Durham E-Theses

Computer algebra and transputers applied to the finite element method

Barbier, Christine

How to cite:

Barbier, Christine (1992) *Computer algebra and transputers applied to the finite element method*, Durham theses, Durham University. Available at Durham E-Theses Online: http://etheses.dur.ac.uk/6112/

Use policy

The full-text may be used and/or reproduced, and given to third parties in any format or medium, without prior permission or charge, for personal research or study, educational, or not-for-profit purposes provided that:

- a full bibliographic reference is made to the original source
- a link is made to the metadata record in Durham E-Theses
- the full-text is not changed in any way

The full-text must not be sold in any format or medium without the formal permission of the copyright holders.

Please consult the full Durham E-Theses policy for further details.

Academic Support Office, Durham University, University Office, Old Elvet, Durham DH1 3HP
e-mail: e-theses.admin@dur.ac.uk Tel: +44 0191 334 6107
http://etheses.dur.ac.uk

Computer Algebra and Transputers Applied to the Finite Element Method

by

Christine Barbier

A Thesis submitted in partial fulfilment
of the requirements for the degree of
Doctor of Philosophy

Computing Service

The University of Durham

1992

The copyright of this thesis rests with the author.
No quotation from it should be published without
his prior written consent and information derived
from it should be acknowledged.

UNIV DUNELM

- 8 SEP 1992

Real programmers don't comment their code.
If it was hard to write, it should be hard to understand.
(Almasi and Gottlieb, 'Highly parallel computing', p150)

Heureux les pauvres d'esprit,
Ceux qui ne cherchent pas à comprendre.
(J. Prevert, 'Paroles')

Woe to the author who always wants to teach!
The secret of being a bore is to tell everything.
(Voltaire, 'De la Nature de l'Homme')

Preface

The work presented in this thesis was carried out in the Computing Service at the University of Durham, between February 1989 and January 1992, under the supervision of Jacqueline A. Bettess from the Computing Service, Alan Craig from the Department of Mathematical Sciences and Peter Bettess from the Department of Marine Technology at the University of Newcastle-upon-Tyne. No part of this work has been previously submitted for any degree either in this or any other university.

The work carried out is believed to be original. The material in chapter 3 is included in a paper published in *Engineering Computations*[†]. The work in chapter 4 is described in a paper submitted for publication in the *Journal of Symbolic Computation*[†]. Chapter 5 forms the basis of a paper published in *Engineering Computations*[†]. The work in chapter 7 is described in a paper written jointly with Ian Applegarth, to be submitted for publication.

[†] Refere to the author index at the end of the thesis. The papers written by the author are marked with an asterix (*)

All the software resulting from the work described in this thesis is available for genuine academic examination from the author or her supervisor at the addresses below:

	Christine BARBIER	or	Jackie BETTESS
	Marine Technology Department		Computing Service
	University of Newcastle-upon-Tyne		University of Durham
	Armstrong Building		South Road
	Newcastle-upon-Tyne		Durham
	NE1 7RU		DH1 3LE
Tel	: (+44) 91 222 6000, ext 8220		(+44) 91 374 2895
Email	: Christine.Barbier @ uk.ac.newcastle		J.A.Bettess @ uk.ac.durham

Relevant information concerning the theory and the implementation on the computer are provided with the software in the form of user guides.

I would like to thank Jackie and Peter Bettess for their friendly supervision throughout the time I spent in Durham. I am also grateful to Alan Craig for his help with the reading of this thesis. I thank Josephine Coleman for her constant availability to resolve the many technical problems that arose. Many thanks go to all my colleagues in Durham too.

I would like to acknowlege and thank my colleague Ian Applegarth, of the Department of Marine Technology at the University of Newcastle, who contributed to the paper on the parallel solvers. I would also like to thank all the other Research Associates of the Department of Marine Technology with whom I had many interesting and stimulating conversations.

Finally, I am pleased to acknowledge the support of the Science and Engineering Research Council through awards GR/E/2099.8 and GR/F/06173.

