City Research Online

City St George's, University of London

Citation: Collins, R. J. (1980). A programming language for the finite element method. (Unpublished Doctoral thesis, The City University)

This is the accepted version of the paper.

This version of the publication may differ from the final published version. To cite this item please consult the publisher's version.

Permanent repository link: https://openaccess.city.ac.uk/id/eprint/37485/

Copyright and Reuse: Copyright and Moral Rights remain with the author(s) and/or copyright holders. Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge, unless otherwise indicated, provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way. For full details of reuse please refer to City Research Online policy.

City Research Online: http://openaccess.city.ac.uk/ publications@citystgeorges.ac.uk

A PROGRAMMING LANGUAGE FOR THE FINITE ELEMENT METHOD

Robert James Collins

Submitted for the degree of
Doctor of Philosophy

Department of Computer Science
The City University
St. John Street
London, EC1V 4PB

February 1980

THE CITY UNIVERSITY LIBRARY,
ST. JOHN STREET, LONDON, E.C.1.

CONTENTS

	ACKNOWLEDGEMENTS		5
	DECLARATION		6
	ABSTRACT		7
	NOTATION		8
1.	INTRODUCTION		9
	1.1	Special Purpose Languages	9
	1.2	Software for the Finite Element Method	12
	1.3	Implementation Considerations	15
	1.4	The Structure of the Thesis	17
2.	THE FINITE ELEMENT METHOD		19
	2.1	Introduction	19
	2.2	Stiffness Matrices and the Assembly Process	21
	2.3	The Basic Solution Process	26
	2.4	Additional Computational Features	28
	2.5	Requirements for an Extensible System	31
3.	THE FRONTAL SOLUTION SYSTEM		33
	3.1	Introduction	33
	3.2	The Front Solution Algorithm	35
	3.3	Housekeeping and Data Structures for the Front Algorithm	39
	3.4	Prescribed Left Hand Sides	41
	3.5	Multiplication	42
	3.6	Further Operations	44
	3.7	Conclusion	48

4.	THE LANGUAGE FEATURES	50
4.1	Introduction	50
4.2	Sysmats and Sysvecs	52
4.3	Type Definitions	54
4.4	Node Definitions	61
4.5	Forward and Backward Loops	64
4.6	Vector Operations	67
4.7	Matrix Operations	69
4.8	The Format of a Complete Program	70
4.9	Conclusion	72
5.	USE OF THE PROGRAMMING SYSTEM	73
5.1	Introduction	73
5.2	Variations on the Linear Theme	75
5.3	Norms and Convergence Tests	78
5.4	Structural Reanalysis	81
5.5	Non-linear Problems	83
5.6	A Simple Eigenproblem	87
5.7	Acceleration Techniques	92
5.8	Conclusion	95
6.	EIGENPROBLEMS	96
6.1	Introduction	96
6.2	Some Eigensolution Methods	98
6.3	Simultaneous Vector Iteration	102
6.4	Two Standard Procedures	105
6.5	The Standard Procedures in Use	108
7.	INPUT AND OUTPUT	113
7.1	Introduction	113
7.2	The General Format of the Input Data	115
7.3	Element and Node Data	116
7.4	Sysvec Data	119
7.5	Output	121
7.6	Conclusion	123
8.	STRUCTURAL DESIGN - THE COMPLETE SOLUTION PROCESS	124
8.1	Introduction	124
8.2	The Basic Design Process	126
8.3	Interactive Facilities	129
8.4	Optimum Structural Design	131

9.	IMPLEMENTATION DETAILS		137
	9.1	Introduction	137
	9.2	An Outline of the Current System	139
	9.3	Frontal Data Handling	141
	9.4	The Preprocessor	144
	9.5	Sysvecs and Sysmats	147
	9.6	Type and Node Definitions	149
	9.7	Loop and Vec Clauses	151
	9.8	The Procedure Visit and Block Clauses	155
	9.9	Conclusion	156
10.	CONCLUSION		157
	10.1	Some Limitations of the Current System	157
	10.2	Final Remarks	159
A1.	A MODIFIED FRONT SOLUTION ALGORITHM		162
	A1.1	Introduction	162
	A1.2	Dynamic Destinations	163
	A1.3	Implementation Considerations	165
A2.	AN ALTERNATIVE DATA STRUCTURE FOR MATRICES		166
	A2.1	Introduction	166
	A2.2	An Alternative Data Structure	167
A3.	THE SYNTAX OF THE EXTENSION TO ALGOL 68-R		168
	A3.1	Introduction	168
	A3.2	New Modes	169
	A3.3	The Syntax Rules	170
A4.	A RESISTIVE NETWORK PROGRAM		171
	A4.1	Introduction	171
	A4.2	A Description of the Program	173
	A4.3	The Resistive Network Program	174
	A4.4	A Set of Input Data for the Program	175
	A4.5	The Algol 68-R Resistive Network Program	178

A5. A TRIANGULAR CONSTANT STRAIN ELEMENT PROGRAM 180

A5.1 Introduction 180
A5.2 A Description of the Program 181
A5.3 The Triangular Constant Strain Element Program 182
A5.4 A Set of Input Data for the Program 183
A5.5 The Algol 68-R Constant Strain Element Program 186

A6. A GEOMETRICALLY NON-LINEAR PLANE FRAME PROGRAM 188

A6.1 Introduction 188
A6.2 A Description of the Program 190
A6.3 The Geometrically Non-linear Plane Frame Program 193
A6.4 An Example of the Use of the Program 195
A6.5 The Algol 68-R Geometrically Non-linear Frame Program 197

REFERENCES 201

ACKNOWLEDGEMENTS

To the Science Research Council for providing financial support.

To the staff of the Computer Unit of The City University for their help in using the computer system.

Finally, to my supervisor Professor V. E. Price for his continuing help and encouragement and in particular for his assistance during the preparation of this thesis.

DECLARATION

The City University Librarian is granted powers of discretion to allow this thesis to be copied in whole or in part without further reference to the author. This permission covers only single copies made for study purposes, subject to normal conditions of acknowledgement.

ABSTRACT

Considerable use of finite element methods is currently made in obtaining numerical solutions of problems in mathematics, science and engineering. The computation is often performed using a software package specially designed for a class of applications or by using a program, written in a general purpose high-level language such as Fortran, for a particular problem.

In an environment of research into finite element methods, it is often desirable to investigate different algorithms for solving a problem. To make such investigations one cannot usually modify any of the programs in a package and so a large number of Fortran programs have been developed for such problems. There is, however, an underlying framework common to finite element methods and a language such as Fortran does not specifically assist the programmer in the task.

A language, which is based on Algol 68, has been developed which facilitates the programming of a wide variety of finite element methods. A series of extensions to Algol 68, which allow a concise representaion of finite element solution algorithms, is described in the thesis. Particular emphasis is given to iterative methods for non-linear problems, though linear and eigenvalue problems are also considered.

An implementation of the language features is reported. This utilises a preprocessor written in the general purpose macro processor ML/1 which converts programs written in the language into Algol 68.

NOTATION

The following notation is used in the thesis:

Description	Examples
Scalar	x
Column vector	$\underline{y}$
Transpose of vector	$\underline{y}^T$
Element of vector	y_i
Matrix	M
Transpose of matrix	M^T
Inverse of matrix	M^{-1}
Element of matrix	m_{ij}
Summation	$\underline{sum}_i\ y_i$
	$\underline{sum}_i^{1,n}\ y_i$
Partial differentiation	$D_x z$
	$D_x \underline{y}$
Iteration number	$\underline{y}^n$
Numerical comparison	$\underline{lt}$
	$\neq$

Chapter 1

INTRODUCTION

1.1 Special Purpose Languages

The main topic of this thesis is the design and implementation of a special purpose programming language with which programs for solving problems using the Finite Element Method[1-5] may be written. Before considering the particular requirements for such a language, however, it would seem appropriate to review some of the reasons which are generally proposed as justifications for the development of a special purpose language[6,7].

Many of the application areas of computers result in the production of programs within which a significant number of common features are present. A special purpose language for an area represents an attempt to factor out these features as special constructs, data types or operations which allow more compact and readable programs to be produced. Early examples of this process are Cobol for business applications and Fortran for scientific work. A language based on concepts appropriate to an application area can remove much of the difficulty of programming by allowing a concise representation of the distinctive facets of a particular problem. It is to be hoped that a good special purpose

language can effectively both mirror and guide the thoughts of the programmer. The removal of the tedium and inefficiency of programming standard housekeeping operations by allowing them to be performed automatically provides additional weight for the argument in favour of special purpose languages when the potential user population and application area are appropriate.

Two major types of special purpose language may be distinguished. In the first class may be put those languages which do not attempt to provide the power of a general purpose programming language. Instead, a set of commands appropriate to the intended area of application is provided. This type of language usually has a simple syntax, often using a verb driven command structure. APT[8], which is used for writing programs to control machine tools, is a language of this class. In its most elementary form, this type of language may simply be a flexible set of input formats to a program or package.

The second class of languages contains those which provide the facilities of a general purpose programming language in addition to the special application oriented features. These languages are often in the form of extensions to existing languages. As an example, Simula[9] is an extended version of Algol 60 which is intended for use in writing programs to perform discrete event simulation. Of course,

many languages such as Lisp[10] do not fall neatly into this classification, however, it does provide a useful guide when a new language is being considered.

1.2 Software for the Finite Element Method

The finite element method is essentially a computational technique which requires the use of a digital computer for its application. Suitable software must therefore be available before the method can be put into practice. Three possibilities arise when the finite element method is to be used in the solution of a particular problem. In the first case, an already existing program is available which can be used to solve the problem. This may be either a special purpose program or one of the large scale general purpose packages. In the second case, software already exists but it requires some modification before it can be used to solve the problem. Finally, a completely new program must be produced. This later possibility can prove time consuming and inefficient. There has been a tremendous amount of research into the finite element method in recent years. Much of the practical work associated with this activity will have been concerned with the development of suitable computer programs, with much consequent duplication of effort.

The large scale finite element systems such as ASKA[11], ASAS[12] and NASTRAN[13] have their main motivation in the solution of structural analysis problems arising in the aero-space and civil engineering industries. The

facilities provided by them are therefore adapted to the requirements of these industries. A wide range of element types are available togther with suites of programs which aid the production of input data and the post-processing and presentation of results. These packages are often very large and complex programming systems whose developement required considerable effort. Indeed, the current ASKA system has involved more than one hundred man years in its production and maintenance[14]. These general purpose systems are very powerful and provide a wide range of facilities. Solution algorithms for non-linear and dynamic problems are often available and some provision for the definition of new element types may be present. The methods for controlling the operation of the programs vary. ASAS is designed for use by engineers with little knowledge of computing and is driven by its input data. In comparison, the user of ASKA must provide a Fortran "steering program" which consists mainly of calls to subroutines of the system. NASTRAN is controlled by a special command language. The complexity of these systems, however, can be a considerable if not impenetrable barrier to a user wishing to perform an analysis outwith the current capabilities of the system by modifying the programs.

The language to be described in this thesis represents an attempt to provide a flexible definition capability

which will allow non-standard finite element solution algorithms to be easily described and implemented. It is therefore aimed mainly at a research or teaching environment, though it is also intended that the facility could provide the basis of a general purpose system. Two systems designed for use in similar environments are FINEL[15] and FINITE[16]. These systems, however, are more concerned with the compact definition of complete problems than with the description of solution algorithms and therefore they have different capabilities. Thus, they provide flexible facilities for defining the mesh and element types to be used to solve a particular problem. They also allow the definition of new element types, but they have essentially fixed solution methods.

The inherent computational complexity of many finite element solution algorithms dictated that the capabilities sought for the language could be best provided as an extension to an existing high level language. Algol 68[17] was selected to be the basic language. This choice was made largely on the grounds of personal preference. Any of the standard languages suitable for scientific programming could have been used to provide a similar basis.

1.3 Implementation Considerations

Three major methods of language implementation may be identified. These are interpretation, compilation and preprocessing into another language. Efficiency considerations tend to rule out an interpretive approach for a language of the type described in the previous section. Finite element solutions often involve large computing times. Though much of this time will be spent in data handling and in the solution of systems of linear equations, both of which may not be greatly affected by the method of implementation since they will be performed by standard system routines, the calculation of the element stiffness matrices could involve a considerable overhead if performed interpretively.

The development of a compiler would have proved excessively time consuming and would have been essentially unrelated to the other parts of the work. Preprocessing was therefore chosen. It is worth noting some of the advantages and disadvantages of this approach.

Unless full syntax and type checking is performed on the source program by the preprocessor, errors will be discovered by the compiler for the target language. Similarly, the run time error messages generated will be related to the target code. The provision of software to interpret

these messages and relate them to the source code, though feasible, would seem somewhat excessive, and therefore, the intermediate form of the program cannot be hidden from the user. If the original program was in an extension of the target language, this should prove no real problem in practice.

An advantage of preprocessing can be transportability. If the preprocessor and the other programs and procedures of the system are written in standard languages, the system can be moved to any machine on which compilers for these languages exist.

1.4 The Structure of the Thesis

This thesis represents an attempt to produce a small set of basic ideas within which finite element solution algorithms may be considered and to show that a programming system based on these concepts will allow many standard and non-standard algorithms to be easily implemented. Of course, methods which are not suitable for implementation by the system are easily found, however, the aim throughout the work has been to demonstrate a practical system which can be used in most contexts rather than attempt an impracticable universality.

The enormous literature of the finite element method precludes any really comprehensive treatment of the computational algorithms which are used and so some selection has had to be made in the choice of examples to be treated. These have been chosen to demonstrate the utility and scope of the system. Though many of the examples have been taken from the field of structural analysis, the solution processes associated with other application areas are very similar.

Chapter 2 presents a general description of the finite element method and defines certain design criteria for a finite element programming language. In the following chapter, a solution method for the sets simultaneous

linear equations which arise from finite element formulations is considered and is shown to be a suitable method on which to base a flexible system. In Chapter 4, a series of extensions to Algol 68 are described which are intended to satisfy the criteria set out in Chapter 2, while the next chapter shows how these features may be used to solve some standard problems. The specialised topic of the solution of eigen-problems is considered in Chapter 6. A suitable solution method is described and its use is illustrated. Chapter 7 treats the input and output requirements of a finite element solution system. The complete solution process as related to structural design is the concern of Chapter 8 which includes a discussion of the use of optimisation techniques. Chapter 9 deals with implementation aspects. Finally, some conclusions are drawn in Chapter 10 and the scope for further work is indicated.

Appendices 1 and 2 enlarge on two points discussed in Chapter 3. A formal syntax of the language is given in Appendix 3. Examples of complete programs and their use can be found in Appendices 4-6.

Chapter 2

THE FINITE ELEMENT METHOD

2.1 Introduction

The finite element method[1-5] originally arose out of a generalisation of matrix methods of structural analysis[18] to problems of elastic continua. The method was later realised to be equivalent to the application of the Rayleigh-Ritz method for finding approximate solutions to variational problems when the basis functions of the approximation are piecewise continuous functions. In practice, the approximating functions are often piecewise polynomials. The method has since been extended to encompass alternative variational formulations. Norrie and de Vries[5] describe how many problems outside of the field of structural analysis, for example, those of fluid dynamics or electrostatics, may be given variational formulations and solved by use of the finite element method. In addition, non-linear problems such as elastoplasticity or problems which give rise to eigenvalue problems such as undamped structural vibration may now be solved.

In this chapter, the computations involved in solving finite elements problems on a digital computer are

introduced. The assembly process whereby the set of equations defining the properties of a complete structure may be formed from the sets of equations defining the individual components is described in Section 2.2 and is illustrated using the example of a simple two-dimensional pin-jointed truss. The basic computational operations of the complete solution process for a standard finite element problem are outlined in Section 2.3. Section 2.4 describes the modifications to these operations which are required for the solution of non-linear or eigenvalue problems. Lastly, Section 2.5 proposes a framework for a flexible finite element programming system.

2.2 Stiffness Matrices and the Assembly Process

The general pattern of the solution procedure involved in applying the finite element method may be introduced by a simple example. The problem to be considered is the 4 element pin-jointed truss structure illustrated in Figure 1. Nodes 1 and 4 are fixed and a force P is being applied at node 3. The structure is treated as an assemblage of elements of the type illustrated in Figure 2.

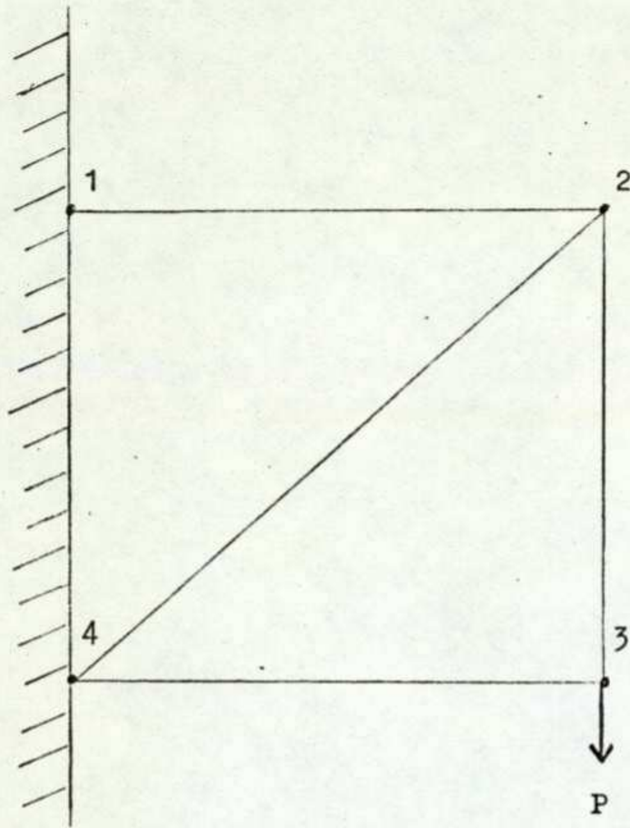
Let the modulus of elasticity and the cross sectional area of the element of Figure 2 be represented by E and A respectively and let:

$$s = \sin a = (y_j - y_i) / L \qquad (2.1)$$

$$c = \cos a = (x_j - x_i) / L \qquad (2.2)$$

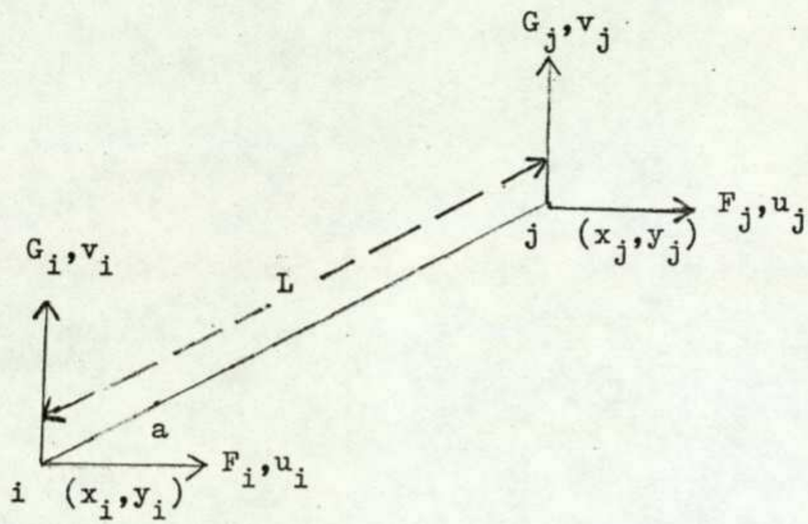
$$L = \underline{\mathrm{sqrt}}\ ((x_j - x_i)^2 + (y_j - y_i)^2) \qquad (2.3)$$

Then, for small displacements, the nodal displacements (u_i, v_i) and (u_j, v_j) may be related to the nodal forces (F_i, G_i) and (F_j, G_j) by the relationship:



A SIMPLE PIN-JOINTED TRUSS STRUCTURE

Figure 1



A SINGLE MEMBER OF A PIN-JOINTED TRUSS STRUCTURE

Figure 2

$$\begin{bmatrix} F_i \\ G_i \\ F_j \\ G_j \end{bmatrix} = EA/L \begin{bmatrix} c^2 & cs & -c^2 & -cs \\ cs & s^2 & -cs & -s^2 \\ -c^2 & -cs & c^2 & cs \\ -cs & -s^2 & cs & s^2 \end{bmatrix} \begin{bmatrix} u_i \\ v_i \\ u_j \\ v_j \end{bmatrix} \qquad (2.4)$$

This may be written as:

$$\underline{f}_e = K_e \, \underline{d}_e \qquad (2.5)$$

where K_e is the element stiffness matrix.

If the element stiffness matrix is written with respect to the nodal freedoms of a complete structure, the expanded form of the element stiffness matrix, K_x, is obtained. As an example, for the element from node 2 to node 4 in the structure of Figure 1, where $s = c = 1 / \underline{sqrt}\ (2)$, this matrix is given by:

$$K_x = \frac{EA}{2L} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & -1 & -1 \\ 0 & 0 & 1 & 1 & 0 & 0 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & -1 & 0 & 0 & 1 & 1 \\ 0 & 0 & -1 & -1 & 0 & 0 & 1 & 1 \end{bmatrix} \qquad (2.6)$$

For a complete structure, the force displacement relations are:

$$\underline{f} = K\,\underline{d} \qquad (2.7)$$

where K is the structural stiffness matrix and $\underline{f}$ is the vector of forces applied to the structure. K is formed by summing the expanded element stiffness matrices. The term assembly is used to describe the formation of the structural or global stiffness matrix from the element stiffness matrices. For the example of Figure 1, this equation is:

$$\begin{bmatrix} q_1 \\ r_1 \\ 0 \\ 0 \\ 0 \\ P \\ q_4 \\ r_4 \end{bmatrix} = \frac{EA}{2L} \begin{bmatrix} 2 & 0 & -2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -2 & 0 & 3 & 1 & 0 & 0 & -1 & -1 \\ 0 & 0 & 1 & 3 & 0 & -2 & -1 & -1 \\ 0 & 0 & 0 & 0 & 2 & 0 & -2 & 0 \\ 0 & 0 & 0 & -2 & 0 & 2 & 0 & 0 \\ 0 & 0 & -1 & -1 & -2 & 0 & 3 & 1 \\ 0 & 0 & -1 & -1 & 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ u_2 \\ v_2 \\ u_3 \\ v_3 \\ 0 \\ 0 \end{bmatrix} \qquad (2.8)$$

if the value of EA/2L is the same for all the elements. The reaction forces at the fixed nodes 1 and 4 are (q_1, r_1) and (q_4, r_4).

