} a_{30} 0 a_{40}	a_{79} 0 a_{89} 0	...

k	1	2	3	4	5	6	7	8	9	...
p_k	0	9	0	8	0	7	0	0	6	...
c_k	1	3	5	7	9	7	9	9	3	...
s_{kqq}	a_{11} a_{12} 0 a_{22}	a_{33} 0 0 a_{44}	a_{55} a_{56} 0 a_{66}	a_{77} a_{78} 0 a_{88}	a_{99} a_{90} 0 a_{00}	a_{37} a_{38} a_{47} a_{48}	a_{39} a_{30} 0 a_{40}	a_{79} 0 a_{89} 0	0 0 a_{45} 0	...

Note: The row and column index 10 has been replaced with 0 for clarity.

Figure 2.5: Dynamic block compressed row storage format example

Steps 1 and 2 usually involve only integer operations (graphs used in ordering and analysis can be represented by sets of integers), whereas steps 3 and 4 involve operations on real numbers. Some steps may be combined depending on the implementation.

Ordering

Ordering produces permutation matrices $\mathbf{P}$ and $\mathbf{Q}$, which are used to reorder the coefficient matrix to allow its efficient storage. The matrix should be reordered on solver input instead of performing the rather impractical matrix multiplication $\mathbf{PAQ}$.

The particular ordering method depends on the solver implementation; profile minimization orderings are best suited for skyline solvers whereas fill-in minimization orderings are best suited for sparse solvers.

The ordering step is usually completely independent on the other steps.

Analysis

The analysis step is necessary to determine the sparsity structure of the coefficient matrix factors in order to allocate appropriate data structures. It can be usually performed together with ordering, but it is dependent on the algorithm used for numerical factorization.

If the coefficient matrix is positive definite, both the ordering and the analysis can be carried out separately from the numerical factorization. Otherwise, analysis has to be performed during the numerical factorization since it may involve pivoting for numerical stability.

Numerical factorization

Numerical factorization is the most computationally difficult part of the solution process. Generally, the decomposition $\mathbf{PAQ} = \mathbf{LU}$ can be performed either by a right-looking or a left-looking algorithm: a right-looking (eager) algorithm updates the elements/columns to the right as soon as the current element/column has been computed, while a left-looking (lazy) algorithm updates the current element/column from earlier elements/columns as late as possible. Both algorithms are equivalent in terms of number of arithmetic operations, and the preference of one over the other depends purely on the particular solver implementation and matrix storage used.

The factors are normally stored in place of the original coefficient matrix for maximum efficiency and minimum storage requirements, i.e., during the factorization, the entries of the original matrix are gradually overwritten with the entries of the factors. In the symmetric case, only a triangular part of the coefficient matrix and the factor needs to be stored.

Substitution

The final step is the solution of the decomposed system. Since the factors are triangular or diagonal, the corresponding linear systems can be solved by a simple substitution.

First, the forward substitution $\mathbf{Ly} = \mathbf{Pb}$ is performed with the right-hand side (permuted with the left permutation vector) and the lower triangular factor to obtain the reduced right-hand side. Second, the back substitution $\mathbf{Uz} = \mathbf{y}$ is performed with the reduced right-hand side and the upper triangular factor to obtain the permuted solution. Last, the solution vector is permuted back into the original system using the right permutation matrix $\mathbf{x} = \mathbf{Qz}$.

The solution in the symmetric case is analogous, however factor $\tilde{\mathbf{L}}^{\mathrm{T}}$ (Cholesky) or factor $\mathbf{DL}^{\mathrm{T}}$ is used instead of factor $\mathbf{U}$ and permutation matrix $\mathbf{P}^{\mathrm{T}}$ instead of permutation matrix $\mathbf{Q}$.

Methods involved in dense and sparse direct solvers are discussed for example in [13], [14], [7] or [23].

2.4.1 Standard implementations

There are four basic classes of sparse direct solvers: frontal solver, skyline (or band) solver, sparse solver, and multifrontal solver.

Frontal solver

Frontal solvers are based on the *frontal solution method* [24, 30], which has emerged from the application of finite element method in structural analysis.

This method involves an auxiliary matrix, called *frontal matrix*, that is used to store only an *active* part of the coefficient matrix. Element matrices are added to the frontal matrix one by one, and the elimination of a pivot is performed as soon as it is fully summed, i.e., there are no contributions from the other elements. The eliminated row is then moved out of the frontal matrix into the factor that is usually stored on disk. Consequently, the assembly and the factorization processes are actually interleaved, and the coefficient matrix is never assembled explicitly.

In non-element applications, rows of the coefficient matrix (equations) are added into the frontal matrix one by one, and a pivot is eliminated when it does not appear in any of the remaining rows.

The frontal solution method uses a right-looking algorithm since the already eliminated matrix rows are moved out of core. The frontal matrix is dense, therefore the factorization within the matrix can be carried out very efficiently. The size of the frontal matrix depends on the *front width*, i.e., the number of simultaneously processed elements (equations). The front width is affected only by the ordering of elements; ordering methods cannot be utilized since the properties of the coefficient matrix such as the profile or the sparsity pattern are irrelevant.

Frontal solvers are quite memory-efficient and are capable of solving large problems, since only the frontal matrix needs to be stored in-core. However, in the case of large problems, the disk storage needed for the factors is usually high, and combined with the slow disk access times the practical usability is reduced.

Skyline solver

Skyline solvers are based on the *active column solution (skyline reduction) method* [5]. This method exploits the fact that the coefficient matrix always retains the same profile (skyline) throughout the factorization, i.e., the fill-in occurs only inside the profile and the entries outside the profile remain zero. Therefore, all the storage space necessary for the factorization can be allocated in advance.

The coefficient matrix is stored in a skyline format (by columns) and a left-looking algorithm is used for factorization. Since the columns are stored as dense vectors, the innermost loops do not require indirect addressing.

A similar approach using a band format (*band solvers*) may be useful in some cases to simplify the factorization algorithm at the cost of higher storage requirements and larger number of arithmetic operations on zero elements.

Although the skyline format is more efficient than the band format, neither of them is feasible for large problems even when a suitable profile minimization ordering is employed.

Sparse solver

Sparse solvers[1] operate only on nonzero entries of the coefficient matrix and the factors and are generally very efficient, especially for large problems, where the overhead required by a sparse storage scheme (for example the compressed column format) is negligible. This approach requires the most complicated algorithms.

The sparse factorization can be both right-looking and left-looking, depending on the particular matrix storage scheme chosen. An indirect addressing has to be used in the innermost loops due to the fact that the columns (or rows) are stored as sparse vectors. Also, an efficient implementation of the fill-in is necessary; some algorithms need the locations of the fill-in to be known in advance, whereas other algorithms allow the addition of new nonzero entries into the matrix storage dynamically.

Ordering methods to reduce the fill-in must be employed to achieve reasonable storage requirements and computational costs. For large problems, an out-of-core implementation is mandatory since it may not be possible, due to the fill-in, to store the whole factors in the memory, even in the case the whole coefficient matrix can be stored.

A summary of the methods used in sparse direct solvers can be found for example in [27] or [37]. The latter work deals in detail with a particular implementation of a sparse direct solver written in C++.

Multifrontal solver

Multifrontal solvers are based on the *multifrontal solution method* [16], which is an extension of the frontal solution method intended for a parallel implementation on high-performance computers. This method is however efficient also on single-processor com-

[1] The term *sparse solver* sometimes means *iterative solver*, but in this work it always refers to a sparse direct solver based on a sparse factorization.

puters, and has lower storage requirements and computational costs than the frontal solution method.

Instead of one frontal matrix multiple frontal matrices are used simultaneously throughout the factorization. For each pivot, a separate frontal matrix is created, eliminated and maintained until it is required by another pivot in subsequent factorization steps. If the coefficient matrix has a block structure, it can be exploited by constructing the frontal matrix for all pivots in a block, reducing the number of fronts and the number of arithmetic operations needed to compute the factors. An assembly tree (similar to a graph in ordering methods) can be used to analyze and optimize the elimination and to merge pivots in fronts.

Multifrontal solvers require more out-of-core data manipulation and more storage for frontal matrices of smaller size than the frontal solvers. However, an important advantage is that any ordering method can be employed to reduce the storage requirements for the factors.

2.4.2 Available software

A recent list of about 50 available sparse direct solver codes is presented in [10]. The list includes main features of the codes, such as used factorization method or ordering method, references to relevant papers and authors' contact information. Another study, given in [23], presents a comprehensive numerical evaluation of 10 available sparse direct solvers for large linear systems.

Commercial codes are not considered since they present 'black box' designs with a limited extensibility, not to mention the need for expensive licenses. Also, the details of commercial implementations are not publicly available.

One particular exception is the PMD finite element system, which is sold commercially, but its source code is available for research purposes to co-developers.

PMD: Package for Machine Design

PMD is a full-featured platform-independent in-house code for finite element analysis of 2-D, 2.5-D and 3-D problems in elasticity, heat transfer, eigenproblems, seismicity, stability, plasticity, creep, contact, etc. It comprises of a set of command-line programs, developed in FORTRAN 77. Each program is designed to perform a part of the finite element computation, and several programs are used in batch depending on the type of the problem to obtain the required solution. This modular design along with the standardized input and output data files allows for an easy extensibility. In the case some modified method needs to be implemented or a new method needs to be tested, either a completely new program can be introduced, or an existing program can be replaced by a more efficient version, both without affecting the rest of the system.

More information can be found in the PMD User Guide [33], the PMD Reference Guide [32] and the PMD Example Manual [31].

Chapter 3

Aims of the Thesis

The primary aim of this thesis is to improve methods and algorithms for the solution of sparse linear equation systems in order to reduce the necessary requirements on computational time and storage space, in particular, when applied to large finite element problems. The focus is on the fundamental methods present in any sparse direct solution process: the storage method for the coefficient matrix, the ordering method, and the solution (factorization) method.

The secondary aim is the implementation of a sparse direct solver for finite element analysis in solid continuum mechanics based on the methods proposed in the theoretical part of this work. The code is intended to be integrated into the PMD finite element system, therefore, an important part is to produce efficient implementation that can fully replace the existing frontal solver.

The proposed methods are restricted to symmetric positive definite linear equation systems that are the most common in the finite element analysis of solids and structures.

To summarize, the aims of the thesis are defined as follows.

1. Based on the critical overview, propose efficient methods and algorithms suitable for the solution of large finite element problems.
   - Generalize the K3 sparse matrix storage scheme.
   - Improve the minimum degree ordering algorithm.
   - Improve the standard $\mathbf{LDL}^{\mathrm{T}}$ factorization algorithm.
2. Implement a sparse direct solver, laying emphasis on the effectiveness of the solution process.
   - Implement both the in-core and out-core version of the solver.
   - Integrate the code into the PMD finite element system.
3. Perform tests and assessments of the solver.
   - Use the standard problems taken from the PMD Example Manual.

- Use large finite element problems taken from real-world engineering applications.
- Compare the sparse direct solver's performace against the frontal solver.

Chapter 4

Applied methods

In this chapter, the theoretical background of the methods used in the work is explained. Proposed modifications and algorithms to enhance the effectiveness of the applied methods are presented in Chapter 5.

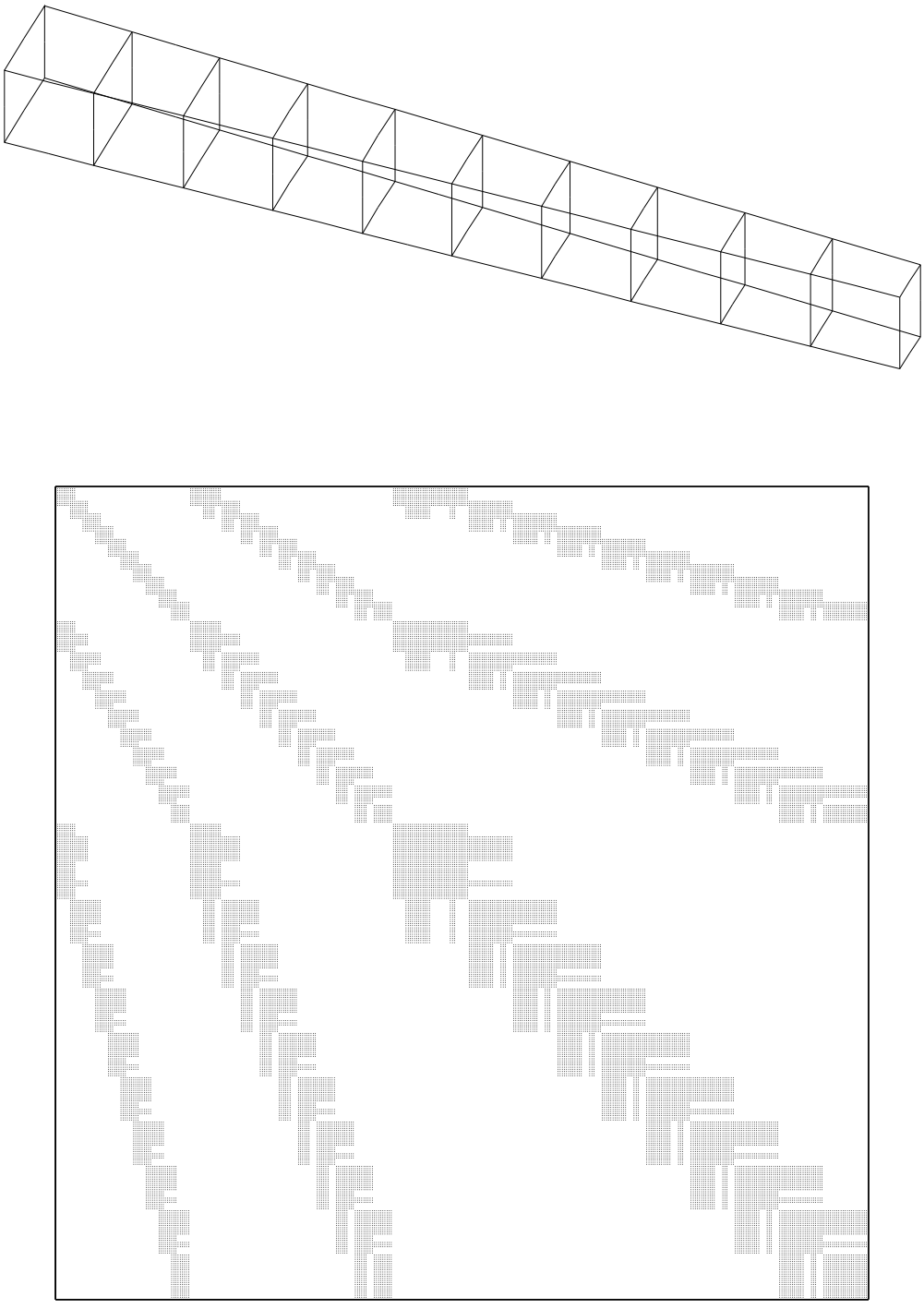
The first three sections describe the selected matrix storage method, the ordering method, and the solution method, respectively. These methods form the basis of any sparse direct solver and are interdependent, meaning that the choice of one method affects (to some degree) the choice of the other methods. Even for large problems, there is no generally best combination of the methods since various approaches are possible. The ultimate efficiency of a sparse direct solver will always highly depend on the particular numerical implementation. The choice of the appropriate methods is therefore a non-trivial issue.

The fourth and final section discusses several concepts and requirements of the PMD system that are necessary for the intended implementation of a sparse direct solver.

4.1 Matrix storage method

Coefficient matrices obtained by finite element discretization are sparse and have a block structure, and in most cases are also symmetric and positive definite, see Figure 4.1. Sparsity results from the problem locality, i.e., the fact that the number of nodes adjacent in the finite element mesh is limited. Since the coefficients in equations correspond to nodes that in turn belong only to a limited number of elements, the number of nonzero coefficients in each equation is usually much lower that the total number of equations. Block structure results directly from the number of degrees of freedom associated with each node. Nodes often have more that one degree of freedom, therefore the corresponding equations have the same nonzero coefficient structure.

The requirements on efficiency stated in Chapter 3 imply that the whole solution process is to be carried out primarily in memory (in-core). Considering the application to large problems, an efficient storage scheme for the coefficent matrix is mandatory. Moreover, the storage scheme must be suitable for both the matrix assembly and matrix factorization. It can be seen from Figure 2.1 that minimum degree ordering does not



Note: The mesh nodes are numbered so that the frontwidth is minimal to facilitate the use of a frontal solver.

Figure 4.1: Example 3-D finite element mesh (top) and corresponding coefficient matrix structure (bottom) for an elastostatic problem

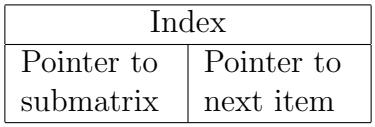


Figure 4.2: Basic element (item) of the K3 data structure

result in a band matrix, and a skyline storage format would not be efficient due to very high and very sparse columns. Compressed row storage format could be used, but would have unreasonably large overhead. However, a general block sparse scheme such as the block compressed row storage format is well suited and efficient. The K3 storage format presents several advantages over the standard block compressed row storage format and is explained in this section in detail.

4.1.1 K3 storage format

The *K3 sparse matrix storage system* [38] exploits the features of newer high-level programming languages such as structured data types, pointers and dynamic memory allocation, to implement a convenient block sparse storage scheme for the use in the finite element method. It is somewhat similar to the dynamic block compressed row storage format (see Subsection 2.3.2).

Considering a finite element mesh with N nodes, where each node has d degrees of freedom, the corresponding coefficient matrix can be partitioned evenly into $N \times N$ nodal submatrices of order d. The use of the mesh topology as means for determining the partitioning of the coefficient matrix into submatrices is advantageous since otherwise a complicated and time-consuming algorithm would be needed to search for some nonzero block pattern.

After the partitioning, nonzero submatrices (i.e., submatrices with at least one nonzero entry) are stored in the K3 data structure, which is composed of *items*. Each item consists of three data members, see Figure 4.2:

1. *Index*. This data member stores either the nodal column index for nondiagonal items or the number of items on the corresponding nodal row for diagonal items.

2. *Pointer to submatrix*. This data member stores the pointer to submatrix entries corresponding to the nodal row and column. The submatrix is stored by rows in a rectangular dense format, or in a triangular dense format in the case of diagonal items.

3. *Pointer to next item*. This data member stores the pointer to the next item in the nodal row. The items on each row are stored in a linked list that starts with the diagonal item and are sorted by ascending order of the column index. If the item is the last item on the row, the pointer has a special *null* value.

The K3 data structure is created as an array of diagonal items, with all pointers initialized to *null* (i.e., they do not point anywhere). The size of the array is fixed and

known since it can be calculated easily using the number of nodes N and number of nodal degrees of freedom d. Nondiagonal items and all submatrices are created dynamically[1] and are referenced by pointers in the other items. The size of dynamically allocated data is unknown, but it is not needed due to the use of memory allocation.

Fundamental operations on the K3 data structure include accessing, update and addition of an item. Addition of new items is done primarily in the assembly, but also in the factorization, when the fill-in occurs and the corresponding item is not present in the storage. The algorithm for assembling a nodal submatrix into the K3 data structure is described below, see Table 4.1 and Figure 4.3.

Step 1 calculates the corresponding nodal row and nodal column indices p and q from the row and column indices of the matrix entry i and j, taking into consideration that only the upper triangular part of the matrix is stored, optionally swapping the indices and transposing the submatrix. If nodal indices p and q are known the first two statements may be skipped.

Step 2 checks whether a diagonal submatrix is requested to see if it can be located immediately in the array of diagonal items. Otherwise the appropriate row must be scanned to see whether the requested item is present.

Step 3 checks whether a new item needs to be inserted after the current item to maintain the ascending order of column indices.

Step 4 advances to the next item on the row and checks if this item is the requested item. Otherwise the scanning of the row is continued.

Step 5 creates a new item and inserts it at the correct column position on the row by updating the appropriate pointers. Corresponding diagonal item's *index* data member is incremented to contain the correct number of nonzero blocks on the row.

Step 6 adds the submatrix to the current item's submatrix, or, if current item's pointer to submatrix is *null*, only assigns the reference to the submatrix.

The algorithm may be extended to access and/or update the whole matrix, for example in the matrix factorization, in which case an outer loop over all p would be added and all steps would be simplified to loop over all storage blocks in each row. Step 6 would be used to perform the factorization on current item's submatrix and step 3 would be changed to contain appropriate conditions for addition of a new storage block that represents the fill-in. In the matrix multiplication, step 5 may be completely removed and step 6 may be used to operate on both submatrices $\mathbf{A}_{pq}$ and $\mathbf{A}_{qp}$.

It can be seen that the dynamic K3 data structure is very flexible in that it is capable of adding and even removing items[2] (submatrices) from the storage on demand. It is however not very suitable for the use with iterative solvers or for the storage on a disk. For these cases a more conventional static data structure (a fixed array of items) can be used that reduces the number of pointers but practically eliminates the ability to add and remove items. Moreover, the size of the static data structure must be known in advance.

Storage requirements of the K3 data structure can be derived from the number of mesh nodes N, number of nodal degrees of freedom d, number of nonzero submatrices

[1] Using language-specific memory allocation functions, such as `malloc` in C.

[2] Removing of submatrices is important in the implementation of an out-of-core solution.

To assemble submatrix $\mathbf{A}_{pq}$ that contains matrix entry a_{ij}:

1. Set $p \equiv i \div d + 1$.
   Set $q \equiv j \div d + 1$.
   If $p > q$ swap $p \Leftrightarrow q$ and transpose $\mathbf{A}_{pq} \Leftrightarrow \mathbf{A}_{qp}$.

2. Set $\mathcal{C} \equiv \mathcal{D}_p$.
   If $p = q$ go to 6.

3. If $\mathcal{C} \rightarrow \mathcal{N} = null$ go to 5.
   If $\mathcal{C} \rightarrow \mathcal{N} \rightarrow \mathcal{I} > q$ go to 5.

4. Set $\mathcal{C} \equiv \mathcal{C} \rightarrow \mathcal{N}$.
   If $\mathcal{C} \rightarrow \mathcal{I} = q$ go to 6.
   Go to 3.

5. Create a new item $\mathcal{E}$.
   Set $\mathcal{E} \rightarrow \mathcal{I} \equiv q$.
   Set $\mathcal{E} \rightarrow \mathcal{S} \equiv null$.
   Set $\mathcal{E} \rightarrow \mathcal{N} \equiv \mathcal{C} \rightarrow \mathcal{N}$.
   Set $\mathcal{C} \rightarrow \mathcal{N} \equiv \mathcal{E}$.
   Set $\mathcal{D}_p \rightarrow \mathcal{I} \equiv \mathcal{D}_p \rightarrow \mathcal{I} + 1$.
   Set $\mathcal{C} \equiv \mathcal{E}$.
   Go to 6.

6. If $\mathcal{C} \rightarrow \mathcal{S} = null$ set $\mathcal{C} \rightarrow \mathcal{S}$ to reference $\mathbf{A}_{pq}$,
   else add $\mathbf{A}_{pq}$ to submatrix referenced by $\mathcal{C} \rightarrow \mathcal{S}$.

Legend:

$\mathbf{A}_{pq}$	nodal submatrix of order d
$\mathcal{C}$	current item
$\mathcal{D}_p$	pth diagonal item
$\mathcal{E}$	new item
$\mathcal{I}$	item *index* data member
$\mathcal{N}$	item *pointer to next item* data member
$\mathcal{S}$	item *pointer to submatrix* data member

Integer division is defined as follows: $X = D \times Q + R$ is the dividend, D is the divisor, $Q = X \div D$ is the quotient and R is the remainder.

Table 4.1: Algorithm for assembling submatrix into K3 data structure

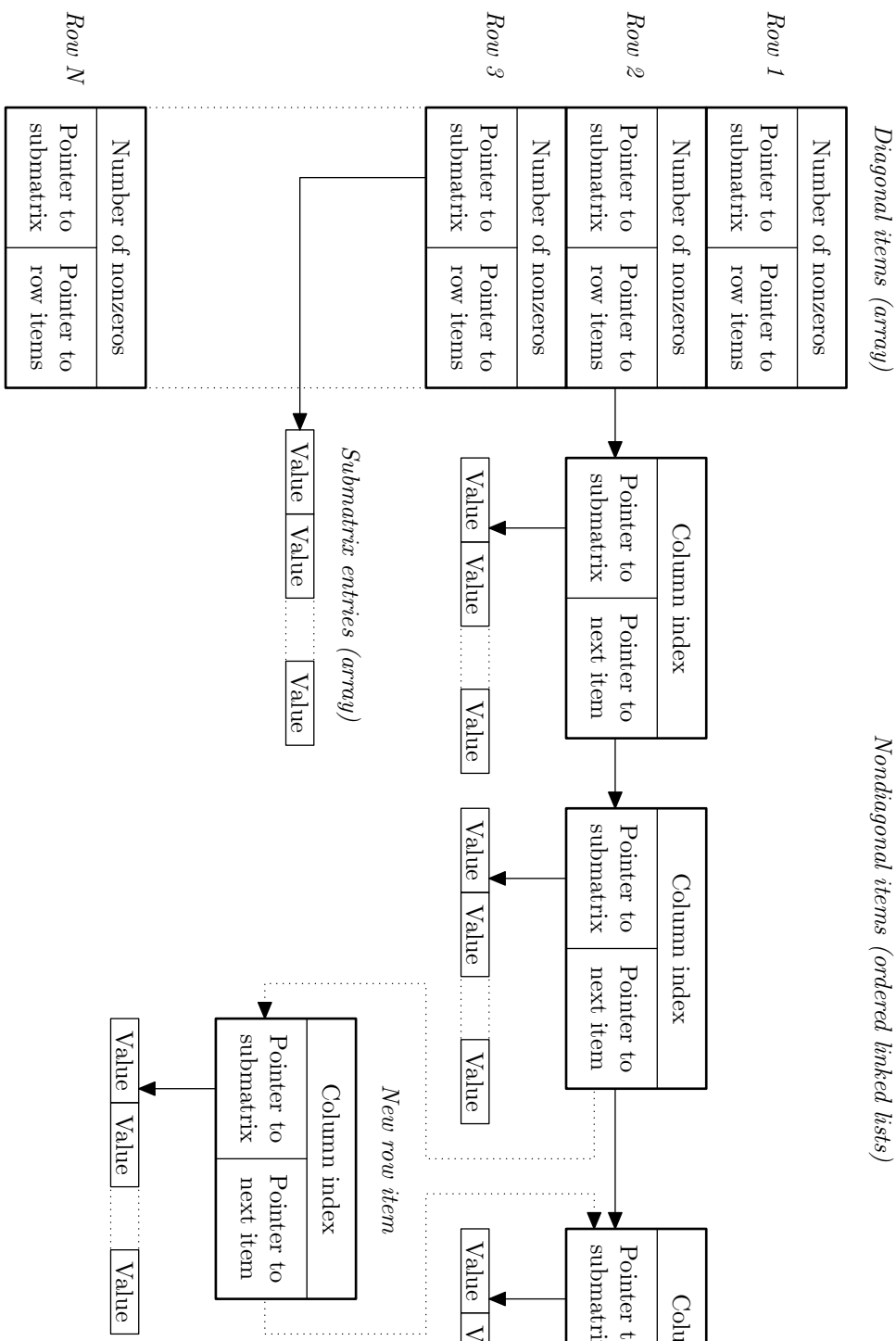


Figure 4.3: K3 storage format data structure

(number of items) N_{nz}, and the size of integer, real, and pointer data types for a given computer platform, and have the form

$$L = N_{nz} \times |\text{integer}| + 2N_{nz} \times |\text{pointer}| + \left(N\frac{d(d+1)}{2} + (N_{nz} - N)d^2 \right) \times |\text{real}|. \quad (4.1)$$

Equation (4.1) cannot be evaluated until the number of items N_{nz} is known, which is normally only after adding all of the items. A distinct advantage of the K3 storage format is that it does not require the storage requirements nor the indices of nonzero submatrices to be known in advance due to the use of memory allocation. Therefore, both the matrix assembly and the matrix factorization can be performed without the need for the computationally expensive fill-in analysis.

Although the K3 storage format is meant for finite element meshes with a constant number of nodal degrees of freedom d, it can be used for variable d as well. If all nodal submatrices are padded with zero entries to match the largest d, the K3 storage format can be used exactly as described earlier, but the efficiency will be affected by the added nonzeros.

Finally, it should be noted that memory allocation is not available in older programming languages, such as FORTRAN,[3] and therefore can present a significant problem for the application of the K3 storage format if the use of such language is mandatory.

4.2 Ordering method

The ordering (also called reordering, preordering, renumbering, relabeling, pivoting or permutation, depending on the context) of a matrix means switching rows and columns of the matrix to obtain a different, preferably more suitable, order of pivots on the main diagonal. Since switching of rows or columns is an elementary matrix operation, it does not change the solution of the associated linear equation system. However, the order of unknowns is generally changed (they are *relabeled*), and the actual numerical implementation must take trace of this relabeling to produce the solution vector in the correct (original) order.