The copyright of this thesis rests with the author. No quotation from it should be published without her prior written consent and information derived from it should be acknowledged.

Abstract

Recent developments in computing technology have opened new prospects for computationally intensive numerical methods such as the finite element method. More complex and refined problems can be solved, for example increased number and order of the elements improving accuracy. The power of Computer Algebra systems and parallel processing techniques is expected to bring significant improvement in such methods. The main objective of this work has been to assess the use of these techniques in the finite element method.

The generation of interpolation functions and element matrices has been investigated using Computer Algebra. Symbolic expressions were obtained automatically and efficiently converted into FORTRAN routines. Shape functions based on Lagrange polynomials and mapping functions for infinite elements were considered. One and two dimensional element matrices for bending problems based on Hermite polynomials were also derived.

Parallel solvers for systems of linear equations have been developed since such systems often arise in numerical methods. Both symmetric and asymmetric solvers have been considered. The implementation was on Transputer-based machines. The speed-ups obtained are good.

An analysis by finite element method of a free surface flow over a spillway has been carried out. Computer Algebra was used to derive the integrand of the element matrices and their numerical evaluation was done in parallel on a Transputer-based machine. A graphical interface was developed to enable the visualisation of the free surface and the influence of the parameters. The speed-ups obtained were good. Convergence of the iterative solution method used was good for gated spillways. Some problems experienced with the non-gated spillways have lead to a discussion and tests of the potential factors of instability.

Foreword

This thesis is organised in three parts each containing several chapters. The parts can be seen either as independent units or as constituents of the whole thesis.

The bibliographic references are organised in a similar fashion. At the end of each chapter a list of references related to that chapter is inserted. References are indicated as numbers in the text. A complete bibliography is provided at the end of the thesis where all references are classified by author's name in alphabetical order. The same reference might appear in more than one chapter but will be accounted for only once in the complete author index at the end.

This thesis has been produced using the typesetting system TEX[†] and the UNIRAS[‡] graphics interactive packages.

[†] Knuth D.E, *The TEX book*, Addison Wesley Publishing Company, 1988

[‡] UNIEDIT, available from UNIRAS

Contents

Chapter I

Introduction

Developments in both computer hardware and software make it possible nowadays to solve more and more complicated physical problems. The design of faster hardware elements, like the Transputer[1], combined with the possibility of associating these elements in a parallel architecture has dramatically increased the computer power available. Parallel processing is one of the latest developments in computing and it has a promising future. Among new software tools, Computer Algebra, which relieves the programmer from tedious and complicated algebra, has recently become more powerful and more widely available. This evolution not only makes it possible to solve present problems better but also new and more complex problems can be considered.

Together with other fields, finite element methods benefit from these developments. As for most numerical techniques, the finite element method depends on the computing power available as this limits the accuracy of the results and the size of the problems which can be solved. The finite element method is also restricted by the complexity of the algebra involved which can become difficult to handle. The use of these new techniques in finite element methods is thus of great interest and is the reason for the investigations described in this thesis.

1.1 The finite element method

1.1.1 Brief review

The basic idea of the finite element method is to replace an actual problem by a simpler one. As this concept is to a large extent physical rather than abstract, it has been present since the beginning of civilization[2]. One of its earliest uses, more than two thousand years ago, was in geometry in such problems as determining the perimeter and area of a circle, where regular polygons were chosen as a substitute problem.

Since these early times, the finite element method has constantly been developed but its major transformation happened about forty years ago with the appearance of computers which enabled it to be applied to solve large problems. As the computers have become more powerful, the finite element method has been refined and its accuracy improved.

A huge amount of work and research has been done in this field[3,4,5,6,7], close to 8000 references ten years ago[3] and many more as internal reports. In its modern form the finite element method started to be used for stress analysis in aircraft structures in the 60's. Since it has spread to other engineering and non-engineering fields like structural and fluid mechanics, semi-conductor design, thermal conduction analysis, bioengineering etc.