In general, the finite element method is used to solve a variational problem defined over a continuous region with certain boundary conditions being applied[3,5]. The region is subdivided into small areas or elements. Certain points in each element are referred to as nodes. Within each element, the unknowns of the problem are expressed in terms of the values of the unknowns at the nodes by means of interpolation formulae. The elements of the region are connected together by having nodes in common. For a structural analysis problem, the unknown quantities could be the displacements of the points of the structure and the problem would then be formulated in terms of the nodal displacements.

If, for example, the variational problem is defined by a quadratic functional, it may be shown[3,5] that using a finite element discretisation to obtain an approximate solution leads to a set of linear equations of the form:

$$K \underline{x} = \underline{y} \qquad (2.9)$$

The vector $\underline{x}$ represents the values of the unknowns of the problems at the nodes. The global "stiffness" matrix K may be formed by assembling the element "stiffness" matrices. If the underlying equations governing the system are linear, other variational formulations can also lead to such a set of linear equations.

The boundary conditions of the original problem must be expressed as constraints on the values of the terms of the vectors $\underline{x}$ and $\underline{y}$ which must be satisfied when the system (2.9) is solved. These constraints may be similar to the applied force and the fixed nodes of the pin-jointed truss of Figure 1 or they may take a more general form. For example, in a structural analysis problem, a distributed force may be present. This must be converted to a set of equivalent nodal forces before it can be applied[1] . After the system of equations (2.9) has been solved, the values of the unknowns within the elements may be determined. In addition, further secondary unknowns may now be evaluated. In a structural analysis problem, these could be the stresses and strains within the elements which may be determined once the displacement field is known.

2.3 The Basic Solution Process

The main operations which take place in the solution of a finite element problem giving rise to a system of equations of the form (2.9) on a digital computer may be listed as follows:

1. Input the data describing the problem. This consists of nodal properties, element properties and nodal connection data and constraints on the values of the problem unknowns at the nodes.

2. Calculate the element stiffness matrices and assemble the global equations (2.9)

3. Solve the global equations.

4. Evaluate the resultant internal states of the elements.

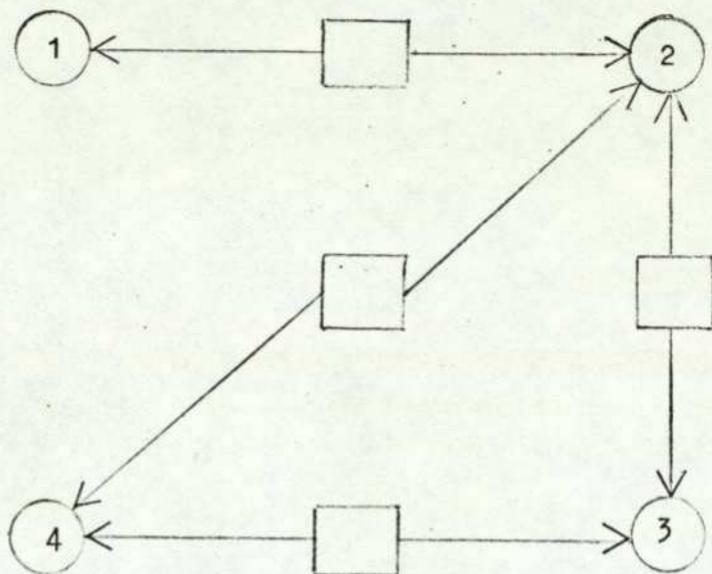
5. Output the results.

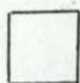
In practice, some of these operations may take place concurrently. Thus, items 4 and 5, the calculation and output of results may proceed together. Indeed,

in the solution method for the system of equations (2.9) which is described in Chapter 3, it will be seen that the assembly and the solution of the equations are overlapped. In Sections 2.4 and 2.5, the modifications to this process which are required for the solution of non-linear or eigenvalue problems are considered.

The program and data structures used in this solution process can be described in terms of an abstraction of a finite element idealisation of a system. The system is viewed as consisting of a set of nodes at which certain numerical values are known and at which certain unknown values are to be determined, and a set of elements which are defined by numerical data, the nodes to which they are connected and the procedures to perform such actions as the calculation and assembly of the element stiffness matrices. This view of the problem illustrated in Figure 1 is shown in Figure 3 where the nodes of an element are indicated by arrows.

These data and program structures provide a basis for performing the 5 steps listed at the start of this section to which standard facilities for data input and output and for the solution of the global equations must be added.





Element - E, A, "node 1", "node 2",
procedure

Node - x, y, u, v, F, G

THE COMPUTER MODEL OF THE TRUSS STRUCTURE OF FIGURE 1

Figure 3

2.4 Additional Computational Features

2.4.1 Large finite element systems such as ASKA[11], ASAS[12] and NASTRAN[13] provide a wide range of options which allow the use of many different element types and properties in the solution of particular problems. A structural analysis formulation may make use of such features as nodal temperatures, or distributed loads such as surface pressures and centripetal forces. The framework described in the previous section of node and element data and element procedures can encompass this type of facility. The necessary changes to the set of global linear equations which may be needed to represent these conditions can be achieved if element procedures can access the global vectors and matrices and the nodal data items and if they contain the coding required to calculate and apply the necessary modifications. This point is considered further in Section 5.2.

2.4.2 Undamped structural vibration problems[1,2,19] can lead to formulations where the set of global equations takes the form of the eigenproblem:

$$K \underline{x} = w^2 M \underline{x} \qquad (2.10)$$

In this equation, K is the structural stiffness matrix and M is the structural mass matrix which is formed by

assembling element mass matrices in the same manner as element stiffness matrices are assembled to form K. The eigenvalues of equation (2.10), $(w_1^2, \ldots w_n^2)$, give the natural frequencies of free vibration of the structure, $(w_1, \ldots w_n)$. The corresponding eigenvectors are the normal modes of vibration of the structure.

The sequence of operations described in Section 2.3 for the solution of linear static problems requires a slight modification for the solution of this type of problem. The only major difference, however, is the global equations take the form (2.10). Suitable solution methods for sets of equations of this form are discussed in Chapter 6.

2.4.3 Another structural analysis problem which gives rise to an eigenproblem is the investigation of structural stability[1,2,19] where for suitable problems a set of equations of the form:

$$K \underline{x} = l K_s \underline{x} \tag{2.11}$$

where K_s is the "initial stress" matrix can be derived. K_s is a function of the stressing state of the structure.

The analysis for problems of this form takes place in two stages. In the first stage, a linear static analysis of the structure is performed for a given loading pattern.

This is used to determine the stressing state of the elements. The element initial stress matrices are then calculated and they are assembled to form K_s. The lowest eigenvalue 1 of equation (2.11) is then determined and this gives the buckling load of the stucture.

2.4.4 For structural analysis problems involving geometric and/or material non-linearity[20,21,22], the governing system of global equations may be expressed as:

$$K \underline{d} = \underline{f} + \underline{g}(\underline{d}) \qquad (2.12)$$

where $\underline{g}(\underline{d})$ represents "pseudo-forces" which are present due to non-linearity. Many different methods have been suggested for the solution of this set of simultaneous non-linear equations. A large number of these methods, however, lead to very similar computational algorithms which involve iteratively setting up and solving sets of global linear equations. These methods are described in Chapter 5, where it is also shown that problems of structural reanalysis may be similarly posed.

2.5 Requirements for an Extensible System

Finite element solution methods are naturally described in terms of operations at two levels. At the first level are the calculations which are associated with elements. These include, for example, the formation and assembly of the element stiffness matrices to form the global stiffness matrix and the further functions which may be necessary to deal with the problems described in the previous section. The second set of operations performs the global solution algorithms which may be specified in terms of the element calculations and operations performed on the global data items. For a simple linear problem, this involves the assembly and solution of the global stiffness equations and the determination of the resultant element states. In contrast, for the iterative algorithms for the solution of non-linear problems which were discussed in Section 2.4.4, the description of the global solution algorithm must specify the setting up and solution of the sets of linear equations and the control of the iterations.

Some desirable features for a language to be used to describe finite element calculations may now be proposed. Firstly, it should be possible to define the node and element data items and the global quantities, which were outlined in Section 2.3. Certain of these items will be required as input data. The language must then

allow the element procedures to be described in terms of these variables and data. Finally, the global solution algorithm must be specified in terms of the global data items and the operations performed by the element procedures. Due to the complexity of the operations which may be required to be performed at the two levels of the solution process, it is desirable that the power of a high level scientific programming language be available for describing these calculations. It is therefore expedient that a system should be provided in terms of a series of extensions to a standard language. Features should be included which allow easy and concise description of the node and element data and variables. Standard routines and operators should be available for the manipulation of local and global data items and to provide an interface between them. Standard procedures should also be present for recurrent requirements such as the solution of sets of global linear equations.

Chapter 3

THE FRONTAL SOLUTION METHOD

3.1 Introduction

A central feature of any finite element system is the linear equation solver. A large proportion of the computing involved in solving a finite element problem is likely to be absorbed in the solution of sets of linear equations, and the choice of an efficient method is very important. A further criterion by which a solution method must be judged is the ability to solve fairly large problems. Finite element formulations frequently require the solutions of thousands of simultaneous linear equations: this is especially true for three dimensional problems. This constraint on the selection of a suitable method makes the use of some form of backing storage mandatory on present day computers. Surveys of methods which have been developed can be found in papers by Schrem[23] and Meyer[24,25].

Of the many different methods of solution which have been described in the literature of the finite element method, two major approaches seemed likely to provide a suitable basis for a finite element system of the

type which is to be described. These were: partitioned matrix methods such as the hypermatrix scheme of ASKA[26]; or the wavefront methods described initially by Irons[27], Helen[28], and Melosh and Bamford[29]. It was felt that an implementation of the latter, that is the frontal solution method, could be produced more quickly, and so this method was finally selected. It will be seen in Chapters 4 and 5 that this choice was to dictate major aspects of the syntax of the language and the way in which programs are written in it.

Section 3.2 contains a description of the basic front solution algorithm while its implementation is discussed in section 3.3. It is then shown that the data structures and data handling techniques required for the basic method can be extended to provide further operations which are needed in a general purpose finite element system. These include the multiplication and triangular factorisation operations which are used in the eigenvalue and eigenvector solution methods considered in Chapter 6. Finally, some of the advantages of wavefront methods are discussed in section 3.7.

3.2 The Front Solution Algorithm

The frontal technique is based on Gaussian elimination without pivoting and can be used with well conditioned positive definite symmetric matrices[28]. The stiffness matrices which are produced by most finite element formulations in current use satisfy these criterea.

To solve a set of simultaneous linear equations:

$$\underline{\text{sum}}_j^{1,n} a_{ij}x_j = b_i \quad i = 1,n \qquad (3.1)$$

or alternatively:

$$A \, \underline{x} = \underline{b} \qquad (3.2)$$

by the method of Gaussian elimination, the unknowns, x_i, are eliminated one by one. The elimination of x_i is achieved by modifying the coefficient matrix A and the right hand side vector $\underline{b}$ according to the following rules:

$$a_{jk} := a_{jk} - a_{ji}a_{ik} \, / \, a_{ii} \quad j,k = i+1,n \qquad (3.3)$$

$$b_j := b_j - a_{ji}b_i \, / \, a_{ii} \quad j = i+1,n \qquad (3.4)$$

The elimination process results in the nth equation having the form:

$$x_n = b_n / a_{nn}$$

The remaining unknowns can now be determined by back substitution according to the rule:

$$x_i := (b_i - \underline{\text{sum}}_j^{i+1,n} a_{ij}x_j) / a_{ii} \qquad i = n-1,1 \tag{3.5}$$

The substitution process starts with x_n and proceeds backwards through the equations to x_1. The terms a_{ij} and b_i correspond to the modified values obtained by the application of operations (3.3) and (3.4).
If the matrix A is symmetric, (3.3) may be modified, becoming:

$$a_{jk} := a_{jk} - a_{ij}a_{ik} / a_{ii} \qquad j = i+1,n; \; k = j+1,n \tag{3.6}$$

The matrices produced by a standard finite element approach are not only positive definite and symmetric but are also sparse. The sparsity is related to the element interconnections. Considering the effect of eliminating x_i on the terms a_{jk} and b_j, two significant facts can be noted from operations (3.4) and (3.6).

1. At the time of elimination, a_{jk} and b_j need not be fully assembled provided all the other terms are.

2. If either a_{ij} or a_{ki} equals zero, a_{jk} will be unaffected by the operation. Similarly, b_j is unchanged if a_{ji} is equal to zero.

The front algorithm takes advantage of these points by alternating assembly and elimination. A variable x_i is eliminated as soon as all the elements whose assembly will alter any of the terms a_{ij} or b_i have been assembled. After the elimination of x_i, the equation:

$$\underline{\text{sum}}_j^{1,n} a_{ij} x_j = b_i$$

will play no further part in the elimination process and so the appropriate terms of A and $\underline{b}$ do not require to remain in main memory. This "row" can therefore be transfered to backing storage. Only the terms corresponding to "active" variables, that is variables x_j for which some a_{jk} or b_j has been affected by assembly but are not yet ready for elimination, need be kept in main memory. In the backward substitution phase, the reduced rows of the matrix with the corresponding terms of $\underline{b}$ are read

from backing store in the reverse order. The sets of active variables occur in the reverse order during this phase, active now indicating that the actual values of the terms x_i have now been calculated. The values of the internal states of the elements can therefore be calculated in the opposite order to that in which they were assembled, these calculations being interspersed with the back substitution.

3.3 Housekeeping and Data Structures for the Front Algorithm

An n by n symmetric matrix A can be considered as a vector $\underline{v}$ of length n(n+1) / 2. The term a_{ij} is equivalent to $v_{l(i,j)}$ where l is a function defined as:

$$l(i,j) = i(i-1) / 2 + j \quad \text{if } i \ \underline{ge} \ j$$
$$= j(j-1) / 2 + i \quad \text{if } i \ \underline{le} \ j$$

In the implementation of the front algorithm, the active portion of the coefficient matrix is held in main memory in a linear work space W of length b(b+1) / 2 where b is the maximum number of variables which are active at any point in the calculation. For each variable x_i in the set of equations to be solved it is necessary to calculate a "destination" d_i which determines where the terms associated with the variable will be placed in the work space. The terms can then be accessed while they are active by applying the function l to their destinations. Thus, if both x_i and x_j are active, the current value corresponding to a_{ij} will be held in $W(l(d_i,d_j))$. The destinations are calculated by preprocessing the elements' nodal connection data. It is possible for destinations to be fixed throughout the activation of a variable or to vary: a comparison of the two approaches is given in Appendix 1.

The set of nodes whose associated variables are active is known as the "front". As the solution process goes through the elements the number of nodes in the front can change. Advantage can be taken of this by only operating on the currently occupied positions within the work space in main memory.

With this housekeeping and data structure, the elimination operations (3.4) and (3.6) may be expressed as:

$$a_{jk} := a_{jk} - a_{ij}\, a_{ik} / a_{ii} \qquad j,k \text{ active}, \neq i; \; d_j \ \underline{\text{gt}}\ d_k \tag{3.8}$$

$$b_j := b_j - a_{ij} b_i / a_{ii} \qquad j \text{ active}, \neq i \tag{3.9}$$

Similarly, the back substitution operations (3.5) now become:

$$x_i := (b_i - \underline{\text{sum}}_j^{\text{active},\neq i}\, a_{ij} x_j) / a_{ii} \tag{3.10}$$

The calculated values x_i are put into positions in a work space for x in main memory corresponding to their precalculated destinations.

3.4 Prescribed Left Hand Sides

In the solution of the sets of simultaneous linear equations - $A \underline{x} = \underline{b}$ - which arise in the application of the finite element method, not all the terms x_i are initially unknown. Certain of the terms x_i have prescribed, usually zero, values which allow the system to be solved since the initial matrix A is usually singular. In the case of a structural stiffness matrix, these constraints remove the rigid body modes.

During the elimination process, when a prescribed x_i is reached, the following operations are performed instead of those defined by (3.8) and (3.9):

$$b_j := b_j - a_{ij} x_i^* \quad j \text{ active}, \neq i \qquad (3.11)$$

where x_i^* is the prescribed value for x_i. During the back substitution phase, the operations:

$$b_i^* := \underline{\text{sum}}_j^{\text{active}} \; a_{ij} x_j - b_i \qquad (3.12)$$

are performed instead of back substitution, where b_i was initialised to zero before the start of elimination. If the value of the unknown b_i^* is not desired, these operations may be omitted.

3.5 Multiplication

3.5.1 To multiply a vector by a matrix - $\underline{x} = A\ \underline{y}$ - the matrix A is assembled as described previously. Instead of elimination, the following operations are performed, where $\underline{x}$ has been initialised to zero:

$$x_i := x_i + \underline{\text{sum}}_j^{\text{active}} a_{ij} y_j \qquad (3.13)$$

$$x_k := x_k + a_{ik} y_i \qquad k \text{ active}, \neq i \quad (3.14)$$

3.5.2 The Gaussian elimination process is equivalent to finding two matrices L and U which are upper and lower triangular respectively such that $A = L\ U$. The diagonal terms of L are equal to 1 and if A is symmetric $l_{ij} = u_{ji} / u_{ii}$.

If the rows of a decomposed matrix are read from backing store in the order of elimination into a work vector $\underline{r}$, $\underline{x} = U\ \underline{y}$ may be calculated using the operations:

$$x_i := \underline{\text{sum}}_j^{\text{active}} r_j y_j \qquad (3.15)$$

Similarly for $\underline{x} = L\ \underline{y}$ the operations are:

$$x_k := x_k + r_k y_i / r_i \qquad k \text{ active} \quad (3.16)$$

3.5.3 Operations (3.15) and (3.16) may be combined allowing the calculation of $\underline{x} = L\ U\ \underline{y} = A\ \underline{y}$ with a decomposed matrix A. The algorithm used in this case is:

$$z := \underline{\text{sum}}_j^{\text{active}} r_j y_j$$

$$x_k := x_k + r_k z \ / \ r_i \qquad k \text{ active} \qquad (3.17)$$

3.6 Further Operations

3.6.1 The solution of sets of linear equations - $A\ \underline{x} = \underline{b}$ - by Gaussian elimination can be divided into three stages:

1. Triangular decomposition (3.8) - $A = L\ U$

2. Forward substitution (3.9) - $\underline{b} = L^{-1}\underline{b}$

3. Backward substitution (3.10) - $\underline{x} = U^{-1}\underline{b}$

Forward substitution is thus equivalent to multiplication of a vector by the inverse of a lower triangular matrix while backward substitution corresponds to multiplcation by the inverse of an upper triangular matrix.

The forward substitution algorithm for a previously decomposed matrix is:

$$b_j := b_j - r_j b_i \;/\; r_i \qquad j \text{ active}, \neq i \tag{3.18}$$

where the work vector $\underline{r}$ is being used to hold row i of U, the upper triangular factor of A.

3.6.2 A symmetric decomposition - $A = L\,L^T$ - of a symmetric matrix may be obtained if the further operations:

$$a_{ij} := a_{ij} \;/\; \underline{\text{sqrt}}(a_{ii}) \quad j \text{ active} \quad (3.19)$$

are performed on row i of A after it has been eliminated. This factorisation is the same as that obtained by a Cholesky decomposition. The use of the standard Cholesky algorithm is not, however, possible with frontal housekeeping.

3.6.3 The iterative methods for solving simultaneous linear equations of Jacobi and Gauss-Seidel[30,31] and the method of succesive over-relaxation[31,32] may be implemented by the use of combinations of the previously described operations. If the set of equations to be solved is $A\,\underline{x} = \underline{b}$, the matrix A is decomposed as follows:

$$A = L + D + U \qquad (3.20)$$

where D is a diagonal matrix, L is a lower triangular matrix with a zero diagonal and $U = L^T$.

The Jacobi iteration is defined as:

$$\underline{x}^{n+1} := D^{-1}\,(\underline{b} - (L + U)\,\underline{x}^n) \qquad (3.21)$$

This is similar in structure to the multiplication operation of (3.13) and (3.14) and may be implemented as:

$$x_i^{n+1} := (b_i - x_i^{n+1} - \underline{sum}_j^{active,\neq i} a_{ij}x_i^n) / a_{ii} \quad (3.22)$$

$$x_k^{n+1} := x_k^{n+1} + a_{ik}x_i^n \qquad k \text{ active}, \neq i \quad (3.23)$$

The Gauss-Seidel iteration is:

$$\underline{x}^{n+1} := (D + L)^{-1} (\underline{b} - U \underline{x}^n) \quad (3.24)$$

This is basically multiplication by an upper triangular matrix followed by forward substitution with matrix (D + L). With frontal housekeeping this becomes:

$$x_i^{n+1} := (x_i^{n+1} + b_i - \underline{sum}_j^{active,\neq i} a_{ij}x_j^n) / a_{ii} \quad (3.25)$$

$$x_k^{n+1} := x_k^{n+1} - a_{ik}x_i^{n+1} / a_{ii} \qquad k \text{ active}, \neq i \quad (3.26)$$

The method of successive over-relaxation is described by:

$$\underline{x}^{n+1} := (D + w\, L)^{-1} (((1 - w)\, D - w\, U)\, \underline{x}^n + w\, \underline{b}) \quad (3.27)$$

where w is the relaxation factor. When w = 1, the method reduces to the Gauss-Seidel iteration. The algorithm for this method is:

$$x_i^{n+1} := (x_i^{n+1} + w\, b_i + (1 - w)\, a_{ii} x_i^n - w\, \underline{\mathrm{sum}}_j^{\mathrm{active},\neq i} a_{ij} x_j^n) \;/\; a_{ii} \tag{3.28}$$

$$x_k^{n+1} := x_k^{n+1} - w\, a_{ik} x_i^{n+1} \;/\; a_{ii} \qquad k \text{ active}, \neq i \tag{3.29}$$

3.7 Conclusion

The front method for the solution of sets of linear equations arising from a finite element formulation has several advantages over a simple band solver[27,28]. The solution process is unaffected by node numbering since it is controlled by the element ordering and interconnections. This allows changes to meshes to be made easily and lessens the difficulty of joining sub-meshes. In comparison, a mesh to be input to a band solver requires careful node numbering to avoid having an excessively large band width. The front solution algorithm is also beneficial when elements with midside nodes are used. The elimination of variables as soon as they are assembled allows a more efficient solution in this case.

Using the basic front algorithm, there is some danger when joining sub-meshes that a temporary increase in the number of active variables may result in zero rows remaining in the front when the number of active variables has diminished. The modified front algorithm described in Appendix 1 avoids this problem.

In contrast to block partitioned matrix schemes, the front algorithm uses less complex data structures and manipulation techniques and involves less computational

overhead. For very large problems where the workspace for the portion of the global stiffness matrix corresponding to the active variables cannot fit into main memory, however, an alternative approach is required.