The ordering is commonly used in the sparse matrix factorization prior to the forward elimination (called *preordering*) to reduce the fill-in. Another common application of the ordering (called *pivoting*) is to improve the numerical stability in the factorization of indefinite matrices.

Let $\mathbf{A}$ be an invertible matrix, $\mathbf{P}$ the corresponding (left) permutation matrix and $\mathbf{Q}$ the corresponding (right) permutation matrix that represent the chosen ordering of rows and columns, respectively. Then the LU factorization (2.3) with preordering takes the form

$$\mathbf{PAQ} = \mathbf{LU}. \quad (4.2)$$

[3]Prior to Fortran 90 standard.

The forward substitution (2.6) takes the form

$$\mathbf{Ly} = \mathbf{Pb} \tag{4.3}$$

and the back substitution (2.7) takes the form

$$\mathbf{Uz} = \mathbf{y}, \tag{4.4}$$

where an additional step is needed to permute the solution back into the original ordering in the form

$$\mathbf{x} = \mathbf{Qz}. \tag{4.5}$$

If matrix $\mathbf{A}$ is symmetric, then

$$\mathbf{Q} = \mathbf{P}^{\mathrm{T}}, \tag{4.6}$$

$$\mathbf{U} = \mathbf{DL}^{\mathrm{T}}, \tag{4.7}$$

and the $\mathbf{LDL}^{\mathrm{T}}$ factorization (2.11) with preordering is obtained.

The choice of the ordering method depends on the used matrix storage method and factorization method. Although sparse factorization can utilize any ordering method, fill-in minimization is mandatory in practice to reduce storage requirements of the matrix factors. The minimum degree algorithm is one of the most widely used orderings due to its effectiveness on a wide range of matrices. Only recently a theoretically superior but computationally expensive ordering method of nested dissections has been shown to be more efficient that minimum degree in some practical cases. However, nested dissections will not be considered in this work.

4.2.1 Minimum degree algorithm

The *minimum degree algorithm* is a symmetric analogue of the Markowitz' method [29] and was first proposed by Tinney and Walker [35] as algorithm S2. It was not renamed to minimum degree until later by Rose [34], who developed a graph theoretical model of the algorithm. In the past four decades, the algorithm has received much attention and a considerable amount of work has been directed towards improving the effectiveness of its numerical implementation. The most important work include that of George and Liu [19, 20], Duff and Reid [17], the multiple minimum degree (MMD) of Liu [28] and the approximate minimum degree (AMD) of Amestoy, Davis and Duff [1, 2].

The minimum degree algorithm analyzes the nonzero structure of a symmetric sparse matrix $\mathbf{A}$ to produce a permutation matrix $\mathbf{P}$ such that the factorization of the reordered matrix $\tilde{\mathbf{A}} = \mathbf{PAP}^{\mathrm{T}}$ exhibits the least fill-in. Unfortunately, the problem of finding the minimum fill-in is NP-complete, therefore the minimum degree algorithm is actually only one of the possible heuristics.

The description of the minimum degree algorithm relies heavily on the theory of graphs, that is briefly mentioned in Section 2.2, and is explained in detail for example in [11].

Elimination graph

The original minimum degree algorithm is based on the *symbolic elimination* using the *elimination graph* of the matrix. Considering an $n \times n$ matrix $\mathbf{A}$, the initial graph $G_0 = (V_0, E_0)$ is constructed according to the nonzero structure of the matrix, where the vertex set $V_0 = \{1, \ldots, n\}$ and the edge set $E_0 = \{(i, j) : a_{ij} \neq 0 \wedge i \neq j\}$. Since the matrix $\mathbf{A}$ is symmetric, the resulting graph G_0 is *undirected* and *simple*.

The symbolic elimination proceeds in steps that simulate the actual numerical elimination (factorization). In the kth elimination step, vertex p corresponding to the kth pivot is removed from the graph $G_{k-1} = (V_{k-1}, E_{k-1})$, i.e., vertex p is removed from V_{k-1} to form the new set V_k and all edges (i, p) are removed from E_{k-1} to form the new set E_k. Then new edges (i, j) are added to E_k for all i, j adjacent to p in G_{k-1}. In other words, new edges are added to the graph to connect all vertices formerly connected to vertex p, creating a *clique* (a fully connected subgraph). This addition of edges to the graph G_k represents the fill-in created in the kth step of the numerical factorization and means that the storage requirements of the elimination graph cannot be known in advance.

The minimum degree algorithm selects the pivot p in the kth step of the symbolic elimination that has a minimum *degree*, defined by

$$t_p = |\mathrm{Adj}_{G_{k-1}}(p)|. \tag{4.8}$$

Selecting p as the kth pivot causes the addition of at most $(t_p^2 - t_p)/2$ new edges in G_k.

The permutation matrix $\mathbf{P}$ is obtained from the sequence in which the vertices p_k, $k = 1, \ldots, n$, were removed from the graph G, i.e., from the sequence of the selected pivots.

Quotient graph

The *quotient graph* (also referred to as *generalized element model*) presents a substantial advantage over the elimination graph, since its storage requirements never exceed the size of the initial graph G_0.

The quotient graph $\mathcal{G}_k = (V_k \cup \overline{V}_k, E_k \cup \overline{E}_k)$ comprises of two distinct types of vertices and edges. The vertices consist of *variables* V_k and *elements* $\overline{V}_k$, where variables are vertices that have not yet been eliminated from the graph and elements are vertices that have been already eliminated. The edges consist of edges between variables E_k and edges between variables and elements $\overline{E}_k$. There are no edges between elements since they are unnecessary. In the initial graph $\mathcal{G}_0$, the sets V_0 and E_0 are identical to the elimination graph G_0, and the sets $\overline{V}_0$ and $\overline{E}_0$ are empty, i.e., $\mathcal{G}_0 = G_0$.

The subscript k denoting the elimination step will be dropped onwards for clarity. Let $\mathcal{A}_i$ be the set of variables adjacent to variable i in $\mathcal{G}$, let $\mathcal{E}_i$ be the set of elements adjacent to variable i in $\mathcal{G}$, and let $\mathcal{L}_e$ be the set of variables adjacent to element e in $\mathcal{G}$. Then

$$\mathcal{A}_i = \{j : (i, j) \in E\}, \tag{4.9}$$

$$\mathcal{E}_i = \{e : (i, e) \in \overline{E}\}, \tag{4.10}$$

$$\mathcal{L}_e = \{i : (i, e) \in \overline{E}\}, \tag{4.11}$$

and the adjacency sets are

$$\mathrm{Adj}_{\mathcal{G}}(i) = \mathcal{A}_i \cup \mathcal{E}_i, \tag{4.12}$$

$$\mathrm{Adj}_{\mathcal{G}}(e) = \mathcal{L}_e. \tag{4.13}$$

The set of variables adjacent to variable i in G, necessary for the computation of the degree (4.8), can be obtained using equations (4.9) to (4.13), and has the form

$$\mathrm{Adj}_G(i) = \left(\mathcal{A}_i \cup \bigcup_{e \in \mathcal{E}_i} \mathcal{L}_e \right) \setminus \{i\}. \tag{4.14}$$

The symbolic elimination proceeds similarly as with the elimination graph. The quotient graph is actually represented using the sets $\mathcal{A}$, $\mathcal{E}$ and $\mathcal{L}$, which is considerably more efficient than using the sets V, $\overline{V}$, E and $\overline{E}$. When variable p is selected as the kth pivot, the set $\mathcal{L}_p = \mathrm{Adj}_G(p)$ is found using equation (4.14). An important implication of equation (4.14) is that all variables adjacent to an element $e \in \mathcal{E}_p$ are also adjacent to the element p, i.e., $\mathcal{L}_e \setminus \{p\} \subseteq \mathcal{L}_p$. The elements $e \in \mathcal{E}_p$ are therefore no longer needed and are *absorbed* into the new element p. The graph is updated in the following way: first, the absorbed elements $e \in \mathcal{E}_p$ are deleted from all sets $\mathcal{E}_i$, and the new element p is added to the sets $\mathcal{E}_i$ for all variables $i \in \mathcal{L}_p$. Next, the sets $\mathcal{A}_p$, $\mathcal{E}_p$ and $\mathcal{L}_e$ for all elements $e \in \mathcal{E}_p$ are deleted. Finally, any entry j in $\mathcal{A}_i$, where both i and j are in $\mathcal{L}_p$, is redundant and is deleted. This results in that the graph $\mathcal{G}_k$ takes the same, or less, storage than the graph $\mathcal{G}_{k-1}$, or formally

$$(|\mathcal{A}_i| + |\mathcal{E}_i| + |\mathcal{L}_e|)_k \leq (|\mathcal{A}_i| + |\mathcal{E}_i| + |\mathcal{L}_e|)_{k-1}\,. \tag{4.15}$$

The set $\mathcal{L}_p$ represents the unpermuted nonzero structure of the kth column (or row) of the factor $\mathbf{L}$, i.e., the factor entry l_{ik} will be nonzero only if $i \in \mathcal{L}_p$. Therefore, the exact nonzero structure of the factor $\mathbf{L}$ (that also includes the fill-in) can be fully determined in the course of the symbolic elimination and used for example to create appropriate data structures for the numerical factorization.

An important property that allows the symbolic elimination algorithm to take advantage of graph cliques is the indistinguishability. Variables i and j are *indistinguishable* in G if

$$\mathrm{Adj}_G(i) \cup \{j\} = \mathrm{Adj}_G(j) \cup \{i\}, \tag{4.16}$$

which also implies they have the same degree. If i is selected as pivot in step k, j can be selected in step $k + 1$ without causing any additional fill-in.

Let $\mathcal{S}_i$ be the set of indistinguishable variables labeled by its *principal variable* i. Then

$$\mathcal{S}_i = \{i\} \cup \{j : \mathrm{Adj}_G(i) \cup \{j\} = \mathrm{Adj}_G \cup \{i\}\} \tag{4.17}$$

represents the *supervariable* i. Any variable from $\mathcal{S}_i$ can be used as a principal variable for labeling the supervariable. Variable i where $\mathcal{S}_i = \{i\}$ is called *simple variable*. Therefore, all variables in the graph can be considered principal variables.

In the initial graph $\mathcal{G}_0$, all variables are simple variables. When variable p is selected as the kth pivot, all variables in the set $\mathcal{S}_p$ are selected as well. The set $\mathcal{S}_p$ is deleted and supervariables are identified and created. When supervariable q is created, the set $\mathcal{S}_q$ is found using equation (4.17), and the sets $\mathcal{A}_i$, $\mathcal{E}_i$ and $\mathcal{S}_i$ for all variables $i \in \mathcal{S}_q \setminus \{q\}$ are deleted. In practice, the indistinguishability is checked in $\mathcal{G}$ rather than in G, since it is faster although some identifications may be missed. Also, only variables in $\mathcal{L}_p$ are checked for indistinguishability in each elimination step.

Selecting all variables in $\mathcal{S}_p$ in one step is called *mass elimination* and it substantially reduces the number of elimination steps necessary to obtain the ordering. Consequently, ordering time is reduced as well as the storage required for the quotient graph.

The *external* degree of a principal variable i is

$$d_i = t_i - |\mathcal{S}_i| + 1 = |\mathrm{Adj}_G(i) \setminus \mathcal{S}_i| = |\mathcal{A}_i \setminus \mathcal{S}_i| + \left| \left(\bigcup_{e \in \mathcal{E}_i} \mathcal{L}_e \right) \setminus \mathcal{S}_i \right|, \tag{4.18}$$

where t_i is the *true* degree defined by (4.8). Selecting the pivot with minimum external degree tends to produce a better ordering than selecting the pivot with minimum true degree, therefore, only the use of external degrees is assumed further on.

Degree computations are the most costly part of the minimum degree algorithm due to the presence of term $\bigcup_{e \in \mathcal{E}_i} \mathcal{L}_e$, which is complicated to evaluate. In practice, the degrees of all variables are computed in graph $\mathcal{G}_0$, and subsequently updated only for variables i adjacent to the pivot p in the actual step of the symbolic elimination, i.e., for variables $i \in \mathcal{L}_p$. This, along with the use of supervariables, results in a considerable reduction of the number of degree computations.

A minimum degree algorithm based on the quotient graph and including element absorption, mass elimination, supervariables, and external degrees is summarized in Table 4.2.

Approximate degree

An *approximate degree* replaces equation (4.18) with a less complex equation that computes an upper bound of the degree instead of the exact degree. While the cost of the degree computation is reduced, the ordering obtained by using approximate degrees is generally worse (results in more fill-in) than the ordering obtained by using exact degrees. Several approximate degrees are explained next in order from the least accurate.

The approximate degree proposed by Gilbert, Moler and Schreiber [22] is

$$\hat{d}_i = |\mathcal{A}_i \setminus \mathcal{S}_i| + \sum_{e \in \mathcal{E}_i} |\mathcal{L}_e \setminus \mathcal{S}_i|. \tag{4.19}$$

The redundancy of the entries in sets $\mathcal{L}_e$ is neglected in equation (4.19), therefore the upper bound can be computed very fast but results in a very coarse upper bound of the exact degree.

$V = \{1, \ldots, n\}$
$\overline{V} = \emptyset$
for $i = 1$ to n do
    $\mathcal{A}_i = \{j : a_{ij} \neq 0 \text{ and } i \neq j\}$
    $\mathcal{E}_i = \emptyset$
    $d_i = |\mathcal{A}_i|$
    $\mathcal{S}_i = \{i\}$
end for
$k = 1$
while $k \leq n$ do
    mass elimination:
    select variable $p \in V$ with minimum d_p
    $\mathcal{L}_p = (\mathcal{A}_p \cup \bigcup_{e \in \mathcal{E}_p} \mathcal{L}_e) \setminus \mathcal{S}_p$
    for each $i \in \mathcal{L}_p$ do
        remove redundant entries:
        $\mathcal{A}_i = (\mathcal{A}_i \setminus \mathcal{L}_p) \setminus \mathcal{S}_p$
        element absorption:
        $\mathcal{E}_i = (\mathcal{E}_i \setminus \mathcal{E}_p) \cup \{p\}$
        compute external degree:
        $d_i = |\mathcal{A}_i \setminus \mathcal{S}_i| + \left|\left(\bigcup_{e \in \mathcal{E}_i} \mathcal{L}_e\right) \setminus \mathcal{S}_i\right|$
    end for
    supervariable detection, pairs found via hash function:
    for each pair i and $j \in \mathcal{L}_p$ do
        if i and j are indistinguishable then
            remove supervariable j:
            $\mathcal{S}_i = \mathcal{S}_i \cup \mathcal{S}_j$
            $d_i = d_i - |\mathcal{S}_j|$
            $V = V \setminus \{j\}$
            $\mathcal{A}_j = \emptyset$
            $\mathcal{E}_j = \emptyset$
        end if
    end for
    convert variable p to element p:
    $\overline{V} = (\overline{V} \cup \{p\}) \setminus \mathcal{E}_p$
    $V = V \setminus \{p\}$
    $\mathcal{A}_p = \emptyset$
    $\mathcal{E}_p = \emptyset$
    $k = k + |\mathcal{S}_p|$
end while

Table 4.2: Minimum degree algorithm based on quotient graph

The approximate degree proposed by Ashcraft, Eisenstat and Lucas [2] is

$$\tilde{d}_i = \begin{cases} d_i & \text{if } |\mathcal{E}_i| = 2 \\ \hat{d}_i & \text{if } |\mathcal{E}_i| \neq 2 \end{cases} . \tag{4.20}$$

The reasoning behind equation (4.20) is that many variables are adjacent to two or fewer elements when their degree is recomputed. Utilizing the exact degree increases the complexity but a significantly better upper bound of the exact degree is obtained than using (4.19).

The approximate degree proposed by Amestoy, Davis and Duff [2] is

$$\overline{d}_i = |\mathcal{A}_i \setminus \mathcal{S}_i| + |\mathcal{L}_p \setminus \mathcal{S}_i| + \sum_{e \in \mathcal{E}_i \setminus \{p\}} |\mathcal{L}_e \setminus \mathcal{L}_p|. \tag{4.21}$$

In equation (4.21) the properties of subsets $\mathcal{L}_e \cap \mathcal{L}_p$ and $\mathcal{L}_e \setminus \mathcal{L}_p$ are exploited to obtain a closer upper bound of the exact degree while maintaining a reasonable complexity. However, some redundant entries in sets $\mathcal{L}_e$ are still counted.

Since the approximate degree is an upper bound of the exact degree, it cannot exceed the number of variables in the actual elimination step $n - k$ nor the worst case fill-in, that can be easily obtained by adding $|\mathcal{L}_p \setminus \mathcal{S}_i|$ to the degree computed in the previous elimination step. Therefore the complete form of equation (4.21) in the kth elimination step is

$$(\overline{d}_i)_k = \min \left\{ \begin{array}{l} n - k \\ (\overline{d}_i)_{k-1} + |\mathcal{L}_p \setminus \mathcal{S}_i| \\ |\mathcal{A}_i \setminus \mathcal{S}_i| + |\mathcal{L}_p \setminus \mathcal{S}_i| + \displaystyle\sum_{e \in \mathcal{E}_i \setminus \{p\}} |\mathcal{L}_e \setminus \mathcal{L}_p| \end{array} \right\}. \tag{4.22}$$

The aproximate degree of Gilbert, Moler and Schreiber can be complemented in the same way, replacing the last row in equation (4.22) with equation (4.19).

Further enhancements

Actual implementations of the minimum degree algorithm include the following enhancements in addition to those already mentioned: *aggressive absorption*, *multiple elimination* and *incomplete degree update*.

In aggressive absorption, any element with $\mathcal{L}_e \setminus \mathcal{L}_p = \emptyset$ is absorbed into element p, even if it is not adjacent to p. Aggressive absorption can improve the computation of degrees since it reduces the size of sets $\mathcal{E}_i$.

In multiple elimination, all independent pivots with minimum degree are selected before any degrees are updated. If a variable is adjacent to two or more pivot elements, its degree is computed only once.

Finally, in incomplete degree update, the degree update of an *outmatched* variable j is avoided until variable i is selected as pivot. Variable j is outmatched if $\mathrm{Adj}_G(i) \subseteq \mathrm{Adj}_G(j)$.

Tiebreaking

A crucial aspect of the minimum degree algorithm is the tiebreaking strategy that is applied when more than one variable with minimum degree can be selected as pivot. The matter has been a topic of research, but so far no significant method has been developed, and no established implementation of the minimum degree ordering uses tiebreaking.[4] The difficulty lies mainly in the complexity of such methods, since the tiebreaking usually considerably increases the time of the symbolic elimination. Therefore, ties are resolved generally by simply selecting the pivot with the lowest vertex number, which does not affect the ordering time.

More on tiebreaking strategies can be found in [19] and [8]. For example, George and Liu suggest in [19] to apply the minimum degree algorithm to the matrix that has been 'preordered' using the reverse Cuthill-McKee ordering (see Section 2.2). This particular procedure is claimed to result in an ordering that is independent on the initial ordering of matrix elements, effectively resolving the ties.

4.3 Solution method

Similarly to the matrix storage method, the solution method must meet the requirements stated in Chapter 3 that the whole solution process is to be carried out by a direct method, preferably in memory (in-core). Another requirement is that the coefficient matrices are restricted to be symmetric and positive definite, which is a reasonable assumption for the given application to the finite element analysis of solids and structures.

The frontal solution method has relatively low memory demands, but it requires a lot of out-of-core data manipulation that would slow down the solution considerably, and particularly it cannot take advantage of sparse matrix ordering.

The multifrontal solution method is able to exploit ordering, but it introduces even more out-of-core data manipulation than the frontal solution method and involves rather complicated algorithms.

The active column solution (skyline reduction) method allows a complete in-core solution, but requires the skyline storage format that would be very inefficient since it would exhibit an excessive number of unused zero entries due to very high and very sparse columns caused by the minimum degree ordering (see Figure 2.1 and Subsection 4.2.1). This would not be resolved even if some block variant of the skyline storage would be introduced.

Finally, the (block) sparse direct solution method is relatively straightforward, allows a complete in-core solution to be used, and an efficient (block) sparse matrix storage scheme and fill-in minimization ordering to be applied. It has however relatively high storage requirements because the whole coefficient matrix has to be stored in memory unless an out-of-core solution is considered.

[4] Strictly speaking, multiple elimination can be considered a somewhat limited form of tiebreaking.

Direct solution methods based on $\mathbf{LU}$, $\mathbf{LDL}^{\mathrm{T}}$ and Cholesky factorizations, that were briefly described in Subsection 2.1.1, are all variants of the Gaussian elimination. The Gaussian elimination is a very important algorithm since it can be used to solve system (2.1) without the need for computing the inverse $\mathbf{A}^{-1}$.

4.3.1 Symmetric block sparse factorization

Standard Gaussian elimination comprises of a *forward elimination*, where matrix $\mathbf{A}$ is reduced to an upper triangular matrix along with the right-hand side vector $\mathbf{b}$,[5] and a *back substitution*, where the solution vector $\mathbf{x}$ is computed from the triangular system obtained in the forward elimination. However, often the solution of the same linear equation system is required to be computed for several different right-hand sides, for example in the finite element method, the problem may be solved for several different loading states. The Gaussian elimination is not suitable for such cases since both the forward elimination and back substitution would be performed for each different right-hand side, which would be very inefficient due to the high computational complexity of the forward elimination. Therefore in practical applications, factorization methods that separate the reduction of the right-hand side from the reduction of the coefficient matrix itself are used. These methods allow the forward elimination to be carried out only once for a given system and then any number of right-hand sides may be used to compute the corresponding solution. Consequently, the solution is divided into two steps: a *forward substitution*, that reduces the right-hand side, and the aforementioned back substitution.[6]

Factorization

The forward elimination (also called reduction, triangularization, factorization or decomposition, depending on the context) of an $n \times n$ matrix $\mathbf{A}$ is performed in $n-1$ steps.

In the kth elimination step, all column entries a_{ik} under the diagonal entry a_{kk}, called the *pivot*, are eliminated (zeroed) by subtracting a multiple of the kth row, called the *pivot row*, from remaining (uneliminated) rows. The multiplier for ith row is

$$l_{ik}^{(k)} = -\frac{a_{ik}^{(k)}}{a_{kk}^{(k)}}, \tag{4.23}$$

where $i = k+1, \ldots, n$ is the row index and the superscript $k = 1, \ldots, n-1$ refers to the actual elimination step.

[5] The elimination is usually done on an augmented matrix $(\mathbf{A}|\mathbf{b})$.

[6] The term *back substitution* is often inaccurately used to refer to both the forward and back substitution, although they are technically different.

The kth elimination step can be written using equation (4.23) in the matrix form

$$\mathbf{A}_{k+1} = \mathbf{L}_k^{-1}\mathbf{A}_k = \\ = \begin{pmatrix} 1 & & & & & 0 \\ & \ddots & & & & \\ & & 1 & & & \\ & & -\frac{a_{k+1,k}^{(k)}}{a_{kk}^{(k)}} & 1 & & \\ & & \vdots & & \ddots & \\ 0 & & -\frac{a_{nk}^{(k)}}{a_{kk}^{(k)}} & 0 & & 1 \end{pmatrix} \begin{pmatrix} a_{11}^{(k)} & \cdots & a_{1k}^{(k)} & \cdots & a_{1n}^{(k)} \\ & \ddots & \vdots & & \vdots \\ & & a_{kk}^{(k)} & \cdots & a_{kn}^{(k)} \\ & & a_{k+1,k}^{(k)} & \cdots & a_{k+1,n}^{(k)} \\ & & \vdots & & \vdots \\ 0 & & a_{nk}^{(k)} & \cdots & a_{nn}^{(k)} \end{pmatrix}, \tag{4.24}$$

where $\mathbf{A}_k$ and $\mathbf{A}_{k+1}$ is the partially reduced matrix in the kth and $(k+1)$th elimination step, respectively, and $\mathbf{L}_k$ is the elementary matrix for elimination of the kth row.

After writing down all steps of the forward elimination the resulting matrix form is

$$\mathbf{U} = \mathbf{A}_n = \mathbf{L}_{n-1}^{-1} \cdots \mathbf{L}_2^{-1}\mathbf{L}_1^{-1}\mathbf{A} = \mathbf{L}^{-1}\mathbf{A}, \tag{4.25}$$

where $\mathbf{U}$ is an upper triangular matrix containing the result of the forward elimination, the reduced matrix $\mathbf{A}$, and $\mathbf{L}^{-1}$ is a unit lower triangular matrix, since the multiplication of unit lower triangular matrices $\mathbf{L}_k^{-1}$ results in a unit lower triangular matrix. Matrix $\mathbf{L}^{-1}$ can be used to perform the forward elimination on matrix $\mathbf{A}$ in a single step.

Combining the leftmost and the rightmost side of (4.25) and realizing that the inverse of a unit lower triangular matrix is also a unit lower triangular matrix, the $\mathbf{LU}$ factorization of matrix $\mathbf{A}$ is obtained in the form

$$\mathbf{A} = \mathbf{LU}. \tag{4.26}$$

Matrix $\mathbf{U}$ can be also expressed as a multiplication of a diagonal matrix $\mathbf{D}$ and a unit upper triangular matrix $\tilde{\mathbf{U}}$. Substituting to (4.26) yields

$$\mathbf{A} = \mathbf{LD}\tilde{\mathbf{U}}. \tag{4.27}$$

If matrix $\mathbf{A}$ is symmetric, the right-hand side of (4.27) must also be symmetric, therefore

$$\tilde{\mathbf{U}} = \mathbf{L}^{\mathrm{T}} \tag{4.28}$$

and hence the $\mathbf{LDL}^{\mathrm{T}}$ factorization of the matrix $\mathbf{A}$ is obtained in the form

$$\mathbf{A} = \mathbf{LDL}^{\mathrm{T}}. \tag{4.29}$$

The matrix factors $\mathbf{L}$ and $\mathbf{U}$ are computed using Gaussian elimination and stored in place of the original matrix $\mathbf{A}$. In the case of a symmetric matrix $\mathbf{A}$, there are two possible approaches how to compute the factors $\mathbf{L}$ and $\mathbf{D}$.

Right-looking and left-looking algorithm

The forward elimination algorithm described above proceeds from top to bottom and from left to right (by rows) and is called *right-looking* (or *eager*) algorithm, since it updates the matrix entries to the right of the pivot column as soon as the entry $a_{ik}^{(k)}$ is eliminated. The elimination of the kth pivot requires the whole submatrix $a_{ij}^{(k)}$, where $i, j = k+1, \ldots, n$, to be updated.

A different forward elimination algorithm can be derived from the definition of the $\mathbf{LDL}^{\mathrm{T}}$ factorization. The equation (4.29) can be rewritten in the form

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{12} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{nn} \end{pmatrix} = \\ = \begin{pmatrix} 1 & & & & 0 \\ l_{21} & 1 & & & \\ l_{31} & l_{32} & 1 & & \\ \vdots & \vdots & \vdots & \ddots & \\ l_{n1} & l_{n2} & l_{n3} & \cdots & 1 \end{pmatrix} \begin{pmatrix} d_{11} & d_{11}l_{21} & d_{11}l_{31} & \cdots & d_{11}l_{n1} \\ & d_{22} & d_{22}l_{32} & \cdots & d_{22}l_{n2} \\ & & d_{33} & \cdots & d_{23}l_{n3} \\ & & & \ddots & \vdots \\ 0 & & & & d_{nn} \end{pmatrix}. \tag{4.30}$$

For an arbitrary entry of the matrix $\mathbf{A}$ the equation (4.30) yields

$$a_{ij} = \sum_{p=1}^{i-1} l_{ip} d_{pp} l_{jp} + d_{ii} l_{ji}. \tag{4.31}$$

From equation (4.31) the relations for the factor entries are derived in the form

$$l_{ji} = \frac{1}{d_{ii}} \left(a_{ij} - \sum_{p=1}^{i-1} l_{ip} d_{pp} l_{jp} \right), \tag{4.32}$$

$$d_{ii} = a_{ii} - \sum_{p=1}^{i-1} l_{ip}^2 d_{pp}. \tag{4.33}$$

Equation (4.33) is obtained by realizing that $l_{ii} = 1$.

The described elimination algorithm proceeds from left to right and from top to bottom (by columns) and is called *left-looking* (or *lazy*) algorithm, since it updates the entry in the pivot column from the entries to the left as late as possible. It is commonly used for column-oriented matrix storage formats, such as the skyline format or the compressed column format, since the sums in equations (4.32) and (4.33) actually represent a scalar product of two columns.

The theoretical complexity of the factorization is $O(n^3)$ and can be evaluated exactly by counting the number of arithmetic operations. The same complexity (roughly $2n^3/3$)

is obtained for both the right-looking and the left-looking algorithm, making the algorithms equivalent. Due to the additional operations that are necessary for handling of the matrix storage and possibly for pivoting, the complexity of the factorization in practice is highly dependent on the particular numerical implementation and, therefore, is difficult to evaluate.

Numerical stability

A potential difficulty in the forward elimination may arise in equation (4.23) or (4.32) if the pivot in the denominator is zero or very small relative to the other entries in the pivot row. In this case the elimination is said to be numerically unstable and generally cannot proceed unless a better pivot can be selected. If the pivot is not zero but very small, the difficulty lies in that the obtained multiplier may be so large that the accuracy is lost due to the finite-precision arithmetic, and a small change in the right-hand side may cause unreasonably large changes in the solution.