Although the mathematical formulation was first stated by Courant[8] in 1943, its practical application did not happen before the 70's with the first book which comprehensively treated the finite element method on both physical and mathematical points of view by Zienkiewicz and Cheung[9]. These new ideas involved variational and weighted residual formulations which could then be applied to a wide range of problem, lifting the method outside the borders of solid mechanics.

In the following paragraphs a brief reminder of the principles of the finite element method will be presented as this is the *leitmotiv* of the work undertaken in this research. The material presented below is a summary of information obtained from references 3, 4, 5, 6 and 7.

1.1.2 Mathematical formulations

The finite element method is mathematically stated as an approximation method for solving a set of equations, usually involving partial differentials, to which boundary values are applied. This can be written as follows:

$$\begin{aligned} F_i(\mathbf{u}) &= 0 \qquad \text{on} \quad \Omega \qquad 0 \le i \le n \\ B_i(\mathbf{u}) &= 0 \qquad \text{on} \quad \Gamma \qquad 0 \le i \le p, \end{aligned} \tag{1.1}$$

where $\mathbf{u}$ is the vector of the unknowns, F_i is an equation relating the unknowns, B_i is an equation describing boundary conditions, Ω is the domain where the equations apply and Γ is the boundary of Ω. Two methods are generally used: the weighted residual method and the variational method. They both transform the differential equations (1.1) into integro-differential equations where the finite element approximation can be applied.

The weighted residual method consists of stating that the weighted average of the residual errors over Ω and Γ is zero. This means that the error introduced by an approximation of the solution **u**, weighted so that the most serious errors are most taken into account, should on average be zero. This can be written as follows:

$$\forall w_1, ...w_n, \tilde{w}_1, ...\tilde{w}_p$$
$$\int_\Omega w_1 F_1(\mathbf{u}) + ...w_n F_n(\mathbf{u}) d\Omega + \int_\Gamma \tilde{w}_1 B_1(\mathbf{u}) + ...\tilde{w}_p B_p(\mathbf{u}) d\Gamma = 0. \tag{1.2}$$

The second method uses the principle of virtual work, or minimum potential energy, associated to a variational principle. A functional Π is written as follows:

$$\Pi = \int_\Omega G(\mathbf{u}) d\Omega + \int_\Gamma E(\mathbf{u}) d\Gamma. \tag{1.3}$$

The solution **u** is a function which makes Π stationary with respect to small changes of $\delta\mathbf{u}$, that is to say $\delta\Pi = 0$. This can be written from equation (1.3) as:

$$\delta\Pi = \int_\Omega \delta\mathbf{u} G'(\mathbf{u}) d\Omega + \int_\Gamma \delta\mathbf{u} E'(\mathbf{u}) d\Gamma = 0. \tag{1.4}$$

This must be true for any variation $\delta\mathbf{u}$:

$$G'(\mathbf{u}) = 0 \qquad \text{and} \qquad E'(\mathbf{u}) = 0. \tag{1.5}$$

If G' and E' can be found so that they correspond to F and B describing the problem equation(1.1) then an integral form of the problem similar to equation (1.2) for the weighted residual method is obtained. For some problems a weighted residual form exists, but not a variational statement.

Once these integral forms (1.2) or (1.4) have been established the actual approximation is achieved by replacing $\mathbf{u}$ in the equations by a piecewise approximation written as follows:

$$\mathbf{u} \approx \tilde{\mathbf{u}} = \sum_{i=1}^{r} N_i \mathbf{a}_i, \tag{1.6}$$

where the N_i are interpolation functions, called shape functions, usually polynomials, and the $\mathbf{a}_i$'s become the unknowns. The integral can then be written as the sum of integrals each related to only one element.

Physically, this process corresponds to splitting up the domain Ω into subdomains, the finite elements, where the unknowns are approximated according to equation (1.6). Often, the $\mathbf{a}_i$'s represent the values of $\mathbf{u}$ at chosen points in the element, the nodes. The integrals are then evaluated element by element. When the equations are linear, they can be written in a matrix form, each integral contributing to one element matrix. A solver for linear equations can then be used. If the equations are not linear more complicated solving schemes must be implemented.