In this chapter the front solution algorithm has been described. It has been shown that the data structures and housekeeping needed for an implementation of the algorithm are compatible with further operations such as multiplication and algorithms for performing these calculations have been given. Together with the obvious methods for addition, multiplication by scalars et cetera, these capabilities provide a basis around which a general purpose finite element system can be built.

Chapter 4

THE LANGUAGE FEATURES

4.1 Introduction

In Chapter 2, some requirements for a flexible finite element system were discussed and it was noted that a suitable method for providing these capabilities is as an extension to a high level scientific programming language. The front solution algorithm for solving the sets of linear equations arising from finite element formulations was then described in Chapter 3. A series of extensions to the basic algorithm were presented which allow the method to be used as a basis for a general purpose system.

A series of extensions to Algol 68R[33] are introduced in this chapter. These extensions are intended to satisfy the criteria set out in Chapter 2 and are designed around the methods of Chapter 3. The syntax of these extensions is described informally. A more formal definition can be found in Appendix 3.

Section 4.2 describes the declaration of the global matrices and vectors which are required in the solution of finite element problems. Sections 4.3 and 4.4 then cover the

definition of the properties of elements and nodes. In Section 4.5 the special clauses which are used with the frontally based algorithms are considered. Operations with global matrices are the subjects of Sections 4.6 and 4.7. Lastly, the format of a complete program is given in Section 4.8.

4.2 Sysmats and Sysvecs

Two new basic modes are introduced to hold the matrices and vectors which are used in the solution of the sets of linear equations which arise from finite element formulations. These are SYSMAT corresponding to the global matrices and SYSVEC corresponding to the global vectors. Both sysmats and sysvecs are held using frontal data structures and they may only be accessed via standard operators and procedures provided in the extension.

As an example of the use of these modes, the declarations needed for the pin-jointed truss problem of Section 2.2 may be used. In this case, one matrix corresponding to the global stiffness and two vectors corresponding to the nodal force and displacement vectors are required. The declaration statements would be:

```
SYSMAT k ;
SYSVEC f, d
```

Sysmat and sysvec declarations cause space to be allocated in both main memory and backing storage to hold their associated values. The actual sizes of these areas depend on: the number of nodes in the problem to be analysed; the number of variables associated with each node; the maximum number of variables which are active

at any point during the frontal handling of the matrices and vectors. The number of variables at each node is held in a standard system variable "nvar" which is one of the program options set as described in Section 4.8. The maximum number of active variables is calculated during preprocessing of the element nodal connection data.

4.3 Type Definitions

Element properties are described using TYPE definitions. For each kind of element, a type definition defines the variables and data associated with each element and gives the number by which elements of this type are to be identified in the input data. In addition, a type definition includes a serial clause. This may describe, for example, the calculation and assembly of the element stiffness matrix for a particular kind of element.

The syntax of a type definition is:

```
TYPE integer = ( parameter list ) :
    BEGIN
    serial clause
    END
```

The integer value following the language word TYPE is the number used to specify the type of an element in the input data. Further details of the input data are given in Chapter 7.

4.3.1 The parameter list describes the data associated with an element. Within the serial clause of a type definition, these parameters are used in a similar manner to that in which the parameters of a procedure

are used within the body of a procedure. The items declared in the parameter list may be of any of the Algol 68-R modes for which transput is possible. Thus, items whose mode definition contains "REF" or "PROC" are not allowed. A further mode, POINTER, may be used to identify nodal connection data.

In addition to the mode, two further attributes may be given to an item in the parameter list. These attributes, which precede the mode of the item they qualify, are MEM which indicates that the item will not occur in the input data but will be set in the serial clause of the type definition, and VAR which indicates that the item will be given an initial value in the input data but that it may be changed. If no attribute is given to an item, it will be set in the input data and may not be changed.

As an example of a complete parameter list, a skeleton type definition for a member of the pin-jointed truss described in Section 2.2 is:

```
TYPE 1 = ([1:2]POINTER node, REAL ea) : (sec)
```

This indicates that each element is connected to two nodes and that one real value ea which is the product of the modulus of elasticity and the cross sectional area of the element is required.

If a 2 by 2 real matrix k with attribute VAR were required in a type definition, its declaration would have the form:

VAR[1:2,1:2]REAL k

4.3.2 The order in which parameters of a type definition are declared dictates their order in the input data. For a particular type of element, it may be desired that not all the items of data associated with each element appear together in the input data. For example, the nodal connection data may have been produced by a mesh generation program while the other data may have been produced separately. A further reason could be that most of the elements have standard material properties and that only a few differ. In this case, if default values for these properties can be defined, only those elements differing from this need be specified.

A more general form of a parameter list has the form:

(GROUPNO integer : parameter declarations)

The data items declared in each of these sub-lists are input together and are identified by the group number given at the start of the list.

The example type definition of Section 4.3.1 could be amended to:

```
TYPE 1 = (( GROUPNO 1 : [1:2] POINTER node ),
          ( GROUPNO 2 : REAL ea )) : (sec)
```

if the nodal connection data for the elements is to be input separately from the values of ea for each element.

4.3.3 The body of a type definition describes the actions to be taken for each element of the particular kind. In addition to using the parameters of the type definition, any identifiers which are in scope may be used. This includes nodal values such as nodal coordinates whose definition is described in Section 4.4.

A special construct, the BLOCK clause, is used to allow the procedural action taken for each element to be varied, while allowing all the properties of a type of element to be described in one place. The syntax of a block clause is:

```
BLOCK serial clause
{NEXT serial clause}
BLOCKEND
```

where the items enclosed in braces may be repeated as

many times as may be desired. The serial clauses are numbered from one in their order of occurrence in the block clause. This number relates to the parameter of PROC(INT) visit which is used to call the appropriate body of the type definition for each element of the current problem. Within a block clause, the serial clause corresponding to the parameter of visit is obeyed.

For example, for a simple element the serial clause of the type definition could have the form:

```
BEGIN
BLOCK
calculate and assemble element stiffness matrix
NEXT
calculate and output element stresses
BLOCKEND
END
```

The first serial clause in the block clause would be called prior to the solution of the global stiffness equations and the second clause would be called afterwards.

4.3.4 Communication between the global matrices and vectors held as sysmats and sysvecs and their element counterparts is achieved by means of standard procedures.

These are:

PROC(REF[,]REAL,REF SYSMAT) madd

and PROC(REF[]REAL,REF SYSVEC) vadd, vequ

Madd performs the assembly operation of element matrices into global matrices while vadd performs the assembly operation for vectors. Vequ sets the values of a global vector corresponding to the nodes of an element in its element vector parameter. The dimensions of the element vector and matrix parameters should be nvar times the number of nodes of the element, that is the number of variables per node times the number of POINTER parameters declared for the type of element. Two "modes", ELVEC and ELMAT, may be used to declare automatically dimensioned element vectors and matrices.

4.3.5 In addition to the standard Algol 68-R library of procedures and operators, a further library of operators taking operands of modes REAL, []REAL and [,]REAL may be used. These operators allow expressions involving real vectors and matrices using the standard monadic and dyadic arithmetic operators together with a few other operations. These operators, which permit a compact and readable definition of many calculations provide run-time checks of compatibility, thus detecting

certain programming errors. Since they are not implemented as in-line code, however, their use incurs certain computational overheads.

These operations are distinct from the operations with global vectors and matrices which are described in Sections 4.6 and 4.7. Examples of the use of operators from the library can be found in the programs listed in Appendices 4-6.

4.4 Node Definitions

NODE definitions are used to describe the data and variables associated with nodes. A node declaration has a similar format to the parameter description portion of a type definition, this format being:

NODE (parameter list)

The data defined in a node declaration are assumed to be known at all nodes. Different groups of data may be defined in separate node statements for clarity, group numbers being given in the same way as for groups of element data. Parameters may be of any kind which may occur in a type definition with the exception of POINTER.

As an example of a node declaration, the following statement could be used to define the nodal coordinates for a two dimensional problem:

NODE ([1:2] REAL c)

4.4.1 Within the serial clause of a type definition, it is possible to refer to the numerical quantities at the nodes of the element. Internally, the nodes of an element are numbered from one in the order of their occurrence in the input data. The nodal data is used

in the serial clause as if it had been declared with an additional dimension to that given in the node declaration. This new dimension becomes the first dimension. Thus, the nodal coordinates defined above would be used as if the statement:

```
[1:number of nodes of element, 1:2] REAL x
```

had occured within the serial clause of the type definition. For example, the second coordinate of the first node of the element would be x [1,2].

4.4.2 Since nodal coordinates are a feature of many problems, a special facility is provided to deal with them. A standard system parameter "dim" may be set as described in Section 4.8. If dim is non-zero, the following node statement is provided automatically:

```
NODE ( [1:dim]REAL coords )
```

Additionally, if dim is between one and three, the following ascriptions apply within type definitions as appropriate:

```
REAL x = coords [,1]
REAL y = coords [,2]
REAL z = coords [,3]
```

Additional checks can be provided for coordinate data such as ensuring that coordinates are given for each node and cross checking with the element nodal connection data.

4.5 Forward and Backward Loops

It has been mentioned in Chapter 3 that the use of the front method for solving the global sets of linear equations involves the interleaving of assembly and elimination. Generally, operations on frontally held data involve a pass through the preprocessed element nodal connection data element by element, this data having been augmented with the destinations of variables in the front areas and other housekeeping information. This pass may take place either in the order in which the elements were input or in the opposite direction. As each element is reached, certain operations may be performed on the element, nodal or global data. These operations may include, for example, calling the body of the type definition for the current element or performing the operations of elimination on a set of global linear equations.

Two special unitary clauses are used to pass forwards and backwards through the elements. Their syntax is:

FORWARD LOOP serial clause POOL

and BACKWARD LOOP serial clause POOL

The constructs automatically perform the global control

and data handling required for a pass through the elements. Within these clauses, operations such as the calling of the type definitions for the elements or elimination are performed by the use of special procedures.

The use of these constructs may be illustrated by an example. It is assumed that a sysmat k and sysvecs d and f have been declared as in Section 4.2. It is also assumed that a type definition with a body of the form illustrated in Section 4.3.3 has been included. The following lines of program will then perform the solution of a problem and the output of the results:

```
FORWARD LOOP
   visit(1) ;
   eliminate(k,d,f)
   POOL ;
BACKWARD LOOP
   backwardsubs(k,d,f) ;
   visit(2)
   POOL
```

In the forward loop clause, the assembly of the global stiffness equations is initiated by the call of visit. The elimination operations on the set of equations $K \underline{d} = \underline{f}$ are then performed by the call of procedure eliminate. In the backward loop, the call of

backwardsubs is used to perform the backward substitution to determine the values of the nodal displacements d. The resultant element stresses are then calculated and output by the second of the serial clauses in the block clause in the type definition which is activated by the call of procedure visit with parameter value 2.

4.6 Vector Operations

When iterative methods for the solution of finite element problems are used, there is often a requirement for calculations involving sysvecs and scalars, such as the evaluation of convergence criteria, to be made. A special unitary clause, within which operators acting on sysvecs and reals may be used, is available in the extension. This is the VEC clause which has the syntax:

```
VEC serial clause CEV
```

These clauses must be used inside one of the two types of loop clause. Within a vec clause, the usual arithmetic operators and asignments of Algol 68-R act between corresponding terms of sysvecs.

As an example, three sysvecs x, y and z are assumed to have been previously declared. It is required that the inner product of x and y be calculated and assigned to the real variable p. Additionally, z is to be set to the term by term sum of x and y. The following piece of program will perform these calculations:

```
p := 0.0 ;
FORWARD LOOP
   VEC
```

```
        p PLUS x * y ;
        z := x + y
    VEC
    POOL
```

An occasional further requirement is access to the terms of sysvecs corresponding to individual degrees of freedom. This may be achieved by declaring a real vector of length nvar which is used inside a vec clause indexed by the standard system identifier "varno". This practice is illustrated in Chapter 5.

4.7 Matrix Operations

A set of procedures to be used within loop clauses in a similar manner to procedures eliminate and backwardsubs can be used to perform all the frontal operations described in Chapter 3. In addition, procedures to perform standard operations such as saving an undecomposed sysmat for later retrieval from backing storage, clearing a sysmat or copying a sysmat are included.

Each sysmat variable is used to refer to both an active workspace in main memory and to the space in backing storage where the inactive rows are stored. The information on backing storage can only be accessed via the standard sysmat procedures. These allocate workspace in main memory to handle the sysmats for the duration of the current loop.

The provision of a full algebra of sysmats, similar to that provided for sysvecs by the use of vec clauses, is both unnecessary and impractical since, for example, the multiplication of banded matrices does not conserve bandwidth.

4.8 The Format of a Complete Program

A complete program in the extension of Algol 68-R has the format:

```
START option list
BEGIN
serial clause
END
FINISH
```

The option list is used to give values to certain global parameters and to indicate if special features are to be used. Items in the option list are separated by commas and are either single keywords or have the form:

```
keyword = value
```

Two single keyword parameters are "statistics" which causes a summary of CPU usuage and I/O transfers caused by certain standard library procedures to be included as part of the output and "data" which causes reports on the input data to be produced. Other parameters may be used if standard element types are to be included or if special procedure libraries are to be used. The parameters nvar which was introduced in Section 4.2 and dim which was described in Section 4.4.2 are set in the option list. Another

parameter "name" may be set to the program title.

Programs must satisfy the Algol 68-R requirement that the declarations of items must precede their use. This applies to type and node definitions which are implicitly used via the procedure visit. A program will generally start with the declaration of global items such as sysmats and sysvecs. This will be followed by node declarations and then type definitions. Finally, the solution algorithms will be specified using loop clauses. Of course, further Algol 68-R coding may be interspersed among these statements.

4.9 Conclusion

In Section 3 of Chapter 2 a conceptual framework for considering finite element formulations was presented. A finite element idealisation was seen to consist of a set of elements each of which is associated with a number of nodes. Both the nodes and the elements may have their properties defined by sets of numerical values. The basic solution method then consists of setting up sets of linear equations using matrices and vectors derived from the element and nodal properties and solving these using appropriate algorithms. Further calculations may then be performed to evaluate the solutions of these sets of equations in terms of the elements and nodes.

In this chapter a programming system has been described in which features have provided which correspond to these basic ideas. Apart from such specialised topics as the solution of eigenproblems which is covered in Chapter 6 or input and output which is described in Chapter 7, the extensions to Algol 68-R are intended to provide the basic finite element programming system. It remains to be shown in the following chapters that this system can be used to produce programs for a wide range of finite element problems.

Chapter 5

USE OF THE PROGRAMMING SYSTEM

5.1 Introduction

In Chapter 2 it was mentioned that there are many variations from the basic linear formulation initially presented. These vary in complexity from the treatment of body forces in a linear elastic problem to the non-linearity inherent in problems of elasto-plasticity or creep. The purpose of this chapter is to show the way in which many of these aspects may be treated within the framework of the language features described in the previous chapter.

The literature of the finite element method is now vast and so the subjects which can be covered in this chapter can only be a small sample of those possible. Reference will be made to survey papers, however, in the hope that this will increase the generality of the treatment. Most of the examples are taken from the field of structural analysis. Essentially similar program structures arise from the application of the finite element method to other fields such as the solution of partial differential equations and flow problems.

Section 5.2 considers the handling of some features which

may be present in a linear problem. The evaluation of convergence criteria, which are necessary when an iterative method of solution is used, are considered in Section 5.3. Iterative methods of solution are then introduced with the simple example of structural reanalysis in Section 5.4 while Section 5.5 contains a general discussion of iterative methods of solution for non-linear problems. Section 5.6 gives as an example, the use of an iterative technique for solving the generalised eigenproblem $A \underline{x} = w^2 B \underline{x}$. Acceleration techniques for speeding up iterative processes are described in Section 5.7.

Further examples of the use of the system can be found in the appendices. Appendices 4 and 5 give complete programs together with sets of input data for two linear problems. Appendix 6 contains a complete program and solution for a geometrically non-linear truss problem.

5.2 Variations on the Linear Theme

5.2.1 The force-displacement equations for an element to be used in the solution of linear elastic problems which was given in Chapter 2:

$$\underline{f}_e = K_e \, \underline{d}_e \qquad (5.1)$$

may be generalised to take account of both distributed loads acting on the element and of initial strains in the element[1]. The equations then take the form:

$$\underline{f}_e = K_e \, \underline{d}_e + \underline{f}_{pe} + \underline{f}_{se} \qquad (5.2)$$

where $\underline{f}_{pe}$ represents the nodal force contribution needed to balance the distributed loads acting on the element and $\underline{f}_{se}$ represents the force contributions needed to balance the initial strains present in the element. The corresponding global stiffness equations may be written in the same format. The assembled global stiffness equations then have the form:

$$\underline{f} = K \, \underline{d} \qquad (5.3)$$

The contributions from the elements due to the terms $\underline{f}_{pe}$ and $\underline{f}_{se}$ of equation (5.2) must therefore be assembled into $\underline{f}$. This may be achieved easily, however, by including

code in the type definitions to calculate the resultant forces and then using procedure vadd to perform the assembly.

5.2.2 Positionally dependent forces acting on elements such as pressure as a function of depth or centripetal force when the structure under consideration is being rotated are similarly handled. If desired, parameters controlling the calculation of these effects can be input using standard Algol 68-R facilities prior to their use within type definitions to calculate and apply their consequences.

5.2.3 The ability to perform calculations in the program outside of type definitions and loop clauses provides a means for handling a large number of special cases. One further example may be cited. In the example problem using triangular constant strain elements illustrated in Appendix 5, all the elements are of the same size and only two different orientations are used. This results in only two numerically different element stiffness matrices being produced. In this case, a gain in efficiency could be achieved if the two different element stiffness matrices were calculated before the assembly of the global stiffness equations. The type definition for the elements would now be needed only to select the appropriate pre-calculated element stiffness matrix and perform the

assembly using procedure madd. Of course, in practice, it would require the frequent occurrence of problems having such special characteristics to justify the production of a special program.

5.3 Norms and Convergence Tests

Various norms have been suggested for use in tests for the convergence of the iterative processes used in the solution of non-linear problems by the finite element method. The inner product norms of reference 20 can be calculated by the method shown in Section 6 of Chapter 4. As an example of the evaluation of the maximum modulus type of norm, the following coding will determine the maximum absolute value obtained by any term in the sysvec x. This value will be held in the real variable norm.

```
norm := 0.0 ;
FORWARD LOOP
   VEC
      IF norm < ABS x
      THEN norm := ABS x
      FI
   CEV
   POOL
```

The modified criteria described by reference 34 are calculated by first dividing each element of a vector by the maximum modulus of the elements of the vector corresponding to the same degree of freedom. A conventional norm is then applied to the resulting

vector. Two sets of calculations within separate loop clauses are required for the evaluation of these conditions. In the first loop clause the maximum absolute values obtained by the terms for each degree of freedom are calculated. If the sysvec x is being used as before, the following code will determine these maxima:

```
[1:nvar]REAL max ;
CLEAR max ;
FORWARD LOOP
   VEC
      IF max [varno] < ABS x
      THEN max [varno] := ABS x
      FI
   VEC
POOL
```

The row of reals of length nvar "max" is used with index varno as described in Section 4.6. When the loop has been left, max will hold the maximum absolute value for each degree of freedom. The second loop clause is used to calculate the norm of the modified vector. If, for example, it were desired to obtain an inner product norm, the following coding could be used:

```
norm := 0.0 ;
FORWARD LOOP
   VEC
      REAL h = x / max[varno] ;
      norm PLUS h * h
   CEV
POOL
```

In an actual program using these methods, the calculations would be made inside loop clauses in which other operations were taking place. These loop clauses could be either forward or backward loops. Thus, in the above example, the computation of max could take place at the end of the loop clause in which the sysvec x is evaluated. The calculation of the norm could then be made at the start of the following loop clause to be obeyed.

5.4 Structural Reanalysis

Structural reanalysis is important in the design of structures where the design process can involve a series of analyses which are undertaken in the search for a structure satisfying certain design criteria. The target structure may be in some sense optimum or may be simply on satisfying certain constraints. A number of different methods of reanalysis have been suggested[35-38], though Kavlie and Powell[35] have indicated that unless the design changes are very small a completely new analysis is likely to prove more efficient. The techniques used, however, provide a useful first example of iterative solution methods for finite element problems.

In structural reanalysis an initial problem:

$$K \underline{d}_0 = \underline{f} \tag{5.4}$$

has been solved and the solution of a modified problem:

$$(K + K') \underline{d}_1 = \underline{f} \tag{5.5}$$

is desired. The matrix K' represents the changes in the structural stiffness matrix due to alterations in the structure. From equation (5.5) the iteration scheme:

$$K\ \underline{d}_1^n := \underline{f} - K'\ \underline{d}_1^{n-1} \qquad (5.6)$$

can be derived. This is the "simple iteration" method of references 35 - 37. This makes use of the previously decomposed matrix K. The matrix K' need never be formed explicitly since the contributions due to it on the right hand side of (5.6) can be evaluated within the type definitions of the elements and applied by use of procedure vadd.

Under relaxation may be used to improve the convergence of simple iteration giving the following iteration[35]:

$$K\ \underline{d}_1^n := \underline{f} - K'\ (l\ \underline{d}_1^{n-1} + (1 - l)\ \underline{d}_1^{n-2}) \qquad (5.7)$$

where l is the relaxation factor.

The coding techniques required to implement these iterative algorithms are illustrated in the example program of Appendix 6 and so they will not be repeated here. The important features are the use of a previously decomposed matrix and the use of vadd to avoid assembling a matrix which is to be used only for multiplication.

5.5 Non-linear Problems

In the formulation of many problems for solution by the finite element method, the resultant global systems of equations obtained may be presented in the form:

$$K_N(\underline{d})\ \underline{d} = \underline{f} \qquad (5.8)$$

or alternatively as:

$$K\ \underline{d} = \underline{f} + \underline{g}(\underline{d}) \qquad (5.9)$$

In equation (5.8), $K_N(\underline{d})$ is the structural stiffness matrix which is a function of $\underline{d}$ due to the non-linearity of the problem. In contrast, the matrix K of equation (5.9) is constant. In this case, the effects of non-linearity are encompassed in the term $\underline{g}(\underline{d})$. Thus, the pseudo-forces $\underline{g}(\underline{d})$ may be given by the equation:

$$\underline{g}(\underline{d}) = (K - K_N(\underline{d}))\ \underline{d} \qquad (5.10)$$

Problems which can be described by these equations include geometric and material non-linearity or a combination of both [1]. Many different algorithms have been proposed for use in the solution of equations (5.8) and (5.9)[21,22,39].