New pivot in the kth step of the elimination can be selected either from the pivot column

$$p_k = \max_{i \geq k} |a_{ik}|, \tag{4.34}$$

by switching corresponding rows, which is called *partial pivoting*, or from the whole remaining uneliminated submatrix

$$p_k = \max_{i,j \geq k} |a_{ij}|, \tag{4.35}$$

by switching corresponding rows and columns, which is called *full pivoting*. Partial pivoting does not require relabeling of unknowns and is usually sufficient in many practical applications.

For diagonally dominant or positive definite matrices the forward elimination is always numerically stable and therefore no pivoting is necessary.

Solution

After the forward elimination is carried out and the factors $\mathbf{L}$ and $\mathbf{U}$ or $\mathbf{D}$ are computed, the solution of the system (2.1) for an arbitrary right-hand side is obtained in two steps.

In the forward substitution, the first triangular system

$$\mathbf{L}\mathbf{y} = \mathbf{b} \tag{4.36}$$

is solved to obtain vector $\mathbf{y}$, the reduced right-hand side. Since $\mathbf{L}$ is a unit lower triangular matrix, y_1 can be solved immediately and substituted into the second equation, and so on, until the last equation where y_n is solved.

In the back substitution, the second triangular system

$$\mathbf{U}\mathbf{x} = \mathbf{D}\mathbf{L}^{\mathrm{T}}\mathbf{x} = \mathbf{y} \tag{4.37}$$

is solved to obtain vector $\mathbf{x}$, the solution. Since $\mathbf{U} = \mathbf{D}\mathbf{L}^{\mathrm{T}}$ is an upper triangular matrix, x_n can be solved immediately and substituted into the second-to-last equation, and so on, until the first equation where x_1 is solved.

The theoretical complexity of the forward and back substitution is $O(n^2)$ and can be evaluated exactly by counting the number of arithmetic operations, which is roughly n^2. Due to the additional operations that are necessary for handling of the matrix storage the complexity of the substitution in practice is highly dependent on the particular numerical implementation and, therefore, is difficult to evaluate.

Block sparse factorization

For application to large symmetric linear equation systems the block sparse $\mathbf{L}\mathbf{D}\mathbf{L}^{\mathrm{T}}$ factorization is the most efficient variant of the $\mathbf{L}\mathbf{U}$ factorization. The algorithms described earlier work directly with matrix entries but the corresponding block variant requires only a few formal changes in the definitions, however, the numerical implementation of block algorithms is substantially more complex.

Let equation (4.29) be rewritten in the block form

$$\begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & \cdots & \mathbf{A}_{1m} \\ \mathbf{A}_{12} & \mathbf{A}_{22} & \cdots & \mathbf{A}_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{A}_{1m} & \mathbf{A}_{2m} & \cdots & \mathbf{A}_{mm} \end{pmatrix} = \\ = \begin{pmatrix} \mathbf{I} & & & & \\ \mathbf{L}_{21} & \mathbf{I} & & & \\ \mathbf{L}_{31} & \mathbf{L}_{32} & \mathbf{I} & & \\ \vdots & \vdots & \vdots & \ddots & \\ \mathbf{L}_{m1} & \mathbf{L}_{m2} & \mathbf{L}_{m3} & \cdots & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{D}_{11} & \mathbf{D}_{11}\mathbf{L}_{21}^{\mathrm{T}} & \mathbf{D}_{11}\mathbf{L}_{31}^{\mathrm{T}} & \cdots & \mathbf{D}_{11}\mathbf{L}_{m1}^{\mathrm{T}} \\ & \mathbf{D}_{22} & \mathbf{D}_{22}\mathbf{L}_{32}^{\mathrm{T}} & \cdots & \mathbf{D}_{22}\mathbf{L}_{m2}^{\mathrm{T}} \\ & & \mathbf{D}_{33} & \cdots & \mathbf{D}_{33}\mathbf{L}_{m3}^{\mathrm{T}} \\ & & & \ddots & \vdots \\ & & & & \mathbf{D}_{mm} \end{pmatrix}, \tag{4.38}$$

where matrices $\mathbf{A}$, $\mathbf{L}$ and $\mathbf{D}$ are partitioned evenly into $m \times m$ blocks and $\mathbf{I}$ is the unit matrix of order m. Then the equations for computing the matrix entry (4.31) and the factors (4.32) and (4.33) have the block form

$$\mathbf{A}_{ij} = \sum_{p=1}^{i-1} \mathbf{L}_{ip}\mathbf{D}_{pp}\mathbf{L}_{jp}^{\mathrm{T}} + \mathbf{L}_{ji}\mathbf{D}_{ii}. \tag{4.39}$$

$$\mathbf{L}_{ji} = \left(\mathbf{A}_{ij} - \sum_{p=1}^{i-1} \mathbf{L}_{ip}\mathbf{D}_{pp}\mathbf{L}_{jp}^{\mathrm{T}} \right) \mathbf{D}_{ii}^{-1}, \tag{4.40}$$

$$\mathbf{D}_{ii} = \mathbf{A}_{ii} - \sum_{p=1}^{i-1} \mathbf{L}_{ip}\mathbf{D}_{pp}\mathbf{L}_{ip}^{\mathrm{T}}. \tag{4.41}$$

The numerical stability of the block factorization depends on the existence of the inverse $\mathbf{D}_{ii}^{-1}$ in equation (4.40). All statements regarding the numerical stability of the factorization indicated earlier are also valid for the block variant.

The advantage of the block sparse factorization lies in that the many of the matrices $\mathbf{A}_{ij}$, $\mathbf{L}_{ij}$ and especially $\mathbf{D}_{ij}$ are zero, therefore large sparse matrices may be processed more efficiently than in the non-block sparse factorization. The sum of triple matrix products in equations (4.40) and (4.41) should be implemented in some efficient way, since the multiplication can be often avoided if either $\mathbf{L}_{ip}$ or $\mathbf{L}_{jp}^{\mathrm{T}}$ is a zero matrix. Matrix-matrix multiplication can be implemented efficiently using highly optimized standard BLAS level 3 functions [6].

As already mentioned in previous sections, a major difficulty of the forward elimination on sparse matrices is the *fill-in*, i.e., that some (often many) of the initially zero entries (or zero matrices in the block variant) become nonzero. Only nonzero entries (submatrices) of the sparse matrix $\mathbf{A}$ are usually stored to achieve minimum storage requirements, making the fill-in very inconvenient since it spoils the effort by increasing the size of the matrix storage. Consequently, although there may be enough storage capacity for the whole original matrix $\mathbf{A}$, the factor $\mathbf{DL^T}$ produced by the factorization may require several orders of magnitude larger storage space, rendering the solution practically unfeasible. Therefore, to reduce the fill-in, a suitable ordering method is normally used prior to the factorization.

4.4 PMD implementation concepts

PMD (Package for Machine Design, version f77.10 as of 2011) is an in-house code for finite element analysis in solid continuum mechanics. It has a long, 30-year tradition, and it is presently developed by VAMET Ltd. and co-developed by the staff of the Academy of Sciences of the Czech Republic.[7] PMD also features own preprocessor and postprocessor GFEM that provides a comfortable graphical user interface comparable to other available commercial software.

The main advantage of the system is its unified modular structure that allows any modifications to existing methods or implementation of new methods to be carried out at any level of the finite element computation. The system features a set of compatible command-line programs where each performs a designated part of the computation. Common subroutines are kept in the main library S3[8] that can be used by any PMD program. A sophisticated method of testing on verified example problems allows checking of the accuracy of the obtained solutions and prevents any modifications that would break the functionality of the system in any way.

The PMD code is written and maintained in FORTRAN 77, largely due to the availability and reliability of optimized compilers for most computer platforms, and it is used for finite element computations on PCs, workstations, and even supercomputers (Cray and NEC). The conversion of the code to a newer version of Fortran is unfeasible due to

[7] The development takes place at the Laboratory of Computational Solid Mechanics, Department of Impact and Waves in Solids, Institute of Thermomechanics ASCR.

[8] The library for 3-D elastostatic analysis. Other auxiliary libraries are available for non-linear problems, dynamic problems, etc.

the overall complexity of the system. Since the inputs and outputs of PMD programs are well defined (see Subsection 4.4.3), it is possible to create a replacement program for any part of the computation with any programming language. The programs constituting the PMD must however conform to the internal rules of development.

The basic concepts required for the implementation of the sparse direct solver in the PMD system are presented in the rest of this section. Most of these concepts are obsolete by present standards, since structured data types and dynamic memory allocation are generally available. Nevertheless, it should be noted that the PMD's framework is quite impressive considering the limited facilities available in FORTRAN 77.

4.4.1 Parameter passing

To share the global parameters of the finite element computation, such as problem parameters (number of mesh nodes, length of the solution, etc.) or program states (open files, array sizes, etc.), one common block is used throughout all subroutines. This is quite practical since otherwise the subroutines would have to include many global parameters in their argument list. A typical declaration of the common block is illustrated by the following code:

```
      COMMON /CPMD/ ICP(160),ICW(160),RCP(48),RCW(336)
      EQUIVALENCE (ICW(8),NNE),(ICP(96),KAVT),(ICP(133),KPOUT),
     1   (ICP(140),KSTR),(ICW(112),IG),(RCP(38),TIGP),(RCW(2),ALF),
     2   (ICP(4),ILINE),(ICW(4),IE)
```

The common block is divided into four parts. Arrays `ICP` and `RCP` are used to store integer and real parameters unique to the problem, while arrays `ICW` and `RCW` can be used to store arbitrary integer and real parameters relevant to the current algorithm. For a convenient handling of the common block the individual array elements are mapped to local variables using the `EQUIVALENCE` statement. The full description of the parameters stored in the common block can be found in the internal PMD documentation.

The main PMD library S3 provides subroutines `WCOMD` and `RCOMD`, which write and read the common block from a file (see Subsection 4.4.3) and allow sharing of the global parameters between separate programs. However, only the arrays `ICP` and `RCP` are stored on the disk, since the arrays `ICW` and `RCW` are considered temporary.

4.4.2 Memory allocation

The common block described in Subsection 4.4.1 is far too small to hold any actual computational data such as the mesh topology or element matrices. For this purpose a large workspace (called *core* in PMD terminology) is allocated at the start of each program, and used throughout all subroutines. A typical initialization of the workspace is illustrated by the following code:

```
      PARAMETER (LI = 10 000 000)
      DIMENSION INT(LI),R(LI/2)
```

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...
j	1		2		3		4		5		6		7		8		...
INT(i)	*integer data*						×	×	×	×	*integer data*				×	×	...
R(j)	×		×		×		*real data*				×		×		*real data*		...

Figure 4.4: Interleaving of integer and real data in PMD program core for ICL = 2

```
EQUIVALENCE (INT(1),R(1))
COMMON /CPMD/ ICP(160),ICW(160),RCP(48),RCW(336)
EQUIVALENCE (ICP(3),ICL)
CALL INIT('FEFS',ISEC)
LR=LI/ICL
```

Parameter LI determines the size of the worspace (core size), i.e., the total memory available to the program. Arrays INT and R represent the workspace (core) and are used to store integer and real data, respectively. The arrays share the same physical memory space (due to the EQUIVALENCE statement) to allow arbitrary interleaving of integer and real data in the workspace. Parameter ICL holds the integer-to-real factor and is defined as

$$\texttt{ICL} = \frac{|\text{real}|}{|\text{integer}|}. \tag{4.42}$$

ICL is constant for a given computer platform, for example ICL = 2 on 32-bit workstations and PCs.[9] Since one element of the real array R spans ICL elements of the integer array INT, indices and lengths can be easily converted between the arrays in practice. ICL is stored in the common block and initialized in the main PMD library S3 subroutine INIT, which is advantageous when the need for recompiling of the PMD code on a different computer platform arises.

The use of the workspace for is illustrated in Figure 4.4. The interleaving of different data types must be done carefully, since changing any of the array elements marked × results in the corruption of the data in the other array. Relations

$$i = (j-1) \times \texttt{ICL} + 1, \tag{4.43}$$

$$j = (i-1) \div \texttt{ICL} + 1, \tag{4.44}$$

can be used to calculate the exact indices of corresponding array elements.

Although the described pseudo-dynamic memory allocation strategy adopted by the PMD code of course cannot be as flexible as the actual dynamic memory allocation provided by newer programming languages, the only practical inconvenience is that the program must be recompiled whenever its core size needs to be increased. However, the basic PMD programs sold commercially have a reasonably large core size and normally do not need recompiling. There is also a special version of PMD programs which features dynamic memory allocation using non-standard platform-dependent functions.

[9] When using double precision floating-point arithmetic.

4.4.3 Input and output files

To ensure the interoperability between PMD programs and facilitate their input and ouput, a simple system of data files is adopted. All data files belonging to the same problem have the same user-chosen name *name*, while the extension is determined by the type of data contained within, as is customary.

The most important are the input files (*name*.I*c*) and the output files (*name*.O*c*), where *c* is an alphanumeric character identifying the particular program or programs that utilize the particular file. Input files are formatted and are used by the user to input all parameters and data needed by the particular program in a readable text format. Input files are general-purpose, meaning that they may contain as little as a few integer and real parameters (for example *name*.I4), or as much as the full description of the finite element mesh (for example *name*.I1). Output files are also formatted and are used by the programs to output the requested results back to the user in a readable text format. Output files usually contain additional information about the program runtime, such as the used CPU time. The full description of the PMD input and output files can be found in [32].

Other data files are unformatted to allow efficient read and write operations, and are intended only for the internal use in PMD programs. For example, file *name*.CMN is used to store the common block described in Subsection 4.4.1. If the structure of a particular data file needs to be known, for example when modifying an existing program or developing a new program, the appropriate description can be found in the internal PMD documentation.

Chapter 5

Results and discussion

In this chapter the theoretical results as well as the practical results of the work are presented and discussed. The first section explains in detail the modifications proposed to the matrix storage method, the ordering method, and the solution method, to enhance theireffectiveness specifically in the case of large finite element problems. The second section describes the particular use of the abovementioned methods in a sparse direct solver designed for the PMD finite element system. The third section presents the numerical results obtained with the sparse direct solver, and also includes a performance comparison with the existing frontal solver of the PMD system, to demonstrate the efficiency of the sparse direct solver.

Results presented in Section 5.1 have been already partially published in papers [39, 40, 41, 42, 43].

5.1 Proposed algorithms

When the size of a finite element problem exceeds a certain level (say, 10^6 equations) the effectiveness of the matrix storage method, the ordering method, and the solution method becomes highly dependent on the effectiveness of the actual algorithms. Practically, there are only limited possibilities for the theoretical improvement of any of the methods, however the underlying algorithms do present a potential for refinement. Nevertheless, the applied methods have been thoroughly analyzed and new algorithms were devised with regard to the solution of large finite element problems. The findings are discussed in this section.

5.1.1 Matrix storage method

The initial matrix storage method chosen for the sparse direct solver was the K3 storage format (see Subsection 4.1.1), to facilitate an efficient in-core solution of large finite element problems. However, during the testing of the sparse direct solver, it has been found that the K3 storage format is severely limited on 32-bit platforms by the 2 GB memory

limit imposed by the operating system[1], and consequently the in-core solution is possible only for finite element problems with about 5×10^5 equations. Since this is clearly unsatisfactory and it also contradicts the aims stated in Chapter 3, the scope of the work has been extended to facilitate an out-of-core solution, allowing finite element problems with 10^6 equations or more to be solved even when the memory is limited. Unfortunately the K3 storage format has been found impractical for an efficient out-of-core solution, and therefore a more suitable matrix storage method has been proposed for the use in the sparse direct solver.

Before continuing with the discussion on the matrix storage method, it is necessary to consider also the 64-bit platform. The theoretical memory limit for 64-bit programs is approximately 8.6×10^9 GB, which is more than enough for finite element problems with even 10^8 equations. In this case the in-core solution is perfectly feasible, since the existing operating systems can transparently emulate any practical amount of virtual memory (if requested physical memory is not available) by using a disk storage. However, the sparse direct solver is intended primarily for finite element computations on existing 32-bit workstations and PCs, and therefore the ability to perform the out-of-core solution is essential.

Modifications to K3 storage format

FORTRAN 77 lacks structured data types and dynamic memory allocation, and therefore the application of the K3 storage format is quite complicated, but not impossible. For the use in the sparse direct solver the original K3 storage format has been adapted to the PMD's memory allocation method (see Subsection 4.4.2).The resulting storage format (let it be called K3/F77 for an easier reference) has the same functionality as the original K3 storage format with the exception of limited item removal (that is not needed in practice anyway), but is considerably more efficient since the items are smaller. The K3/F77 storage format data structure can be easily written and read from a sequential unformatted file, item by item.

Unlike the K3 storage format, the K3/F77 storage format takes into account the possible different number of nodal degrees of freedom in the finite element mesh, using an additional array d of size N to store the degrees instead of a single value of d. Moreover, the submatrix entries are stored directly in the item, therefore the only overhead needed is the *index* data member and the *pointer to next item* data member (cf. Figure 4.2). Consequently, the items have only minimum necessary size and thus larger coefficient matrices can be stored than in the K3 storage format, considering the same storage space. This efficiency is particularly significant in the factorization, since the submatrices added due to the fill-in are smaller.

The K3/F77 storage format data structure is illustrated in Figure 5.1 for `ICL` $= 2$ (cf. Figure 4.4). Unfortunately, due to the interleaving of integer and real data, the description is somewhat complicated. For clarity, the origin of the data structure is shown coincident with the origin of the workspace `INT(1)` and `R(1)`. Since the pointers p_{ij} (and q_{ij}) are

[1]Regardless of operating system and platform.

	Pointers to first row item							
u	1	2	3	...	$N-2$	$N-1$	N	...
v	$\times$							...
INT(u)	p_{11}	p_{22}	p_{33}	...	$p_{N-2,N-2}$	$p_{N-1,N-1}$	p_{NN}	...
R(v)	$\times$							...

		Diagonal item					
u	...	p_{ii}	$p_{ii}+1$	$p_{ii}+2$	$\times$	$p_{ii}+k_{ii}-1$	...
v	...	$\times$		q_{ii}	...	$q_{ii}+l_{ii}-1$	...
INT(u)	...	$*$	i	$\times$			...
R(v)	...	$\times$		$\mathbf{A}_{ii}$			...

		Nondiagonal item					
u	...	p_{ij}	$p_{ij}+1$	$p_{ij}+2$	$\times$	$p_{ij}+k_{ij}-1$	...
v	...	$\times$		q_{ij}	...	$q_{ij}+l_{ij}-1$	...
INT(u)	...	$*$	j	$\times$			...
R(v)	...	$\times$		$\mathbf{A}_{ij}$			...

Legend:

N	number of nodes
i	nodal row index (row corresponding to node i)
j	nodal column index (column corresponding to node i)
p_{ij}	pointer to item corresponding to nodes i, j
k_{ij}	length of item corresponding to nodes i, j
q_{ij}	pointer to nodal submatrix corresponding to nodes i, j
l_{ij}	length of nodal submatrix corresponding to nodes i, j
$\mathbf{A}_{ij}$	nodal submatrix corresponding to nodes i, j
$*$	pointer to the next item on the row, 0 if it is the last item
$\times$	not applicable indices or elements
u, v	workspace array element indices

Figure 5.1: K3/F77 storage format data structure

always relative to the origin of the data structure, it can be placed (almost) anywhere in the workspace. In practice, the data structure is placed at the end of the allocated workspace, since it uses the whole remaining part of the workspace for the storage of items.

The first N array elements of the data structure are reserved for row pointers[2] (pointers to the first item of the row). The items are stored as in the K3 storage format by rows using linked lists, beginning with the diagonal item and continuing with the nondiagonal items sorted ascendingly by the column index. However, all items in the K3/F77 storage format are stored dynamically, unlike the K3 storage format, where the diagonal items are stored separately in a static array. Another difference is that all items store the nodal column index in the *index* data member, to simplify storage algorithms (the number of nonzero items is not needed in practice anyway).

The initial size of the data structure is

$$L_0 = [(N + \texttt{ICL} - 1) \div \texttt{ICL}] \times \texttt{ICL}, \tag{5.1}$$

where `ICL` (integer-to-real factor, see Subsection 4.4.2) is used to align integer and real data to the same length. All row pointers are initially zero (indicating a *null* value). New items are added to the data structure simply beginning from index $L+1$, and the storage size L is then increased by the size of the added item, whichis

$$k_{ij} = (l_{ij} + h) \times \texttt{ICL}, \tag{5.2}$$

where h is the size of item *overhead*, defined as

$$h = 1 \div \texttt{ICL} + 1. \tag{5.3}$$

In the K3/F77 storage format, the overhead is the two additional integers that are stored with the submatrix, i.e., the *index* data member and the *pointer to next item* data member. The size of a nodal submatrix is

$$l_{ij} = \begin{cases} d_i(d_i+1)/2 & \text{when } i = j \\ d_i d_j & \text{when } i \neq j \end{cases}, \tag{5.4}$$

where d_i and d_j are the number of degrees of freedom of node i and node j, respectively.

Any item can be located only by using the integer indices (pointers), without any additional calculations. However, after locating the required item in the data structure, the index (pointer) must be converted from the integer array `INT` to the real array `R` to access the submatrix entries, using equation

$$q_{ij} = (p_{ij} - 1) \div \texttt{ICL} + h. \tag{5.5}$$

The size of the K3/F77 storage format data structure is

$$\begin{aligned} L &= |\text{degrees}| + |\text{row pointers}| + |\text{items}| = \\ &= (N + L_0 + N_{nz}h \times \texttt{ICL} + L_{nz} \times \texttt{ICL}) \times |\text{integer}|, \end{aligned} \tag{5.6}$$

[2]Actual pointers are replaced with array indices in FORTRAN 77.

where N_{nz} is the number of nonzero submatrices (number of items), and L_{nz} is the summed length of all nonzero submatrices. Considering two practical cases `ICL` $= 1$ and `ICL` $= 2$, and separating integer and real parts, the size of the data structure is approximately

$$L \doteq (2N + 1 + 2N_{nz}) \times |\text{integer}| + L_{nz} \times |\text{real}|. \tag{5.7}$$

As with the K3 storage format, the data structure is created dynamically and therefore L does not need to be known in advance.

In the out-of-core solution, submatrices of the coefficient matrix need to be written to disk and read back to the memory efficiently. One possibility for an efficient out-of-core disk storage is to use a direct-access file with N_{nz} records, assign each item a number, say s, and write each item to the record number s. The problem is that the item number would have to be stored within the item, augmenting the overhead h substantially due to data alignment. Considering the complicated index conversion in equation (5.5) and the need for careful data alignment, the K3 storage format (and its modification) is impractical for an out-of-core sparse direct solver in FORTRAN 77.

Although the K3/F77 storage format is replaced in the sparse direct solver by the following proposed storage format, its concept of dynamic allocation of items is reused in the minimum degree ordering to perform symbolic assembly of the coefficient matrix and construct the quotient graph (see Subsection 5.1.2).

Proposed block sparse storage format

A simple and efficient matrix storage method for the out-of-core sparse direct solver is proposed in this work. It is partially based on the findings acquired from the K3/F77 storage format, and partially on the compressed row storage format and the dynamic block compressed row storage format, described in Section 2.3.2.

The proposed storage format has the following features:

- Only one-dimensional arrays are employed that are easy to understand and implement in FORTRAN 77 using the PMD memory allocation strategy.
- Integer and real data are stored in separate arrays and thus no index conversions are necessary.
- General finite element meshes with variable number of nodal degrees of freedom as well as the prescribed boundary conditions are taken into account, i.e., nodal submatrices can be of variable size.
- Nodal submatrices are numbered implicitly to allow an efficient out-of-core data manipulation using direct-access file.
- The size of the data structure is the same or less than in the K3/F77 storage format.

The proposed storage format uses four arrays: r_i, c_k, p_k and s_l, where $i = 1, \ldots, N+1$ (N is the number of nodes in the finite element mesh), $k = 1, \ldots, N_{nz}$ (N_{nz} is the number of nonzero submatrices in the coefficient matrix) and $l = 1, \ldots, L_{nz}$ (L_{nz} is the sum of the lengths of all nonzero submatrices). Arrays r and c store row pointers and column indices similar to the compressed row storage format, array s stores the submatrices, and array p stores the pointers to the submatrices in array s.

Aside from the four abovementioned arrays, an important additional array d of size N is necessary to store the number of free nodal degrees of freedom. Since equations corresponding to zero boundary conditions are removed from the coefficient matrix, it is efficient to store only the remaining active equations. The submatrices corresponding to nodes that have some of the degrees fixed have smaller size, and lower d_i results in lower fill-in size in nodal row i. Obviously, completely fixed nodes ($d_i = 0$) are not stored in the proposed storage format.

The size of the proposed storage format data structure is

$$L = |d| + |r| + |c| + |p| + |s| = (2N + 1 + 2N_{nz}) \times |\text{integer}| + L_{nz} \times |\text{real}|, \tag{5.8}$$

that is indeed no larger that the size of K3/F77 storage format given by equation (5.7).

One difficulty is that the nonzero structure of the coefficient matrix must be known in advance in order to create arrays r and c. However, in practice, the nonzero structure of the assembled coefficient matrix can be obtained from the finite element mesh topology, while the nonzero structure of the factorized matrix can be obtained from the ordering (see Subsection 5.1.2).

When the storage data structure is created, arrays r and c are initialized according to the nonzero structure of the matrix, and all elements in array p are initialized to 0 to indicate that the corresponding submatrix has not yet been stored in array s. The initial length of the data in array s is $L_s = 0$.

In the matrix assembly, when a nodal submatrix corresponding to nodes i and j is assembled, its index k is found in c_q, where $q = r_i, \ldots, r_{i+1} - 1$. If $p_k = 0$ the submatrix is stored to entries s_t where $t = L_s + 1, \ldots, L_s + l_{ij}$ and L_s is increased by the length of the submatrix l_{ij} given inequation (5.4). If $p_k > 0$ the submatrix is added to entries s_t where $t = p_k, \ldots, p_k + l_{ij} - 1$.

In the matrix factorization, it is only necessary to handle the condition $p_k = 0$, i.e., the addition of fill-in. In this case the entries s_t where $t = L_s + 1, \ldots, L_s + l_{ij}$ are initialized to 0 and L_s is increased the same way as in the matrix assembly.

If there is enough memory to store all nonzero submatrices, i.e., if array s has length L_{nz}, an in-core solution can be performed without difficulty.Otherwise, when the actual size of array s is smaller than L_{nz}, an out-of-core solution is necessary. In the out-of-core solution, the situation when a submatrix needs to be added to the storage but there is no more space in array s must be implemented efficiently. One possible way (although not too efficient) is to write all submatrices k for which $p_k > 0$ to disk, set $p_k = 0$ for all k, set $L_s = 0$, and then continue with the solution. Of course, the out-of-core factorization algorithm must check pointer array p to ensure all submatrices required for the actual elimination step are present in the storage array s, and load them from the

disk if necessary. This concept can be easily extended to both the assembly algorithm and the (forward and back) substititon algorithm.

In conclusion, the proposed block sparse storage format is very efficient in both the storage overhead (resulting in low matrix storage size) and the algorithmic overhead (resulting in quick accessibility of matrix entries). It allows the sparse direct solver to perform all necessary operations on the coefficient matrix (assembly, factorization and substitution) either in-core or out-of-core, depending on the amount of available memory. Efficient out-of-core data manipulation is provided by using direct-access file.

5.1.2 Ordering method

The ordering method chosen for the sparse direct solver is the approximate minimum degree ordering algorithm (see Subsection 4.2.1), to minimize the fill-in introduced in the sparse factorization, and consequently to reduce the storage requirements and the time needed to obtain the solution. Although established implementations of the approximate minimum degree ordering are available, their use in a commercial sparse direct solver is prohibited,and the possibility for modifications is limited. Therefore, an original version of the minimum degree algorithm is proposed for the use in the sparse direct solver. This version is capable of selective switching of various features (supervariables, approximate degrees, etc.) to allow the efficiency of the algorithm to be analyzed thoroughly.

Proposed minimum degree algorithm

Common implementations of the minimum degree algorithm work with the nonzero structure of the matrix, i.e., directly with the matrix entries. In the application to the sparse direct solver, it is advantageous to perform the ordering on the *block* nonzero structure of the matrix instead. Since the fill-in can occur only in blocks due to the block sparse storage format, the ordering performed on matrix entries is unnecessarily expensive. Exploiting of the block structure reduces the number of graph vertices and edges involved in the minimum degree algorithm considerably. An important consequence is that the ordering can be carried out only using the finite element mesh topology (with only integer arithmetic involved), without assembling the coefficient matrix explicitly. For large finite element problems the ordering on matrix blocks is significantly faster compared to the ordering on matrix entries.