1.2 Reasons and nature of the investigations

Nowadays, the finite element method often involves thousand of unknowns which means very big systems of equations have to be established and solved which is very time consuming. The algebra for deriving the shape functions, their derivatives and the element matrices can be tedious and complicated, especially with the current interest for finer meshes, higher order elements, and adaptive and hierarchical elements. The recent developments in parallel processing and Computer Algebra raises interest in using these techniques in finite element methods to increase the accuracy of the solution and the difficulty of the problem. Investigations have already taken place on these topics and this research aims at further developments. The work has focused on three subjects:

1. Automatic generation of interpolation functions and element matrices using Computer Algebra.
2. Development of fast solvers for systems of linear equation using parallel processing.

3. Study of a free surface flow over a spillway using finite element analysis.

The thesis is divided into three parts, each related to one of the subjects above. Within each part, the first chapter is devoted to introducing in detail the particular technique used and survey previous work in that field. The following chapters describe the theory and implementation on the computer. Each chapter is ended by a presentation of results and conclusions, and by a list of references.

The first part of the work is concerned with the automatic production of FORTRAN code for shape functions for finite elements, mapping functions for infinite elements and element matrices based on Hermite polynomials. Traditionally, the mathematical expressions for these functions were derived and coded into the computer by hand which was time consuming and error-prone. The present approach is concerned with the obtention of the computer program in an automatic and error-free manner. The implementation has been carried out with the language REDUCE which is introduced at the beginning of part one.

It is not uncommon that the systems of equations which arise in finite element method are linear and of large dimensions – thousands of equations and more. Even in cases when these systems are not linear an approximation method is generally used to linearise the system and the problem is reduced to solving a linear set of equations. The intention of the second part is thus to present the implementation in parallel of solvers for systems of linear equations. Both symmetrical and unsymmetrical solvers have been considered. The parallel machines used are Transputer-based machines progammed using a parallel version of standard FORTRAN.

The third and last part of the work focuses on the study of a free surface flow over a spillway. A lot of work has been done on this subject by trial and error methods but not as many studies have involved the automation of the method on the computer. The finite element method has been used to evaluate the position of the surface and the values of the stream function, thus the pressure distribution on the spillway. The Newton-Raphson method has been employed to solve the system of non-linear equations that arose. The analytical form of the element matrices has been derived using Computer Algebra and their numerical evaluation has been carried out in parallel. The complete program has been coded in FORTRAN and graphic routines have been used for the display of the mesh and the position of the surface. A library of routines concerned with the implementation of parallel concepts has also been used.

References

1. The Transputer is manufactured by INMOS Limited, member of the SGS–Thomson Microelectronics Group, 1000 Aztec West, Almondsbury, Bristol, BS12 4SQ, UK.
2. Martin H.C and Carey G.F, *Introduction to Finite Element Analysis, Theory and application*, McGraw-Hill Book Company, 1973.
3. Zienkiewicz O.C., *The Finite Element Method*, Third Edition, McGraw-Hill Book Company, 1977.
4. Filho J.S.R.A, 'The Use of Transputer Based Computers in Finite Element Calculations', *PhD thesis*, University of Wales, University College of Swansea, Department of Civil Engineering, September 1989.
5. Desai C.S and Abel J.F, *Introduction to the Finite Element Method (a numerical method for engineering analysis)*, Van Nostrand Reinhold Company, 1972.
6. Akin J.E., *Finite Element Analysis for Undergraduates*, Academic press, 1986.
7. Rao S.S., *The Finite Element Method in Engineering*, Second Edition, Pergamon Press, 1989.
8. Courant R., 'Variational methods for the solution of problems of equilibrium and vibration', *Bulletin of American Methematical Society*, **49**. 1–23, 1943.
9. Zienkiewicz O.C and Cheung Y.K, *The Finite Element Method in Structural and Continum Mechanics*, McGraw-Hill, London, 1967.