A simple approach which is suggested by the form of equation (5.9) is the method of "successive approximations"[21,22]. In this method, the load is applied incrementally, as illustrated in the example of Appendix 6, and at each load increment the following iteration is performed until satisfactory convergence is obtained:

$$K \underline{d}^{n} = \underline{f} + \underline{g}(\underline{d}^{n-1}) \qquad (5.11)$$

This iteration is essentially the same as that used in the "simple iteration" algorithm for structural reanalysis presented in Section 5.4. A feature of this method is that the structural stiffness matrix K need be decomposed only once. Unfortunately, convergence has been found to be slow if the non-linearity is too great[21]. Zienkiewicz and Nayak[20,40] have presented a complete treatment for the solution of elasto-plastic problems which is based on an improved version of this algorithm where an acceleration scheme which is based on the non-linearity of the problem is adopted.

At the opposite extreme from this method is the use of the Newton-Raphson technique[20,21,22]. This method involves the calculation and inversion of the tangent stiffness matrix:

$$K_T = D_{\underline{d}} \underline{f}_u \qquad (5.12)$$

where:

$$\underline{f}_u = \underline{f} - K_N(\underline{d})\ \underline{d} \qquad (5.13)$$

$\underline{f}_u$ is the current unbalanced force. The Newton-Raphson iteration is then:

$$\underline{d}^n = K_T^{-1}\ \underline{f}_u^n + \underline{d}^{n-1} \qquad (5.14)$$

This method is used in the program in Appendix 6. It is very costly, however, in terms of computer usage to decompose a new stiffness matrix at each iteration, and so some algorithms perform this inversion less frequently.

The methods described so far attempt to find an "exact" solution of equations (5.8) and (5.9). An alternative approach is to apply the load incrementally and at each increment of the load to calculate a value for the change in the vector of nodal displacements by using the current tangent stiffness matrix. The paper by Marcal and King[41] describes the solution of problems in elasto-plasticity by this method. Further variants including "self correcting" procedures can be found in references 21 and 22.

It should be possible to describe and hence program all

of the above mentioned methods using the language features described in Chapter 4. Vec clauses together with the sysmat procedures provide the facilities for the necessary manipulation of global data while the calculation of element properties can make use of the full power of Algol 68-R if necessary. All of the communication between element data and global data falls within the prescribed limits set by the availability of the procedures vadd, vequ and madd.

5.6 A Simple Eigenproblem

Though the provision of a facility for the efficient solution of eigenproblems is the subject of Chapter 6, the calculation of eigenvalues and eigenvectors using only the previously described language features provides a useful illustration of the use of the system. The inherent complexity of elasto-plastic problems would be likely to obscure the relative simplicity of algorithm description which is possible. This problem gives a chance to display the use of additional features which are not required for the example of Appendix 6, such as procedures save and cholesky.

The problem to be considered is that of undamped structural vibration. In this case, the governing equation takes the form[1]:

$$K \underline{x} = w^2 M \underline{x} \tag{5.15}$$

where K is the structural stiffness matrix, M is the mass matrix of the structure and w is the frequency of vibration. The equation may be transformed to the standard form:

$$A \underline{y} = l \underline{y} \tag{5.16}$$

If L is the lower triangular Cholesky factor of K, then:

$$A = L^{-1} M (L^T)^{-1} \qquad (5.17)$$

and $$\underline{y} = L^T \underline{x} \qquad (5.18)$$

A simple way to obtain the eigenvalue and eigenvector of this equation corresponding to the largest value of l is the method of "direct iteration"[30-32]. This algorithm uses the iterative cycle:

$$\underline{z}^n := A \underline{y}^{n-1} \qquad (5.19)$$

$$\underline{y}^n := \underline{z}^n / \max(\underline{z}^n) \qquad (5.20)$$

If it is assumed that suitable type definitions have been provided and that the procedure call "visit(1)" causes the assembly of sysmats k and m, a skeleton program to use this method is:

```
SYSMAT k, m ;
SYSVEC yold, ynew, evec ;
REAL h, w, tolerance, norm, max ;
INT maxiter ;
C
   form sysmats m and k
   find the cholesky factors of k
   initialise the sysvecs
C
```

```
FORWARD LOOP
   visit(1) ;
   save(m) ;
   cholesky(k) ;
   VEC
      ynew := 1.0 ;
      yold := 0.0
   CEV
POOL ;
C
   the main iteration
C
norm := tolerance + 1.0 ;
max := 1.0 ;
TO maxiter WHILE norm > tolerance DO
   BEGIN
   norm := 0.0 ;
   BACKWARD LOOP
      VEC
         ynew DIV max ;
         h := ABS (yold - ynew) ;
         yold := ynew ;
         IF norm < h
         THEN norm := h
         FI
      CEV ;
      backward subs(k, yold, ynew)
```

```
      POOL ;
      FORWARD LOOP
         multiply(ynew, m, yold) ;
         forward subs(k, ynew, ynew) ;
         VEC
            h := ABS ynew ;
            IF max < h
            THEN max := h
            FI
         CEV
      POOL
      END ;
   C
      calculate the unnormalised eigenvector of
      equation (5.15) evec and the value of w
   C
   w := sqrt(1.0 / max) ;
   BACKWARD LOOP
      backward subs(k, evec, ynew)
   POOL
```

When the above coding has been obeyed, if the iteration has converged, the lowest fundamental frequency of vibration of the structure will be held in the real variable w and the unnormalised mode of vibration will be held in the sysvec evec. Two features used in the

coding which may be noted are the use of procedure multiply to perform the multiplication by the previously saved mass matrix m and the use of procedures forwardsubs and backwardsubs to perform the multiplication by the inverses of the Cholesky factors of the sysmat k.

5.7 Acceleration Techniques

5.7.1 Several techniques for accelerating the convergence of the iterative processes used in the solution of certain problems with the finite element method have been described [40,42,43]. The method developed by Zienkiewicz and Nayak[40] for the solution of problems of elasto-plastic analysis has already been mentioned in Section 5.5. The acceleration technique adopted uses the known value of the non-linear structural stiffness matrix to calculate a diagonal matrix which is used to modify an initial estimate of the increment in the nodal displacements calculated using the initial linear stiffness matrix for the structure. The computational features needed to apply this method are the multiplication of sysvecs by the non-linear stiffness matrix and their manipulation within vec clauses plus the solution of sets of linear equations using the previously decomposed linear stiffness matrix. This process, which can therefore be programmed by making use of previously decribed coding techniques, avoids the need to decompose the current structural stiffness at each iteration while it makes better use of the available information about the non-linearity of the problem than the simple iteration technique.

5.7.2 Jennings has described a more general technique for the acceleration of matrix iterative processes[42]

which is based on Aitken acceleration[30]. Jennings and Boyle have shown that this method can be useful in the context of elasto-plastic stress analysis[44]. The equations governing the acceleration used in this study were:

$$\underline{x} = \underline{x}^n + s\,(\underline{x}^n - \underline{x}^{n-1}) \qquad (5.21)$$

$$\text{where } s = (\underline{x}^{n-2} - 2\,\underline{x}^{n-1} + \underline{x}^n)^T\,(\underline{x}^{n-1} - \underline{x}^n)\,/$$

$$(\underline{x}^{n-2} - 2\,\underline{x}^{n-1} + \underline{x}^n)^T\,(\underline{x}^{n-2} - 2\,\underline{x}^{n-1} + \underline{x}^n) \qquad (5.22)$$

The accleration is applied once every three iterations. The terms $\underline{x}^n$, $\underline{x}^{n-1}$ and $\underline{x}^{n-2}$ represent the values of the basic iterates and $\underline{x}$ is the modified value. Since the evaluation of the inner products of sysvecs are required for the above calculation, two loop clauses are needed. The essential coding for the application of this acceleration technique is:

```
SYSVEC xn, xnm1, xnm2, x ;
REAL s, h, p1, p2 ;
p1 := p2 := 0.0 ;
FORWARD LOOP
   VEC
      h := xnm2 - 2 * xnm1 + xn ;
```

```
            p1 PLUS h * (xnm1 - xn) ;
            p2 PLUS h * h
         CEV
      POOL

  and s := p1 / p2 ;
      FORWARD LOOP
         VEC
            x := xn + s * (xn - xnm1)
         CEV
      POOL
```

In the above coding, the sysvecs xn, xnm1, xnm2 and x are used to represent the vectors $\underline{x}^{n}$, $\underline{x}^{n-1}$, $\underline{x}^{n-2}$ and $\underline{x}$ of equations (5.21) and (5.22)

5.8 Conclusion

While it has in no way been possible to prove that the series of extensions to Algol 68-R described in the previous chapter can be used to describe succinctly and comprehensibly a wide range of finite element problems, this chapter has demonstrated their utility. The sample programs in the appendices should also aid this contention. The ultimate test of any programming system, however, is its adoption and use by others than its designers when its general applicability can be more convincingly shown.

Chapter 6

EIGENPROBLEMS

6.1 Introduction

It has been noted in Chapter 2 that in addition to standard linear problems or problems whose solution algorithm can be formulated in terms of the solution of a series of linear problems, there is a further major class of problems whose governing equations take the form of the eigenproblem:

$$A \underline{x} = l B \underline{x} \tag{6.1}$$

where A and B are symmetric matrices and A is positive definite. An important problem of this class is the problem of undamped structural vibration.

Section 6.2 compares various methods for the solution of equation (6.1). Special consideration is given to two important methods, simultaneous vector iteration and the determinant search technique, which have been developed for use when equation (6.1) has arisen from a finite element formulation and which take advantage of the sparse structure of matrices A and B. It is concluded that simultaneous vector iteration is most

suited for inclusion in a system based on the front solution algorithm described in Chapter 3, and the method is described in detail in Section 6.3. Section 6.4 defines two standard procedures based on simultaneous vector iteration. An example illustrating the use of the procedures is then given in Section 6.5.

6.2 Some Eigensolution Methods

6.2.1 The problem of equation (6.1) may be transformed to give an equivalent standard eigenproblem:

$$C \underline{y} = m \underline{y} \tag{6.2}$$

where C is a symmetric matrix if:

$$C = L^{-1} B (L^T)^{-1} \tag{6.3}$$

$$\underline{y} = L^T \underline{x} \tag{6.4}$$

$$m = 1 / l \tag{6.5}$$

where L and L^T are the Cholesky factors of A. The standard solution algorithm for the problem of equation (6.2) is Householder reduction followed by QR iteration to determine the complete set of eigenvalues of A[30,45]. The corresponding set of eigenvectors may then be determined by inverse iteration. This method, however, is not suitable in this context since it does not take advantage of the sparsity of matrices A and B which is present in a finite element formulation. A further reason is that in many cases the complete set of eigenvalues and eigenvectors is not required. Thus, for the structural vibration problem governed by equation (2.10) of Chapter 2,

only the lowest values of w^2 may be of interest. Similarly, for the structural buckling problem described by equation (2.11), the first buckling load which is given by the lowest eigenvalue l will be of greatest interest.

6.2.2 The method of direct iteration coupled with orthogonalisation techniques for calculating the eigenvalues and eigenvectors of equation (6.1) corresponding to the lowest values of l has been used as an illustrative example in Chapter 5. In the case of closely spaced eigenvalues, however, the rate of convergence may be very slow. Anderson[19] developed a complete solution system which was essentially based on this algorithm enhanced with acceleration techniques. A further refinement included the use of the eigenvalue economiser technique[46,47] which reduces the original system of equations (6.1) to obtain the modified eigenproblem:

$$A^* \underline{z} = l^* B^* \underline{z} \qquad (6.6)$$

where the eigenvalues of equation (6.6) approximate the low valued eigenvalues of the original system. This reduction of the order of the equation system allowed iterations to be carried out completely in main memory. This approach has now been superseded by the following two methods to be described.

6.2.3 The basic determinant search technique is described in reference 48 though other variants exist[49,50]. The basis of the method is the fact that an eigenvalue l of equation (6.1) is a root of the characteristic polynomial:

$$p(z) = \underline{\det}\ (A - z\ B) \qquad (6.7)$$

This may be evaluated by performing an L U factorisation of A - z B using Gaussian elimination as described in Chapter 3. Since the diagonal terms of L are all equal to 1, the value of the determinant is equal to the product of the diagonal terms of U. Using the fact that the leading principal minors of A - z B possess the Sturm sequence property[30], various interpolation techniques and search methods allow specific eigenvalues to be determined. Once an eigenvalue l has been determined, corresponding eigenvectors may then be found by inverse iteration:

$$(A - l\ B)\ \underline{x}^{n+1} := B\ \underline{x}^{n} \qquad (6.8)$$

coupled if necessary with orthogonalisation with respect to previously determined eigenvectors. Further details of this approach can be found in the references listed above. In principle, this method could be implemented as part of the programming system being described.

6.2.4 Bathe and Wilson in a survey of methods for the solution of eigenvalue problems arising in structural mechanics[51], recommend that the determinant search technique is suitable for use in the solution of problems where matrices A and B are of narrow bandwidth and where the problem can be solved in main memory. For problems where the solution algorithm makes use of backing storage, they recommend the simultaneous vector iteration method which is the topic of the following section.

6.3 Simultaneous Vector Iteration

Many different versions of simultaneous vector iteration exist[51-56]. Only one method, that of Rutishauser[53], will be described. The other algorithms differ in details which do not affect the feasibility of implementing them within the basic programming system.

The method of Rutishauser may be used to calculate the eigenvectors and eigenvalues of:

$$A \underline{x} = l \underline{x} \tag{6.9}$$

corresponding to the p largest eigenvalues l_i, $i = 1,p$ by iterating simultaneously with p trial vectors $(\underline{x}_1 \ldots \underline{x}_p) = X$. An initial n by p matrix X^0 such that:

$$(X^0)^T X^0 = I \tag{6.10}$$

is used to start the iteration, where I is the p by p identity matrix. The following operations are then performed at each stage of the process:

$$\text{(i)} \quad Z^k := A X^k \tag{6.11}$$

$$\text{(ii)} \quad B^k := (Z^k)^T Z^k \tag{6.12}$$

(iii) Calculate the eigenvectors Q^k of B^k so that:

$$(Q^k)^T B^k Q^k = (D^k)^2 \qquad (6.13)$$

(iv) $$X^{k+1} := Z^k Q^k (D^k)^{-1} \qquad (6.14)$$

The columns of X^k are the current estimates of the eigenvectors of A and the diagonal terms of the diagonal matrix D^k are the estimates of the corresponding eigenvalues. The p by p eigenproblem of operation (iii) above is usually solved in main memory using the Jacobi method.

A standard modification to the basic sequence of operations listed above which improves the efficiency of the solution process involves performing the following operations before applying the basic sequence given above:

(v) $$Y^{k+m} := (A)^m X^k \qquad (6.15)$$

(vi) $$X^{k+m} := Y^{k+m} R^{k+m} \qquad (6.16)$$

such that:

$$(X^{k+m})^T X^{k+m} = I \qquad (6.17)$$

The orthogonalisation operation specified by step (vi) above is performed by the Schmidt process.

Further details of simultaneous vector iteration methods, including the choice of the initial matrix of trial vectors X^0 and convergence tests are given in References 51 - 56.

6.4 Two Standard Procedures

6.4.1 Two procedures based on the method of simultaneous vector iteration can usefully be added to the basic system described in Chapter 4. The first of these will solve the eigenproblem of equation (6.1) which often arises from finite element formulations. In this case, the matrix A may represent a structural stiffness matrix which will be effectively positive definite when the constraints are applied as described in Chapter 3. The procedure is:

```
PROC (REF SYSMAT, REF SYSMAT, REF[]SYSVEC,
      REF[]SYSVEC, REF[]REAL, INT) eigen
```

This procedure automatically performs the operations necessary to transform (6.1) to the standard form (6.9) and then applies the algorithm described in Section 6.3. The matrix A of equation (6.9) need never be explicitly formed. It is only used in the multiplication operations (6.11) and (6.15), which may be performed by the use of the frontal multiplication, backward substitution and forward substitution operations defined in Chapter 3.

The parameters of the procedure consist of the two matrices A and B of (6.1), the trial vectors and a vector workspace of equal dimension, the calculated eigenvalues and the

number of eigenvalues and eigenvectors required. Further parameters could be used in a similar procedure to provide control over the tolerances for the various values calculated during the operation of the procedure. Two sets of sysvecs are required because the frontal multiplication algorithm needs separate input and output vectors.

For fast convergence, it is desirable that the number of iteration vectors used should be greater than the number of eigenvalues and eigenvectors required. Bathe and Wilson[56] have used the formula:

$$q = \underline{\min}\ (2p,\ p+8) \qquad (6.18)$$

as an ad hoc method to determine the number of vectors q which should be used if p eigenvectors are desired. If the integer parameter of the procedure is set to zero, this rule can be assumed.

On exit from the procedure, the calculated eigenvalues are returned in the low numbered positions of the row of reals parameter and they are given in order of decreasing magnitude. The corresponding eigenvectors are given in the first rows of the sysvecs parameter.

6.4.2 A more flexible procedure which solves the

eigenproblem (6.9) is:

```
PROC (PROC (REF[]SYSVEC, REF[]SYSVEC), REF[]SYSVEC,
        REF[]SYSVEC, REF[]REAL, INT) gen eigen
```

In this case, the transformation of a problem to the standard form (6.9) must be performed by the user. In addition, the user must supply a procedure to perform the multiplication operations which are required by (6.11) and (6.15). This procedure will place in its first parameter, the sysvec resulting if its second parameter is multiplied by the matrix A. The procedure parameter replaces the two sysmat parameters of procedure eigen. The other parameters have the same meaning for both procedures.

6.5 The Standard Procedures in Use

6.5.1 As an example of the use of the two standard procedures, it will be shown how the problem of structural vibration, previously treated in Section 5.6, can be solved using these procedures. The governing equation for this problem is:

$$K \underline{x} = w^2 M \underline{x} \tag{6.19}$$

For both examples, it will be assumed that the following declarations have been made, where the number of eigen-vectors to be calculated is given by nev:

```
INT nwork := min (nev + 8, 2 * nev) ;
SYSMAT k, m ;
[1: nwork] SYSVEC ev1, ev2 ;
[1: nwork]REAL evals ;
```

6.5.2 The piece of program required to calculate the first nev vibration modes and frequencies using procedure eigen would then be:

```
C
   form the sysmats k and m and store
   them on backing storage
C
```

```
FORWARD LOOP
   visit(1) ;
   save(k) ;
   save(m)
POOL ;
C
   solve the eigenproblem
C
eigen(k, m, ev1, ev2, evals, nev)
```

It is assumed in the above coding that the call "visit(1)" performs the assembly of the sysmats k and m. On exit from procedure eigen, the calculated eigenvalues and eigenvectors will be held in the first nev places of evals and ev1 respectively.

6.5.3 The coding required to use procedure gen eigen is:

```
C
  define the procedure to perform the multiplication
   by matrix A of equation (6.9)
C
PROC times = (REF[]SYSVEC v1, v2) :
   BEGIN
   INT nvec = UPB v1 ;
   BACKWARD LOOP
```

```
      FOR i TO nvec DO
         BEGIN
         backward subs(k, v2[i], v2[i])
         END
   POOL ;
   FORWARD LOOP
      FOR i TO nvec DO
         BEGIN
         multiply(v1[i], m, v2[i]) ;
         forward subs(k, v1[i], v1[i])
         END
   POOL
   END ;
C
   form sysmats m and k and find the
   Cholesky factors of k
C
FORWARD LOOP
   visit(1) ;
   save(m) ;
   cholesky(k)
POOL ;
C
   solve the transformed eigenproblem
C
gen eigen(times, ev1, ev2, evals, nev) ;
```

```
C
    obtain the eigenvectors of the original
    problem defined by equation (6.19)
C
BACKWARD LOOP
    FOR i TO nev DO
        BEGIN
        backward subs(ev2[i], k, ev1[i])
        END
POOL
```

Procedure times performs the operation v1 := A * v2. On exit from the above piece of program, the unnormalised eigenvectors will be held in the first nev places of the row of sysvecs ev2. The final loop clause is needed because procedure gen eigen is used to solve the transformed eigenproblem (6.2) which is obtained by the transformations described in Section 6.2.1.

6.5.4 An example of a situation when the use of procedure gen eigen would be advantageous is when the problem of equation (6.19) is formulated using a lumped mass matrix instead of a consistent mass matrix[1,2]. In this case, the diagonal mass matrix M may be held as a sysvec. The multiplication by M can then be performed by multiplying the corresponding terms of

the sysvecs. In addition to the benefits in storage requirement and efficiency associated with the mass matrix M, there is the added gain that two sets of trial vectors are no longer necessary since sysmat multiplication is not used.

Chapter 7

INPUT AND OUTPUT

7.1 Introduction

A finite element system of the type which is being proposed should have flexible input and output capabilities which can be adapted to the needs of particular problem areas. The input data should be easily related to the basic language constructs while also allowing a natural grouping of data items by the user of a specific program.

The basic data requirements of a program are implicit in the use of the special features of the extension of Algol 68-R and their input need not be explicitly programmed. Any further data required can of course be read using the standard input facilities of Algol 68-R. The data associated with the various constructs has already been discussed in general terms in Chapter 4. In the current chapter, a more detailed description of the actual input formats is given and these are related to the corresponding language features.

Data preprocessors and postprocessors are much used in finite element analysis to ease the effort needed for

data preparation and checking or needed to present the results produced to the user in a suitable form. These facilities will be considered in Chapter 8. It is a requirement of any system that it should be compatible or easily interfaced with such programs.

In contrast, directly produced data is likely to be used in many applications. In this case, the ability to give default values to items is a useful feature. It is also valuable if suitable vetting of the data can be performed.

Some desirable features for the input and output abilities of a finite element system have been listed above. This chapter contains a description of a possible set of input formats and output procedures to be used with the system being described which is an attempt to satisfy these criteria. In Section 7.2 the general format for all sets of input data is outlined. Element and node data is then covered in Section 7.3. Sysvec data is discussed in Section 7.4. This is used to input prescribed or initial values for sysvecs, for example, prescribed nodal displacements for use in the solution of the force-displacement equations of a structure. Finally, some standard output utilities are described in Section 7.5. Examples of complete input data sets can be found in Appendices 4-6.

7.2 The General Format of the Input Data

Each group of data is preceded by a header card. This card identifies the data which is to follow. Thus, it may indicate e.g. element data group number two. Some additional parameters may be given on a header card indicating that certain options will apply to the following group of data. For example, a choice may be made between identifying sets of data by numbers or by the order of presentation. Other parameters control the checks that are to be performed on the data or select input formats other than the standard free format. Each group of data is terminated by an "END" card.