This almost natural improvement is surprisingly not commonly mentioned, one exception being [27]. It is probably because many solvers are standalone, i.e., they are designed to work with any supplied matrix without knowing anything about the origin or the structure of the matrix. Some solvers optionally allow the user to input the information about the matrix structure, or use special algorithms to identify and exploit the block nonzero pattern of the matrix themselves (however, such algorithms are costly, especially for large matrices). The presented sparse direct solver is integrated into the PMD system, and is therefore capable to utilize the properties of the finite element mesh directly, without user intervention.

The established approximate minimum degree ordering algorithm (AMD)[3] by Amestoy, Davis andDuff [1] uses an array to store the quotient graph, which requires *garbage collection* to compress the data occasionally, depending on the size of the array provided by the user. Supervariables are detected using a hash table and placed in a special linked list sorted by their degree to accelerate the search for minimum degree, and the degrees are computed approximately using equation (4.22).

It is important to realize that the AMD algorithm works with the nonzero structure of an explicitly given matrix, hence it can use a simple array to store the quotient graph. Unlike the AMD algorithm, the proposed minimum degree algorithm has to obtain the nonzero (block) structure of the coefficient matrix by analyzing the finite element mesh and assembling the coefficient matrix symbolically, which cannot be done efficiently using an array. Consequently, the proposed minimum degree algorithm uses linked lists to store the quotient graph and thus does not require the garbage collection, but has higher storage requirements compared to the AMD algorithm. However, the storage space required for the ordering is significantly smaller than the storage space required for the subsequent (numerical) assembly of thecoefficient matrix, and therefore higher storage requirements of the ordering does not constitute any difficulty to the sparse direct solver.

The proposed minimum degree algorithm is summarized in Table 5.1. The quotient graph is constructed by a symbolic assembly of the block nonzero structure of the coefficient matrix using the finite element mesh topology. Array d of size N, where N is the number of mesh nodes, is used to store the number of unconstrained nodal degrees of freedom (to account for boundaryconditions, see proposed block sparse storage format in Subsection 5.1.1). The initial quotient graph is defined by sets $\mathcal{V}_i$, where $j \in \mathcal{V}_i$ only if $i \neq j$, $d_i > 0$, $d_j > 0$, and both i and j are nodesof the same mesh element. Fully constrained nodes (i.e., nodes k where $d_k = 0$) do not contribute to the quotient graph, since the corresponding nodal submatrices are not included in the coefficient matrix anyway. The vertices corresponding to fully constrained nodes are removed prior to the symbolic elimination.

The symbolic elimination is performed using the quotient graph constituting of sets $\mathcal{V}$. Arrays $\mathcal{D}$, $\mathcal{N}$, $\mathcal{P}$, $\mathcal{S}$ and $\mathcal{V}$ have size N, however $\mathcal{V}$ and $\mathcal{S}$ represent linked lists that are stored in the quotient graph workspace of size $2N_{nz}$. For a comparison, although inappropriate, the AMD algorithm has a minimum workspace size $N_{nz} + N$ ($1.2N_{nz} + N$ is recommended by the authors). Linked lists $\mathcal{V}$ and $\mathcal{S}$ are implemented in a similar way to the K3/F77 storage format (see Subsection 5.1.1), but are considerably simpler since there is no need to store any real numbers. Only two integers are stored in a linked list item: the pointer to the next item and the vertex number. Operations on linked list, i.e., unions, complements, the detection and construction of supervariables, and of course the degree computations, present computationally expensive parts of the algorithm, which has been substantially optimized at the implementation level to achieve high efficiency.

The proposed minimum degree algorithm incorporates mass elimination, element absorption, supervariable detection, and external degree computation. Moreover, unlike

[3]FORTRAN implementation of AMD version 2.2 is considered.

Algorithm for the kth elimination step:

1. *Minimum degree search.*

   Select supervariable p with minimum degree $\mathcal{D}_p$.

2. *Mass elimination.*

   Add variable p and variables in $\mathcal{S}_p$ to the permutation vector.

3. *Element absorption.*

   Create element p:
   Set $\mathcal{P}_i = p$ for elements $\mathcal{P}_i \in \mathcal{V}_p$.
   Set $\mathcal{V}_p = \left(\bigcup_{\mathcal{P}_i=p} \mathcal{V}_i\right) \setminus \{j\}$ where $\mathcal{P}_j \neq j$.
   Set $k = k + \mathcal{N}_p$, $\mathcal{D}_p = N$, $\mathcal{N}_p = |\mathcal{V}_p|$ and $\mathcal{S}_p = -1$.

   Update variables adjacent to element p:
   Set $\mathcal{V}_i = \mathcal{V}_i \setminus \{j\}$ where $\mathcal{P}_j \neq j$ and $i \in \mathcal{V}_p$.

4. *Supervariable detection.*

   Test all pairs of variables $i \in \mathcal{V}_p$ and $j \in \mathcal{V}_p$ for indistinguishability:
   If $|\mathcal{V}_i| = |\mathcal{V}_j|$ and $\sum(\mathcal{V}_i) = \sum(\mathcal{V}_j)$ compare $\mathcal{V}_i$ and $\mathcal{V}_j$ entry by entry (using $\mathcal{P}$).

   If variables i and j are indistinguishable create supervariable i:
   Set $\mathcal{S}_i = \mathcal{S}_i \cup \mathcal{S}_j$, $\mathcal{V}_j = -1$, $\mathcal{P}_j = i$ and $\mathcal{D}_j = N$.

5. *Degree update.*

   Update $\mathcal{D}_i$ of variables $i \in \mathcal{V}_p$.

Legend:

$\mathcal{D}_i$ Degree of supervariable i. If i is not a supervariable $\mathcal{D}_i = N$. Initially $\mathcal{D}_i = |\mathcal{V}_i|$.

$\mathcal{N}_i$ If i is a supervariable $\mathcal{N}_i = |\mathcal{S}_i| + 1$. If i is an element $\mathcal{N}_i = |\mathcal{V}_i|$. Initially $\mathcal{N}_i = 1$.

$\mathcal{P}_i$ If i is an element absorbed into element j, or i is a non-principal variable of supervariable j, $\mathcal{P}_i = j$. Initially $\mathcal{P}_i = i$.

$\mathcal{S}_i$ Pointer to the set of non-principal variables in supervariable i. If i is an element $\mathcal{S}_i < 0$. Initially $\mathcal{S}_i = 0$.

$\mathcal{V}_i$ Pointer to the set of vertices (variables and elements) adjacent to vertex (variable or element) i. If i is a non-principal variable $\mathcal{V}_i < 0$. Initially $\mathcal{V}_i$ is formed according to the finite element mesh topology.

Table 5.1: Proposed minimum degree algorithm

the AMD algorithm, it is possible to selectively switch off supervariables and external degrees, and select between four types of degree computations (exact degrees and three approximate degrees, see Subsection 4.2.1). Multiple elimination and incomplete degree update are not efficient when used together with approximate degrees and therefore are not included in the proposed minimum degree algorithm (they are not included in the AMD algorithm for the same reasons). Aggressive absorption is not used in the presented sparse direct solver, since its benefits for the proposed minimum degree algorithm are debatable due to the following assessment.

Assessment of the proposed minimum degree algorithm

Numerical tests were carried out using the sparse direct solver (see Section 5.2) to verify the proposed minimum degree algorithm and to assess its performance. The individual test configurations are denoted by a letter and a number that represent the selected algorithm options and the type of degree computation, respectively, for a total of 12 configurations. The test configurations are:

A. True degrees are used (no supervariables).

B. Supervariables are used with true degrees.

C. Supervariables are used with external degrees.

1. Degrees are computed exactly.

2. Degrees are computed approximately according to Amestoy, Davis and Duff.

3. Degrees are computed approximately according to Ashcraft, Eisenstat and Lucas.

4. Degrees are computed approximately according to Gilbert, Moler and Schreiber.

To summarize the terminology from Subsection 4.2.1,

- *true degree* is the number of variables adjacent to a variable,
- *external degree* is the number of variables adjacent to a supervariable without counting the variables in the supervariable,
- *exact degree* is either a true degree or an external degree computed exactly, and
- *approximate degree* is either a true degree or an external degree computed approximately.

The ordering time for 10 selected large finite element problems (approximately from 50,000 to 2,000,000 equations) is listed in Table 5.2 and plotted in Figure 5.2. The plot is divided into three parts for clarity. Values corresponding to the same problem are connected by a solid line that has no real meaning but to show the trend. Configuration A1 corresponds to the original quotient graph-based minimum degree algorithm and serves

as the reference configuration. The results of all configurations are normalized to the reference configuration to allow the assessment.

It can be seen that with supervariables the minimum degree algorithm is about 60–95% faster than without supervariables, and also the difference in ordering time between exact and approximate degrees is negligible. Without supervariables, the difference in ordering time between exact and approximate degrees is distinct and well corresponds to the theory. Configuration A1 is of course the slowest. Configuration A4 is the fastest, but the gain may be as low as only 10%. Configuration A3 is rather slow (since exact degrees are involved) and the gain is only 10–40%. Finally, configuration A2 is generally substantially faster than A1 or A3 and the gain can be as high as 60%. The use of external degrees does not have any significant effect on the ordering time. To summarize, the proposed minimum degree algorithm is most efficient when the supervariables are used, i.e., in configurations B or C. To determine the effect of external degrees and approximate degrees, the ordering must be analyzed with regard to the fill-in.

The amount of the fill-in obtained by applying the ordering is listed in Table 5.3 and plotted in Figure 5.3 in the same manner as the ordering time results. It can be seen that without supervariables, the fill-in in configurations A2 and A3 is about ±5% while in configuration A4 it grows unreasonably. This is not surprising since approximate degree 4 neglects the duplicities in edge sets and therefore results in a very high bound of the degree. In configurations B4 and C4 the bound can be reasonable in some cases but is still generally the highest. With supervariables but without external degrees, the fill-in is only slightly lower (about 3%) but can be also surprisingly higher, up to 10% in some cases. Configurations B1-B3 result in the same amount of the fill-in in most cases. With supervariables and external degrees, the fill-in is much lower, up to 20%. The differences in fill-in between configurations C1-C3 are generally small.

To summarize the assessment, the proposed minimum degree algorithm is most efficient when the supervariables and external degrees are used, i.e., in configuration C. The minimum degree algorithm is a heuristic, hence it is difficult to select the 'best' configuration since different configurations can yield the 'best' ordering for different matrices, as can be seen from the results. However, from the implementation point of view, configurations C2 and C3 require additional computations to obtain the approximate degrees, and therefore the configuration C1, which does not require any additional computations, can be considered the fastest and is recommended for the use with the sparse direct solver in practice.

In conclusion, the proposed minimum degree algorithm is a reasonable option to the established AMD algorithm for the use in the sparse direct solver. The comparison of both algorithms cannot be done directly since each has a slightly different scope. While AMD is a general-purpose algorithm, the proposed minimum degree algorithm is closely connected to the finite element method. Some form of direct comparison would require non-trivial modifications of both algorithms which is out of scope of this work, but will be considered for future research.

Problem name	Number of equations	Number of nodes	Configuration											
			A1	A2	A3	A4	B1	B2	B3	B4	C1	C2	C3	C4
M386_01	65,304	21,768	52	20	39	3	1	1	1	1	1	1	1	1
P6_QUAD	74,742	24,914	152	57	106	6	1	1	1	1	1	1	1	1
ADYN	129,636	43,212	584	213	409	24	3	3	3	3	3	3	3	2
K1	136,354	31,338	14	7	9	3	1	1	1	1	1	1	1	1
ADS	359,504	82,760	18	17	16	16	7	7	7	7	7	7	7	7
K3	362,262	83,142	84	35	54	18	7	8	7	7	7	7	7	7
A98ABT001_1	365,043	121,681	64	44	59	46	11	11	11	11	11	11	11	11
DOCHL	387,829	89,797	32	25	22	19	9	9	9	9	9	9	9	9
G1R90L	931,728	310,576	804	434	676	524	71	70	69	69	70	70	69	69
BUBEN	1,739,211	579,737	3,041	1,675	2,612	1,267	173	171	170	169	171	170	168	170

Table 5.2: Proposed minimum degree algorithm results, ordering time (in seconds)

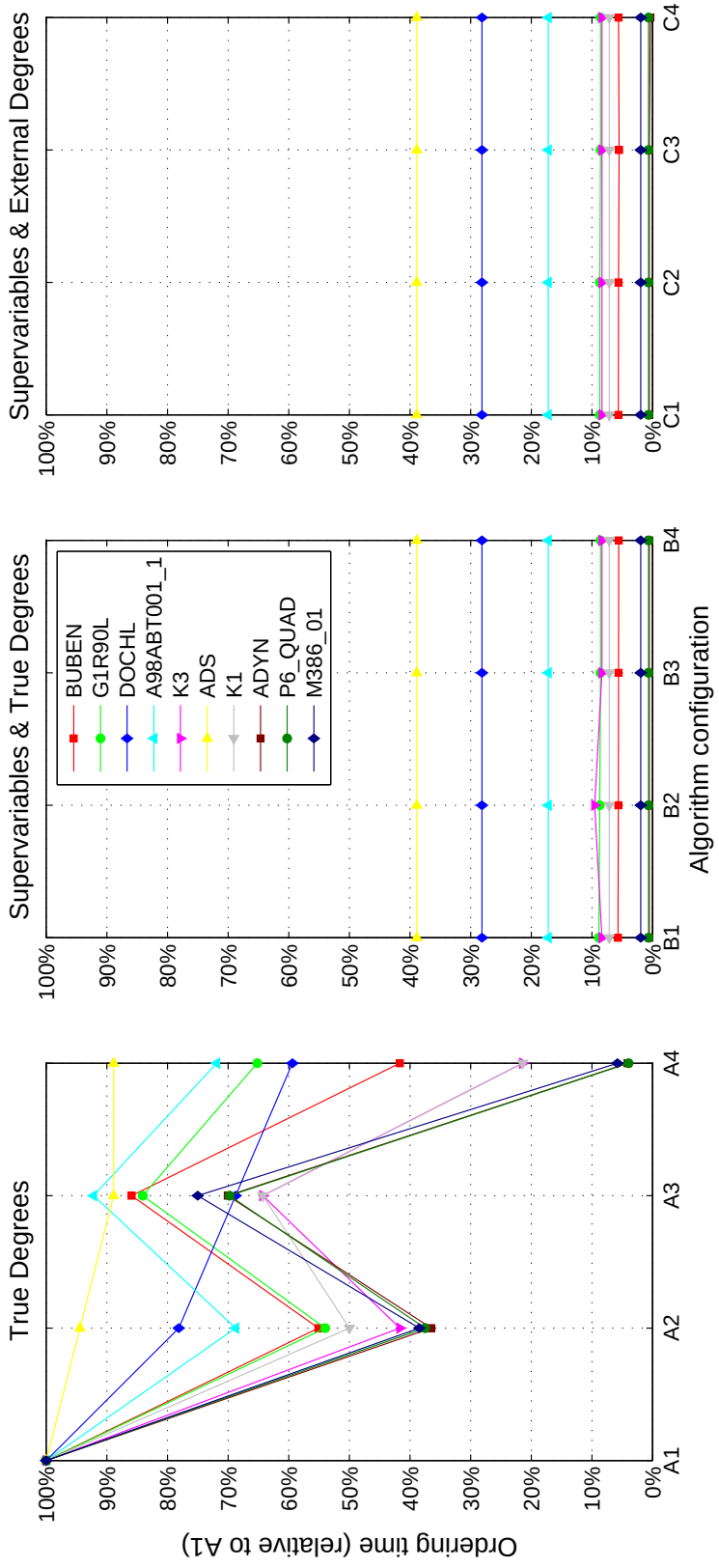


Figure 5.2: Proposed minimum degree algorithm results, ordering time

Problem name	Configuration			
	A1	A2	A3	A4
M386_01	5,982,349	5,982,349	5,982,349	14,650,301
P6_QUAD	10,323,907	10,323,907	10,323,907	38,290,062
ADYN	25,945,540	25,945,540	25,945,540	127,845,208
K1	2,398,524	2,331,624	2,488,882	4,584,039
ADS	2,474,447	2,502,239	2,617,352	3,740,716
K3	7,701,439	7,251,902	7,405,601	21,492,594
A98ABT001_1	10,399,405	10,399,405	10,399,405	92,028,223
DOCHL	3,748,508	3,748,508	3,949,206	7,775,260
G1R90L	65,763,296	65,763,296	65,763,296	865,617,009
BUBEN	152,424,435	152,424,435	152,424,435	1,327,233,745
Problem name	Configuration			
	B1	B2	B3	B4
M386_01	5,955,917	5,955,917	5,955,917	5,360,717
P6_QUAD	10,242,470	10,242,470	10,242,470	12,426,400
ADYN	26,672,325	26,672,325	26,672,325	35,793,018
K1	2,392,517	2,457,648	2,552,796	2,895,375
ADS	2,474,447	2,508,712	2,578,933	2,586,777
K3	7,619,303	7,656,445	7,622,608	8,761,111
A98ABT001_1	10,429,947	10,429,947	10,429,947	11,231,726
DOCHL	3,858,251	3,858,251	4,044,940	3,888,210
G1R90L	65,763,296	65,763,296	65,763,296	74,504,419
BUBEN	146,983,198	146,983,198	146,983,198	215,181,680
Problem name	Configuration			
	C1	C2	C3	C4
M386_01	5,433,559	5,433,559	5,433,559	5,222,393
P6_QUAD	8,475,559	8,475,559	8,475,559	9,289,914
ADYN	20,627,159	20,627,159	20,627,159	25,823,628
K1	2,147,350	2,121,899	2,213,134	2,205,559
ADS	2,289,191	2,340,394	2,346,700	2,410,171
K3	6,460,907	6,397,210	6,455,584	8,166,626
A98ABT001_1	9,366,981	9,366,981	9,366,981	9,597,389
DOCHL	3,412,780	3,412,780	3,636,798	3,686,526
G1R90L	56,651,950	56,651,950	56,651,950	59,229,362
BUBEN	129,267,717	129,267,717	129,267,717	165,069,878

Table 5.3: Proposed minimum degree algorithm results, ordering fill-in (in blocks)

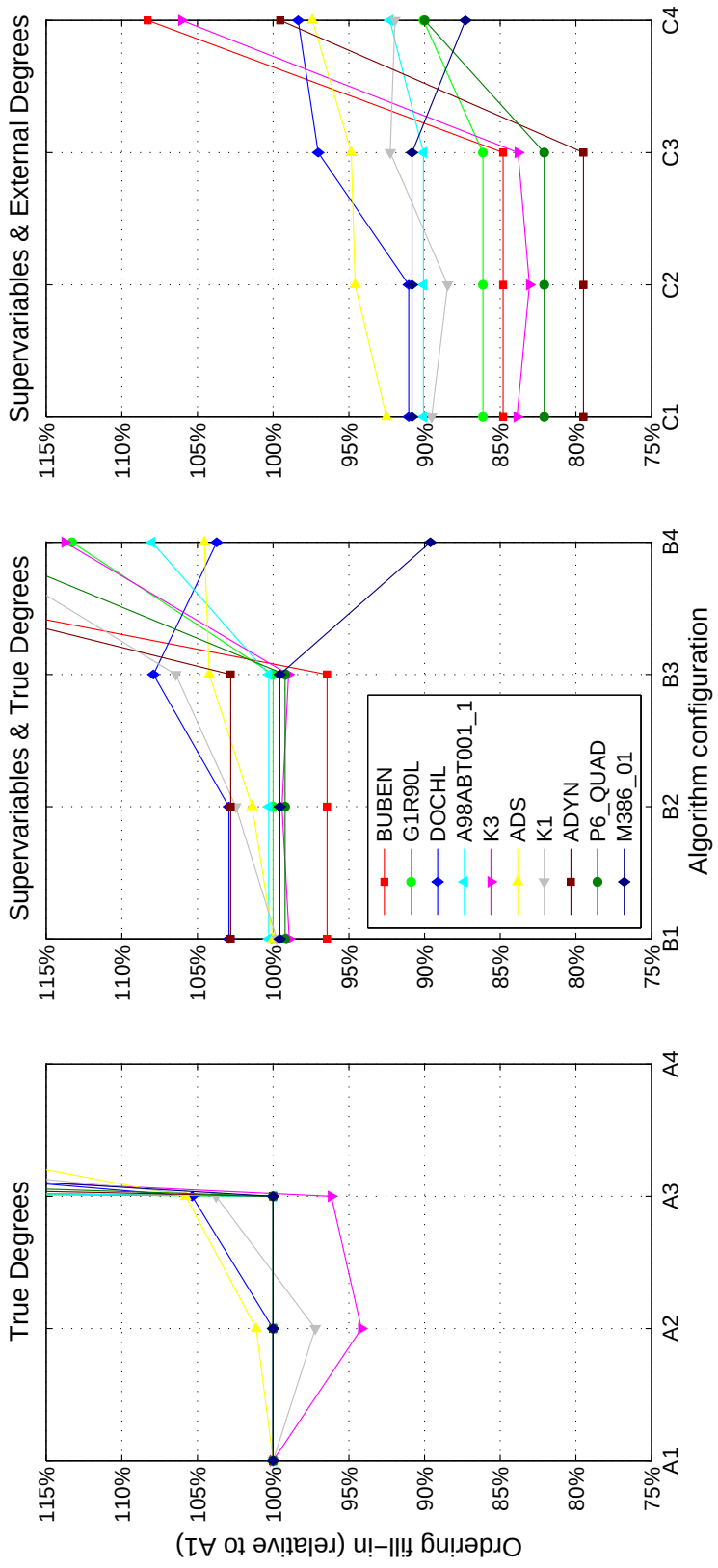


Figure 5.3: Proposed minimum degree algorithm results, ordering fill-in

5.1.3 Solution method

The solution method chosen for the sparse direct solver is the block $\mathbf{LDL}^{\mathrm{T}}$ factorization (see Subsection 4.3.1). In Subsection 5.1.1 the practical impossibility to solve large finite element problems in-core have been explained. Therefore, an out-of-core symmetric block sparse factorization algorithm is proposed for the use in the sparse direct solver.

Proposed out-of-core symmetric block sparse factorization algorithm

There are four possibilities to perform the $\mathbf{LDL}^{\mathrm{T}}$ factorization. It can be carried out either by a left-looking algorithm (using factor $\mathbf{L}$) or a right-looking algorithm (using factor $\mathbf{DL}^{\mathrm{T}}$), and either by rows or by columns. The number of arithmetic operations is equivalent in all cases, therefore the most suitable algorithm can be selected based on other properties. Obviously, for an out-of-core factorization the most important question is which part of the coefficient matrix must be present in a factorization step if the whole coefficient matrix cannot be stored in-core. It is of course possible to design the out-of-core algorithm so that only two or three submatrices (blocks) that are involved in the actually summed term are present in the memory at any given time, which is very efficient in terms of the storage space but extremely inefficient in terms of the out-of-core data manipulation, since many submatrices must be read from and/or written to disk repetitively and disk access is slow compared to memory access. A reasonable compromise is when all submatrices involved in the computation of the actual submatrix are present in the memory, since additional reads and writes are still required but reduced considerably. The most suitable out-of-core case, in which the efficiency of the factorization is not affected, is when the whole active part of the coefficient matrix (active submatrix) is present in the memory. In any case, the out-of-core factorization algorithm checks the presence of blocks required for the actual factorization step in the matrix storage and reads them from the disk if not present. In the case the matrix storage is full all blocks are written to disk and discarded before reading the required blocks.

The partially reduced coefficient matrix after the kth factorization step in the right-looking algorithm has the form

$$\mathbf{A}_{k+1}|_{\text{row}}^{\text{right}} = \mathbf{A}_{k+1}|_{\text{column}}^{\text{right}} = \left(\begin{array}{ccc|ccc} d_{11} & \cdots & d_{11}l_{k1} & d_{11}l_{k+1,1} & \cdots & d_{11}l_{n1} \\ & \ddots & \vdots & \vdots & & \vdots \\ \hline & & d_{kk} & d_{kk}l_{k+1,k} & \cdots & d_{kk}l_{nk} \\ & & & d_{k+1,k+1} & \cdots & d_{k+1,k+1}l_{n,k+1} \\ & & & a_{k+2,k+2}^{(k+1)} & \cdots & a_{k+2,n}^{(k+1)} \\ & & & \vdots & & \vdots \\ 0 & & & a_{n,k+1}^{(k+1)} & \cdots & a_{nn}^{(k+1)} \end{array}\right). \tag{5.9}$$

Matrix (5.9) is identical for both the row-wise approach (common Gaussian elimination) and the column-wise approach (uncommon, impractical). Rows $1, \ldots, k$ already contain factor entries d_{ii} and l_{ji} that are however not used in any of the subsequent factorization

steps. Row $k+1$ also contains factor entries but is required in the $(k+1)$th factorization step. Rows $k+2,\ldots,n$ are only partially factorized, and row i is required in all factorization steps $k \leq i$. Therefore, in practice, row k can be removed from memory and written to disk after completed kth factorization step. The size of the active submatrix in the kth factorization step, delimited by lines in (5.9), is

$$L_k|_{\text{row}}^{\text{right}} = L_k|_{\text{column}}^{\text{right}} = \frac{(n-k+1)(n-k+2)}{2}. \tag{5.10}$$

Storage requirements are highest in step $k = 1$, where 100% of the matrix entries is required.

The partially reduced coefficient matrix after the kth factorization step in the column-wise left-looking algorithm (i.e., skyline reduction or active column solution) has the form

$$\mathbf{A}_{k+1}|_{\text{column}}^{\text{left}} = \left(\begin{array}{cccc|ccc} d_{11} & l_{21} & \cdots & l_{k1} & a_{1,k+1} & \cdots & a_{1n} \\ & d_{22} & \cdots & l_{k2} & a_{2,k+1} & \cdots & a_{2n} \\ & & \ddots & \vdots & \vdots & & \vdots \\ & & & d_{kk} & a_{k,k+1} & \cdots & a_{kn} \\ \hline & & & & a_{k+1,k+1} & \cdots & a_{k+1,n} \\ & & & & \vdots & & \vdots \\ 0 & & & & a_{n,k+1} & \cdots & a_{nn} \end{array}\right). \tag{5.11}$$

Matrix (5.11) is defined by equations (4.32) and (4.33). Columns $1,\ldots,k$ already contain factor entries d_{ii} and l_{ji} that are however used in all subsequent factorization steps. Columns $k+1,\ldots,n$ contain the original matrix entries a_{ij}, and column j is not required until jth factorization step. Therefore, in practice, no column may be removed from memory and written to disk unless it is read back later, which reduces efficiency. The size of the active submatrix in the kth factorization step, delimited by lines in (5.11), is

$$L_k|_{\text{column}}^{\text{left}} = \frac{k(k+1)}{2}. \tag{5.12}$$

Storage requirements are highest in step $k = n$, where 100% of the matrix entries is required.

Finally, the partially reduced coefficient matrix after the kth factorization step in the row-wise left-looking algorithm has the form

$$\mathbf{A}_{k+1}|_{\text{row}}^{\text{left}} = \left(\begin{array}{cccc|ccc} \boxed{d_{11}} & l_{21} & \cdots & l_{k1} & l_{k+1,1} & \cdots & l_{n1} \\ & \boxed{d_{22}} & \cdots & l_{k2} & l_{k+2,1} & \cdots & l_{n2} \\ & & \ddots & \vdots & \vdots & & \vdots \\ & & & d_{kk} & l_{k+1,k} & \cdots & l_{nk} \\ \hline & & & & a_{k+1,k+1} & \cdots & a_{k+1,n} \\ & & & & \vdots & & \vdots \\ 0 & & & & a_{n,k+1} & \cdots & a_{nn} \end{array}\right). \tag{5.13}$$

Matrix (5.13) is also defined by equations (4.32) and (4.33). Columns $1, \ldots, k$ already contain factor entries d_{ii} and l_{ji}, however entries l_{ji} are not used in any of the subsequent factorization steps. Rows $k+1, \ldots, n$ contain the original matrix entries a_{ij}, and row i is not required until ith factorization step. The submatrix with entry indices $i = 1, \ldots, k$ and $j = k+1, \ldots, n$ also contains factor entries but columns $j = p, \ldots, n$ are required until pth factorization step is completed.

Therefore, in practice, column k can be removed from memory (except for the diagonal entry) and written to disk after completed kth factorization step. Moreover, rows $k+1, \ldots, n$ need not to be present in memory until they are needed. The size of the active submatrix in the kth factorization step (including diagonal entries), delimited by lines in (5.13), is

$$L_k|_{\text{row}}^{\text{left}} = k(n-k+2) - 1. \tag{5.14}$$

Storage requirements are highest in step $k = n/2$, where only about 50% of the matrix entries is required.

In conclusion, the most efficient algorithm for an out-of-core factorization is the left-looking algorithm performed by rows, which is surprisingly not commonly mentioned. Presented conclusions can be easily extended to a block sparse coefficient matrix, in which case the storage requirements will be substantially lower due tosparsity. The minimum required storage size for factorization (i.e., the maximum size of the active submatrix) can be obtained from the symbolic elimination (using minimum degree ordering).