Part I

Computer Algebra and Finite Element Method

Chapter II

Introduction to Computer Algebra

2.1 Generalities

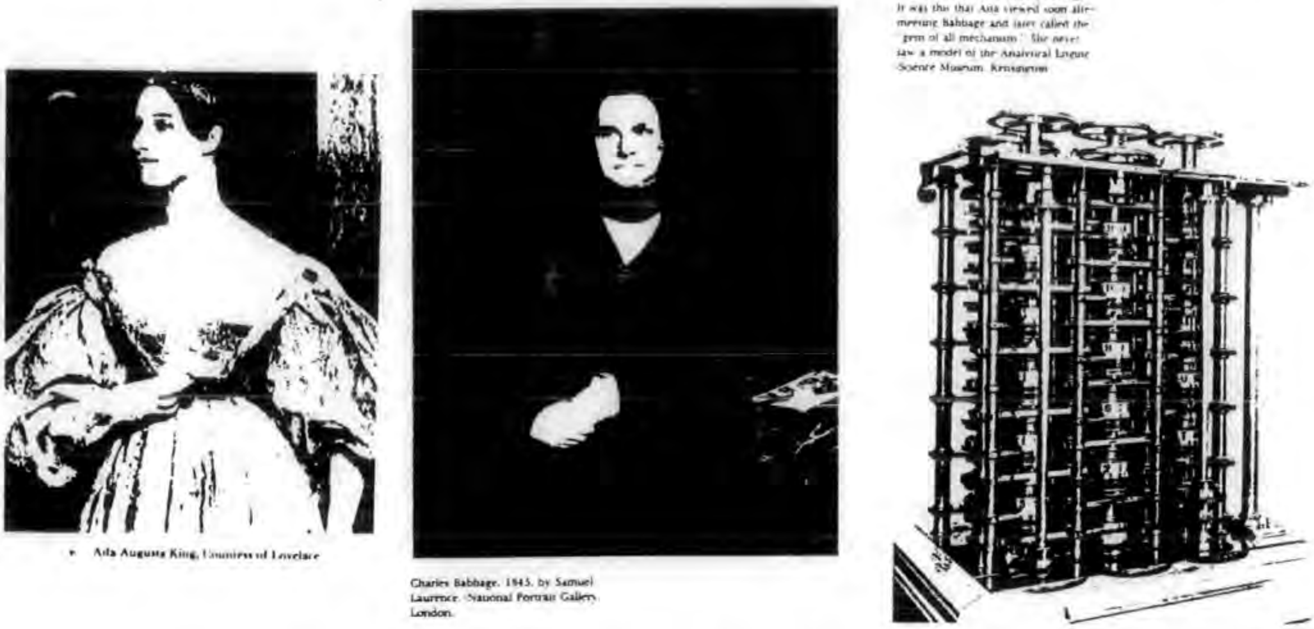
2.1.1 Definition

Computer Algebra is the discipline which concerns itself with the design, analysis, implementation and application of algebraic algorithms for computers[1]. It is the part of Computer Science which focuses on the processing of algebraic expressions, the automation of algebraic calculation or, in other words, non-numerical calculation on the computer. The name of this discipline has long varied, being designated as Symbolic Manipulation, Algebraic Computation, Symbolics Mathematics, Symbolics Algebra, Formula Manipulation and Analytic Calculation to settle down as Computer Algebra in English and Calcul Formel in French.

2.1.2 Brief history

This idea that computers could perform algebraic calculation was first suggested nearly a century and a half ago by Lady Ada Lovelace[2], who was the patron of Charles Babbage[3], usually credited with the development of the world's first computer (see Figure 2.1). It was not before the 50's that a practical attempt at Computer Algebra was made by Kahrimanian and Nolan[4] (1953) with their pioneering work on differentiation. In the 60's hardware and software capabilities became sufficient for the development of complete packages[5,6].

At that time, the new discipline was the frontier between several fields, hence the various names it has taken during its history. Mathematicians were involved through the design of effective algorithms. The initial objectives were in the field of Artificial Intelligence, even if the methods are moving away from it nowadays[7]. Physicists and engineers have also contributed to these sytems by writting special packages related to their own field of interest. Computer Algebra has finally become a discipline in its own right.