If the default option is selected, the default values are given first in the input group. Following this come the values for those data items which differ from the default.

A set of global data is the first input group. This is used to provide a run title and to initialise certain global values such as the number of elements or nodes in the current run.

7.3 Element and Node Data

The relationship between node and type declarations and their input data is best illustrated by a simple example. It is assumed that only one element type is defined in the program and that the type definition header is:

```
TYPE 3 = ((GROUPNO 1 :  1:3 POINTER p),
          (GROUPNO 2 : REAL e, v))
```

This type header, which defines two input data groups, is similar in structure to the example described in Section 4.3.2. Only one node definition is present. This uses the standard input group number zero and is:

```
NODE (REAL temp)
```

indicating that one real value temp is to be input for each node. The input data is to be for sixty-six nodes and one hundred elements. The input data for the nodes and elements, where explicit numerical values have been given and free format input is being used, could then be:

```
ELEMENTS(GROUPNO=1,NUMBERED)
1  3  1  2  3
```

```
2  3 2 4 5
      .
      .
100 3 55 65 66
END

ELEMENTS(GROUPNO=2,DEFAULT)
DEFAULT  3  10.0  0.25
31  3  8.0  0.3
52  3  8.0  0.3
END

NODES(DEFAULT)
DEFAULT  20.0
10  25.0
11  25.0
12  25.0
END
```

In the above input data, the first ELEMENTS input group is used to provide the element nodal connection data which was specified as element data group one in the type definition. The option "NUMBERED" indicates that the set of data for each element is to be preceded by the number of the element. The data for each element is given one element per line. The first item in each line is the number of the element. This is followed

by a number giving the element type: this is 3 corresponding to the number given in the type definition in the program. Finally, the three nodal pointers for each element are given.

The second ELEMENTS data group shows the use of default values. In this case, only elements 31 and 52 have values of e and v differing from 10.0 and 0.25. The use of the default facility is indicated by the parameter "DEFAULT" in the header card. The first data card in the group then gives the default values of e and v for element type 3. The following two cards give the values for elements 31 and 52.

In the third data group, the values of the nodal data for the standard input group are given. Again, the default option is used. The default value for temp is 20.0 with nodes 10, 11 and 12 having a value of 25.0 for temp.

7.4 Sysvec Data

Sysvecs are used in the solution of sets of linear equations having the form:

$$K \underline{x} = \underline{y} \quad (7.1)$$

In a structural analysis problem $\underline{x}$ would represent nodal displacements and $\underline{y}$ would represent nodal forces.
Since the basic systems of equations arising from standard finite element formulations involve a singular matrix K, it is usual to impose constraints on the values of some terms of the left hand side vector $\underline{x}$ which allow a solution to be obtained. This usually involves prescribing that certain terms of $\underline{x}$ have a value of zero, though non-zero values may also be used.

To reflect the essential difference between left hand side or "displacement" constraints and right hand side or "force" values, two distinct input groups are used. These are LHS and RHS. A further group ZEROLHS may also be used for values constrained to zero. A parameter "DEGREENO" is used if values for only a particular degree of freedom are to be input. In this way, a nodal displacement may be constrained in one direction only. Different data groups having the same group number are permitted, the values being treated internally as if

they originated from a single group.

In order to allow simple problems to be expressed easily, the input values corresponding to the unnumbered or zero input groups are automatically applied by the appropriate sysmat procedures such as eliminate and backwardsubs. The other input groups must be invoked explicitly by using further options of these procedures which allow scalar multiples of these values be applied.

RHS input groups may be used independently of the sysmat procedures to input sysvecs. Two procedures provide the interface. These are:

PROC(REF SYSVEC, INT, REAL) vec plus, vec equal

which add or assign the data values to the sysvec parameter. The integer parameter gives the number of the RHS input group to be used and the real parameter gives the scalar multiple of the values to applied.

The input of sysvec data is illustrated in Appendices 4-6. In particular, the program in Appendix 6 shows the use of procedure vec plus to apply a load incrementally.

7.5 Output

Though the basic reporting of results can be achieved by use of the standard output procedures of Algol 68-R, the provision of certain utility procedures is useful. It may be desired, for example, to group together results and data for post processing such as graph plotting or for other forms of report generation.

A helpful procedure is:

```
PROC(REF CHARPUT, [ ] INT) output
```

Standard system identifiers have been ascribed unique integer values in the standard system prelude and these identifiers are used in the row of integers parameter of the above procedure to select the output to be obtained. The identifiers include "nodes", "elements" and "sysvecs" which cause the output of all the node, element and sysvec data respectively. Two further identifiers "printer" and "punch" are used to indicate the type of output which is desired. If the option "punch" is used, the output will be in 80 column card format suitable for input to a further program. if the option "printer" is used, the output will be formatted for output on a line printer and suitable headings will be included.

In the case of element output, all the data items declared in the parameter list of the type definition will be listed. Any calculated values pertaining to elements for which output is desired should therefore be declared as MEM variables in the type definition of the element.

Further outputs which can prove useful when debugging a program or a set of data are some snapshots which can be produced as a program runs by the standard system procedures. These include listing the sets of element or node data or the current values of the sysvecs as the execution of a loop clause proceeds or the values of any element matrices or vectors which are being assembled by procedures madd or vadd. These snapshots are controlled by the global options of the input data. Of course, a certain restraint must be applied in their use since such facilities are inevitably lavish in their use of output media, especially when iterative solution methods are being considered.

7.6 Conclusion

While rarely being the most exciting aspect of system design, the input and output facilities are often one of the most important features and can prove to be the crux of system viability since it is this aspect which will ultimately affect the user of any programs produced most. The capabilities described here are adequate for small/medium problems and they can interface satisfactorily with the more complex data presentation and preparation methods for large problems.

Chapter 8

STRUCTURAL DESIGN - THE COMPLETE SOLUTION PROCESS

8.1 Introduction

Much of the initial impetus for the development of the finite element method came from the field of structural analysis. Though the method is now used in many diverse fields, this remains one of the most important application areas. So far, the discussion of the finite element method has been mainly restricted to the actual solution algorithms involved while the ancillary operations which are inevitably associated with the complete solution process have not been treated in any detail. Structural design provides a useful context in which these matters may be considered.

The design of structures is usually an iterative process involving repeated analyses in an attempt to obtain an improved design. This can be carried out in batch mode or interactively, the analyst examining the results of each solution, deciding on some possible improvements and submitting the altered design for reanalysis. Alternatively, the process of design may be partially automated with many iterations taking place without interference from the designer.

This chapter contains a brief look at the subject of structural design as it relates to the finite element programming language described in the previous chapters. Section 8.2 indicates the basic operations involved in solving a structural design problem using a standard finite element system and shows how the language may be used to provide comparable facilities. In Section 8.3, the use of interactive methods is considered and some limitations of the current system are indicated. Finally, the relationship between the techniques of optimum structural design and the language features is covered in Section 8.4.

8.2 The Basic Design Process

8.2.1 Users of proprietry finite element solution systems such as ASKA[11], NASTRAN[13] or ASAS[12] are provided with a range of elements with which a structure may be modelled. A finite element idealisation of a structure to be analysed is created using a compatible collection of these elements. The preparation of the data defining the problem may be performed entirely by hand, or it may be aided by a mesh generation program like that of Zienkiewicz and Phillips[57] or by a program written specially for the current problem. Most designs are analysed under several loading conditions and so data describing these must also be produced. The complete set of data is then assembled in the standard format of the system being used and the resulting job is then submitted to the computer. For the ASKA system, it is also required that the user prepare a short control program which calls the standard subroutines of the system. The results obtained may simply be in the form of tables. A graphic output device such as a graph plotter may provide an alternative form of output. At this point, the user may decide that the current design is capable of improvement and the above process can be repeated for the modified structure.

8.2.2 No major change in this sequence of operations

should be necessary if the language being described is used. Standard element definitions can be easily prepared and utilities for maintaining a library of these would be part of a full system. Type definitions could then be linked into a particular program via the option list at its head. These element types would be defined in terms of standard global variables declared in the system prelude included in all programs. Indeed, for the solution of a basic linear problem, a complete program could be:

```
START option list including element types
    to be used
BEGIN
solve problem
END
FINISH
```

where "solve problem" is a standard procedure which will perform the operations necessary for the solution of the problem.

8.2.3 One point which requires further mention is the handling of multiple load cases since the number of these should be determined at run time by the problem and should not be a feature of the program. A further parameter, SIZE, on the header cards of data input

groups gives the number of load cases if multiple load cases are used. Corresponding to this in the program is a monadic operator, SIZE, which takes an integer parameter, the group number, and returns the number of load cases set in the input data. This permits both rows of sysvecs and the parameters of elements and nodes to have the evaluation of their bounds delayed to run time. It is, however, convenient to insist that the use of this operator to determine the size of an array be explicit with the application of the operator taking place within the bounds part of a declaration. This allows the actual dimensions to be calculated in the "active prelude" of the program which is described in Chapter 9.

8.3 Interactive Facilities

Various parts of the design process can usefully be performed interactively for certain applications, the designer communicating directly with the computer via an interactive terminal. This often consists of some form of computer graphics terminal. The portion of the process which is done interactively can vary from a completely interactive system which can be used for small problems, to the use of the interactive capability to aid the preparation and checking of data or the examination of results.

The basic system described in Chapter 4 is well suited to providing the solution module of a semi-interactive system. In this mode of operation, a separate program would be used to examine the results of an analysis and set up the input data for a modified problem. Communication between the programs would be achieved by use of the standard input file described in Chapter 7 and a suitably formatted output file. The global control of the suite of programs would be provided by the job control language of the computer system. Of course, some delay would be likely between the definition of a modified structure and the examination of the results.

The language described in Chapter 4 is not designed to be used to implement fully interactive systems with which the effects of changes to the idealisation or loading of simple structures may be quickly investigated. This type of operation is better suited by the use of a solution algorithm which operates entirely in main memory. The system organisation described in Chapter 9 makes use of backing storage to hold both the problem description in terms of the nodes, elements and loading, and the inactive rows of the structural stiffness matrix and the elements of its associated vectors. A further problem is that the organisation required by the front solution algorithm means that any change in the mesh defining the element interconnections necessitates the recalculation of the nodal destinations, and this will affect the structure of the basic data files. Of course, this does not preclude the possibility of developing a system providing a similar user interface but designed for interactive applications.

8.4 Optimum Structural Design

8.4.1 In recent years, a large amount of research in the field of optimum structural design has taken place and many different optimisation techniques have been applied[58,59]. It seems possible, however, to divide the methods into two distinct classes with respect to the computational algorithms involved. In the first class may be included the stress ratio method of fully stressed design[60] and the optimality criteria methods[61]. With these techniques, the design improvement algorithm proceeds by local considerations which can be made at an element level. In contrast, the second class contains those methods where global considerations are used to determine the current set of design changes. These methods often require the evaluation of the gradients of the constraints with respect to the design variables.

8.4.2 The stress ratio technique of fully stressed design may be used to illustrate the type of operation which is involved in applying a method of the first class. A fully stressed design is one in which each member reaches its limiting stress under at least one of the applied loading conditions. The resulting design may not be optimal, however, the algorithm has good convergence properties and produces a good design in most cases. A simple example is given by the design

of a pin-jointed truss with the design variables being the cross-sectional areas of the members. If a_i^k represents the cross-sectional area of element i at iteration k, a new value for the area is given by the stress ratio formula:

$$a_i^{k+1} := a_i^k \underline{\max}_l \left(s_{il}^k \,/\, s_i^*\right) \qquad (8.1)$$

In the above, s_i^* represents the limiting stress in element i and s_{il}^k represents the stress in element i at design stage k if loading case l is applied.

This type of algorithm is suited to being implemented as a single program. The member cross-sectional areas would be declared with the VAR attribute defined in Section 4.3.

8.4.3 With problems of structural continua, the requirements of inter-element continuity can prevent each element from having its own design variable. This type of problem often involves much fewer design variables. Zienkiewicz and Campbell[62] give an example of a hollow gravity dam specified by three variables defining its geometrical configuration. In this case, the relationship between the individual element properties and the values of the design variables is no longer so simply defined. A different organisation for the optimisation process

is therefore required. The following sequence of operations is often adopted:

0. Assign initial values to the design variables.

1. Produce the input data to the structural analysis program corresponding to the current values of the design variables.

2. Analyse the current structure. If desired, calculate the derivatives of the constraints with respect to the design variables.

3. Evaluate the results of step 2. If the current design is satisfactory, stop. Otherwise, calculate a new set of values for the design variables and return to step 1.

Steps 1, 2 and 3 of the above can conveniently be performed by separate programs. Step 1 can be highly problem dependent. Steps 2 and 3, however, may be performed by standard programs, the structural analysis system being used for step 2, with different programs corresponding to different design tools being employed for step 3.

For example, Zienkiewicz and Campbell[62] describe how

the technique of approximation programming may be used. Using this approach, the derivatives of the constraints and the problem objective function with repect to the design variables are used to create a sequence of linear programming problems. Suitable bounds on any changes in the design variables are set, and then the solution of the linear programming problem is used to predict a set of new trial values. Other optimisation algorithms may be used in a similar manner.

8.4.4 An important feature of the structural analysis package used in step 2 of the sequence of operations proposed in Section 8.4.3 is the capability to evaluate the derivatives of the constraint functions with respect to the design variables. In a finite element formulation of a structural analysis problem, the fundamental unknowns are usually the nodal displacements and so the basic operation is the calculation of the derivatives of the nodal displacements with respect to the design variables. The further derivatives needed can then be expressed in terms of these quantities.

If the basic problem is:

$$K \underline{d} = \underline{f} \tag{8.2}$$

then, partially differentiating with respect to the

design variable x gives:

$$K\,D_x\underline{d} + D_xK\,\underline{d} = D_x\underline{f} \qquad (8.3)$$

The change in the nodal displacements $\underline{d}$ with respect to a change in x may therefore be expressed as:

$$D_x\underline{d} = K^{-1}(D_x\underline{f} - D_xK\,\underline{d}) \qquad (8.4)$$

This method of calculation avoids the need for the complete solution of a linear system of the form (8.2) for each design variable, which would be necessary if the derivatives were estimated by a finite difference method. The quantity in brackets in equation (8.4) is known as the force derivative. It can be evaluated using the methods described in Reference 62.

The calculations to determine the derivatives $D_x\underline{d}$ of equation (8.4) may be organised in two ways. In the first of these, the structural analysis package is entered twice. An initial analysis of the problem (8.2) is performed and then another program is used to calculate the force derivatives. The structural analysis program is then entered to perform a reanalysis using the previously decomposed structural stiffness matrix K and treating the force derivatives as a set of multiple loading cases.

An alternative approach could be adopted if the language described in Chapter 4 were used to develop the analysis package. In this case, the program could be designed to accept initial input data describing the affect on the basic problem of changes in the design variables. For example, this data could specify the changes in the nodal coordinates resulting from a change in the design. The number of design variables and hence the number of sets of data would be indicated using the SIZE feature described in Section 8.2.3. This organisation would allow the program used to generate the input data for the structural analysis program to have a much simpler construction since it would not need to take account of the internal characteristics of the elements used to model the structure.

If the design constraints take the form of limiting stresses within elements or groups of elements, the derivatives of the constraint functions can then be evaluated in terms of the previously calculated derivatives of the nodal displacements with respect to the design variables. The effects of design changes on the value of the objective function can usually be determined directly.

Chapter 9

IMPLEMENTATION DETAILS

9.1 Introduction

A preliminary implementation of some of the language features described in Chapter 4 has been produced. This realisation of the system is a pilot version which can indicate the feasibility and utility of such a system. Thus, while the criteria of efficiency or ease of use have not been ignored, there are many details which would require further attention in the development of a production version.

Though reference to this initial work will be made, the main purpose of this chapter is not to detail this existing system. The aim is to show how the constructs and data types of Chapter 4 may be mapped into Algol 68-R by means of a preprocessor and also to describe the ancillary programs and modules which are required.

Section 9.2 gives a general introduction to the programs of the system. Section 9.3 then discusses the file structures used for the frontally held data. The program preprocessor is described in Section 9.4. The mappings from the extension into Algol 68-R for sysvecs

and sysmats, type and node definitions, and loop and vec clauses are given in Sections 9.5, 9.6 and 9.7 respectively. Section 9.8 describes the procedure visit and block clauses. The actual Algol 68-R programs produced by the current preprocessor for the example programs of Appendices 4 - 6 are included in these appendices.

9.2 An Outline of the Current System

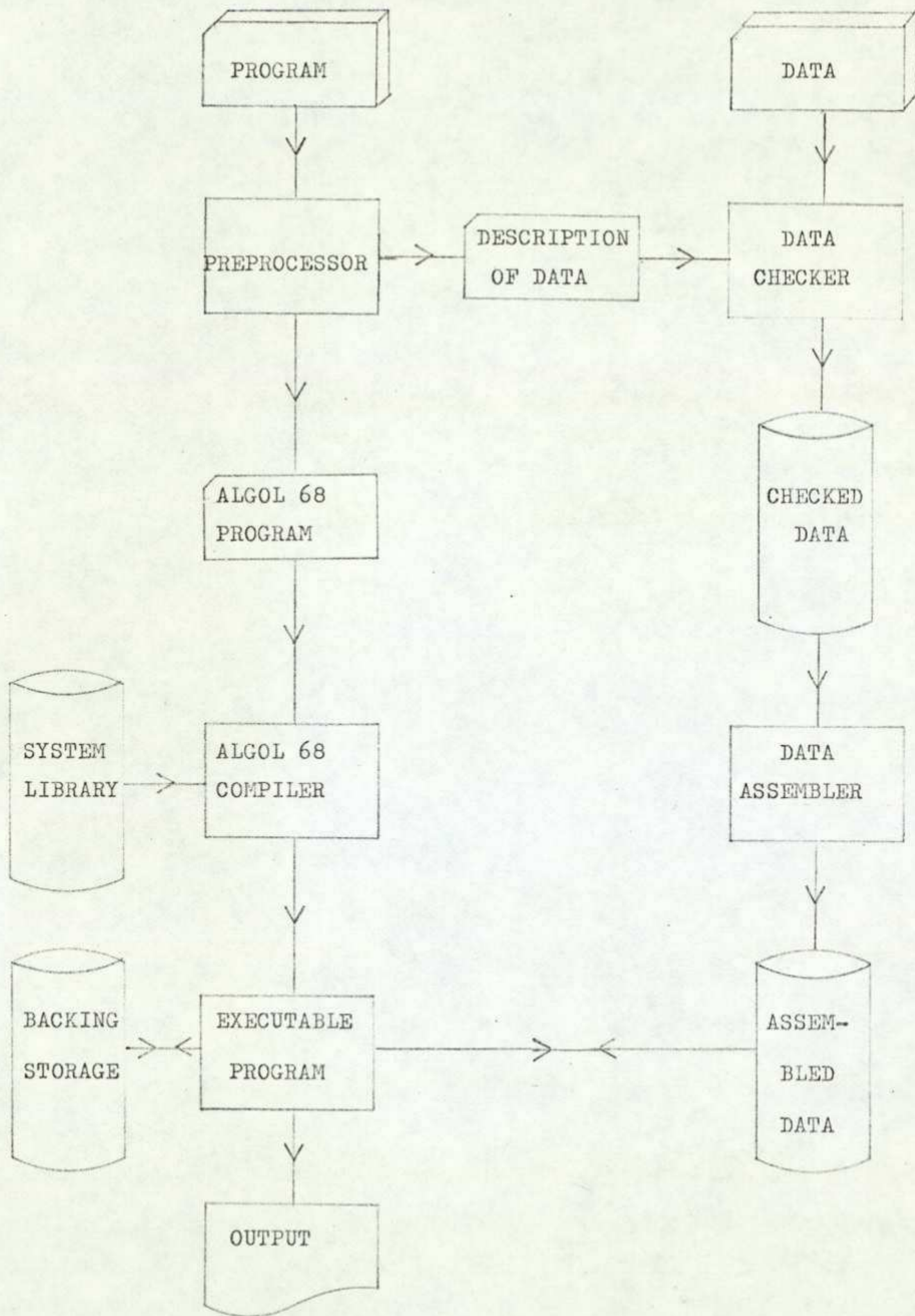
The present implementation involves the use of three separate programs together with a set of precompiled segments which are loaded with the final Algol 68-R program produced. A diagrammatic representation of the operation of the system can be found in Figure 4.

The first program to be invoked is the preprocessor which transforms the source program which is written in the extension of Algol 68-R described in Chapter 4 into an Algol 68-R program. The preprocessor is written in the language ML/1[63,64]. The preprocessor also produces a description of the types, nodes and other features which are used in the source program. Ideally, the preprocessor would be written in Algol 68-R and would perform full syntax and mode checking on the source program. With the present configuration, most of these errors will not be detected until the output of the preprocessor has been processed by the Algol 68-R compiler.

The second output of the preprocessor is used by the data checking program which operates on the input data of the system. If the tests applied by this program are satisfied, it produces an output file containing the input data in direct access format for use by the data

assembling program. The data assembler then sets up a file containing the data in a suitable format for use by the Algol 68-R program output by the preprocessor. This formatting relates to the frontal algorithms and housekeeping used, and it will be described in Section 9.3. The prefrontal operations are performed by this program.

Finally, the Algol 68-R program output by the preprocessor is compiled and loaded together with any of the precompiled segments required. These segments contain the basic run-time procedures of the system such as the equation solving routines and the data housekeeping routines together with any special operator or procedure libraries which may be used.



THE OPERATIONS OF THE COMPLETE SYSTEM

Figure 4

9.3 Frontal Data Handling

In Chapter 3, it was described how the nodes of a finite element idealisation may be divided into three classes at each stage during a pass over the elements. These classes are:

1. The nodes which have been completely processed. All the elements in which these nodes occur have been reached.

2. The active nodes. These nodes have occurred in elements which have been processed and which will recur in elements which have yet to be processed.

3. The nodes which only occur in elements which have yet to be processed.

Thus, if the operations being performed are the assembly and elimination of a set of global stiffness equations, the rows of the matrix and the terms of the vectors corresponding to nodes in class 1 have been fully assembled and have been eliminated and written to backing storage; the rows corresponding to nodes in class 2 are currently being assembled and are held in main memory; the rows corresponding to nodes in class 3 have yet to

have assembly into them commence. During the backward substitution phase, the same sets of nodes exist when the same element is being processed. In this case, the sets of nodes in classes 1 and 3 are reversed. For backward substitution, however, it is only necessary to have one row of the decomposed matrix in main memory at any stage since no assembly or elimination is taking place.