5.2 Solver implementation

Sparse direct solver presented in this work is based on methods and algorithms described in Chapter 4 and, in particular, Section 5.1. The sparse direct solver (program FESD) is implemented according to the requirements of the PMD system, described in Section 4.4, and is designed to seamlessly replace the existing PMD's frontal solver (program FEFS).

In short, the sparse direct solver reads the mesh topology and element matrices (preprocessed by other PMD programs), solves the associated linear equation system, and writes displacements and reaction forces (postprocessed by other PMD programs to calculate stresses, strains, etc.). Sparse direct solver's input file, output file and intermediate data files are documented in Appendix B. The intermediate data files used by the sparse direct solver are different from the frontal solver, therefore, both solvers can coexist in the same directory (provided the user is aware that files *name*.I4, *name*.O4 and `IDSOL` are shared between the two solvers). Consequently, the coefficient matrix factorized by the sparse direct solver cannot be used by the frontal solver and vice versa. To facilitate program restarts and allow the computation of additional load cases (right-hand sides), a new parameter `KFES` was added to the PMD common to store sparse direct solver's status.

The presented sparse direct solver is divided logically into four distinct phases (ordering, assembly, factorization and solution) which are described in the following sections. The source code of the sparse direct solver is documented in Appendix C.

5.2.1 Ordering phase

The ordering phase is perhaps the most elaborate part of the program. It comprises of the following steps:

1. `INET`, `NNET`, `NNDF` and `IFIXV` are read from file `IDP`.

2. `MBD`, `NPIV`, `NVAR`, `LEMTX` and `LSMTX` are calculated using `INET`, `NNET`, `NNDF` and `IFIXV`.

3. A quotient graph is created using `INET` and `NNET`.
   Simultaneously, `NBLK1` and `SBUF1` are calculated using `MBD`.

4. `IPNOD` is calculated using the minimum degree ordering on the quotient graph. Optionally, the (reverse) Cuthill-McKee preordering is applied.
   Simultaneously, `MRP` and `MCI` are calculated and written to file `IDEQI`.
   Simultaneously, `NBLK2` and `SBUF2` are calculated using `MBD`.

5. `IPSOL` is calculated using `NNDF`, `MBD` and `IPNOD`.

6. `NPIV`, `NVAR`, `NBLK1`, `NBLK2`, `SBUF1`, `SBUF2`, `LSMTX`, `LROW`, `LEMTX`, `KMET`, `IPNOD`, `IPSOL` and `MBD` are written to file `IDEQC` to facilitate solver restarts.

7. `MRP` and `MCI` are read from file `IDEQI`, permuted using `IPNOD`, and written back to file `IDEQI`.

Note that the column indices written to file `IDEQI` in step 4 are not permuted since the full permutation vector `IPNOD` is not known until the ordering is complete. Therefore, the indices are permuted and file `IDEQI` is rewritten in step 7 prior to the assembly phase.

The ordering is performed using the minimum degree algorithm proposed in Subsection 5.1.2, optionally coupled with the (reverse) Cuthill-McKee preordering. The output of the ordering phase is the nonzero block structure of the coefficient matrix in the factorization. Consequently, appropriate storage structures can be prepared for both the assembly and the factorization phases. The implemented minimum degree ordering algorithm is highly optimized, particularly regarding the operations on edge sets (adding and removing elements, joining sets, etc.).

5.2.2 Assembly phase

The assembly phase is relatively simple and comprises of the following steps:

1. Element stiffness matrix is read from file `IDELM` and partitioned into nodal submatrices using `INET`, `NNET`, `NNDF` and `IPSOL`.

2. Nodal submatrices are assembled into the coefficient matrix using `IPNOD`, `MRP`, `MCI`, `MBP` and `BUF`.

3. Steps 1 and 2 are repeated for all elements `NELEM`.

The coefficient matrix is assembled using the block sparse storage format proposed in Subsection 5.1.1, where $N \equiv$ NPIV, $N_{nz} \equiv$ NBLK2, $L_{nz} \equiv$ SBUF1, $d \equiv$ MBP, $r \equiv$ MRP, $c \equiv$ MCI, $p \equiv$ MBP and $s \equiv$ BUF. Entries corresponding to constrained degrees of freedom (zero boundary conditions) are excluded from nodal submatrices (corresponding submatrices are therefore smaller). Nodal submatrices corresponding to fully constrained nodes are not included in the coefficient matrix, since they have zero size. Thus, the output of the assembly phase is the assembled coefficient matrix excluding the boundary conditions.

5.2.3 Factorization phase

The factorization phase comprises of the following steps:

1. The coefficient matrix is factorized using MRP, MCI, MBP, MBD and BUF.

2. BUF is written to file IDEQR.

The factorization is performed using the block sparse factorization algorithm proposed in Subsection 5.1.3. The output of the factorization phase is the factorized coefficient matrix. The implemented block sparse factorization algorithm is highly optimized, particularly regarding the arithmetic operations performed on blocks.

After this phase is successfully finished it is possible to restart the solver with additional right-hand sides using the factorized coefficient matrix (without the need to factorize it again).

5.2.4 Solution phase

The solution phase comprises of the following steps:

1. RHS is read from file IDRHS and permuted using IPSOL.

2. Displacements are solved for all load cases NASV by forward and back substitution using RHS, MRP, MCI, MBP, MBD and BUF.

3. Solved displacements are permuted back using IPSOL.

4. Reaction forces are solved for all load cases NASV by assembling the appropriate equations directly from file IDELM using INET, NNET, NNDF, IPSOL and the displacements.

5. The displacements and the reaction forces are written to file IDSOL.

The output of the solution phase are displacement vectors and reaction force vectors for all load cases (right-hand sides).

Name	Type	Operating system	Fortran compiler
`hastrman`	Server	HP Tru64 UNIX 5.1B	HP Fortran 5.5A-3548
`rusalka`	Server	HP Tru64 UNIX 5.1B	HP Fortran 5.5A-3548
`ds9`	PC	Windows XP 5.1.2600	Intel Fortran 12.0-1291
`babylon5`	PC	Windows 7 6.1.7601	Intel Fortran 12.0-1291

Name	Processor(s)	Memory†	Disk†
`hastrman`	4x DEC Alpha EV7	4,096	81,920
`rusalka`	4x DEC Alpha EV6	1,024	9,216
`ds9`	Intel Core 2 Duo T8100	2,048	277,118
`babylon5`	Intel Core i7 950	8,192	895,345

†*Storage available for program use, in megabytes (MB)*

Table 5.4: Computers used in FEM analysis

Memory consideration

If there is enough memory available the coefficient matrix is assembled, factorized and solved in-core, otherwise it is processed out-of-core. Intermediate out-of-core data are stored within file `IDEQR`, a direct-access file with `NBLK2` records of length `LSMTX`. Thus, reads and writes are performed efficiently (block i is stored in record i).

The factorization phase requires substantially more memory than the assembly phase, because the factorized coefficient matrix is substantially larger than the assembled coefficient matrix due to the fill-in. The forward and back substitutions need even slightly more memory than the factorization because of the storage of right-hand side vectors and displacement vectors. The substitutions are performed for all right-hand sides together to minimize out-of-core data manipulation, therefore it is possible, if needed, to increase the memory available for the solution of displacements by reducing the number of right-hand sides solved in one run. Finally, the solution of reaction forces requires only little memory compared to the solution of displacements since it does not require the coefficient matrix.

5.3 FEM applications

The sparse direct solver had been tested on various example problems before it was used in real engineering applications of the finite element method. All computations were complemented using the existing frontal solver to allow a comparison of performace and accuracy for both solvers. Numerical results obtained, which are presented later in this section, confirm that the sparse direct solver gives accurate results while being substantially faster in most cases. Therefore, the sparse direct solver can fully substitute the existing frontal solver of the PMD system.

The FEM analysis was performed on four different computers listed in Table 5.4. Two of the computers were UNIX-based servers while the others were Windows-based PCs, all located in the Institute of Thermomechanics ASCR. Use of the UNIX-based computers

Problem name	Problem size	Frontal solver		Sparse direct solver	
		Memory	Disk	Memory	Disk
BEAM6S1	46	803,488	800,420	5,920	4,956
BEAM4S1	54	803,168	800,484	5,688	5,140
TUBE7TS1	106	804,848	801,096	7,448	5,236
BEAM56S1	168	833,840	801,436	67,152	55,968
BEAM61S1	95	821,808	800,832	19,256	24,976
BEAM53S1	30	816,736	800,300	2,472	2,328
BEAM71S1	184	895,248	801,564	71,984	137,376

Storage sizes are in bytes (B)

Table 5.5: FEM analysis results, storage size, example problems

was further limited by the administrator, hence the low available memory and disk storage. The Windows-based computers had no limitations except the 2 GB memory limit in the case of 32-bit Windows. Collectively, these various computer configurations simulated practical environments where the PMD system can be used.

On all four computers, a FORTRAN 77-compatible compiler was used to compile the PMD source code (including the frontal solver and the sparse direct solver) with maximum possible optimizations. Only the 32-bit version of PMD was tested, because it is used almost exclusively, and the 64-bit version of PMD is currently not up-to-date.

The sparse direct solver is capable of utilizing up to 8 GB of memory (depending on the compiler and operating system used), given the 32-bit integer size (4 bytes) and the maximum array index ($2^{31} - 1$). More memory could be utilized by switching to 64-bit integers, but this would require considerable changes to the source code, which would be unfortunately backwards incompatible with the 32-bit version. However, as already mentioned in Subsection 5.1.1, the sparse direct solver would benefit from the 64-bit memory space since it could perform all matrix operations in-core regardless of physical memory size.

Finally, it is important to note that there would be no benefits from the 64-bit version of the frontal solver.

With regard to the conclusions of the minimum degree algorithm assessment in Subsection 5.1.2, the sparse direct solver has been configured to use supervariables with exact external degrees (i.e., `KMET` = 1 in the input file *name*.I4) for all solved FEM problems presented in this section.

5.3.1 Example problems

All finite element problems[4] from the PMD Example Manual [31] were solved in order to verify the correct function of the sparse direct solver. The problems were not limited only

[4]Heat transfer problems were not included since they use a separate solver in PMD.

Problem name	Problem size	Frontal solver		Sparse direct solver	
		Memory	Disk	Memory	Disk
BEAM56D1	168	833,872	801,476	62,912	47,344
BEAM61D1	95	821,808	800,840	18,784	22,292
BEAM53D1	30	816,736	800,300	2,440	2,328
BEAM71D1	184	895,280	801,604	67,736	114,416
BEAM56D2	168	833,872	801,476	62,912	47,344
BEAM53D2	30	816,736	800,300	2,440	2,328
BEAM53D3	30	816,736	800,300	2,440	2,328
BEAM53D4	30	816,736	800,300	2,440	2,328
BEAM6P1	46	803,664	800,588	2,972	356
BEAM56P1	168	834,320	801,924	25,664	8,944
BEAM6P2	46	803,488	800,420	5,920	4,956
BEAM56P2	168	833,840	801,436	67,152	55,968
BEAM6P3	46	803,488	800,420	6,168	4,956
BEAM56P3	168	833,840	801,436	67,152	55,968
BEAM6P4	46	803,488	800,420	6,168	4,956
BEAM56P4	168	833,840	801,436	67,152	55,968
BEAM6P5	46	803,488	800,420	6,168	4,956
BEAM56P5	168	833,840	801,436	67,152	55,968
BEAM6P6	46	803,488	800,420	6,168	4,956
BEAM56P6	168	833,840	801,436	67,152	55,968
BEAM6P7	46	803,504	800,436	5,632	4,652
BEAM56P7	168	833,888	801,492	61,000	54,432
BEAM6P8	46	803,488	800,420	6,168	4,956
BEAM56P8	168	833,840	801,436	67,152	55,968
TUBE7P1	106	804,848	801,096	8,296	5,236
TUBE56P1	384	838,448	803,832	67,912	68,112
BEAM6C1	46	803,488	800,420	5,920	4,956
BEAM6C2	46	803,504	800,436	5,392	4,652
BEAM56C1	168	833,840	801,436	67,152	55,968
BEAM56C2	168	833,840	801,436	67,152	55,968
BEAM56C3	168	833,840	801,436	67,152	55,968
TUBE7C1	106	804,848	801,096	8,296	5,236
TUBE56C1	384	838,448	803,832	67,912	68,112
BEAM6G1	46	803,488	800,420	5,920	4,956
BEAM4G1	54	803,168	800,484	6,120	5,140
BEAM56G1	168	833,808	801,412	69,408	55,968
BEAM61G1	95	821,808	800,840	18,672	24,976
BEAM56G2	168	833,808	801,412	69,408	55,968
CUBE55K1	126	833,168	801,204	32,888	31,732
PRESS7K1	112	804,816	801,020	12,408	12,880

Table 5.5: (continued from page 86)

Problem name	Problem size	Frontal solver		Sparse direct solver	
		Memory	Disk	Memory	Disk
P6_LIN	19,656	10	217	54	56
P6_QUAD	74,742	74	2,359	612	616
K1	136,354	15	1,601	284	414
ADS	359,504	15	2,840	316	449
K3	362,262	49	6,972	852	1,248
A98ABT001_1	365,043	74	8,361	720	683
DOCHL	387,829	42	4,601	450	668
C5	393,774	48	6,054	1,245	1,181
C3	559,125	130	13,347	2,391	2,270
10	752,778	227	23,782	1,862	2,722
COUV9	1,129,747	242	40,308	1,779	2,646
BUBEN	1,739,211	123	44,695	9,887	9,387
SHVO	1,909,577	141	47,866	4,307	6,282
C2	1,973,550	679	113,919	16,635	15,797
SH	3,022,848	549	198,772	8,673	12,705

Storage sizes are in megabytes (MB)

Table 5.6: FEM analysis results, storage size, engineering problems

to elastostatic analysis, since a linear solver is required in the solution of all types of finite element problems. The example problems are listed in more detail in Appendix A.1.

The results are listed in Table 5.5. Solution times (and ordering times for the sparse direct solver) are not listed since they are less than 1 second for all problems. The example problems are quite small, but it can be seen that the sparse direct solver has (surprisingly) lower memory storage requirements and (unsurprisingly) lower disk storage requirements for all problems compared to the frontal solver due to the efficient storage scheme.

The solution vectors of both solvers were compared and it has been verified that the implemented sparse direct solver gives accurate results and correctly solves all types of finite element problems.

5.3.2 Engineering problems

To conclude the numerical analysis, the sparse direct solver has been used to solve several finite element problems from various real-world applications to assess its performace on large problems (selected engineering problems are listed in Appendix A.2). The numerical analysis was performed on computers described in the beginning of this section.

Storage requirements of the sparse direct solver and frontal solver are listed in Table 5.6. It can be seen that the memory requirements of the sparse direct solver are quite high compared to the frontal solver, since, as already mentioned several times, the sparse direct solver stores the whole coefficient matrix in memory. When the memory storage

Problem name	Problem size	Frontal solver			
		`hastrman`	`rusalka`	`ds9`	`babylon5`
P6_LIN	19,656	00:01:27	00:06:00	00:01:15	00:00:19
P6_QUAD	74,742	00:47:55	03:20:50	00:40:28	00:12:55
K1	136,354	00:15:27	00:47:04	00:10:05	00:02:45
ADS	359,504	00:09:15	00:19:51	00:08:02	00:01:57
K3	362,262	01:20:06	05:38:19	01:13:23	00:21:39
A98ABT001_1	365,043	02:10:53	09:19:15	01:58:30	00:36:59
DOCHL	387,829	00:52:46	02:29:43	00:34:34	00:09:40
C5	393,774	00:53:01	03:14:18	00:46:12	00:12:57
C3	559,125	03:35:49	×	03:10:00	01:00:32
10	752,778	12:51:31	×	08:43:43	02:46:46
COUV9	1,129,747	19:16:08	×	14:01:37	04:29:09
BUBEN	1,739,211	14:25:31	×	11:05:21	03:33:40
SHVO	1,909,577	18:14:08	×	12:41:30	04:07:03
C2	1,973,550	×	×	×	22:22:54
SH	3,022,848	×	×	×	17:09:56
Problem name	Problem size	Sparse direct solver			
		`hastrman`	`rusalka`	`ds9`	`babylon5`
P6_LIN	19,656	00:00:16	00:00:30	00:00:11	00:00:03
P6_QUAD	74,742	00:10:41	00:24:01	00:03:23	00:01:42
K1	136,354	00:03:01	00:05:52	00:01:13	00:00:32
ADS	359,504	00:02:51	00:05:03	00:01:31	00:00:21
K3	362,262	00:15:31	00:29:36	00:05:21	00:02:23
A98ABT001_1	365,043	00:07:59	00:17:48	00:03:47	00:01:17
DOCHL	387,829	00:05:12	00:09:18	00:01:53	00:00:43
C5	393,774	00:19:39	00:43:28†	00:06:47	00:02:42
C3	559,125	00:45:51	01:51:05†	01:36:12†	00:06:37
10	752,778	00:54:06	01:50:58†	00:11:54	00:05:50
COUV9	1,129,747	01:10:07	02:28:41†	00:16:40	00:07:04
BUBEN	1,739,211	06:12:40†	×	05:29:27†	01:59:37†
SHVO	1,909,577	03:09:20†	06:35:12†	04:18:55†	00:22:22
C2	1,973,550	×	×	×	07:11:10†
SH	3,022,848	12:55:01†	×	14:06:15†	04:59:21†

× *Solution not performed due to insufficient storage capacity*
†*Solution performed out-of-core*

Table 5.7: FEM analysis results, solution time, engineering problems

Problem name	Problem size	Sparse direct solver			
		`hastrman`	`rusalka`	`ds9`	`babylon5`
P6_LIN	19,656	00:00:00	00:00:00	00:00:00	00:00:00
P6_QUAD	74,742	00:00:01	00:00:04	00:00:01	00:00:00
K1	136,354	00:00:04	00:00:09	00:00:00	00:00:00
ADS	359,504	00:00:30	00:01:08	00:00:03	00:00:01
K3	362,262	00:00:31	00:01:09	00:00:03	00:00:01
A98ABT001_1	365,043	00:00:45	00:01:41	00:00:05	00:00:02
DOCHL	387,829	00:00:41	00:01:32	00:00:04	00:00:02
C5	393,774	00:01:22	00:03:06	00:00:10	00:00:05
C3	559,125	00:02:39	00:05:56	00:00:22	00:00:10
10	752,778	00:02:02	00:04:35	00:00:13	00:00:07
COUV9	1,129,747	00:05:45	00:10:53	00:00:28	00:00:17
BUBEN	1,739,211	00:22:16	00:32:47	00:02:23	00:00:47
SHVO	1,909,577	00:19:09	00:35:09	00:02:00	00:00:49
C2	1,973,550	00:58:09	01:34:43	00:06:49	00:01:59
SH	3,022,848	01:01:33	01:32:26	00:06:04	00:01:59

Table 5.8: FEM analysis results, ordering time, engineering problems

is not sufficient to hold the whole coefficient matrix the sparse direct solver processes the matrix out-of-core by moving data between memory and disk as needed. No additional disk storage is required for the out-of-core processing besides the amount listed in Table 5.6.

On the other hand, the disk storage requirements of the sparse direct solver are substantially lower than that of the frontal solver, about 80% in most cases. It is due to the efficient block sparse storage and especially due to the use of the fill-in minimization ordering (i.e., the minimum degree algorithm). The disk storage requirements of the sparse direct solver are sometimes slighly higher that its memory requirements because unlike the memory storage, fixed-size blocks have to be used in the disk storage.

Solution times of the sparse direct solver and the frontal solver are listed in Table 5.7. It can be seen that the sparse direct solver is indeed quite efficient, the block sparse solution can be up to 90% faster than the frontal solution in some cases. Largest problems achieved at least about 50% faster block sparse solution, although the solution times were biased by the disk access speed because of the necessary out-of-core processing.

Finally, ordering times of the sparse direct solver are listed in Table 5.8 (the frontal solver does not use comparable internal ordering). It can be seen that the ordering times are practically negligible compared to the solution times, therefore, they do not present a significant disadvantage for the sparse direct solver in the overall comparison to the frontal solver.

In conclusion, the implemented sparse direct solver based on the proposed block sparse matrix storage format, minimum degree algorithm and block sparse factorization has been

confirmed to be an efficient and adequate complement to the existing frontal solver of the PMD system. It has high memory requirements, but in practice, this is not a difficulty for present computers and operating systems, and the savings in time and disk space required for the solution can be up to 80% or even more. The sparse direct solver is a suitable option especially in cases of very large problems where the frontal solver starts to have unreasonable requirements on disk storage due to large frontwidth. However, it should be noted that there are some large problems where the frontal solution is faster, particularly when the frontwidth is small. As already mentioned, the minimum degree algorithm is not guaranteed to give the best ordering for any arbitrary problem, since it is after all only a heuristic.

Chapter 6

Conclusions

An efficient sparse direct solver for finite element analysis of very large problems in continuum mechanics was designed. As a starting point, today's numerical methods, which played the central role in direct solution approaches, were critically analyzed and on the basis of this, new algorithms were proposed. In the next step, the sparse direct solver was implemented, thouroughly tested and employed in practical applications to have proven itself in real-world engineering problems. The contribution of the work presented in this thesis was thus both of theroretical and practical nature.

The theoretical contribution lies primarily in the proposed efficient algorithms for the storage, ordering, and solution of large sparse linear systems. Although the appropriate mathematical methods have been around for some time, their application to sparse linear systems with 10^6 equations or more becomes problematic even on contemporary computers, and therefore it requires special consideration and careful approach. The overall quality of the implementation of involved algorithms has a substantial impact on the solution time and storage size and may result in unfeasibility of the solution if inappropriate numerical procedures were applied.

The original matrix storage method selected for the sparse direct solver was the promising K3 storage format, which was, however, found unsuitable for the intended application during the course of the work. Therefore, it was replaced by an efficient custom storage format that was based on a combination of several storage schemes including the K3 storage format. The proposed matrix storage format exploits sparsity and block structure resulting from the finite element method to achieve minimum storage requirements, while using simple arrays that can be easily implemented in any relevant programming language. Moreover, the proposed storage format allows for an out-of-core implementation of the matrix assembly, factorization and resolution, which are the primary matrix operations performed by the sparse direct solver. An out-of-core solver of course involves much more complicated algorithms than an in-core solver, and it should be emphasized that the implemented possibility for an out-of-core solution is crucial for the practical usability of the sparse direct solver for large finite element problems.

The minimum degree algorithm, which is one of the most commonly used ordering methods due to its relative directness and efficiency, was implemented in a different way than in most established codes. It was tailored to the finite element method by perform-

ing symbolic assembly and elimination on the block nonzero structure of the coefficient matrix, which turned out to be significantly more efficient than analyzing the nonzero structure of the matrix entries. The linked-list based quotient graph used in the proposed minimum degree algorithm was highly optimized to allow the operations on the edge sets to be carried out in an efficient manner. The proposed minimum degree algorithm employs all features found in the established implementations, such as supervariables, element absorption, external degrees, and approximate degrees used in the state-of-the-art approximate minimum degree algorithm (AMD). For research purposes and fine-tuning of the minimum degree algorithm, several features were set optional by outletting them to the solver's input file. Subsection 5.1.2 presents a comprehensive assessment of the performance of the proposed minimum degree algorithm that was used to choose the best type of degree computation for the practical applications of the sparse direct solver. The ordering times achieved by the sparse direct solver on large problems demonstrated the effectiveness and usability of the proposed minimum degree algorithm.

The chosen solution method ($\mathbf{LDL}^{\mathrm{T}}$ factorization) is also well known but its common application is not overly well suited to large systems because of its high demands on the storage space. To minimize the memory requirements of the factorization, it was proposed in Subsection 5.1.3 to employ a rather non-traditional left-looking algorithm performed on rows instead of on columns as was standard in skyline solvers. This reduced the size of the active submatrix that had to be present in the memory (core) during the factorization significantly and, therefore, made the out-of-core solution more feasible since much less out-of-core data manipulation was needed. Moreover, the solution algorithm made use of the proposed block sparse matrix storage format in order to reduce the number of floating-point operations during the factorization and resolution to minimum by skipping submatrix multiplications where any of the factors was zero.

On practical part, a sparse direct solver code compatible with the PMD finite element system was made, based on the aforementioned algorithms. The implemented FORTRAN 77 code has been integrated into the PMD system, and now presents a fully featured linear solver capable of efficient solving of large problems. The sparse direct solver was designed to perform the three primary matrix operations (assembly, factorization and resolution) all fully in-core but, in addition, to automatically use out-of-core algorithms whenever available memory is limited. This allowed very large finite element problems to have been solved. One should note that by implementing this complex out-of-core sparse direct solver, the initial requirements on the solver were significantly surpassed.

Numerical results obtained by the sparse direct solver were carefully analyzed and compared to the existing frontal solver of the PMD system and, therefore, the correct and accurate operation of the new solver was verified. Finally, numerical tests were carried out on several large real-world engineering problems using the finite element method. Their results, presented in Subsection 5.3.2, clearly demonstrated the abundant effectiveness of the sparse direct solver compared to the existing frontal solver.

In conclusion, while **the aims of this thesis were accomplished**, further work should be directed towards the integration of the new solution procedures into the dynamic and non-linear branch of the PMD system to fully exploit the effectiveness of this fast linear solver. Another interesting line of development is the implementation of the sparse

direct solver on 64-bit platforms, where the benefits of the sparse factorization can be utilized fully due to virtually unlimited available memory.

Bibliography

[1] Amestoy P. R., Davis T. A. and Duff I. S. (2004). Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Transactions on Mathematical Software*, **30**, 3, pp. 381–388.

[2] Amestoy P. R., Davis T. A. and Duff I. S. (1996). An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, **17**, 4, pp. 886–905.

[3] Bai Z., Demmel J., Dongarra J., Ruhe A. and van der Vorst H., editors (2000). *Templates for the solution of algebraic eigenvalue problems: A practical guide*. SIAM, Philadelphia.

[4] Barnard S. T., Pothen A. and Simon H. D. (1993). A spectral algorithm for envelope reduction of sparse matrices. *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pp. 493–502.

[5] Bathe K. J. (1996). *Finite element procedures*. Prentice-Hall, Upper Saddle River, NJ.

[6] Blackford L. S., Demmel J., Dongarra J., Duff I. S., Hammarling S., Henry G., Heroux H., Kaufman L., Lumsdaine A., Petitet A., Pozo R., Remington K., and Whaley R. C. (2002). An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software*, **28**, 2, pp. 135–151.

[7] Botsch M., Bommes D. and Kobbelt L. (2005). Efficient linear system solvers for mesh processing. In: Martin R. R., Bez H. E. and Sabin M. A., editors. *IMA Conference on the Mathematics of Surfaces*, **3604**, pp. 62–83. Springer, Berlin.

[8] Cavers I.A. (1989). *Using deficiency measure for tiebreaking the minimum degree algorithm*. Technical Report TR-89-02. University of British Columbia, Vancouver, BC.

[9] Cuthill E. and McKee J. (1969). Reducing the bandwidth of sparse symmetric matrices. *Proceedings of the 1969 24th national conference*, pp. 157–172.

[10] Davis T. A. (2009). *Summary of available software for sparse direct methods*. `http://www.cise.ufl.edu/research/sparse/codes/`. Retrieved on April 1, 2009.

[11] Diestel R. (2000). *Graph theory*, 2nd edition. Springer, New York.

[12] Duff I. S. and Scott J. A. (2005). *Towards an automatic ordering for a symmetric sparse direct solver*. Technical Report RAL-TR-2006-001. Rutherford Appleton Laboratory, Oxon.

[13] Duff I. S. (1998). *Matrix methods*. Technical Report RAL-TR-1998-076. Rutherford Appleton Laboratory, Oxon.

[14] Duff I. S. (1998). *Direct methods*. Technical Report RAL-TR-1998-054. Rutherford Appleton Laboratory, Oxon.

[15] Duff I. S. (1997). *Sparse numerical linear algebra: direct methods and preconditioning*. Technical Report RAL-TR-96-047. Rutherford Appleton Laboratory, Oxon.

[16] Duff I. S. and Reid J. K. (1983). The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Transactions on Mathematical Software*, **9**, 3, pp. 302–325.

[17] Duff I. S. and Reid J. K. (1982). *MA27—A set of Fortran subroutines for solving sparse symmetric sets of linear equations*. Technical Report AERE R10533. Her Majesty's Stationery Office, London.

[18] George A. and Liu J. W. H. (1981). *Computer solution of large sparse positive definite systems*. Prentice-Hall, Englewood Cliffs, NJ.

[19] George A. and Liu J. W. H. (1989). The evolution of the minimum degree ordering algorithm. *SIAM Review*, **31**, 1, pp. 1–19.

[20] George A. and Liu J. W. H. (1980). A fast implementation of the minimum degree algorithm using quotient graphs. *ACM Transactions on Mathematical Software*, **6**, 3, pp. 337–358.

[21] George A. (1973). Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, **10**, 2, pp. 345–363.

[22] Gilbert J. R., Moler C. and Schreiber R. (1992). Sparse matrices in MATLAB: design and implementation. *SIAM Journal on Matrix Analysis and Applications*, **13**, 1, pp. 333-356.