Model of Babbage's Difference Engine. It was this that Ada viewed soon after meeting Babbage and later called the "germ of all mechanism." She never saw a model of the Analytical Engine. Science Museum, Kensington.

Ada Augusta King, Countess of Lovelace

Charles Babbage, 1845, by Samuel Laurence. National Portrait Gallery, London.

Figure 2.1: *Lady Ada de Lovelace, Charles Babbage and his differential engine*

2.1.3 Classification

Wooff[8] divides algebraic programming languages into three groups, just as numerical programming languages are often designated as first, second, third, fourth and fifth generation.

The first group comprises systems written to solve specialized problems where the algebraic calculations necessary were extremely tedious and time consuming such as Mathematical Physics, General Relativity, Quantum Theory, Celestial Mechanics, and High Energy Physics. The computer systems had a significant impact in their chosen field where the algebra, which used to consume months of a good mathematician's time, could be derived in a very short time and reliably compared to pages of hand calculation. These systems were not easy to manipulate and were non-interactive. They were only worth using on genuinely difficult problems. Among the best known members are SCHOONSCHIP, CAMAL, CLAM and ASHMEDAI which are described in van Hulzen's article[9].

The second group includes more general purpose systems. Compared to the first group they were easier to use, portable, interactive and not restricted to a particular application. Because of these criteria, this second grouping is much smaller, containing systems like REDUCE[10], MACSYMA[11], Scratchpad[12], SMP[13] and its successor Mathematica[14].

Members of the third group are characterised by smaller memory requirements, a wide range of built in functions and simplicity of use. There are currently two principal members Maple[15] and muMATH[16] with its successor Derive[17]. Current developments will probably increase the number of members of this grouping.

2.1.4 Characteristics of Computer Algebra systems

Nowadays, members of all three groups are still in use although only eight systems are up-to-date and widely available[18]. They are Derive, MACSYMA, Maple, Mathematica, muMATH, REDUCE, Scratchpad and SMP. These systems vary in their abilities to solve particular type of problems, in their availabilities and their price. They, nevertheless, all provide some basic facilities which include polynomial manipulation, recognition of transcendental functions, exact arithmetic calculation, analytic differentiation, integration, equation solving and substitution, matrix manipulation, definition of new rules, interactive use and file handling.

There is not, however, any standardisation of the Computer Algebra languages. Some of the systems like REDUCE and Maple are available for a wide range of machines from micro to mainframe computers. Others have a more restricted range like muMATH and Derive which mainly run on microcomputers, MACSYMA on UNIX workstations and SMP and Mathematica on minicomputers. Scratchpad is significantly different from the other packages with a design based upon abstract datatypes. It is currently an IBM internal research project but it is considered as a possible IBM product.

2.1.5 Survey

A complete up-to-date survey[18] of the main Computer Algebra systems is available through the Computer Algebra Support Project in Liverpool[19]. Other information can be found in specialized journals which include SIGSAM[†], a bul-

[†] Special Interest Group in Symbolic and Algebraic Manipulation

letin of the world-wide organisation group of ACM[20], proceedings of congresses EUROCAM and EUROSAM organised by the European group SAME[‡], the review CALSYF edited by M. Mignotte from the University of Strasbourg in France and the Journal of Symbolic Computation which has been published since 1985. Books available include some concerned with the use a specific language such as muMATH[8], REDUCE[21] and Mathematica[22], others dealing with the design of mathematical algorithms[23,24,25,26] and those orientated towards a general grasp of the field[7].

So many type of books are available because, unlike numerical languages, it is necessary for the user to have a general idea on how to use a particular language as well as what are the algorithms implemented in order to use these systems efficiently. A typical example is intermediate expression swelling, which is the unwanted expansion of intermediate expressions in a calculation, which can sometimes be avoided by a restatement of the initial problem. Some systems might fail to give satisfactory answers when a simple answer exists because of memory requirements if the user is not able to predict a failure. Once a system has been mastered moving to another one does not normally present many problems except for special features which might be implemented in a different way.