Access to all sysmat, sysvec and node data is governed by these sets. For node and sysvec data, all the values corresponding to the active nodes are held in main memory. For a sysmat, either all the active rows or only one row may be needed in main memory, depending on the operations being performed on the sysmat in the current loop clause.

The data can be arranged in direct access storage so that during a forward pass through the elements it is either read or written sequentially. Each group of data corresponding to a particular node contains a backwards pointer giving the start position in the file of the previous group. This is used during a backward pass through the elements. In the current implementation, data corresponding to node statements is read sequentially during a forward pass, having been preformatted by the data assembly program, while matrix and vector data is written sequentially. The use of backward pointers allows groups to differ in size while it is economical

in the use of space on backing storage. This is particularly relevant to rows of sysmats since the current front width can vary considerably.

In the present implementation, space is always allocated in main memory to hold the values of the node and sysvec data for the active nodes. Space for sysmats is allocated by the standard sysmat procedures used in the current loop clause and is released on exit from the loop.

9.4 The Preprocessor

The present preprocessor program functions as a series of macros which transform the new language features of Chapter 4 into Algol 68-R. The general purpose macro processor ML/1[63,64] was ideally suited to producing a preliminary implementation of this program. Of course, if more extensive checks were to be performed on the source program, this method of implementation would prove innappropriate. The preprocessor also generates descriptions of the type and node definitions occurring in the source program for use by the data inputting and vetting program.

The program output by the preprocessor is compiled by the Algol 68-R compiler and is loaded together with an "active prelude" segment in addition to the standard prelude and library segments. This segment initialises the files used by the program and performs some initial data input. The active prelude also contains the declarations of the basic global variables of the system onto which the preprocessor maps the local variables of the element types and the nodes, and the sysmat and sysvec variables. The global variables are used for data transput by the standard procedures of the system.

The use of global variables will become more apparent when the detailed descriptions of the mappings of the individual constructs are given. For clarity, the representations used will not be the actual ones used in the current implementation of the system, but they will resemble them in all essential details. Descriptive variable and procedure names will be used, avoiding the need for the explicit definition of all the names included. Examples of the mappings produced by the present system can be found in Appendices 4 and 7.

Two particular variables meriting special mention are:

[1:max number of nodes] INT pointers, destinations

which are used to hold the nodal pointers and destinations for the current element. In line with the standard convention[27,28], the sign bits of the nodal pointers are used to indicate the last occurrences of nodes during a forward pass through the elements. Similarly, a negative destination indicates the first occurrence of a node during a forward pass or the last occurrence during a backward pass. These indications are used to control the data transput procedures and the operations in vec clauses.

The various mappings will be illustrated by example. A completely general description would of necessity be of excessive length while a formal definition would require a descriptive tool of the same order as ML/1 itself.

9.5 Sysvecs and Sysmats

9.5.1 Each sysvec requires a work area in main memory of sufficient size to hold the terms for the maximum number of variables which can be simultaneously active. Also needed is an area of backing storage to hold the inactive values. The preprocessor is used to count the number of sysvecs defined in the source program while the data assembler calculates the size of work area which is necessary for each sysvec. The following declaration is made in the active prelude segment:

```
[1:no of sysvecs, 1:max no of active variables]
                                          REAL sysvecs
```

Then, for example, on encountering the declarations:

```
SYSVEC a, b ;
[1:10]SYSVEC c ;
```

in the source program, the preprocessor replaces them with:

```
REF[]REAL a = sysvecs[1,] ;
REF[]REAL b = sysvecs[2,] ;
REF[,]REAL c = sysvecs[3:12,] ;
```

Transput is performed for all sysvecs together by standard procedures called automatically from within loop clauses.

9.5.2 Sysmats are organised differently. A new mode is defined in the active prelude. This is a structure which will contain the addresses of the limits of the currently assigned work area in main memory and of the area in backing storage allocated to hold the rows of the matrix. Also included are various other items of status information for the sysmat. After preprocessing, for example, the declaration:

```
SYSMAT e, f ;
```

would become:

```
SYSMAT e, f ;
initialise sysmat((e,f)) ;
```

The procedure "initialise sysmat" takes a single parameter of mode []REF SYSMAT and it is used to initialise the status information for sysmats. The transput of sysmats is not performed automatically, but is invoked via the standard sysmat routines such as eliminate which outputs the rows of the decomposed matrix to backing storage, or backwardsubs which reads in the decomposed rows.

9.6 Type and Node Definitions

9.6.1 As described in Section 4.3, the header of a type definition gives the order and nature of the data to be read or simply held for each element of the particular type. This information is collected by the preprocessor and is passed on to the data input and checking program in a suitably coded form.

The macro expansion associated with a type definition may be illustrated by:

```
TYPE 1 = ( [1:3]REAL a, REAL b, INT c, d,
                          [1:3]POINTER p ) :
   BEGIN
   serial clause
   END
```

which becomes:

```
type procedure [1] := VOID :
   BEGIN
   REF[ ]REAL a = type reals [1:3] ;
   REF REAL b = type reals [4] ;
   REF INT c = type integers [1] ;
   REF INT d = type integers [2] ;
```

```
REF[]INT p = pointers[1:3] ;
the statements of the serial clause
END
```

9.6.2 Node definitions are processed in a similar manner to type definitions. In order to allow the simple naming convention for nodal values, two data areas are associated with nodal quantities. The first holds the nodal values for all the active nodes while the second holds the values for the nodes of the current element. The names declared in the node declaration are ascribed to the second of the two data areas. The appropriate data is copied to these variables before any element processing is performed.

As an example, the node declaration:

```
NODE([1:4]REAL e, INT f)
```

would become after preprocessing:

```
REF[,]REAL e = node real vars[1:4,] ;
REF[]INT f = node int vars[1,] ;
```

The extra dimension used in comparison with the treatment of the parameters of a type definition corresponds to the naming of the values for each of the nodes of the element.

9.7 Loop and Vec Clauses

9.7.1 Loop clauses are used to perform the inscribed statements for each element in turn. They also perform the transput for element, node and sysvec data.

A forward loop:

```
FORWARD LOOP
   serial clause
POOL
```

becomes after preprocessing:

```
initialise loop ;
forwards := TRUE ;
FOR elno TO number of elements DO
   BEGIN
   INT dummy variable ;
   element number := elno ;
   input from backing store ;
   the statements of the serial clause ;
   output to backing store
   END
```

Backward loops are treated similarly. The procedure "input from backing store" is used to input the element

data for the current element and any node or sysvec data for nodes which have become active. It also copies the node data to the element storage area. The procedure "output to backing store" is used to copy to backing store any information which may have been changed, that is sysvec data and any MEM or VAR data for nodes or elements. The nodally based data is copied when the nodes cease to be active after the current element. The boolean variable "forwards" allows coding obeyed within a loop clause to be dependent on whether the clause is a forwards or backwards loop.

9.7.2 Vec clauses are used to perform arithmetic operations between corresponding terms of sysvecs and between sysvecs and scalars. The operations for the variables of a node are applied during the processing of the element containing the last occurrence of the node. The results of the sysmat procedures become available at this stage and so this allows a natural sequencing of operations within loop clauses.

A vec clause:

```
VEC
   serial clause
CEV
```

becomes after macro expansion by the preprocessor:

```
FOR nodeno TO number of nodes DO
    IF (forwards AND pointers[nodeno] < 0) OR
       (NOT forwards AND destinations[nodeno] < 0)
    THEN INT index := (ABS destinations[nodeno] - 1)
                      * number of variables per node ;
        FOR ivar TO number of variables per node DO
            BEGIN
            INT varno := ivar ;
            index PLUS 1 ;
            the processed serial clause
            END
    FI
```

The condition of the IF statement in the above coding based on the signs of the elements of the arrays pointers and destinations is used so that the operations on the terms of the sysvecs take place when the node is making its last appearance in an element. The processing of the serial clause within a vec clause involves the addition of a first subscript "index" to all occurrences of sysvecs. For example, if [1:3] SYSVEC a and SYSVEC b occurred in a vec clause in the form:

```
b := b * a[3] ;
```

this line would be replaced with:

```
b[index] := b[index] * a[index,3] ;
```

9.8 The Procedure Visit and Block Clauses

The procedure visit is used to call the type procedure appropriate to the current element. Visit is defined as:

```
PROC visit = (INT i) :
  BEGIN
  block number := i ;
  type procedure [type number]
  END
```

A block clause is simply replaced by a CASE statement branching on the global variable "block number".

9.9 Conclusion

This chapter has contained a description of an initial implementation of the language features described in earlier chapters. This consisted of an outline of the functions of the main computer programs involved. The source program preprocessor was treated in greater detail since it is a less standard piece of software. Not all aspects of the treatment of specific language features has been covered. Full documentation for a system of this nature would be excessively lengthly in this context. One essential point is the collection of certain items of information by the preprocessor for use by the other programs of the system. This allows the housekeeping associated with data input and storage to be hidden from the user while still permitting a fairly natural representation of problems.

Chapter 10

CONCLUSION

10.1 Some Limitations of the Current System

It would not seem amiss to indicate some constraints imposed by the approach adopted, especially with respect to the method of solution of linear equations used, that is Irons' frontal method. The constructs described in Chapter 4 severly restrict the patterns of access to the global vectors and matrices. It has been shown in Chapter 5, however, that the allowable methods are adequate for many algorithms. Indeed, this restriction may be easily construed as having distinct advantages, since it helps to guide the programmer to the production of fairly efficient code, whereas free access to vectors and matrices held largely on backing storage could cause large overheads to be incurred inadvertently.

There is one particular class of problems which cannot be solved using the current programs. Some formulations require the solution of sets of linear equations with a non-symmetric matrix. For these, a variant of the front algorithm developed by Hood[65] could be incorporated into a modified system.

The use of substructures[66,67] is another operation which is not possible with the basic system. For structures which are composed of repeated components, substructuring techniques can lead to an increase in the efficiency of the solution process for linear static and dynamic analyses[68,69]. A further advantage is that substructuring can prove to be a natural way for users to model a complex structure since the specification and node numbering of the individual components can proceed largely independently. Much of this flexibility, however, can be provided by a suitably powerful data preprocessor. Some investigation was made into the feasibility of adding a multi-level substructuring capability to the basic system described in Chapters 4 and 7. It was quickly realised that a considerable increase in the complexity of the system would be necessary due to the additional data handling operations required and that implementation of these facilities would be impossible in the time available. A further justification for not including substructuring in the system is that the technique has little advantage for non-linear analyses of the type described in Section 5.5. Indeed, the method can lead to a loss of efficiency for problems of this type since it can force constraints on the order of elimination of variables in the solution of sets of linear equations which can result in a larger bandwidth.

10.2 Final Remarks

In the finite element method,as in many other fields of the application of computers, the development of new techniques has far outstripped their utilisation by the majority of practitioners and this trend seems destined to continue. Proprietary systems, while providing many features, cannot hope to be exhaustive. A prospective user of a new technique is therefore forced to make a choice between modifying existing software; producing a completely new program; or attempting to solve his problem by a method for which facilities are already available. The large size of general systems makes them relatively inaccessible to programmers other than their initial designers and implementers while the production of completely new programs can be a lengthy process which can only serve to sidetrack those whose central interests and occupations are not directly concerned with computer programming. Though many new techniques will turn out to be of little general interest or will be superseded later by more powerful approaches, it would seem to be a useful exercise to attempt to increase the speed and ease with which new algorithms can be implemented. This can apply to both active researchers and analysts interested in obtaining beter results.

It has been the aim of the work described herein to show that a flexible environment within which many algorithms may be imbedded can be provided. A particular system has been implemented and its main features have been described. There would seem to be no reason, however, why such a capability could not be based on an existing finite element system since many of the basic features are common to all standard packages. Since most of these are written in FORTRAN, a preprocessor could be used to provide an interface between programs writen in an extended FORTRAN and the basic subroutines and functions of the system. This, together with a more general method of data input, could provide the user of such a system with greater freedom than presently while the cost necessary to provide this type of facility could be a relatively minor portion of that involved in producing the basic system. For example, the ASKA linear elastic-static analysis system consisted of approximately 160,000 FORTRAN statements[14]. In comparison, the implementation described here was achieved in just over 6,000 lines of program. Of course these figures do not give an entirely unbiased picture since there is a considerable difference between a prototype and a "user-proofed" production system, however, they would seem to indicate that the effort required to provide a considerable degree of flexibility need not be prohibitive.

APPENDICES

Appendix 1

A MODIFIED FRONT SOLUTION ALGORITHM

A1.1 Introduction

The standard front solution algorithm has been described in Chapter 3. It has been noted by Irons and Kan[70] that changes in the front width can lead to the occurrence of zero rows within the front area in memory causing inefficiency in the solution process. This is a consequence of variables allocated destinations at the extremities of the area outliving variables assigned interior destinations.

Yeo[71] has presented a strategy for the allocation of destinations by longlevity considerations where short lived nodes are given the higher order destinations. This results in an improvement in efficiency but still does not completely prevent zero rows from occurring. It is shown here that a further gain in efficiency can be achieved by moving the extreme rows to fill any zero rows arising.

A1.2 <u>Dynamic Destinations</u>

The effect of eliminating row k of a matrix M on the term M_{ij} is described by the expression:

$$M_{ij} := M_{ij} - M_{ik} M_{kj} / M_{kk} \qquad (A1.1)$$

Thus M_{ij} is only altered if both M_{ik} and M_{kj} are non zero. The front method takes advantage of the fact that only M_{ik}, M_{kj} and M_{kk} need be fully summed, allowing the elimination of row k as soon as all the elements affecting it have been assembled. The element sequencing normally used with the front algorithm is such that fill-in is almost complete within the active portions of the front.

The redundant operations which arise during the elimination of a row due to the presence of p zero rows in a front of width n may now be considered. Tests for the presence of zero terms are made in the outer loop of the procedure for eliminating a row. If q is the average number of zero items occuring in a row which are not due to the presence of zero rows, an approximate value for the number of redundant operations is given by:

$$N_R \cong (n - p - q - 1)(p + q) / 2 \qquad (A1.2)$$

In comparison, the number of terms which would have to be moved if rows at the edge of the front were to be moved to fill the zero rows prior to elimination is:

$$N_M = (2n - p + 1) \, p \, / \, 2 \qquad (A1.3)$$

The number of redundant operations during elimination after compression of the front area may be estimated as:

$$N_{RC} \cong (n - q - 1) \, q \, / \, 2 \qquad (A1.4)$$

Hence, an approximate value for the number of operations avoided is:

$$\begin{aligned} N_A &= N_R - N_{RC} \\ &\cong (n - p - 2q - 1) \, p \, / \, 2 \qquad (A1.5) \end{aligned}$$

It can easily be seen, without a detailed examination of the machine operations implied, that the computing required to remove zero rows is likely to be cancelled out during the first elimination.

A1.3 Implementation Considerations

If dynamic allocation of destinations is to be implemented a choice must be made between moving a row after each elimination or moving rows on an element basis. A consideration of relations (A1.3) and (A1.5) indicates that there would be little to choose between the two methods on the grounds of efficiency since the major objective is to avoid the continued presence of zero rows.

Calculation of the moves by element is less complex and allows program modularity to be more easily maintained. The actual calculation of moves is made in the prefront routine. A move is only required if a space will remain in the front after the assembly of the next element containing the last occurrence of a variable.

Appendix 2

AN ALTERNATIVE DATA STRUCTURE FOR MATRICES

A2.1 Introduction

A basic data structure for matrices which are to be treated by frontal algorithms has been described in Chapter 3. Each row of a matrix is held for the complete active portion of the front. This structure is used in both the main memory of the computer and backing storage. If a matrix is to be decomposed by the Gaussian elimination space must be left in the row for the terms which will become non-zero due to fill-in. In contrast, if a matrix is not to be decomposed, many zero terms may remain once the assembly is complete and hence many redundant operations will occur if, for example, the matrix is used for multiplication of a vector. When repeated multiplication by a matrix occurs as in the Simultaneous Vector Iteration algorithm described in Chapter 6, this is likely to prove expensive in terms of computer usuage.

A2.2 An Alternative Data Structure

A suitable data structure for these matrices holds each term as a pair of numbers. The first number is the numerical value of the term; the second number holds the destination of the term in the current row. The computing operations to apply the basic algorithms of Chapter 3 require little modification to use this data structure, the standard routines having two modes of operation dependent on which form of matrix is being used.

While in principle the maximum number of non-zero terms occurring in each row can be calculated at the prefrontal stage allowing assembly to take place into the packed data structure, a simpler approach would seem to be advantageous. A new procedure "compress" which reconfigures a matrix can be applied after a matrix has been assembled. This method will take advantage of zero terms which would be included if the organisation of the packed data were precalculated.

Appendix 3

THE SYNTAX OF THE EXTENSION TO ALGOL 68-R

A3.1 Introduction

An informal description ot the extension to Algol 68-R has been given in Chapter 4. In this appendix, the syntax is presented using the same conventions as are used in Appendix 3 of the Algol 68-R Users Guide[33]. If these rules are added to those of Algol 68-R and if the production for "segment" is deleted, a complete syntax for the extended language will be obtained. It should be noted, however, that although these rules are defined in the usual recursive manner, only certain expansions will be meaningful. The name "optionlist" is not expanded but it has already been described in Chapter 4.

A3.2 New Modes

Five new basic modes are defined. These are:

SYSMAT

SYSVEC

ELMAT

ELVEC

POINTER

A3.3 The Syntax Rules

program

```
START optionlist (sec) FINISH
```

primary

```
FORWARD LOOP sec POOL

BACKWARD LOOP sec POOL

VEC sec CEV

BLOCK sec {NEXT sec etc} BLOCKEND
```

declaration

```
TYPE posint = typedenotation

NODE ({GROUPNO posint :} paramtype identifier-,etc--,etc)
```

typedenotation

```
(paramtype identifier-,etc--,etc):(sec)

((GROUPNO posint : paramtype identifier-,etc--,etc)-,etc):(sec)
```

posint

```
digit {digit etc}
```

paramtype

```
mode §

MEM mode §

VAR mode §
```

Appendix 4

A RESISTIVE NETWORK PROGRAM

A4.1 Introduction

Zienkiewicz[1] used the example of a resistive network as a simple introduction to some of the computational aspects of the finite element method. This problem similarly provides a useful first example to show the use of the programming system.

The voltages and the currents entering at the ends of a resistive element are related by the formula:

$$\begin{bmatrix} i_1 \\ i_2 \end{bmatrix} = 1/R \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \tag{A4.1}$$

$$= K^e \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

Using the fact that, for a complete resistive network, the total current entering the system at each node must be equal to the sum of the currents entering the elements connected to the node, the formula:

$$\underline{i} = K \, \underline{v} \tag{A4.2}$$

may be derived, where K is formed by assembling the element "stiffness" matrices K^e as described in Chapter 2.

In this appendix a program and input data to solve a simple resistive network problem are given. Slight differences can be found between the representations described in Chapters 4, 7 and 9, and the actual formats used in the programs and data in this appendix and in Appendices 5 and 6. This is simply due to small details of the initial implementation. Section A4.2 contains a description of the program. The actual program is listed in Section A4.3. The input data for the example problem is given in Section A4.4. Finally, the Algol 68-R program which results after the preprocessing of the program of Section A4.3 is listed in Section A4.5.

A4.2 A Description of the Program

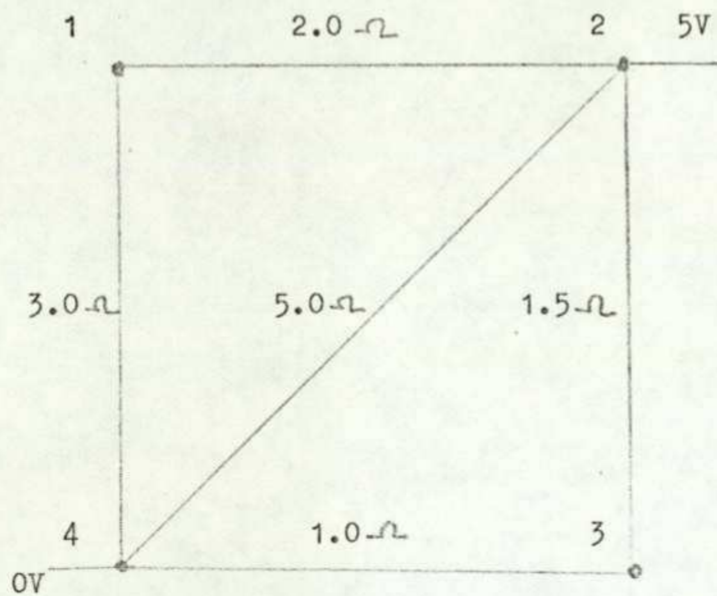
The first line of the program gives the program options. It indicates that nvar = 1, dim = 0 and that operations between real matrices, vectors and scalars are to be used. Following this, the sysmat and sysvec declarations define the global matrix K and the global vectors I and V. The type definition for the resistive element indicates that each element has two nodes and that one real value, the element resistance R, is used to define its properties. A block clause is used to give two sets of operations for the element. The first set describes the calculation and assembly of the element matrix EK. The second set performs the calculation and output of the current through the element. Finally, there are two loop clauses. The first performs the assembly and elimination of the global equation (A4.2). The second loop clause performs the backward substitution and the calculation of the resultant currents.

A4.3 The Resistive Network Program

```
'START' 1,0,MATOPS
'BEGIN'

'SYSMAT' K ;
'SYSVEC' I,V ;

'TYPE' 1 =([1:2]'POINTER' N,'REAL' R):
      'BEGIN'
      'BLOCK1' 'ELMAT' EK ;
            EK:=((1,-1),(-1,1)) ;
            EK:=EK*(1/R) ;
            MADD(EK,K)
      'BLOCK2' 'ELVEC' EV ;
            VEQU(EV,V) ;
            'REAL' C ;
            C:=(EV[1]-EV[2])/R ;
            PRINT((NEWLINE,NEWLINE,
                  "ELEMENT NO",ELEMENT NO,NEWLINE,
                  "FROM",N[1]," TO",N[2],NEWLINE,
                  NEWLINE,"CURRENT",C))
      'BLOCKEND'

      'END' ;

'FORWARD''LOOP'
      VISIT(1) ;
      ELIMINATE(K,V,I)
'POOL' ;

'BACKWARD' 'LOOP'
      BACKWARDSUBS(K,V,I) ;
      VISIT(2)
'POOL'

'END'
'FINISH'
```

A4.4 A Set of Input Data for the Program

A4.4.1 The resistive network illustrated in Figure 5 is described by the input data given in Section A4.4.2. The first data card is a comment which is followed by a card giving the number of elements in the problem, the number of nodes and the maximum node number to be used. The individual resistive elements are then defined. Since only one type of element has been defined in the program, the option NOTYPENOS is used allowing the type number field of an element definition to be omitted. The boundary conditions are defined in the LHS input group. These are analogous to constrained displacements and fix the voltages at nodes 2 and 4 at 5V and OV respectively.