[23] Gould N. I. M., Scott J. A. and Hu Y. (2007). A numerical evaluation of sparse direct solvers for the solution of large sparse symmetric linear systems of equations. *ACM Transactions on Mathematical Software*, **33**, 2.

[24] Irons B. M. (1970). A frontal solution program for finite-element analysis. *International Journal for Numerical Methods in Engineering*, **2**, 1, pp. 5–32.

[25] Karypis G. and Kumar V. (1998). A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, **20**, 1, pp. 359–392.

[26] Khaira M. S., Miller G. L. and Sheffler T. J. (1992). *Nested Dissection: A survey and comparison of various nested dissection algorithms*. Technical Report CSD. Carnegie Mellon University, Pittsburgh, PA.

[27] Kruis J. (2006). *Domain decomposition methods for distributed computing*. Saxe-Coburg, Glasgow.

[28] Liu J. W. H. (1985). Modification of the minimum-degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software*, **11**, 2, pp. 141–153.

[29] Markowitz H. M. (1957). The elimination form of the inverse and its application to linear programming. *Management Science*, **3**, 3, pp. 255–269.

[30] Okrouhlík M., editor (2008). *Numerical methods in computational mechanics*. Institute of Thermomechanics ASCR, Prague.

[31] Plešek J. and Gabriel D. (2000). *PMD Example Manual*. Institute of Thermomechanics ASCR, Prague.

[32] *PMD Reference Guide*. VAMET Ltd., Prague.

[33] *PMD User Guide*. VAMET Ltd., Prague.

[34] Rose D. J. (1970). *Symmetric elimination on sparse positive definite systems and the potential flow network problem*. Doctoral Thesis. Harvard University, Cambridge, MA.

[35] Tinney W. F. and Walker J. W. (1967). Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proceedings of the IEEE*, **55**, 11, pp. 1801–1809.

[36] Ueberhuber C. W. (1994). *Numerical computation*. Springer, Berlin.

[37] Vondráček R. (2008). *Use of a sparse direct solver in engineering applications of the finite element method*. Doctoral Thesis. Czech Technical University in Prague, Prague.

[38] Vondrák V. (2000). *Description of the K3 sparse matrix storage system*. Technical Report, unpublished. Technical University in Ostrava, Ostrava.

Author's publications

[39] Pařík P. and Plešek J. (2009). Assessments of the implementation of the minimum degree ordering algorithms. *Pollack Periodica, International Journal for Engineering and Information Sciences*, **4**, 3, pp. 121–128.

[40] Pařík P. (2009). Performance tests of the minimum degree ordering algorithm. *Engineering Mechanics 2009*, pp. 929–935.

[41] Pařík P. (2008). Implementation of a sparse direct solver for large linear systems. *Výpočty konstrukcí metodou konečných prvků 2008*, pp. 98–101.

[42] Pařík P. (2007). Sparse direct solver with fill-in optimization. *Engineering Mechanics 2007*.

[43] Pařík P. (2005). Numerická implementace lineárního řešiče na bázi algoritmu AMD. *Summer Workshop of Applied Mechanics 2005*, pp. 75–84.

Appendix A

Selected FEM problems

A.1 Example problems

A.1.1 Elastostatics

Problem name	LSOL	NNOD	NELEM	NVAR	NPIV	Problem description
BEAM6S1	46	23	4	42	22	Plane stress
BEAM4S1	54	27	8	50	26	Plane stress
TUBE7TS1	106	53	10	53	53	Axisymmetric thermal stress
BEAM56S1	168	56	4	154	55	Surface traction
BEAM61S1	95	23	4	86	22	Edge traction
BEAM53S1	30	5	4	24	4	Concentrated force, body force
BEAM71S1	184	56	7	170	55	Transition elements

A.1.2 Dynamics

Problem name	LSOL	NNOD	NELEM	NVAR	NPIV	Problem description
BEAM56D1	168	56	4	144	48	Natural frequencies
BEAM61D1	95	23	4	84	20	Natural frequencies
BEAM53D1	30	5	4	24	4	Natural frequencies
BEAM71D1	184	56	7	160	48	Natural frequencies
BEAM56D2	168	56	4	144	48	Transient response
BEAM53D2	30	5	4	24	4	Steady-state response, mode synthesis
BEAM53D3	30	5	4	24	4	Steady-state response, direct integration
BEAM53D4	30	5	4	24	4	Steady-state response, response spectrum method

A.1.3 Plasticity

Problem name	LSOL	NNOD	NELEM	NVAR	NPIV	Problem description
BEAM6P1	46	23	4	0	0	Uniaxial strain
BEAM56P1	168	56	4	48	48	Uniaxial strain
BEAM6P2	46	23	4	42	22	Uniaxial stress
BEAM56P2	168	56	4	154	55	Uniaxial stress
BEAM6P3	46	23	4	42	22	Isotropic hardening
BEAM56P3	168	56	4	154	55	Isotropic hardening
BEAM6P4	46	23	4	42	22	Kinematic hardening
BEAM56P4	168	56	4	154	55	Kinematic hardening
BEAM6P5	46	23	4	42	22	Cyclic hardening
BEAM56P5	168	56	4	154	55	Cyclic hardening
BEAM6P6	46	23	4	42	22	Cyclic softening
BEAM56P6	168	56	4	154	55	Cyclic softening
BEAM6P7	46	23	4	38	21	Residual stress
BEAM56P7	168	56	4	140	54	Residual stress
BEAM6P8	46	23	4	42	22	Thermo-plasticity
BEAM56P8	168	56	4	154	55	Thermo-plasticity
TUBE7P1	106	53	10	53	53	Penalty method, axisymmetric
TUBE56P1	384	128	10	203	128	Penalty method

A.1.4 Creep

Problem name	LSOL	NNOD	NELEM	NVAR	NPIV	Problem description
BEAM6C1	46	23	4	42	22	Long-term creep
BEAM6C2	46	23	4	38	21	Relaxation
BEAM56C1	168	56	4	154	55	Bina's model, strain hardening
BEAM56C2	168	56	4	154	55	Bina's model, time hardening
BEAM56C3	168	56	4	154	55	Bina's model, damage softening
TUBE7C1	106	53	10	53	53	Penalty solution, axisymmetric
TUBE56C1	384	128	10	203	128	Penalty solution

A.1.5 Geomerically nonlinear problems

Problem name	LSOL	NNOD	NELEM	NVAR	NPIV	Problem description
BEAM6G1	46	23	4	42	22	Large deflection
BEAM4G1	54	27	8	50	26	Large deflection
BEAM56G1	168	56	4	160	55	Stability of a column
BEAM61G1	95	23	4	84	22	Stability of a column
BEAM56G2	168	56	4	160	55	Buckling

A.1.6 Contact

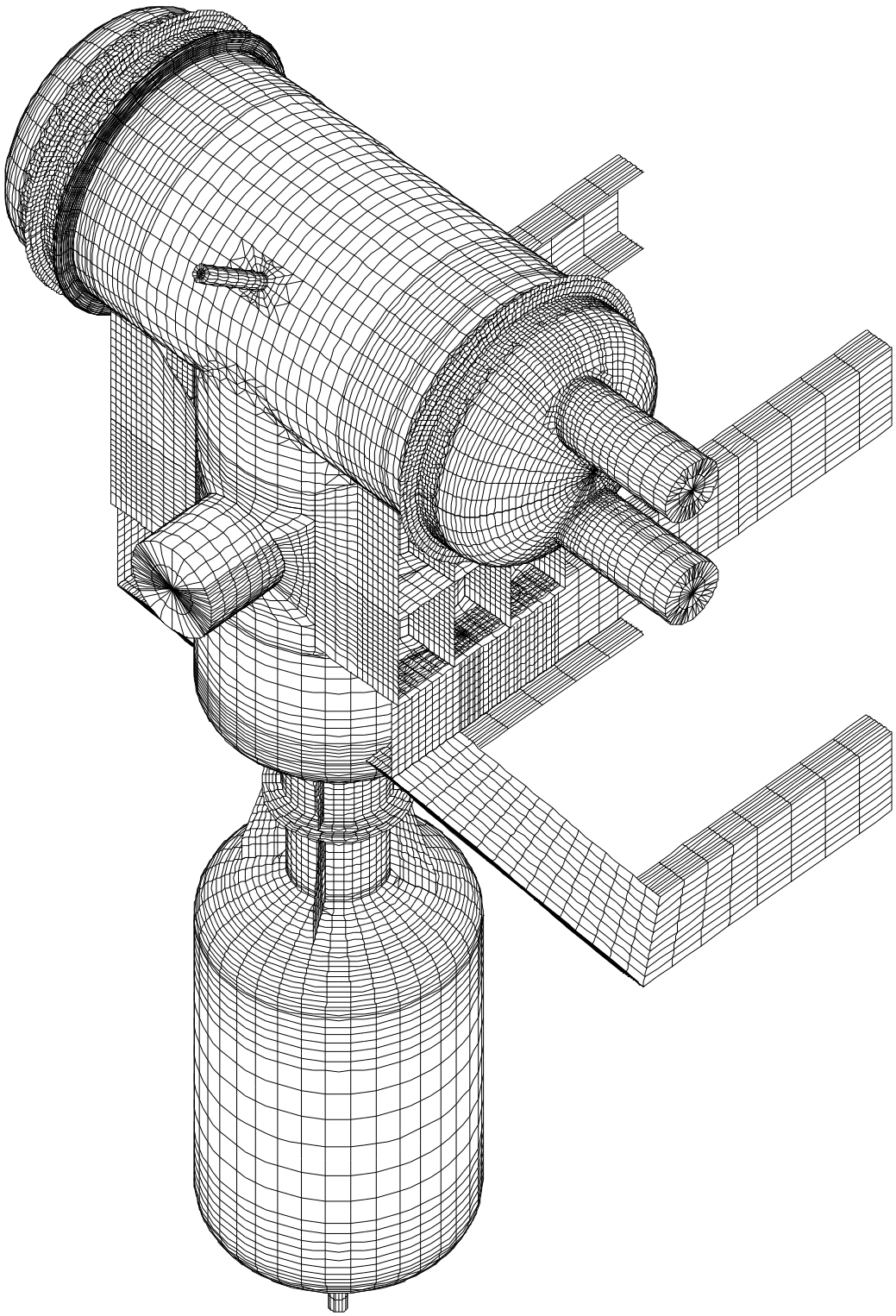
Problem name	LSOL	NNOD	NELEM	NVAR	NPIV	Problem description
CUBE55K1	126	42	3	86	41	Simple contact
PRESS7K1	112	56	10	90	56	Press fit connection

A.2 Engineering problems

Problem name	LSOL	NNOD	NELEM	NVAR	NPIV	Element type(s)
10	752,778	173,070	59,570	747,630	172,450	semi-loof
A98ABT001_1	365,043	121,681	19,250	365,031	121,681	pentahed.,hexah.
ADS	359,504	82,760	28,240	357,968	82,504	semi-loof
BUBEN	1,739,211	579,737	140,816	1,739,198	579,736	pentahed.,hexah.
C2	1,973,550	657,850	412,054	1,973,358	657,850	tetrahedral
C3	559,125	186,375	108,886	558,833	186,375	tetrahedral
C5	393,774	131,258	70,629	393,630	131,210	tetrahedral
COUV9	1,129,747	257,861	97,579	1,129,211	257,725	semi-loof
DOCHL	387,829	89,797	27,610	387,741	89,797	semi-loof
K1	136,354	31,338	10,978	136,354	31,338	semi-loof
K3	362,262	83,142	29,510	362,114	83,118	semi-loof
P6_LIN	19,656	6,552	5,330	18,334	6,552	hexahedral
P6_QUAD	74,742	24,914	5,330	70,924	24,914	hexahedral
SH	3,022,848	695,050	242,461	3,020,479	695,050	semi-loof
SHVO	1,909,577	439,287	152,297	1908134	439287	semi-loof

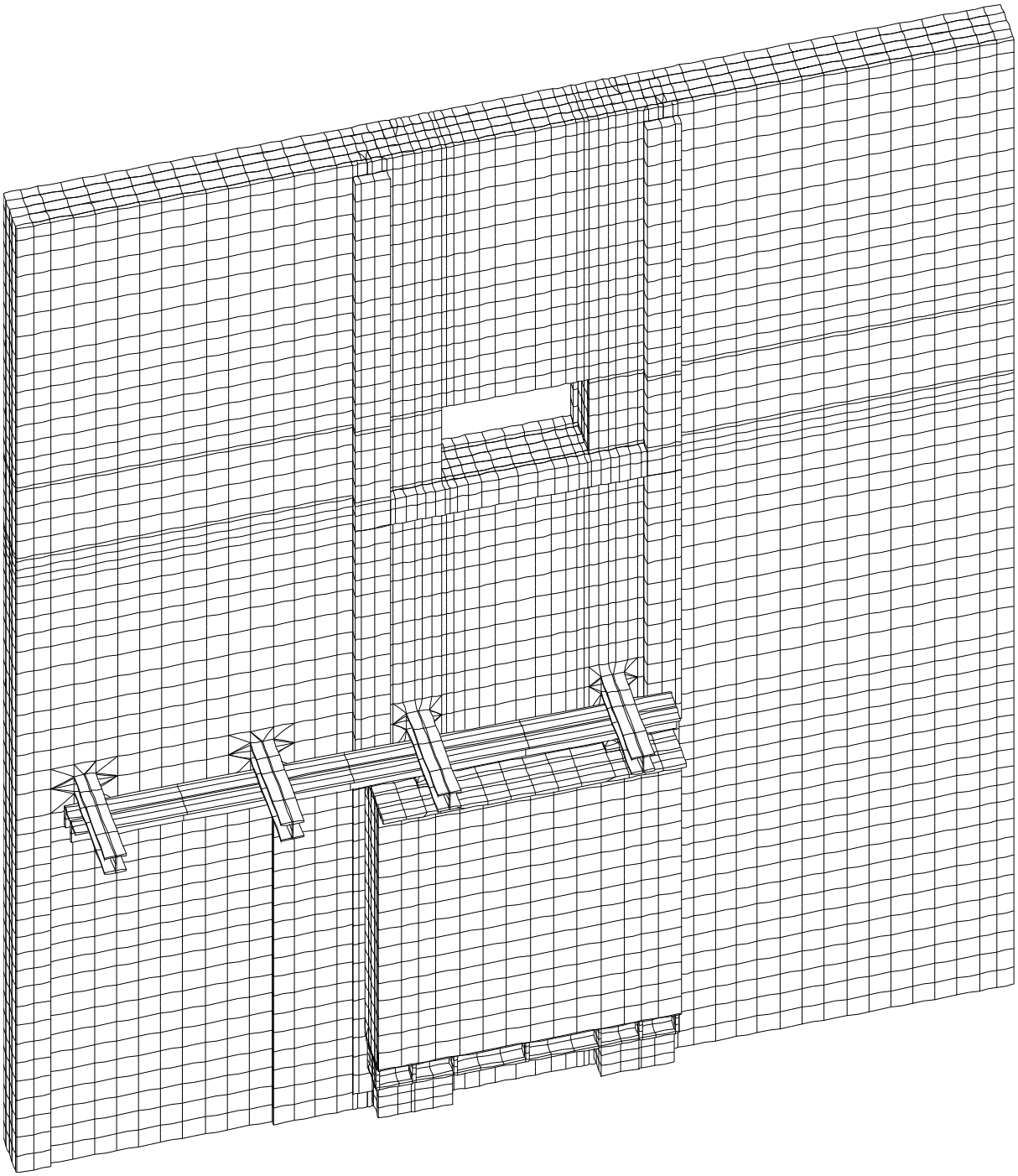
A.2.1 10

Elastostatic analysis of a degasser.



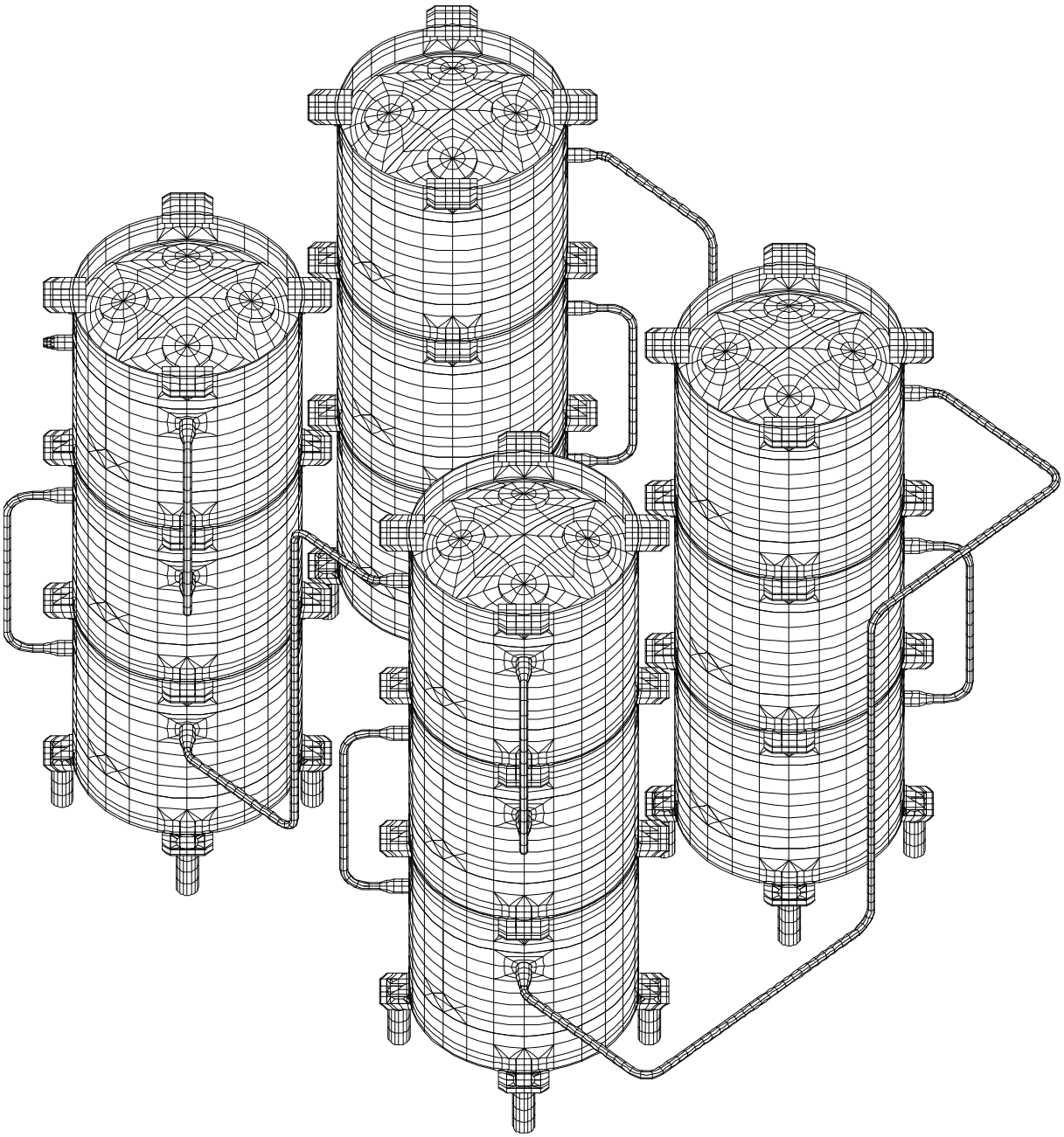
A.2.2 A98ABT001_1

Seismic analysis of a reactor shielding door from a nuclear power plant.



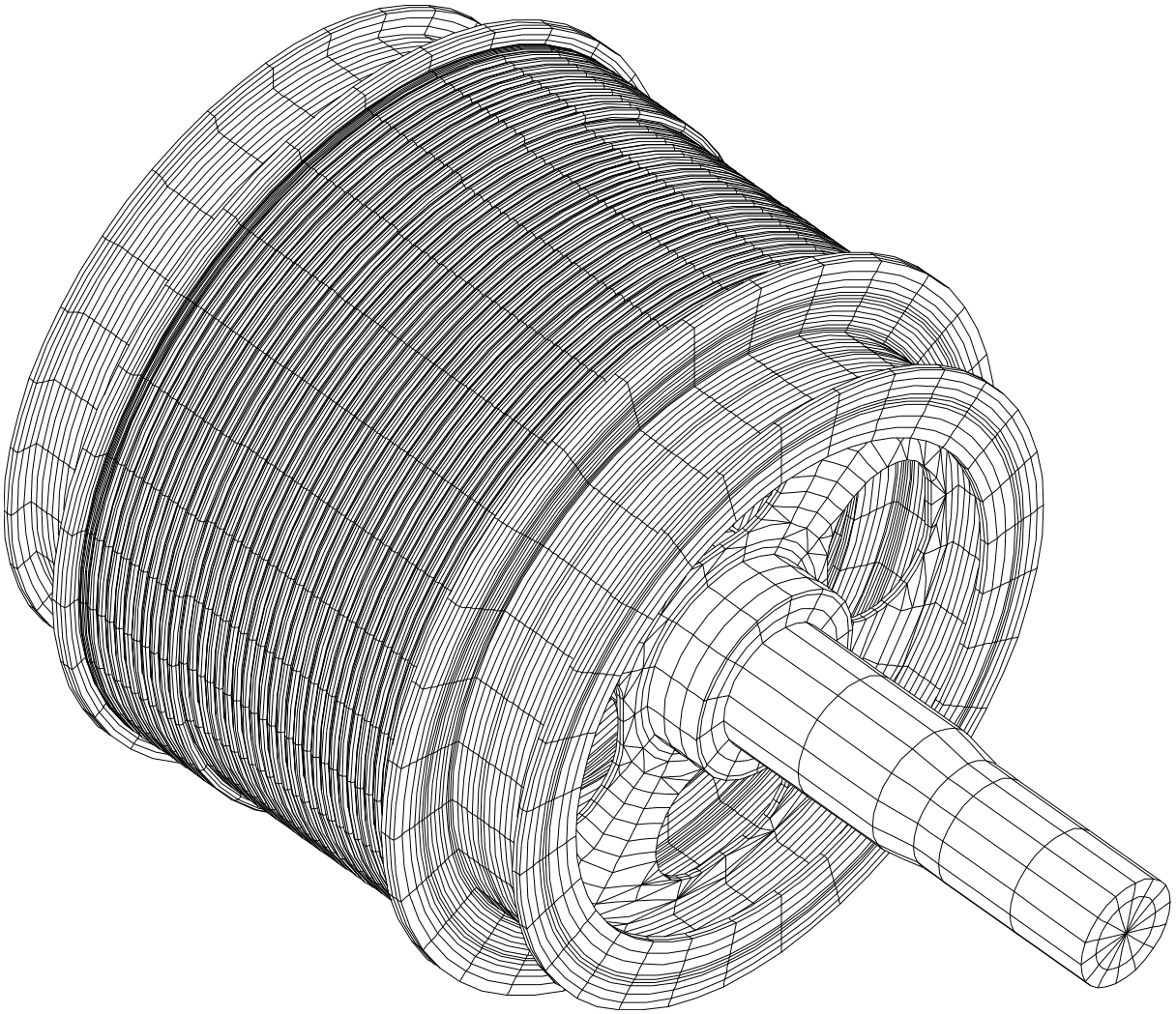
A.2.3 ADS

Elastostatic, dynamic and seismic analysis of an adsorber cell from a nuclear power plant.



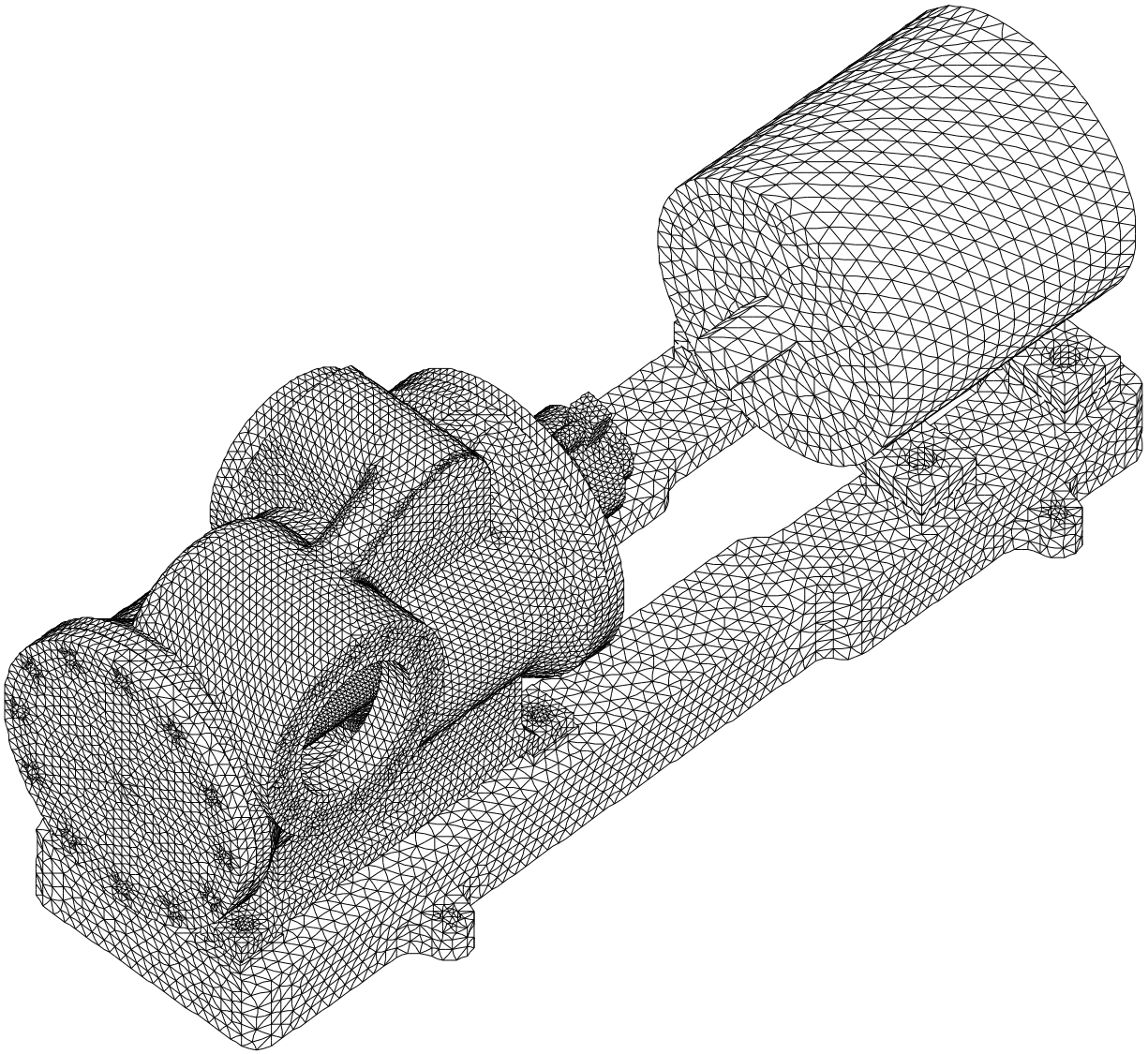
A.2.4 BUBEN

Elastostatic analysis of a winding engine drum.



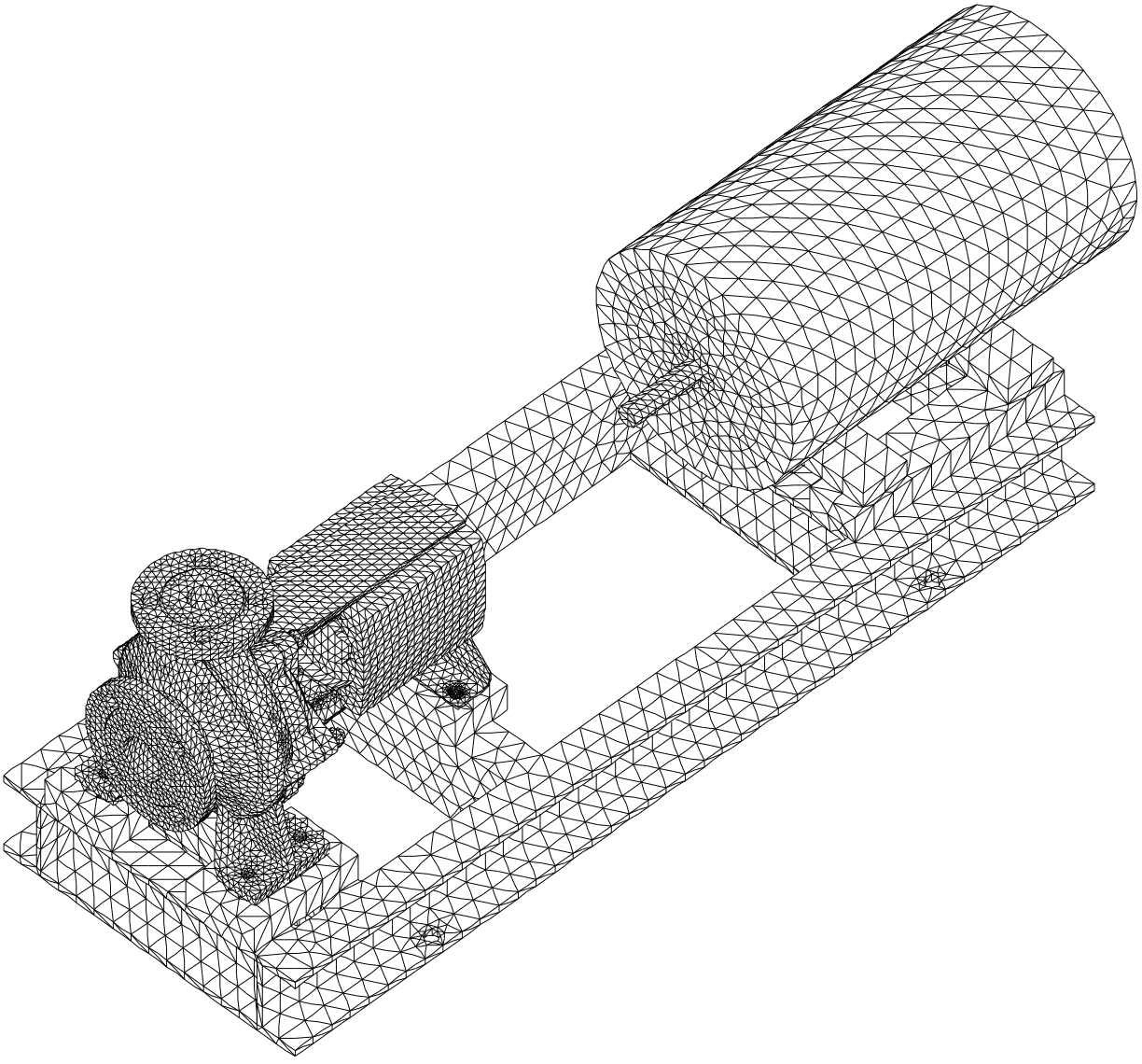
A.2.5 C2

Elastostatic, dynamic and seismic analysis of an oil pump from a nuclear power plant.



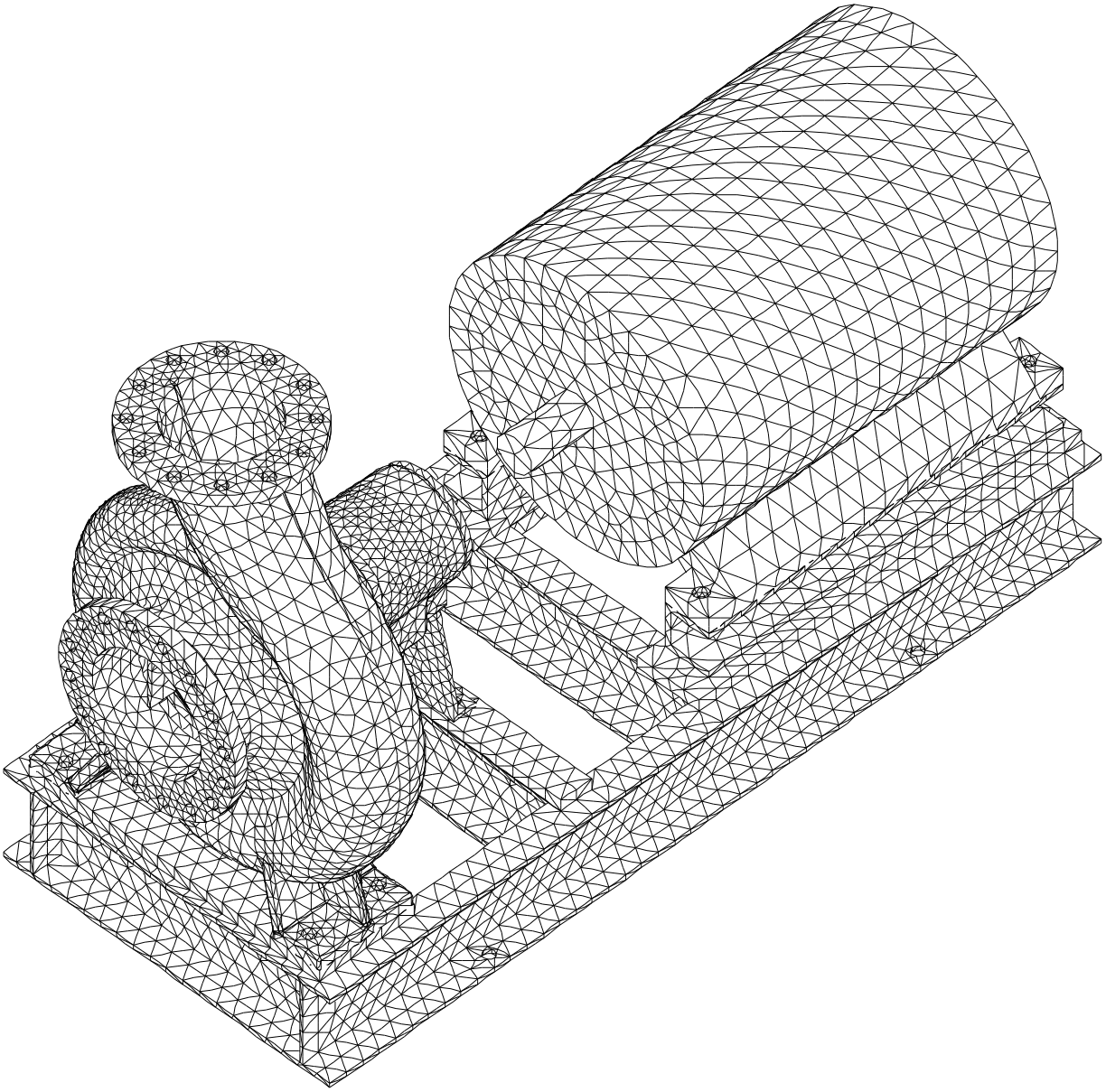
A.2.6 C3

Elastostatic and dynamic analysis of a waste water pump from a nuclear power plant.



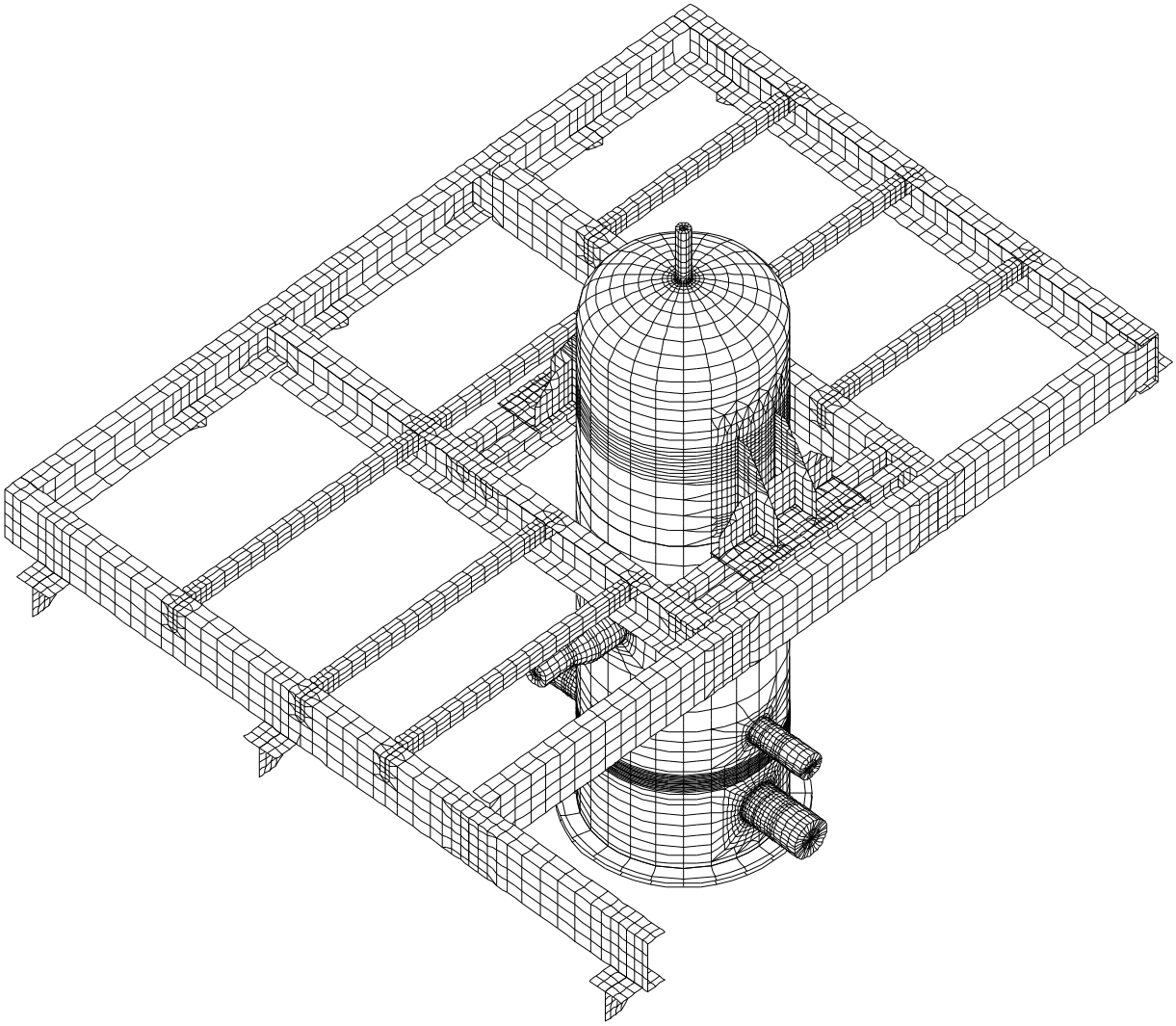
A.2.7 C5

Elastostatic, dynamic and seismic analysis of a sprinkler system pump from a nuclear power plant.



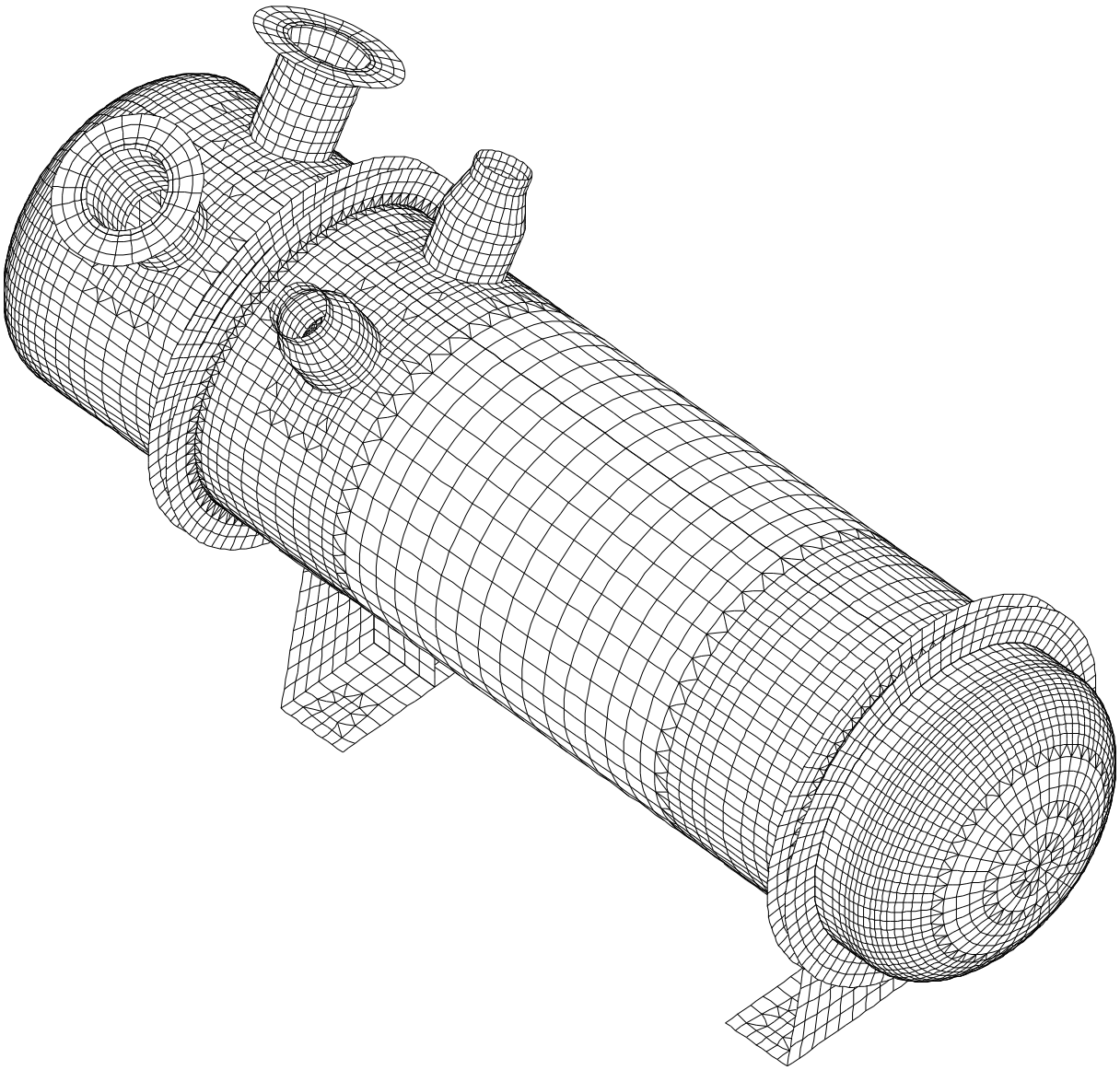
A.2.8 COUV9

Elastostatic and seismic analysis of a steam heat exchanger.



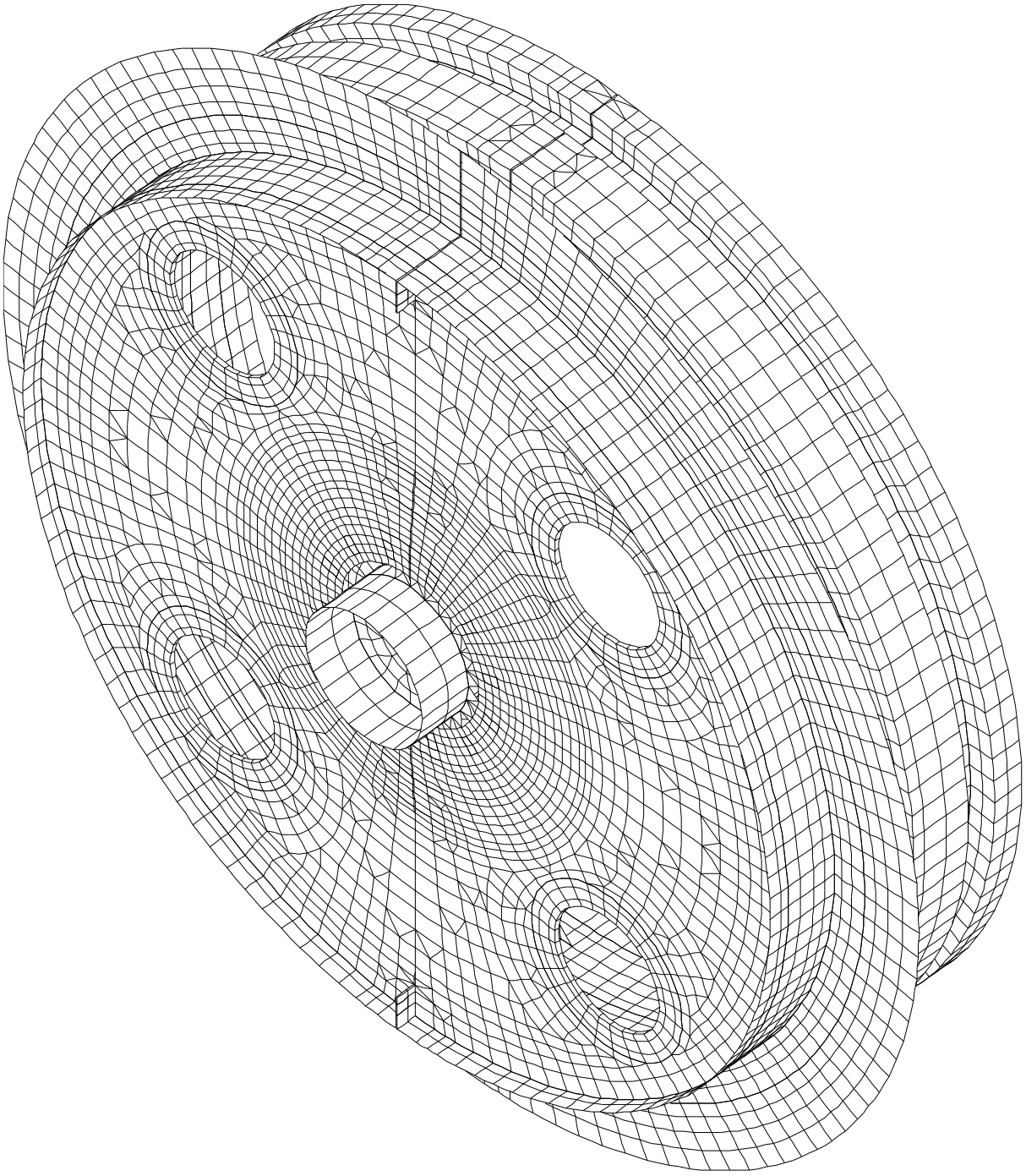
A.2.9 DOCHL

Elastostatic analysis of a heat exchanger.



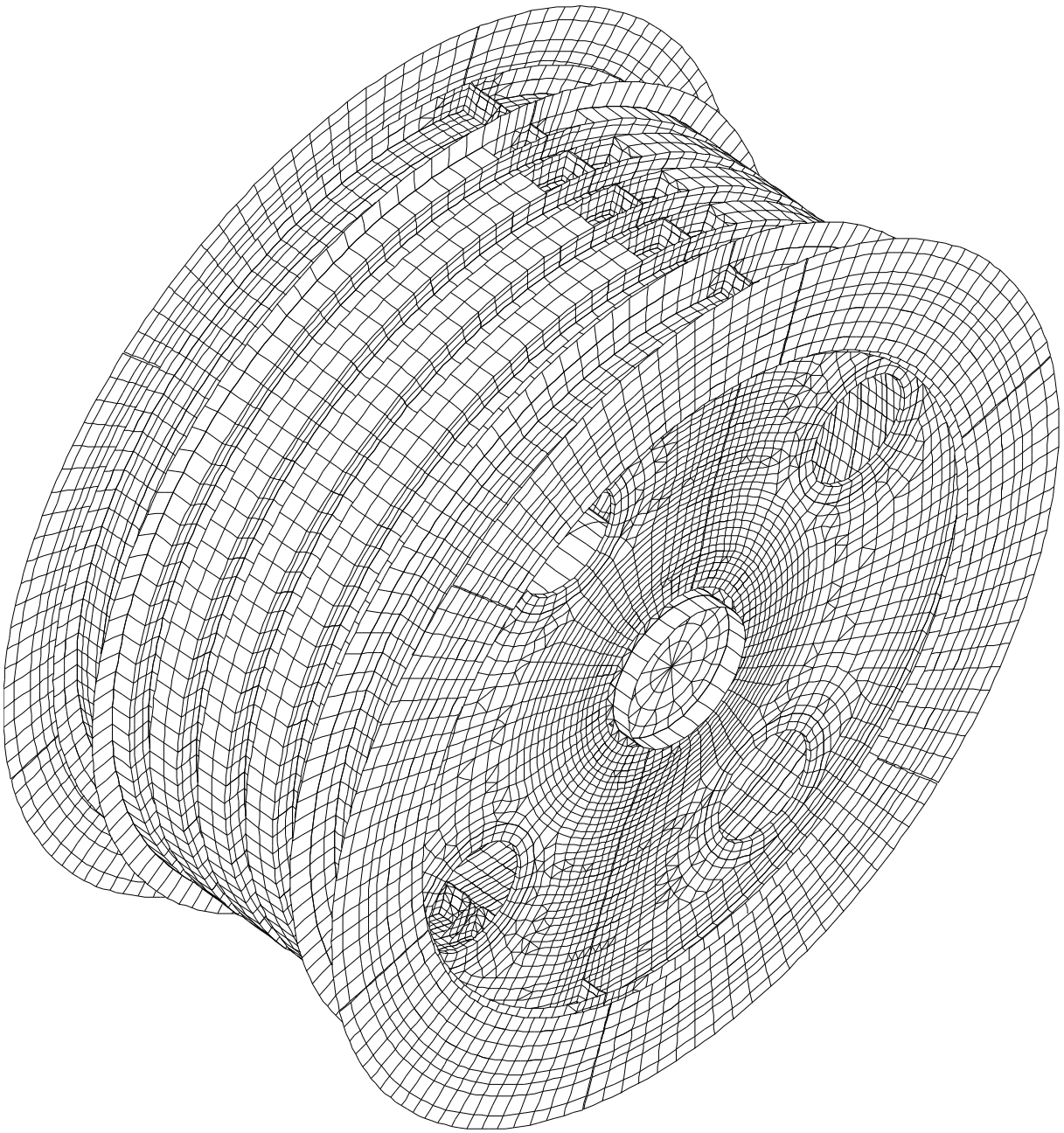
A.2.10 K1

Elastostatic analysis of a winding engine drum.



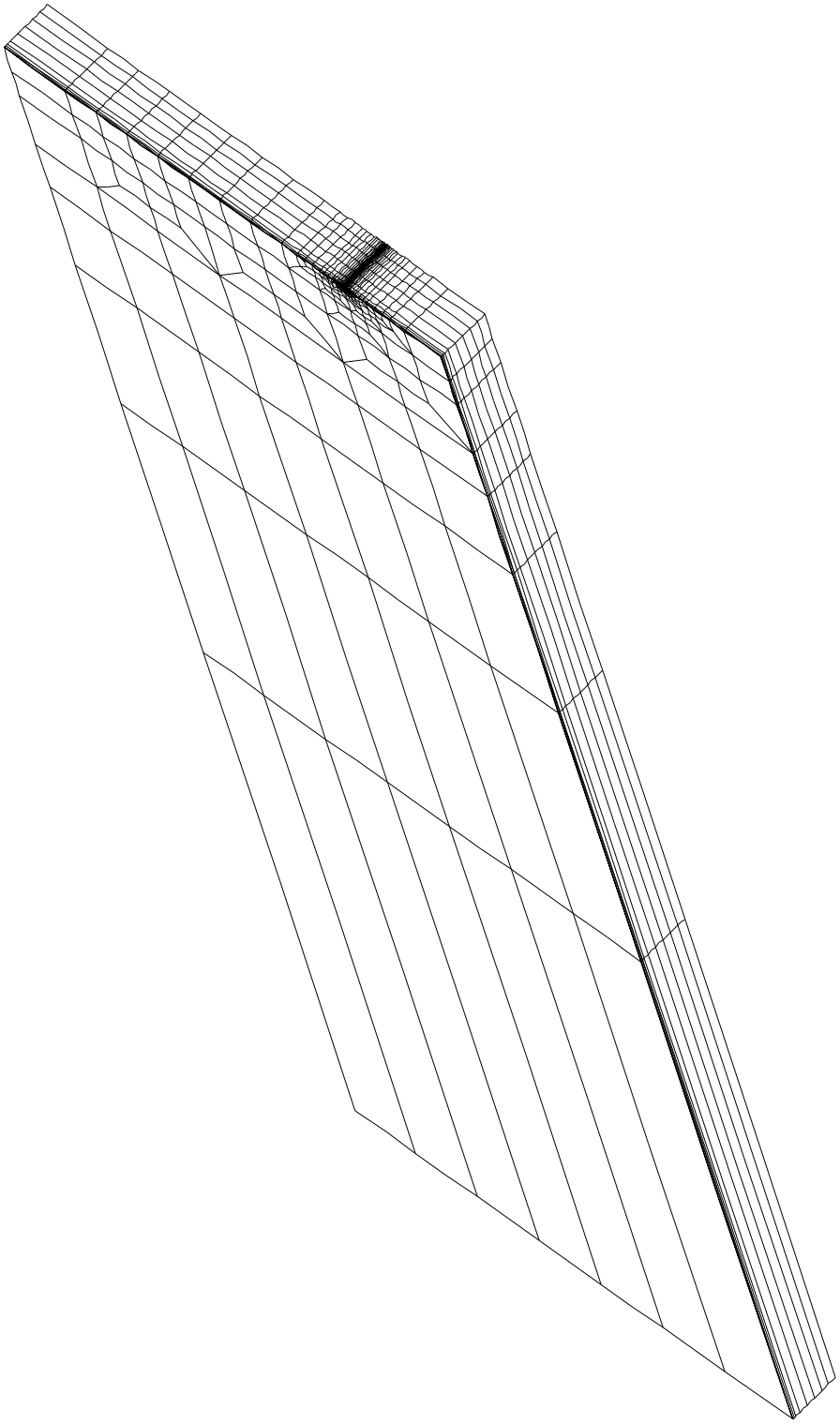
A.2.11 K3

Elastostatic analysis of a winding engine drum.



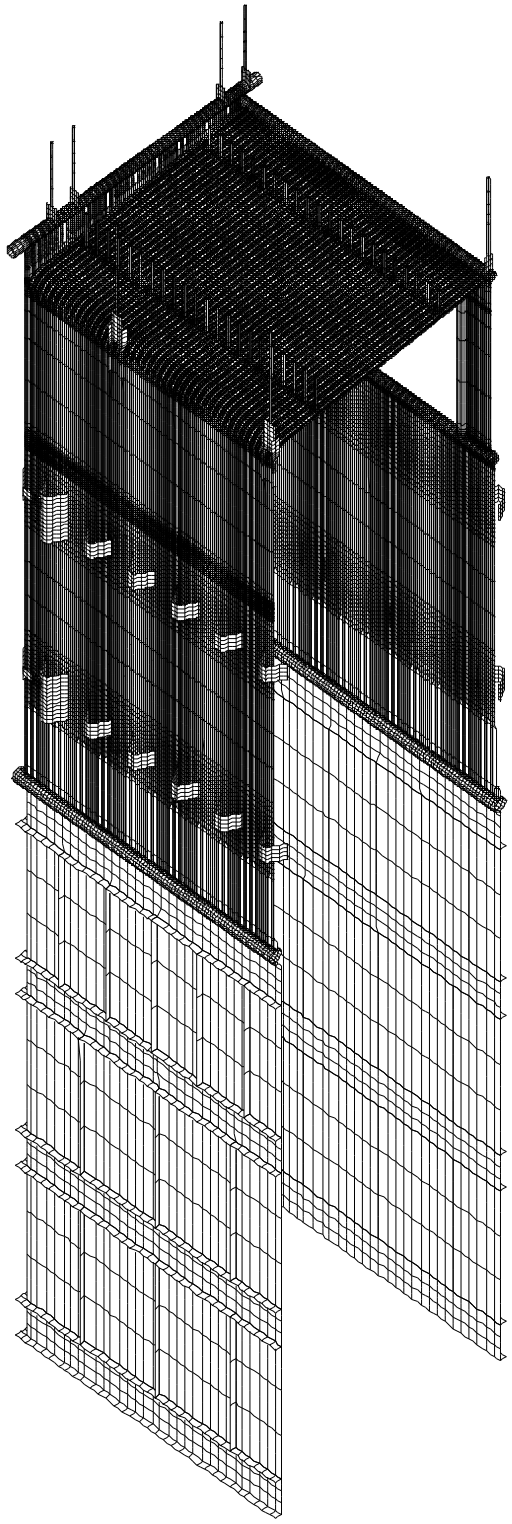
A.2.12 P6_LIN, P6_QUAD

Plasticity analysis of a crack with contact searching.