A SIMPLE RESISTIVE NETWORK

Figure 5

A4.4.2 The input data is as follows:

```
#RESITIVE NETWORK DATA
INITIALISE(NELEMENTS=5,NNODES=4,MAXNODENO=4)
ELEMENTS(ORDERED,NOTYPENOS)
 1  2   2.0
 2  3   1.5
 3  4   1.0
 1  4   3.0
 2. 4   5.0
ALL
LHS(NUMBERED)
 2   5.0
 4   0.0
ALL
STOP
```

A4.4.3 The following output is obtained:

```
ELEMENT NO + 5
FROM -2 TO -4
CURRENT +1.00000&+0

ELEMENT NO +4
FROM -1 TO +4
CURRENT +1.00000&+0

ELEMENT NO +3
FROM -3 TO +4
CURRENT +2.00000&+0

ELEMENT NO +2
FROM +2 TO +3
CURRENT +2.00000&+0

ELEMENT NO +1
FROM +1 TO +2
CURRENT -1.00000&+0
```

A4.5 <u>The Algol 68-R Resistive Network Program</u>

```
'WITH' PREL , MATOPS 'FROM' PANALALGOL68
'BEGIN'

'SYSMAT' K ;
START 'OF' K:=0 ;
SYSMATS[1]:=K ;

'REF'[]'REAL' I ,V ;
I := SYSVECS[1] ;
V := SYSVECS[2] ;


ELEMENT PROCEDURE[1] := 'VOID' :
'BEGIN'
'REF'[]'INT'  N =  POINTERS [1:2 'AT' 1] ;
'REF''REAL'  R =  TYPE REAL CSTS [1] ;



'C' ******  BLOCK1  ****** 'C'
'CASE' BLOCKNO 'IN'
 [1:ESIZE,1:ESIZE]'REAL' EK ;
            EK:=((1,-1),(-1,1)) ;
            EK:=EK*(1/R) ;
            MADD(EK,K)

'C' ******  BLOCK2  ****** 'C' ,
 [1:ESIZE]'REAL' EV ;
            VEQU(EV, V) ;
            'REAL' C ;
            C:=(EV[1]-EV[2])/R ;
            PRINT((NEWLINE,NEWLINE,
                   "ELEMENT NO",ELEMENT NO,NEWLINE,
                   "FROM",N[1]," TO",N[2],NEWLINE,
                   NEWLINE,"CURRENT",C))

'OUT' FAULT(ERR1)
'ESAC'
'C' ******  BLOCKEND  ****** 'C'


      'END' ;



'C' ******  FORWARD LOOP  ****** 'C'
START LOOP(FORW) ;
'FOR' ENO 'TO' NO OF ELEMS FOUND 'DO'
'BEGIN'
```

```
'INT'ERESTING ;
ELEMENT NO := ENO ;
LOOP INPUT ;

      VISIT(1) ;
      ELIMINATE(K,V,I)
; LOOP OUTPUT
'END'
'C' ****** POOL ****** 'C'
;

'C' ****** BACKWARD LOOP ****** 'C'
START LOOP(BACK) ;
'FOR' ENO 'FROM' NO OF ELEMS FOUND 'BY' -1 'TO' 1 'DO'
'BEGIN'
'INT'ERPLANETARY ;
ELEMENT NO:=ENO ;
LOOP INPUT ;

      BACKWARDSUBS(K,V,I) ;
      VISIT(2)
; LOOP OUTPUT
'END'
'C' ****** POOL ****** 'C'

'END'
'FINISH'
```

Appendix 5

A TRIANGULAR CONSTANT STRAIN ELEMENT PROGRAM

A5.1 Introduction

The triangular constant strain element was one of the first elements to be used with the finite element method. The derivation of the element stiffness matrix is described by Zienkiewicz in Chapter 4 of Reference 1. Section A5.3 contains a program for solving plane stress problems using this element. This program is described in Section A5.2. A set of input data which corresponds to the problem given in pages 467-471 of Reference 1 is listed in Section A5.4. Lastly, the Algol 68-R program produced by the preprocessor from the program of Section A5.3 is listed in Section A5.5.

A5.2 A Description of the Program

The first line of the program indicates that nvar = 2, that dim = 0 and that matrix and vector operations are to be used. The global stiffness matrix K and the global displacement and force vectors D and F are then declared. The type definition header indicates that a three noded element requiring three real values, E the modulus of elasticity, V Poisson's ratio and T the thickness, is to be described. The following code upto the block clause is used to calculate items which are used in both the formation of the element stiffness matrix and in the calculation of the element stress after the solution of the global stiffness equations. These are the matrix B, which relates strain in the element to the nodal displacements and the elasticity matrix DM, which relates stress to strain. Also calculated is A which is twice the area of the element. The system variables X and Y hold the coordinates of the nodes of the element. The first serial clause in the block clause calculates and assembles the element stiffness matrix EK. The second clause determines the element nodal displacements U and uses this to calculate and output the resultant stress components within the element. The operator 'T' performs the transposition of a matrix. Finally, the two loop clauses perform the assembly and solution of the global stiffness equation and activate the elements to obtain the listing of the results.

A5.3 The Triangular Constant Strain Element Program

```
'START' 2,2,MATOPS
'BEGIN'

'SYSMAT' K ;
'SYSVEC' D,F ;

'TYPE' 2 = ([1:3]'POINTER' N,'REAL' E,V,T) :
      'BEGIN'
      'REAL' A, B1, B2, B3, C1, C2, C3 ;
      [1:3,1:6]'REAL' B ;
      [1:3,1:3]'REAL' DM ;
      B1:=Y[2]-Y[3] ;
      B2:=Y[3]-Y[1] ;
      B3:=Y[1]-Y[2] ;
      C1:=X[3]-X[2] ;
      C2:=X[1]-X[3] ;
      C3:=X[2]-X[1] ;
      A:='DET'((1,X[1],Y[1]),
               (1,X[2],Y[2]),
               (1,X[3],Y[3])) ;
      B:=((B1,0,B2,0,B3,0),
          (0,C1,0,C2,0,C3),
          (C1,B1,C2,B2,C3,B3)) ;
      B:=B*(1/A) ;
      DM:=((1,V,0),(V,1,0),(0,0,(1-V)/2)) ;
      DM:=DM*(E/(1-V↑2)) ;
      'BLOCK1' 'ELMAT' EK ;
               EK:='T'B*DM*B*(T*A/2) ;
               MADD(EK,K)
      'BLOCK2' [1:3]'REAL' STRESS ;
               'ELVEC' U ;
               VEQU(U,D) ;
               STRESS:=DM*B*U ;
               PRINT((NEWLINE,"ELEMENT",ELEMENTNO,
                      NEWLINE,"STRESS=",STRESS))
      'BLOCKEND'
      'END' ;

'FORWARD''LOOP'
      VISIT(1) ;
      ELIMINATE(K,D,F)
'POOL' ;

'BACKWARD' 'LOOP'
      BACKWARDSUBS(K,D,F) ;
      VISIT(2)
'POOL'

'END'
'FINISH'
```

A5.4 A Set of Input Data for the Program

A5.4.1 The set of input data given in Section A5.4.2 corresponds to the plane stress problem illustrated in Figure 6. All the elements are of unit thickness and have a modulus of elasticity of 1.0 and a Poisson's ratio of 0.25. The coordinate input group gives the coordinates for nodes 1 to 10. Following this, the nine elements of the problem are defined. The nodes constrained to zero displacement are given in the HELD input group. All data input is in free format, allowing the two node numbers, 1 and 4, to be input on the same line. The last input group defines the force of 10 units which is applied to node 10 in the Y coordinate direction.

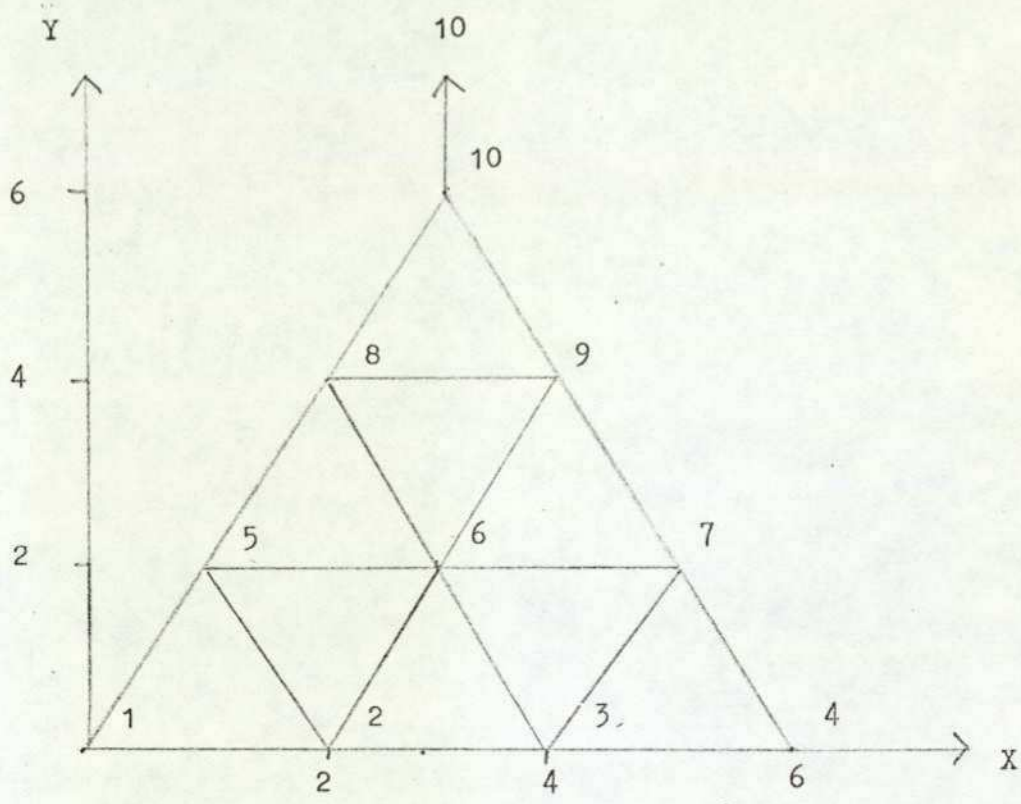
It should be noted that HELD and FORCES are synonyms for ZEROLHS and RHS respectively. In addition, DISPLACEMENTS may be used in place of LHS.

A5.4.2 The input data is as follows:

```
# PLANE STRESS PROBLEM
INITIALISE(NELEMENTS=9,NNODES=10,MAXNODENO=10)
COORDS(ORDERED)
0  0
2  0
4  0
6  0
1  2
3  2
5  2
2  4
4  4
3  6
ALL
ELEMENTS(ORDERED,NOTYPENOS)
1 2 5   1.  .25  1.
2 3 6   1.  .25  1.
3 4 7   1.  .25  1.
2 6 5   1.  .25  1.
3 7 6   1.  .25  1.
5 6 8   1.  .25  1.
6 7 9   1.  .25  1.
6 9 8   1.  .25  1.
8 9 10  1.  .25  1.
ALL
HELD(NUMBERED)
 1  4
ALL
FORCES(NUMBERED)
 10  0.0  10.0
ALL
STOP
```

A5.4.3 The following output is obtained:

```
ELEMENT +9
STRESS= +1.67942&+0  +1.00000&+1  +6.83267&-11
ELEMENT +8
STRESS= -2.94915&-1  +2.10266&+0  +5.82077&-11
ELEMENT +7
STRESS= +1.80775&+0  +3.94867&+0  -1.38450&+0
ELEMENT +6
STRESS= +1.80775&+0  +3.94867&+0  +1.38450&+0
ELEMENT +5
STRESS= +9.50317&-1  +5.18942&-1  -6.73333&-1
ELEMENT +4
STRESS= +9.50317&-1  +5.18942&-1  +6.73333&-1
ELEMENT +3
STRESS= +1.49023&+0  +3.77270&+0  -3.11365&+0
ELEMENT +2
STRESS= -7.39929&-1  +1.41672&+0  +9.89642&-11
ELEMENT +1
STRESS= +1.49023&+0  +3.77270&+0  +3.11365&+0
```



A PLANE STRESS PROBLEM USING TRIANGULAR ELEMENTS

Figure 6

A5.5 The Algol 68-R Constant Strain Element Program

```
'WITH' PREL , MATOPS 'FROM' BANALALGOL68
'BEGIN'

'SYSMAT' K ;
START 'OF' K:=0 ;
SYSMATS[1]:=K ;

'REF'[]'REAL' D ,F ;
D := SYSVECS[1] ;
F := SYSVECS[2] ;


ELEMENT PROCEDURE[1] := 'VOID' :
'BEGIN'
'REF'[]'INT'  N =  POINTERS [1:3 'AT' 1] ;
'REF''REAL'   E =  TYPE REAL CSTS [1] ;
'REF''REAL'   V =  TYPE REAL CSTS [2] ;
'REF''REAL'   T =  TYPE REAL CSTS [3] ; '


        'REAL' A, B1, B2, B3, C1, C2, C3 ;
        [1:3,1:6]'REAL'B;
        [1:3,1:3]'REAL'DM;
        B1:=Y[2]-Y[3] ;
        B2:=Y[3]-Y[1] ;
        B3:=Y[1]-Y[2] ;
        C1:=X[3]-X[2] ;
        C2:=X[1]-X[3] ;
        C3:=X[2]-X[1] ;
        A:='DET'((,X[1],Y[1]),
                 (1,X[2],Y[2]) ,
                 (1,X[3],Y[3])) ;
        B:=((B1,0,B2,0,B3,0),
            (0,C1,0,C2,0,C3),
            (C1,B1,C2,B2,C3,B3)) ;
        B:=B*(1/A) ;
        DM:=((1,V,0),(V,1,0),(0,0,(1-V)/2)) ;
        DM:=DM*(E/(1-V↑2)) ;

'C' ****** BLOCK1 ****** 'C'
'CASE' BLOCKNO 'IN'
 [1:ESIZE,1:ESIZE]'REAL' EK ;
                  EK:='T'B*DM*B*(T*A/2) ;
                  MADD(EK,K)

'C' ****** BLOCK2 ****** 'C'  ,
 [1:3]'REAL'STRESS;
                  [1:ESIZE]'REAL' U ;
                  VEQU(U,D) ;
```

```
                    STRESS:=DM*B*U ;
                    PRINT(NEWLINE,"ELEMENT",ELEMENTNO,
                        NEWLINE,"STRESS=",STRESS))

'OUT' FAULT(ERR1)
'ESAC'
'C' ******  BLOCKEND  ****** 'C'

        'END' ;



'C' ******  FORWARD LOOP  ****** 'C'
START LOOP(FORW) ;
'FOR' ENO 'TO' NO OF ELEMS FOUND 'DO'
'BEGIN'
'INT'ERESTING ;
ELEMENT NO := ENO ;
LOOP INPUT ;

        VISIT(1) ;
        ELIMINATE(K,D,F)
; LOOP OUTPUT
'END'
'C' ******  POOL  ****** 'C'
 ;


'C' ******  BACKWARD LOOP  ****** 'C'
START LOOP(BACK) ;
'FOR' ENO 'FROM' NO OF ELEMS FOUND 'BY' -1 'TO' 1 'DO'
'BEGIN'
'INT'ERPLANETARY ;
ELEMENT NO:=ENO ;
LOOP INPUT ;

        BACKWARDSUBS(K,D,F) ;
        VISIT(2)
; LOOP OUTPUT
'END'
'C' ******  POOL  ****** 'C'


'END'
'FINISH'
```

Appendix 6

A GEOMETRICALLY NON-LINEAR PLANE FRAME PROGRAM

A6.1 Introduction

The solution of geometrically non-linear plane frame problems provides a simple example of the implementation of an iterative solution method. The element properties used are described by Jennings[72]. The method of solution used is to apply the load incrementally and to perform Newton-Raphson iterations at each loading stage until convergence is obtained. The iteration at loading stage i is thus:

$$K_T^n \delta \underline{d}^{n+1} := \underline{l}^i - \underline{f}^n \qquad (A6.1)$$

$$\underline{d}^{n+1} := \underline{d}^n + \delta \underline{d}^{n+1} \qquad (A6.2)$$

where $\underline{l}^i$ is the applied load at stage i, $\underline{f}^n$ represents the total force applied by the elements in their current state of deformation and $\underline{d}^n$ represents the nodal displacements. This method requires the calculation of the tangent stiffness matrix K_T^n and the solution of equation (A6.1) at each iteration.

The computer program is described in Section A6.2 and it

is listed in Section A6.3. In Section A6.4 a set of input data for the solution of a simple toggle problem using two elements is given and the results obtained are presented. Finally, the Algol 68-R program generated by the preprocessor is listed in Section A6.5.

A6.2 A Description of the Program

Line one of the program listed in Section A6.3 indicates that there three degrees of freedom at each node and that each node has two coordinates. These freedoms at nodes are the displacements in the X and Y axes and the joint rotations at each node. The tangent stiffness matrix of the structure K is then declared followed by the three sysvecs D, F and DELTAD representing the nodal displacements, forces and increments in displacement. These sysvecs correspond to the terms $\underline{d}^{n+1}$, $\underline{l}^{i} - \underline{f}^{n}$ and $\delta\underline{d}^{n+1}$ in equations (A6.1) and (A6.2) respectively.

The integer variable NINC is used to hold the number of load increments to be applied. This number is read from the auxiliary input file.

The type definition is for an element of the kind illustrated in Figure 7, with three displacements u,v and t at each node and with three corresponding components of force. Two real values, EI the product of the modulus of elasticity and the second moment of inertia of the element cross section and, EA the product of the modulus of elasticity and the cross sectional area of the element are required to define the properties of an element. In addition, the nodal coordinates

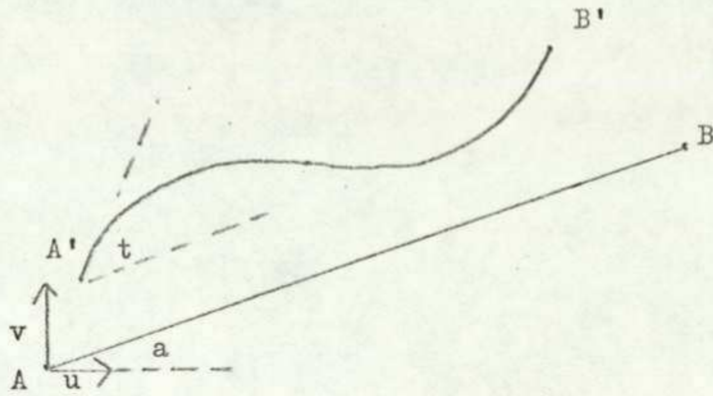
X and Y for the element in its initial undeformed condition and the nodal displacements ED are needed to evaluate its current state.

The operations within the first clause of the block clause in the type definition are used to calculate two items. These are the tangent stiffness matrix for the element, LEK and the nodal forces of the element in in its current state of deformation, EF. These are calculated by transforming the nodal displacements to local coordinates which reflect the deformation of the element giving the vector EU and by using the elastic properties of the element defined by the matrix EK. The nodal forces, EF, are assembled into the global force vector, F and the tangent stiffness matrix, LEK, is assembled into the global tangent stiffness matrix, K.

The second clause of the block clause in the type definition is used to print out the nodal displacements for each element after convergence has been obtained at each loading stage.

The DO clause in the coding defining the global solution algorithm loops over the increments of applied force. At each loading level, the Newton-Raphson iteration defined by equations (A6.1) and (A6.2) is performed

until NORM, the maximum absolute value of any change in nodal displacement, DELTAD, for any degree of freedom, is less than 0.001. Each iteration is performed by two loop clauses. The first clause applies the current load, assembles the element nodal forces into the sysvec F, assembles the tangent stiffness matrix K and then performs the elimination operations on equation (A6.1). The second loop clause then performs the backward substitution operations to determine the change in the nodal displacements. The vec clause is used to update the nodal displacement vector, D, and to calculate NORM. The sysvecs F and DELTAD are then initialised for the next iteration. The final loop clause is used to print out the nodal displacements for each element once the iteration has converged.



Initial position AB

Displaced position A'B'

A SINGLE MEMBER OF A PLANE FRAME STRUCTURE

Figure 7

A6.3 <u>The Geometrically Non-Linear Plane Frame Program</u>

```
'START' 3,2,MATOPS
'BEGIN'
'SYSMAT' K ;
'SYSVEC' D,F,DELTAD ;
'REAL' NORM ;
'INT' NINC, COUNT ;
READ(NINC) ;

'TYPE' 3 = ([1:2]'POINTER' PS,'REAL' EI,EA) :
      'BEGIN'
      'BLOCK1' [1:4,1:6]'REAL' T ;
            [1:3,1:4]'REAL' ABAR ;
            [1:4]'REAL' U ;
            [1:6]'REAL' ED,EF ;
            [1:3]'REAL' EU,ER ;
            [1:3,1:3]'REAL' EK ;
            [1:4,1:4]'REAL' DM ;
            [1:6,1:6]'REAL' LEK ;
            'REAL' P ;
            'REAL' XL = X[2]-X[1], YL = Y[2]-Y[1] ;
            'REAL' L = SQRT(XL*XL+YL*YL), COSA = XL/L,
                  SINA = YL/L ;
            VEQU(ED,D) ;
            T:=((-COSA,-SINA,0,COSA,SINA,0),
                (SINA,-COSA,0,-SINA,COSA,0),
                (0,0,1,0,0,0),
                (0,0,0,0,0,1)) ;
            U:=T*ED ;
            ABAR:=((1,U[2]/L,0,0),
                   (0,-1/L,1,0),
                   (0,-1/L,0,1)) ;
            EU:=(U[1]+U[2]*U[2]/(2*L), U[3]-U[2]/L,
                 U[4]-U[2]/L) ;
            EK:=((EA/L,0,0),
                 (0,4*EI/L,2*EI/L),
                 (0,2*EI/L,4*EI/L)) ;
            ER:=EK*EU ;
            P:=ER[1] ;
            EF:= -'T'T*'T'ABAR*ER ;
            VADD(EF,F) ;
            DM:=((0,0,0,0),
                 (0,P/L,0,0),
                 (0,0,0,0),
                 (0,0,0,0)) ;
            LEK:='T'T*('T'ABAR*EK*ABAR+DM)*T ;
            MADD(LEK,K)

      'BLOCK2' [1:6]'REAL' ED ;
            VEQU(ED,D) ;
            PRINT((NEWLINE,NEWLINE,"ELEMENT NO",ELEMENT NO,
```

```
                    NEWLINE,"NODES",FS,NEWLINE,
                    "DEFLECTIONS",ED))

     'BLOCKEND'
     'END' ;

'FOR' I 'TO' NINC 'DO'
     'BEGIN'
     COUNT:=0 ;
     'WHILE' NORM:=0 ;
             COUNT 'PLUS' 1 ;
             'FORWARD''LOOP'
                     VEC PLUS FORCE(F,1,I/NINC) ;
                     VISIT(1) ;
                     ELIMINATE(K,DELTAD,F)
                     'POOL' ;
             'BACKWARD''LOOP'
                     BACKWARD SUBS(K,DELTAD,F) ;
                     'VEC' D:=D+DELTAD ;
                           'IF' NORM<'ABS'DELTAD
                           'THEN' NORM:='ABS'DELTAD
                           'FI' ;
                           F:=DELTAD:=0
                     'CEV'
                     'POOL' ;
             NORM>0.001
     'DO' 'SKIP' ;

     'TO' 5 'DO' NEWLINE(STANDOUT) ;
     PRINT(("LOAD INCREMENT NO",I,NEWLINE,
             "NO OF ITERATIONS",COUNT)) ;

     'FORWARD''LOOP'
             VISIT(2)
             'POOL'
     'END' ;

'SKIP'
'END'
'FINISH'
```

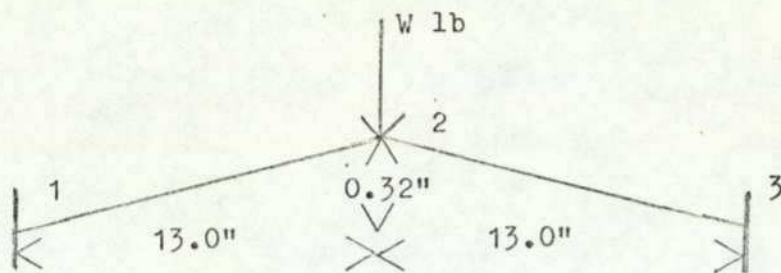
A6.4 <u>An Example of the Use of the Program</u>

A6.4.1 The toggle problem illustrated in Figure 8 was chosen as an example since published results were available. The load deflection characteristics for for the toggle using two elements are presented in Figure 9. This graph agrees well with the results obtained by Jennings[72].