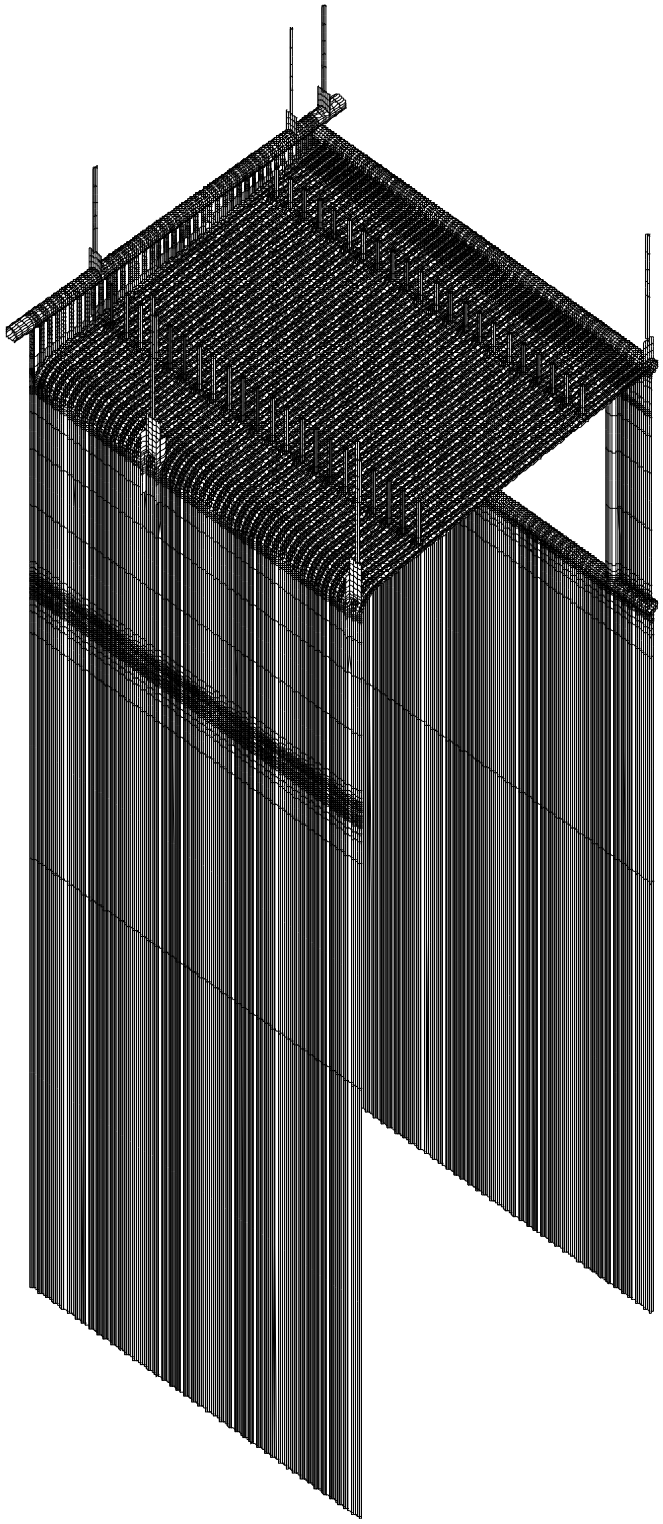
A.2.13 SH

Elastostatic analysis of a boiler membrane wall from a thermal power plant.



A.2.14 SHVO

Elastostatic analysis of a boiler membrane wall from a thermal power plant.



Appendix B

Solver file formats

B.1 Formatted files

The only formatted files used by the sparse direct solver are the standard PMD input and output files. This section is intended to update the PMD Reference Guide.

B.1.1 *name*.I4 (FEFS, FESD)

The sparse direct solver utilizes the same input file as the frontal solver, i.e., *name*.I4, making it trivial for the user to move from one solver to the other. The corresponding output file is *name*.O4.

It is possible, if desired, to use several separate input and output files by adding an arbitrary alphanumeric character to the input file extension, i.e., *name*.I4A. The output file is also named accordingly.

Input file format

```
; integer parameters
IP KREST KMET KPRIN

; real parameters
RP PIVOT

; end of input
EN
EN
```

Description

KREST	Restart key
= 1	Factorize the coefficient matrix and solve for all load cases.
= 2	Solve for additional load cases.

`KMET`	Method key (controls minimum degree ordering algorithm)
$= 0$	Use default method.
$= xx1$	Compute exact degrees.
$= xx2$	Compute approximate degrees according to Amestoy, Davis and Duff.
$= xx3$	Compute approximate degrees according to Ashcraft, Eisenstat and Lucas.
$= xx4$	Compute approximate degrees according to Gilbert, Moler and Schreiber.
$= x0x$	Do not preorder.
$= x1x$	Preorder with Cuthill-McKee algorithm.
$= x2x$	Preorder with reverse Cuthill-McKee algorithm.
$= 0xx$	Use supervariables with external degrees.
$= 1xx$	Use supervariables with true degrees.
$= 2xx$	Use true degrees (no supervariables).

`KPRIN`	Print key (controls output to file *name*.O4)
$= 0$	Do not print debug information (default).
$= 1$	Print runtime and memory allocation information.
$= 2$	Also print computed permutation vector.

`PIVOT`	Minimum pivot value (default $= 10^{-6}$)

Notes

- `KREST` has the same function in both solvers.

  Necessary conditions for `KREST` $= 2$ are

  - the program has already finished factorization with `KREST` $= 1$, and
  - additional load cases were processed by programs RPD3 and SRH3 (or RPD2 and SRH2, respectively) with `KREST` $= 2$.

- `KMET` is used only by the sparse direct solver.

  The default value, recommended for general use, is `KMET` $= 1$. Other values may occasionally yield better orderings, but are usually slower.

  Prefixing `KMET` with a minus sign causes the sparse direct solver to finish after the ordering phase, giving the user the opportunity to check the computed ordering. `KMET` < 0 should be used together with `KPRIN` ≥ 1.

  When solving very large problems, it may be worthwhile to test different degree computation methods and/or preordering methods this way to find the best suited value of `KMET` before commiting the factorization, because savings in computational time and storage space may be significant.

- `KPRIN` is used only by the sparse direct solver.

- `PIVOT` has the same function in both solvers.

  To prevent numerical instability, the factorization is stopped when the pivot in the absolute value is too small ($|p| \leq$ `PIVOT`). A warning message is issued into the output file *name*.O4 whenever the pivot is negative or small ($p \leq 10^2 \times$ `PIVOT`) to notify the user. Such conditions are usually caused by errors in the input data.

B.2 Unformatted files

The sparse direct solver uses several intermediate unformatted files[1] to store the coefficient matrix factor (factorized equations) and other internal data. Intermediate files used by the frontal solver are not accessed in any way. This section is intended only for PMD developers.

B.2.1 `IDEQC`

File `IDEQC` is a sequential file used to store various internal data of the sparse direct solver related to ordering, assembly and factorization. It is written after the ordering phase and contains four records:

1. Common parameters: `NPIV`, `NVAR`, `NBLK1`, `NBLK2`, `SBUF1`, `SBUF2`, `LSMTX`, `LROW`, `LEMTX` and `KMET`.

2. Permutation of nodes: array `IPNOD(NNOD)`.

3. Permutation of degrees (equations): array `IPSOL(LSOL)`.

4. Dimensions of matrix blocks: array `MBD(NPIV)`.

B.2.2 `IDEQI`

File `IDEQI` is a sequential file used to store the indices of nonzero blocks in the factorized coefficient matrix, necessary to reconstruct the arrays `MRP(NPIV+1)` and `MCI(NBLK2)`. It is written during the ordering phase, but initially it contains unpermuted indices. The file is rewritten with correctly permuted indices prior to the assembly phase.

There are `NPIV` records; each record contains $N+1$ values of type `INTEGER*4`, where the first value is N, and the rest N values are indices of nonzero blocks in the corresponding coefficient matrix block row.

[1] In the PMD documentation, unformatted files are commonly referred to by the corresponding source code symbol consisting of prefix `ID` and the file extension, for example `IDELM` for file *name*.ELM.

B.2.3 IDEQR

File IDEQR is a direct-access file used to store the data of nonzero blocks in the coefficient matrix factor. The internal structure is different from file IDEQ1 used by the frontal solver, therefore, equations factorized by the sparse direct solver cannot be directly used by the frontal solver and vice versa.

The file is normally written after the factorization phase, however, in the out-of-core mode, it may be written as soon as during the assembly phase. There are NBLK2 records; each record represents one block (submatrix) and stores LBLK values of type REAL*8. Some space is wasted in smaller blocks but an efficient out-of-core data manipulation is possible.

Appendix C

Solver source code

The sparse direct solver is coded in standard FORTRAN 77. No third-party code is used except for the standard subroutines from PMD library S3 (ERM2, FINISH, IFILL, IMOV, INIT, IZERO, RDISC, RMOV, RRDISC, RTMPC, RZERO, SIAR, WCOMD, WDISC and WRDISC). The sparse direct solver uses several data files (IDELM, IDP, IDRHS, IDSOL and IDCOM) that are preprocessed or postprocessed by other PMD programs, and also several own intermediate files (IDEQC, IDEQI and IDEQR).

C.1 Program FESD

Program FESD contains all subroutines required for solving a linear problem in PMD. There is little reason in listing the full source code (about 3,000 lines), thus, a shortened version, without actual code but with comments on the purpose of each subroutine, is presented, as a reference for PMD developers.

The main program (listed in full) contains only few necessary statements; the actual code is executed via a call to subroutine RUN. The amount of memory available to the program is specified by parameter LI.

Main program

```
PROGRAM FESD
PARAMETER (LI = 100 000 000)
COMMON /CPMD/ ICP(160),ICW(160),RCP(48),RCW(336)
EQUIVALENCE (ICP(3),ICL)
DIMENSION INT(LI),R(LI/2)
EQUIVALENCE (INT(1),R(1))
CALL INIT('FESD',ISEC)
CALL RUN(INT,LI,R,LI/ICL)
CALL FINISH('FESD',ISEC)
END
```

Subroutines

```
      SUBROUTINE CIPS(NNDF,LNNDF,MBD,IPNOD,NNOD,IPSOL,LSOL,ITMP)
      DIMENSION NNDF(LNNDF),MBD(NNOD),IPNOD(NNOD),IPSOL(LSOL),
     *ITMP(NNOD+1)
C
C     Computes permutation vector IPSOL (ordering of equations) using
C     permutation vector IPNOD (ordering of nodes) and mesh data NNDF and MBD.
C
      END
      SUBROUTINE CLEMTX(INET,LINET,NNET,LNNET,NNDF,LNNDF,NELEM,LEMTX)
      DIMENSION INET(LINET),NNET(LNNET),NNDF(LNNDF)
C
C     Computes maximum length of element matrix LEMTX exactly. LEMTX from the
C     PMD common block is not used since it is often set to the absolute
C     maximum.
C
      END
      SUBROUTINE CMBD(NNDF,LNNDF,IFIXV,NFIXV,IPSOL,LSOL,MBD,NNOD)
      DIMENSION NNDF(LNNDF),IFIXV(NFIXV),IPSOL(LSOL),MBD(NNOD)
C
C     Computes number of variables in blocks MBD using mesh data NNDF and
C     IFIXV. Also initializes IPSOL for subroutine CIPS.
C
      END
      FUNCTION IINT2R(IINT)
      COMMON /CPMD/ ICP(160),ICW(160),RCP(48),RCW(336)
      EQUIVALENCE (ICP(3),ICL)
C
C     Converts index of integer array element into index of corresponding real
C     array element. The index is adjusted so the real element does not
C     overwrite any preceding integer elements.
C     Note: The sizes of integer and real data are platform-dependent.
C
      END
      FUNCTION IR2INT(IR)
      COMMON /CPMD/ ICP(160),ICW(160),RCP(48),RCW(336)
      EQUIVALENCE (ICP(3),ICL)
C
C     Converts index of real array element into first index of corresponding
C     integer array element.
C     Note: The sizes of integer and real data are platform-dependent.
C
      END
      SUBROUTINE ISORT(IA,LIA)
      DIMENSION IA(LIA)
C
```

```
C     Sorts elements of integer array IA using the insertion sort algorithm.
C
      END
      FUNCTION LINT2R(LINT)
      COMMON /CPMD/ ICP(160),ICW(160),RCP(48),RCW(336)
      EQUIVALENCE (ICP(3),ICL)
C
C     Converts length of integer array into length of corresponding real array.
C     Note: The sizes of integer and real data are platform-dependent.
C
      END
      FUNCTION LR2INT(LR)
      COMMON /CPMD/ ICP(160),ICW(160),RCP(48),RCW(336)
      EQUIVALENCE (ICP(3),ICL)
C
C     Converts length of real array into length of corresponding integer array.
C     Note: The sizes of integer and real data are platform-dependent.
C
      END
      SUBROUTINE PADD(IA,IB,LIAB)
      DIMENSION IA(LIAB),IB(LIAB)
C
C     Combines two successive permutations defined by vectors IA and IB into
C     a single permutation vector and stores the result back in array IA.
C
      END
      SUBROUTINE PIAR(IA,IB,IP,LIABP)
      DIMENSION IA(LIABP),IB(LIABP),IP(LIABP)
C
C     Permutes integer array IA using permutation vector IP and stores the
C     result in array in IB.
C
      END
      SUBROUTINE PIARR(IA,IB,IP,LIABP)
      DIMENSION IA(LIABP),IB(LIABP),IP(LIABP)
C
C     Permutes integer array IA using inverse permutation vector IP and stores
C     the result in array in IB. Permutation is done without computing the
C     inverse.
C
      END
      SUBROUTINE PINV(IA,IB,LIAB)
      DIMENSION IA(LIAB),IB(LIAB)
C
C     Computes inverse transformation (ordering) to IA and stores it in IB.
C
```

```
      END
      SUBROUTINE PRAR(A,B,IP,LABP)
      DIMENSION A(LABP),B(LABP),IP(LABP)
C
C     Permutes real array A using permutation vector IP and stores the result
C     in array in B.
C
      END
      SUBROUTINE PRARR(A,B,IP,LABP)
      DIMENSION A(LABP),B(LABP),IP(LABP)
C
C     Permutes real array A using inverse permutation vector IP and stores the
C     result in array in B. Permutation is done without computing the inverse.
C
      END
      SUBROUTINE RUN(INT,LI,R,LR)
      COMMON /CPMD/ ICP(160),ICW(160),RCP(48),RCW(336)
      EQUIVALENCE (ICP(1),MW),(ICP(3),ICL),(ICP(10),NELEM),
     *(ICP(11),NNOD),(ICP(18),LINET),(ICP(19),LNNET),(ICP(20),LNNDF),
     *(ICP(28),NFIXV),(ICP(34),LSOL),(ICP(62),IDP),(ICP(63),IDCOM),
     *(ICP(66),IDELM),(ICP(67),IDRHS),(ICP(73),IDSOL),(ICP(84),NASV),
     *(ICP(86),KPRIN),(ICP(122),KFES)
      EQUIVALENCE (ICW(1),NPIV),(ICW(2),NVAR),(ICW(3),NBLK1),
     *(ICW(4),NBLK2),(ICW(5),LSMTX),(ICW(6),LEMTX),(ICW(7),LROW),
     *(ICW(8),IDEQC),(ICW(9),IDEQI),(ICW(10),IDEQR),(ICW(11),LBBUF),
     *(ICW(12),LEBUF),(ICW(13),LCBUF)
      EQUIVALENCE (RCW(1),SBUF1),(RCW(2),SBUF2)
      DIMENSION INT(LI),R(LR)
C
C     Manages the operation of the sparse direct solver.
C
      END
      SUBROUTINE SDAMD(IG,LIG,ID,IS,IN,IP,MBD,IPNOD,ITMP,JTMP,KTMP,
     *NBLK,SBUF,LROW,IDISC,KDG,KSV,KED)
      COMMON /CPMD/ ICP(160),ICW(160),RCP(48),RCW(336)
      EQUIVALENCE (ICP(1),MW),(ICP(11),NNOD),(ICP(86),KPRIN),
     *(ICP(132),KMET)
      DIMENSION MBD(NNOD),IPNOD(NNOD),ITMP(NNOD),JTMP(NNOD),
     *KTMP(NNOD),IG(LIG),ID(NNOD),IS(NNOD),IN(NNOD),IP(NNOD)
C
C     Computes permutation vector IPNOD (ordering of nodes) using minimum degree
C     ordering algorithm. Also computes memory requirements for factorization
C     and writes indices of nonzero matrix blocks into file IDEQI.
C
      END
      SUBROUTINE SDASM(MRP,MCI,MBP,BUF,IPSOL,INET,NNET,NNDF,ITMP,TMP,
```

```
     *SBUF)
      COMMON /CPMD/ ICP(160),ICW(160),RCP(48),RCW(336)
      EQUIVALENCE (ICP(1),MW),(ICP(3),ICL),(ICP(10),NELEM),
     *(ICP(11),NNOD),(ICP(18),LINET),(ICP(19),LNNET),(ICP(20),LNNDF),
     *(ICP(34),LSOL),(ICP(66),IDELM),(ICP(86),KPRIN)
      EQUIVALENCE (ICW(1),NPIV),(ICW(2),NVAR),(ICW(3),NBLK1),
     *(ICW(4),NBLK),(ICW(6),LEMTX),(ICW(11),LBBUF),(ICW(12),LEBUF),
     *(ICW(14),NA),(ICW(15),NR),(ICW(16),NW),(ICW(17),NF)
      DIMENSION MRP(NPIV+1),MCI(NBLK),MBP(NBLK),BUF(LEBUF),
     *IPSOL(LSOL),INET(LINET),NNET(LNNET),NNDF(LNNDF),
     *ITMP(NNOD),TMP(LEMTX),NDF(20),IDF(20)
C
C     Assembles the coefficient matrix using element matrices from file IDELM.
C
      END
      SUBROUTINE SDBKS(MRP,MCI,MBP,MBD1,BUF,RHS,LRHS,NRHS)
      COMMON /CPMD/ ICP(160),ICW(160),RCP(48),RCW(336)
      EQUIVALENCE (ICW(1),NPIV),(ICW(2),NVAR),(ICW(4),NBLK),
     *(ICW(12),LEBUF)
      DIMENSION MRP(NPIV+1),MCI(NBLK),MBP(NBLK),MBD1(NPIV+1),
     *BUF(LEBUF),RHS(LRHS,NRHS)
C
C     Performs back substitution for all right-hand sides RHS. The result is
C     stored back in RHS.
C
      END
      SUBROUTINE SDBUF1(MRP,MCI,MBP,BUF,IPSOL,NNDF,ITMP,
     *IBLK,LBLK,IPTR)
      COMMON /CPMD/ ICP(160),ICW(160),RCP(48),RCW(336)
      EQUIVALENCE (ICP(11),NNOD),(ICP(20),LNNDF),(ICP(34),LSOL)
      EQUIVALENCE (ICW(1),NPIV),(ICW(2),NVAR),(ICW(4),NBLK),
     *(ICW(10),IDEQR),(ICW(11),LBBUF),(ICW(12),LEBUF),(ICW(13),LCBUF),
     *(ICW(14),NA),(ICW(16),NW),(ICW(17),NF),(ICW(15),NR)
      DIMENSION MRP(NPIV+1),MCI(NBLK),MBP(NBLK),BUF(LEBUF),
     *IPSOL(LSOL),NNDF(LNNDF),ITMP(NNOD)
C
C     Adds submatrix to buffer BUF (optionally first reads matrix block from
C     file IDEQR). When the buffer is full, writes all matrix blocks into file
C     IDEQR and resets the buffer.
C
      END
      SUBROUTINE SDBUF2(MRP,MCI,MBP,MBD,BUF,ITMP,IPIV,IROW)
      COMMON /CPMD/ ICP(160),ICW(160),RCP(48),RCW(336)
      EQUIVALENCE (ICW(1),NPIV),(ICW(4),NBLK),(ICW(10),IDEQR),
     *(ICW(11),LBBUF),(ICW(12),LEBUF),(ICW(13),LCBUF),
     *(ICW(14),NR),(ICW(15),NW),(ICW(16),NF)
```

```
      DIMENSION MRP(NPIV+1),MCI(NBLK),MBP(NBLK),MBD(NPIV),BUF(LEBUF),
     *ITMP(NPIV)
C
C     Reads row of matrix blocks from file IDEQR into buffer BUF. When the
C     buffer is full, writes all matrix blocks into file IDEQR and resets the
C     buffer.
C
      END
      SUBROUTINE SDBUF3(MBP,BUF,IBLK,LBLK)
      COMMON /CPMD/ ICP(160),ICW(160),RCP(48),RCW(336)
      EQUIVALENCE (ICW(4),NBLK),(ICW(10),IDEQR),(ICW(11),LBBUF),
     *(ICW(12),LEBUF),(ICW(13),LCBUF),(ICW(14),NR),(ICW(15),NF)
      DIMENSION MBP(NBLK),BUF(LEBUF)
C
C     Reads matrix block from file IDEQR into buffer BUF. When the buffer is
C     full, resets the buffer.
C
      END
      SUBROUTINE SDCHK(LI,LMAX,LCOR,LBUF)
      COMMON /CPMD/ ICP(160),ICW(160),RCP(48),RCW(336)
      EQUIVALENCE (ICP(1),MW),(ICP(86),KPRIN)
C
C     Checks memory requirement LMAX against available memory LI. If the
C     available memory is insufficient the program is aborted.
C
      END
      SUBROUTINE SDDIS(MRP,MCI,MBP,MBD1,BUF,RHS,TMP,SBUF)
      COMMON /CPMD/ ICP(160),ICW(160),RCP(48),RCW(336)
      EQUIVALENCE (ICP(1),MW),(ICP(3),ICL),(ICP(34),LSOL),
     *(ICP(84),NASV),(ICP(86),KPRIN)
      EQUIVALENCE (ICW(1),NPIV),(ICW(2),NVAR),(ICW(4),NBLK),
     *(ICW(11),LBBUF),(ICW(12),LEBUF),(ICW(14),NR),(ICW(15),NF)
      DIMENSION MRP(NPIV+1),MCI(NBLK),MBP(NBLK),MBD1(NPIV+1),
     *BUF(LEBUF),RHS(LSOL,NASV),TMP(NVAR)
C
C     Computes solution for all right-hand sides RHS using forward and back
C     substitution. The result is stored back in RHS.
C
      END
      SUBROUTINE SDFAC(MRP,MCI,MBP,MBD,BUF,ITMP,SBUF)
      COMMON /CPMD/ ICP(160),ICW(160),RCP(48),RCW(336)
      EQUIVALENCE (ICP(1),MW),(ICP(3),ICL),(ICP(86),KPRIN)
      EQUIVALENCE (ICW(1),NPIV),(ICW(4),NBLK),(ICW(10),IDEQR),
     *(ICW(11),LBBUF),(ICW(12),LEBUF),
     *(ICW(14),NR),(ICW(15),NW),(ICW(16),NF)
      DIMENSION MRP(NPIV+1),MCI(NBLK),MBP(NBLK),MBD(NPIV),BUF(LEBUF),
```

```
     *ITMP(NPIV),D(6),G(6,6)
C
C     Factorizes the coefficient matrix. The factor is stored in the place of
C     the original matrix.
C
      END
      SUBROUTINE SDFWS(MRP,MCI,MBP,MBD1,BUF,RHS,LRHS,NRHS,TMP)
      COMMON /CPMD/ ICP(160),ICW(160),RCP(48),RCW(336)
      EQUIVALENCE (ICW(1),NPIV),(ICW(2),NVAR),(ICW(4),NBLK),
     *(ICW(12),LEBUF)
      DIMENSION MRP(NPIV+1),MCI(NBLK),MBP(NBLK),MBD1(NPIV+1),
     *BUF(LEBUF),RHS(LRHS,NRHS),TMP(NVAR)
C
C     Performs forward substitution for all right-hand sides RHS. The result is
C     stored back in RHS.
C
      END
      SUBROUTINE SDGRF(IG,LIG,INET,NNET,MBD,NBLK,SBUF)
      COMMON /CPMD/ ICP(160),ICW(160),RCP(48),RCW(336)
      EQUIVALENCE (ICP(10),NELEM),(ICP(11),NNOD),(ICP(18),LINET),
     *(ICP(19),LNNET)
      DIMENSION IG(LIG),INET(LINET),NNET(LNNET),MBD(NNOD)
C
C     Constructs a quotient graph for the minimum degree ordering algorithm and
C     computes memory requirements for the assembly.
C
      END
      SUBROUTINE SDORD(IG,LIG,INET,NNET,MBD,IPNOD,IPNOD1,ITMP,
     *KPR,KED,KSV,KDG)
      COMMON /CPMD/ ICP(160),ICW(160),RCP(48),RCW(336)
      EQUIVALENCE (ICP(1),MW),(ICP(11),NNOD),(ICP(18),LINET),
     *(ICP(19),LNNET),(ICP(86),KPRIN)
      EQUIVALENCE (ICW(3),NBLK1),(ICW(4),NBLK2),(ICW(7),LROW),
     *(ICW(9),IDEQI)
      EQUIVALENCE (RCW(1),SBUF1),(RCW(2),SBUF2)
      DIMENSION IG(LIG),INET(LINET),NNET(LNNET),MBD(NNOD),
     *IPNOD(NNOD),IPNOD1(NNOD),ITMP(NNOD*3)
C
C     Preorders nodes using subroutine SDRCM (if requested) and then calculates
C     permutation vector IPNOD using subroutine SDAMD.
C
      END
      SUBROUTINE SDPIV(PIVOT,DIAG,ISOL)
      COMMON /CPMD/ ICP(160),ICW(160),RCP(48),RCW(336)
      EQUIVALENCE (ICP(1),MW)
      EQUIVALENCE (RCP(32),PIVAL)
```

```
C
C     Checks pivot value PIVOT of equation ISOL and stores its reciprocal value
C     in DIAG. If the pivot is equal or less than PIVAL the program is aborted.
C
      END
      SUBROUTINE SDPMI(MRP,NPIV,MCI,NBLK,IPNOD,NNOD)
      DIMENSION MRP(NPIV+1),MCI(NBLK),IPNOD(NNOD)
C
C     Permutes column indices of matrix blocks MCI using permutation vector
C     IPNOD (ordering of nodes).
C
      END
      SUBROUTINE SDPPV(IG,LIG,ID,KQ,KL,IROOT)
      COMMON /CPMD/ ICP(160),ICW(160),RCP(48),RCW(336)
      EQUIVALENCE (ICP(11),NNOD)
      DIMENSION IG(LIG),ID(NNOD),KQ(NNOD),KL(NNOD)
C
C     Calculates the pseudo-peripheral vertex of the ordering graph starting
C     at vertex IROOT. The pseudo-peripheral vertex is returned in IROOT.
C
      END
      SUBROUTINE SDRCM(IG,LIG,ID,IPNOD,ITMP,KPR)
      COMMON /CPMD/ ICP(160),ICW(160),RCP(48),RCW(336)
      EQUIVALENCE (ICP(11),NNOD),(ICP(86),KPRIN)
      DIMENSION IG(LIG),ID(NNOD),IPNOD(NNOD),ITMP(NNOD)
C
C     Computes permutation vector IPNOD (ordering of nodes) using Cuthill-McKee
C     ordering algorithm. Reverse algorithm is used if KPR is nonzero.
C
      END
      SUBROUTINE SDREF(INET,NNET,NNDF,IPSOL,SOL,REF,ITMP,TMP)
      COMMON /CPMD/ ICP(160),ICW(160),RCP(48),RCW(336)
      EQUIVALENCE (ICP(10),NELEM),(ICP(11),NNOD),(ICP(18),LINET),
     *(ICP(19),LNNET),(ICP(20),LNNDF),(ICP(34),LSOL),(ICP(66),IDELM),
     *(ICP(84),NASV)
      EQUIVALENCE (ICW(2),NVAR),(ICW(6),LEMTX)
      DIMENSION INET(LINET),NNET(LNNET),NNDF(LNNDF),IPSOL(LSOL),
     *SOL(LSOL,NASV),REF(LSOL,NASV),ITMP(NNOD),TMP(LEMTX),IS(60)
C
C     Computes reaction forces REF for all load cases SOL. Corresponding
C     equations are assembled from file IDELM.
C
      END
      SUBROUTINE SDREQ(CMIN,CMAX,DMAX,KPR)
      COMMON /CPMD/ ICP(160),ICW(160),RCP(48),RCW(336)
      EQUIVALENCE (ICP(1),MW),(ICP(3),ICL),(ICP(11),NNOD),
```

```
     *(ICP(18),LINET),(ICP(19),LNNET),(ICP(20),LNNDF),(ICP(28),NFIXV),
     *(ICP(34),LSOL),(ICP(84),NASV),(ICP(86),KPRIN)
      EQUIVALENCE (ICW(1),NPIV),(ICW(2),NVAR),(ICW(3),NBLK1),
     *(ICW(4),NBLK2),(ICW(5),LSMTX),(ICW(6),LEMTX),(ICW(7),LROW)
      EQUIVALENCE (RCW(1),SBUF1),(RCW(2),SBUF2)
C
C     Calculates the memory requirements of the sparse direct solver.
C
      END
      SUBROUTINE SDRMI(MRP,NPIV,MCI,NBLK,IDISC)
      DIMENSION MRP(NPIV+1),MCI(NBLK)
C
C     Reads the indices of the matrix blocks from file IDISC.
C
      END
      SUBROUTINE SDWMI(MRP,NPIV,MCI,NBLK,IDISC)
      DIMENSION MRP(NPIV+1),MCI(NBLK)
C
C     Writes the indices of the matrix blocks into file IDISC.
C
      END
```

C.2 Program FESDA

Since the recompilation of the source code in order to change parameter `LI` is usually not feasible to normal user, program FESDA, which uses dynamic memory allocation, is also available. The program differs only in the main program code, all subroutines are identical to program FESD. It requires a Fortran 90 compiler, however, the compilation needs to be done only once.

To change the default value of `LI`, use additional parameters on the command line:

```
FESDA name -MLI value
```

where `name` is the name of problem data and `value` is the requested value of `LI`. The value can be specified also in megabytes (2^{20} bytes) using parameter `-M` instead of `-MLI`. If the specified value is 0, all available memory is allocated, subject to the limits of the compiler, operating system and computer configuration.