The input data used is listed in Section A4.6.2. The GROUPNO parameter set in the header card for the FORCES input group is used so that the values given may be identified in the call of procedure VEC EQ FORCE in the program which applies the load at each stage of the calculations.

A6.4.2 The input data is as follows:

```
#TOGGLE PROBLEM
INITIALISE(NELEMENTS=2, NNODES=3, MAXNODENO=3)
COORDS(ORDERED)
 0  0
 13  0.32
 26  0
ALL
ELEMENTS(ORDERED, NOTYPENOS)
 1  2  9.27&3  1.885&6
 2  3  9.27&3  1.885&6
ALL
HELD(NUMBERED)
 1  3
ALL
FORCES(GROUPNO=1, NUMBERED)
 2  0  -120  0
ALL
STOP
```

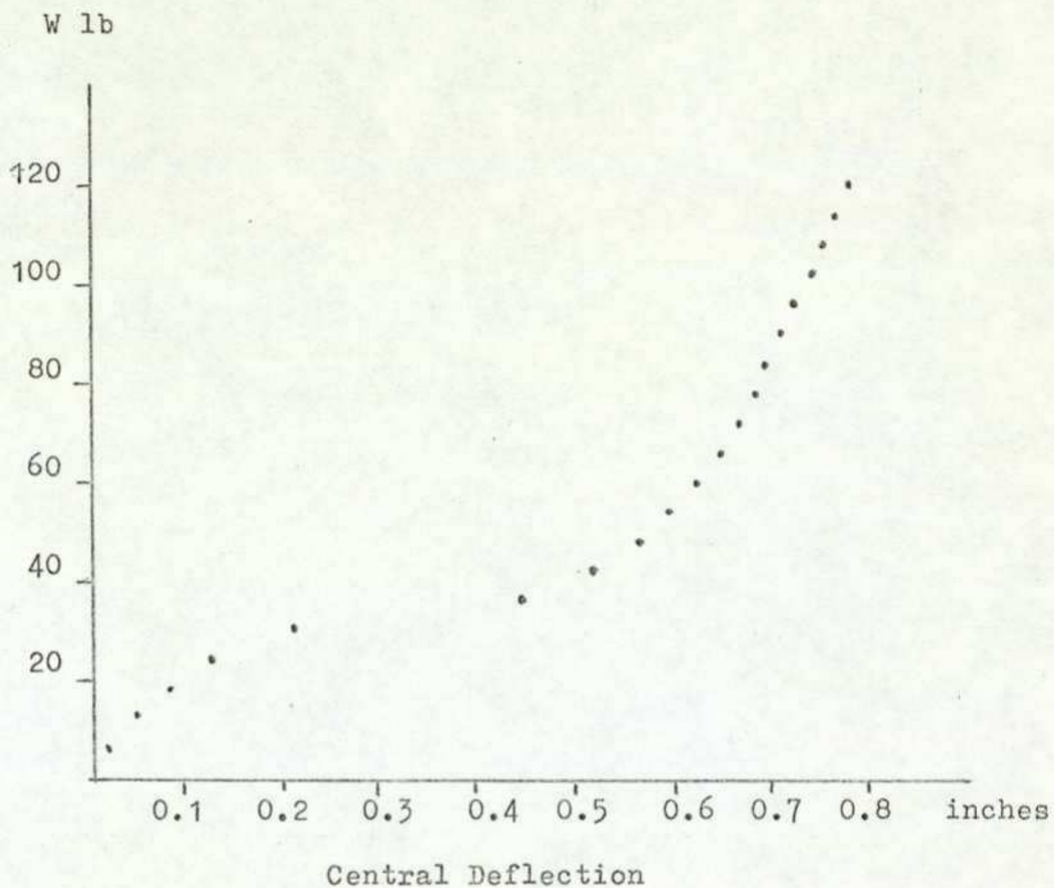


EA = 1.885×10^6 lb

EI = 9.27×10^3 lb per sq in

A TOGGLE COMPOSED OF TWO MEMBERS

Figure 8



THE LOAD DEFLECTION CHARACTERISTICS OF

THE TOGGLE OF FIGURE 8

Figure 9

A6.5 The Algol 68-R Geometrically Non-Linear Frame Program

```
'WITH' PIFL , MATOPS 'FROM' BANALALGOL68
'BEGIN'
'SYSMAT' K ;
START 'OF' K:=0 ;
SYSMATS[1]:=K ;

'REF'[]'REAL' D ,F ,DELTAD ;
D := SYSVECS[1] ;
F := SYSVECS[2] ;
DELTAD := SYSVECS[3] ;

'REAL' NORM ;
'INT' NINC, COUNT ;
READ(NINC) ;

ELEMENT PROCEDURE[1] := 'VOID' :
'BEGIN'
'REF'[]'INT' PS = POINTERS [1:2 'AT' 1] ;
'REF''REAL' EI = TYPE REAL CSTS [1] ;
'REF''REAL' EA = TYPE REAL CSTS [2] ;



'C' ****** BLOCK1 ****** 'C'
'CASE' BLOCKNO 'IN'
[1:4,1:6]'REAL'T;
           [1:3,1:4]'REAL'ABAR;
           [1:4]'REAL'U;
           [1:6]'REAL'ED,EF;
           [1:3]'REAL'EU,EA;
           [1:3,1:3]'REAL'EK;
           [1:4,1:4]'REAL'DM;
           [1:6,1:6]'REAL'LEK;
           'REAL' P ;
           'REAL' XL = X[2]-X[1], YL = Y[2]-Y[1] ;
           'REAL' L = SQRT(XL*XL+YL*YL), COSA = XL/L,
                  SINA = YL/L ;
           VEQU(ED,D) ;
           T:=((-COSA,-SINA,0,COSA,SINA,0),
               (SINA,-COSA,0,-SINA,COSA,0),
               (0,0,1,0,0,0),
               (0,0,0,0,0,1)) ;
           U:=T*ED ;
           ABAR:=((1,U[2]/L,0,0),
                  (0,-1/L,1,0),
                  (0,-1/L,0,1)) ;
           EU:=(U[1]+U[2]*U[2]/(2*L), U[3]-U[2]/L,
               U[4]-U[2]/L) ;
           EK:=((EA/L,0,0),
```

```
                    (0, 4*EI/L, 2*EI/L),
                    (0, 2*EI/L, 4*EI/L)) ;
               ER:=EK*EU ;
               P:=ER[1] ;
               EF:= -'T'T*'T'ABAR*ER ;
               VADD(EF, F) ;
               DM:=((0, 0, 0, 0),
                    (0, P/L, 0, 0),
                    (0, 0, 0, 0),
                    (0, 0, 0, 0)) ;
               LEK:='T'T*('T'ABAR*EK*ABAR+DM)*T ;
               MADD(LEK, K)

'C' ******  BLOCK2  ****** 'C' ,
 [1:6]'REAL'ED;
               VEQU(ED, D) ;
               PRINT((NEWLINE, NEWLINE, "ELEMENT NO", ELEMENT NO,
                     NEWLINE, "NODES", PS, NEWLINE,
                     "DEFLECTIONS", ED))

'OUT' FAULT(ERR1)
'ESAC'
'C' ******  BLOCKEND  ****** 'C'

       'END' ;

'FOR' I 'TO' NINC 'DO'
      'BEGIN'
      COUNT:=0 ;
      'WHILE' NORM:=0 ;
             COUNT 'PLUS' 1 ;

'C' ******  FORWARD LOOP  ****** 'C'
START LOOP(FORW) ;
'FOR' ENO 'TO' NO OF ELEMS FOUND 'DO'
'BEGIN'
'INT'ERESTING ;
ELEMENT NO := ENO ;
LOOP INPUT ;

                    VEC PLUS FORCE(F, 1, I/NINC) ;
                    VISIT(1) ;
                    ELIMINATE(K, DELTAL, F)
                    ; LOOP OUTPUT
'END'
'C' ******  POOL  ****** 'C'
 ;
```

```
'C' ******  BACKWARD LOOP  ****** 'C'
START LOOP(BACK) ;
'FOR' ENO 'FROM' NO OF ELEMS FOUND 'BY' -1 'TO' 1 'DO'
'BEGIN'
'INT'EPPLANETARY ;
ELEMENT NO:=ENO ;
LOOP INPUT ;

                    BACKWARD SUBS(K,DELTAD,F) ;

'C' ******  VEC  ****** 'C'
'FOR' IVEC 'TO' NNODES 'DO'
'IF' (FORWARDS 'AND' POINTERS[IVEC]<0) 'OR'
     (BACKWARDS 'AND' DESTS[IVEC]<0)
'THEN' 'INT' INDK:=('ABS' DESTS[IVEC]-1)*NVAR ;
'FOR' JVEC 'TO' NVAR 'DO'
'BEGIN'
'INT' VARNO = JVEC ;
INDK 'PLUS' 1 ;

 D[INDK]:=D[INDK]+DELTAD[INDK] ;
                         'IF' NORM<'ABS'DELTAD[INDK]
                         'THEN' NORM:='ABS'DELTAD[INDK]
                         'FI' ;
                         F[INDK]:=DELTAD[INDK]:=0

'END'
'FI'
'C' ******  CEV  ****** 'C'


                  ; LOOP OUTPUT
'END'
'C' ******  POOL  ****** 'C'
 ;
          NORM>0.001
     'DO' 'SKIP' ;

     'TO' 5 'DO' NEWLINE(STANDOUT) ;
     PRINT(("LOAD INCREMENT NO",I,NEWLINE,
           "NO OF ITERATIONS",COUNT)) ;


'C' ******  FORWARD LOOP  ****** 'C'
START LOOP(FORW) ;
'FOR' ENO 'TO' NO OF ELEMS FOUND 'DO'
'BEGIN'
'INT'ERESTING ;
ELEMENT NO := ENO ;
```

```
LOOP INPUT ;

                VISIT(2)
                ; LOOP OUTPUT
'END'
'C' ******  POOL  ****** 'C'

      'END' ;

'SKIP'
'END'
'FINISH'
```

REFERENCES

1. O. C. Zienkiewicz, The Finite Element Method in Engineering Science, McGraw-Hill, 1971.

2. Robert D. Cook, Concepts and Applications of the Finite Element Method, Wiley, 1974.

3. Gilbert Strang and George J. Fix, An Analysis of the Finite Element Method, Prentice Hall, 1973.

4. A. R. Mitchell and R. Wait, The Finite Element Method in Partial Differential Equations, Wiley, 1977.

5. Douglas H. Norrie and Gerard de Vries, The Finite Element, Academic Press, 1973.

6. J. E. Sammet, 'An overview of programming languages for specialised application areas', AFIPS SJCC, 299-311, 1972.

7. F. B. Thompson and B. H. Dostert, 'The future of specialized languages', AFIPS SJCC, 313-319, 1972.

8. R. A. Kelley, APT-ADAPT, American Machinist, Special Report No. 554, 1964.

9. Ole-Johan Dahl and Kirsten Nygaard, 'SIMULA - an ALGOL Based Simulation Language', Comm. ACM 9, 671-678, 1966.

10. W. D. Maurer, The Programmer's Introduction to Lisp, Macdonald/American Elsevier, 1972.

11. M. Konig, E. Schrem and G. Dietrich, The ASKA Program System, ASKA UM 214, Stuttgart, 1975.

12. R. K. Henrywood and J. D. Sheffield, 'ASAS, a general purpose finite element system', Atkins Research and Development, 1975.

13. MCS/NASTRAN Users Manual, MacNeal-Schwendler Corporation, 1976.

14. H. Knapp, 'Experience in sharing large finite element software on a variety of computer systems', 3rd Post Conference on Computational Aspects of the Finite Element Method, 1-18, 1975.

15. D. Hitchings, 'FINEL - a finite element language for teaching, research and development', 3rd Post Conference on Computational Aspects of the Finite Element Method, 59-67, 1975.

16. L. A. Lopez, 'Finite: an approach to structural mechanics systems', Int. J. Num. Meth. Eng. 10, 851-866, 1976.

17. A. van Wijngaarden et al, Revised Report on the Algorithmic Language Algol 68, Springer-Verlag, 1976.

18. R. K. Livesly, Matrix Methods in Structural Analysis, Pergamon Press, 1964.

19. R. G. Anderson, A Finite Element Eigenvalue Solution System, Ph. D. Thesis, University of Wales, Swansea, 1968.

20. G. C. Nayak and O. C. Zienkiewicz, 'Elasto-plastic stress analysis, a generalisation for various constitutive relations including strain softening', Int. J. Num. Meth. Eng. 5, 113-135, 1972.

21. James A. Stricklin, Walter E. Haisler and Walter A. von Riesemann, 'Evaluation of solution procedures for material and/or geometrically non-linear structural analysis', AIAA Journal 11, 292-299, 1973.

22. Robert Kao, 'A comparison of Newton-Raphson and incremental procedures for geometrically non-linear analysis', Computers and Structures 4, 1091-1097, 1974.

23. Ernst Schrem, 'Computer Implementation of the Finite-Element Procedure', Numerical and Computer Methods in Structural Mechanics, Academic Press, 1973.

24. Christian Meyer, 'Solution of Linear Equations - State-of-the-Art', ASCE J. Struct. Div. 99, 1507-1526, 1973.

25. Christian Meyer, 'Special Problems Related to Linear Equation Solvers', ASCE J. Struct. Div. 101, 869-890, 1975.

26. G. von Fuchs and J. R. Roy, Solution of the Stiffness Equations in ASKA, I. S. D. Stuttgart, Report No. 50, 1968.

27. B. M. Irons, 'A frontal solution program for finite element analysis', Int. J. Num. Meth. Eng. 2, 5-32, 1970.

28. T. K. Helen, 'A frontal solution for finite element techniques', C.E.G.B. Report No. RD/B/N1469, 1969.

29. Robert J. Melosh and Robert M. Bamford, 'Efficient solution of load-deflection equations', ASCE J. Struct. Div. 95, 661-676, 1969.

30. Anthony Ralston, A First Course in Numerical Analysis, McGraw-Hill, 1965.

31. L. Fox, An Introduction to Numerical Linear Algebra, Oxford University Press, 1964.

32. Richard S. Varga, Matrix Iterative Analysis, Prentice-Hall International, 1962.

33. P. M. Woodward and S. G. Bond, Algol 68-R Users Guide, H.M.S.O., 1972.

34. Pal G. Bergan and Ray W. Clough, 'Convergence Criteria for Iterative Processes', AIAA Journal 10, 1107-1108, 1972.

35. Dag Kavlie and Graham H. Powell, 'Efficient reanalysis of modified structures', ASCE J. Struct. Div. 97, 377-392, 1971.

36. Uri Kirsch and Moshe F. Rubinstein, 'Structural reanalysis by iteration', Computers and Structures 2, 497-510, 1972.

37. Suresh R. Phansalkar, 'Matrix iterative methods for structural reanalysis', Computers and Structures 4, 779-800, 1974.

38. Ahmed K. Noor and Harold E. Lowder, 'Approximate techniques of structural reanalysis', Computers and Structures 4, 801-812, 1974.

39. Walter E. Haisler, James A. Stricklin and Frederick J. Stebbins, 'Development and Evaluation of Solution Procedures for Geometrically Nonlinear Structural Analysis', AIAA Journal 10, 264-272, 1972.

40. G. C. Nayak and O. C. Zienkiewicz, 'Note on the 'alpha' - constant stiffness method for the analysis of non-linear problems', Int. J. Num. Meth. Eng. 4, 579-582, 1972.

41. P. V. Marcal and I. P. King, 'Elastic-plastic analysis of two-dimensional stress systems by the finite element method', Int. J. Mech. Sci. 9, 143-155, 1967.

42. Alan Jennings, 'Accelerating the Convergence of Matrix Iterative Processes', J. Inst. Maths. Applics. 8, 99-110, 1971.

43. Bruce M. Irons and Robert C. Tuck, 'A version of the Aitken accelerator for computer iteration', Int. J. Num. Meth. Eng. 1, 275-277, 1969.

44. E. F. Boyle and A. Jennings, 'Accelerating the convergence of elastic-plastic stress analysis', Int. J. Num. Meth. Eng. 7, 232-235, 1973.

45. J. H. Wilkinson, The Algebraic Eigenvalue Problem, Oxford University Press, 1965.

46. R. J. Guyan, 'Reduction of Stiffness and Mass Matrices', AIAA Journal 3, 380, 1965.

47. B. M. Irons, 'Structural Eigenvalue Problems - Elimination of Unwanted Variables', AIAA Journal 3, 961, 1965.

48. Klaus-Jurgen Bathe and Edward L. Wilson, 'Eigensolution of Large Structural Systems with Small Bandwidth', ASCE J. Eng. Mech. Div. 99, 467-479, 1973.

49. G. Peters and J. H. Wilkinson, 'Eigenvalues of $A x = \lambda B x$ with band symmetric A and B', Comp. J. 12, 398-404, 1969.

50. K. K. Gupta, 'Vibration of frames and other structures with banded stiffness matrix', Int. J. Num. Meth. Eng. 2, 221-228, 1970.

51. Klaus-Jurgen Bathe and Edward L. Wilson, 'Solution methods for eigenvalue problems in structural mechanics', Int. J. Num. Meth. Eng. 6, 213-226, 1973.

52. Heinz Rutishauser, 'Computational Aspects of F. L. Bauer's Simultaneous Iteration Method', Numer. Math. 13, 4-13, 1969.

53. H. Rutishauser, 'Simultaneous Iteration Method for Symmetric Matrices', Numer. Math. 16, 205-233, 1970.

54. R. B. Corr and A. Jennings, 'A simultaneous iteration algorithm for symmetric eigenvalue problems', Int. J. Num. Meth. Eng. 10, 647-663, 1976.

55. V. Giuriutiu and R. O. Stafford, 'An improvement on simultaneous vector iteration', 3rd Post Conference on Computational Aspects of the Finite Element Method, 171-182, 1975.

56. Klaus-Jurgen Bathe and Edward L. Wilson, 'Large eigenvalue problems in dynamic analysis', ASCE J. Eng. Mech. Div. 98, 1471-1485, 1973.

57. O. C. Zienkiewicz and D. V. Phillips, 'An automatic mesh generation scheme for plane and curved surfaces by 'isoparametric' co-ordinates', Int. J. Num. Meth. Eng. 3, 519-528, 1971.

58. R. H. Gallagher and O. C. Zienkiewicz, eds., Optimum Structural Design, Wiley, 1973.

59. Structural Optimization, North Atlantic Treaty Organisation, Advisory Group for Aerospace Research and Development, AGARD-LS-70, 1974.

60. Richard H. Gallagher, 'Fully Stressed Design', Optimum Structural Design, 19-32, Wiley, 1973.

61. L. Berke and N. S. Khot, 'Use of optimality criteria methods for large scale systems', Structural Optimization, AGARD-LS-70, 1974.

62. O. C. Zienkiewicz and J. S. Campbell, 'Shape Optimization and Sequential Linear Programming', Optimum Structural Design, 109-126, Wiley, 1973.

63. P. J. Brown, 'The ML/I Macro Processor', Comm. ACM 10, 618-623, 1967.

64. P. J. Brown, ML/I user's manual, University of Kent at Canterbury, 1974.

65. P. Hood, 'Frontal solution program for unsymmetric matrices', Int. J. Num. Meth. Eng. 10, 379-399, 1976.

66. R. Ford, 'Multi-level component analysis', 2nd International Asas Users' Conference, 91-95, 1976.

67. T. Furuike, 'Computerized multiple level sub-structuring analysis', Computers and Structures 2, 1063-1073, 1972.

68. F. W. Williams, 'Comparison between sparse stiffness matrix and substructure methods', Int. J. Num. Meth. Eng. 5, 383-394, 1973.

69. J. M. Siddall, R. E. Yeardon and T. Harrison, 'The application of sub-structure techniques to the static and frequency analysis of structural systems', 3rd Post Conference on Computational Aspects of the Finite Element Method, 201-212, 1975.

70. B. M. Irons and David K. Y. Kan, 'Equation solving algorithms for the finite element method', Numerical and Computer Methods in Structural Mechanics, 499-511, Academic Press, 1973.

71. Michael F. Yeo, 'A more efficient front solution: allocating assembly locations by longlevity considerations', Int. J. Num. Meth. Eng. 7, 570-573, 1973.

72. Alan Jennings, 'Frame analysis including change of geometry', ASCE J. Struct. Div. 94, 627-644, 1968.