2.1.6 Applications for Computer Algebra

The range of application for Computer Algebra is very wide as algebraic manipulation is used in most scientific and engineering fields. The list of application domains is too long to be exhaustively presented here. Several survey articles on this subject have been published[27,28]. Alongside the use of Computer Algebra as a tool to develop or speed up new mathematical calculations for a particular area of science, another important application of these systems is in teaching. A Computer Algebra system can help a teacher to present examples and encourage guesswork and implicit comprehension. There is, however, argument among those concerned about whether Computer Algebra systems should be used in teaching[29].

[‡] Symbolic and Algebraic Manipulations in Europe

2.2 Computer Algebra and mechanical engineering: a survey

A subject of particular interest for the work presented in this thesis is computational mechanics with the use of finite element methods for which some of the algebra can automatically and reliably be derived using Computer Algebra systems.

Computer Algebra systems have been used in computational mechanics for at least twenty years. References can be found as early as 1971[30]. Most of the major work in this field started in the mid-seventies. References of work done in Denmark by Pedersen and others[31,32,33,34] show that systems like FORMAC[6] were used to derive element matrices for various finite element problems. The functionalities of the Computer Algebra systems used included matrix multiplication, matrix inversion, analytical derivation and integration of multivariable polynomial functions. When functions were more complicated than polynomials a mixed scheme of numerical integration and analytical derivation of the integrands was adopted[32].

At that time Computer Algebra sytems were powerful enough to deal with such calculations but were still too cumbersome to be easily used. Pedersen[32] mentions that his program was not 'very user-oriented'. Other references from the same period can be found in other countries such as the USA[35,36,37]. It is at that time that people like Jensen[34] observed that there were 'more people designing [Computer Algebra] systems than using them'.

Although the technique has evolved since these early days and is more widespread, it seems that it has not become yet an obvious tool that everybody uses. While some people[38] today claim that Computer Algebra systems have been used for at least ten years in their university as part of the undergraduate curriculum, particularly in America, others have never heard about it and appear quite interested in the prospects the technique opens. A recent seminar on case studies of the use of Computer Algebra in industry has taken place[39] which claims to be the first of this kind. The aim was to demonstrate how Computer Algebra in industry can be useful and cost effective. The speakers were mainly involved with research in both academic and industrial worlds. This suggests that the claim made previously on how Computer Algebra is not widely used yet is in a certain way justified and that Computer Algebra is still an active area of research.

From personal observations, it seems that where Computer Algebra packages have been introduced in the early days through, for example, the proximity of a computer science department involved in the design of such a package, or through contact with computer scientists involved in the field, they are nowadays used on an everyday basis. This implies that from a research point of view the topic of using Computer Algebra in engineering is not new and most of the concepts are quite old. Nevertheless, the idea is not fully widespread yet, particularly in the UK and it is worth using it to experiment with its implications to finite element analysis.

Recent work has been undertaken in this field[40,41,42,43,44,45] but the total number of papers published remains small. It seems likely that with the advent of Computer Algebra packages which can run on PC's and the spread of powerful workstations that this will change. Nowadays, languages like Mathematica, Derive and REDUCE are compact enough to fit in a desktop machine. An interesting new development, which runs on a workstation like the SUN SPARCstation, is the SENAC[†] package[46] which is an algebraic environment providing an easy-to-use interface to the NAG FORTRAN and Graphics libraries. It claims to be the first system of its type and to be a new concept in scientific computations[47]. It integrates the advantages of a robust library with the power of symbolic computation.

The advantage of using Computer Algebra packages to allow 'the analyst to concentrate on more meaningful tasks, such as the establishment of physical assumptions, without being sidetracked by the tedious and trivial details of the algebraic manipulations' was pointed out by Crespo Da Silva[37]. In that sense Computer Algebra has been used in this work to show the benefits of such a technique in a scientific and educational context.

In the following section, a detailed description of the algebraic system used in this work – REDUCE – is described. The chapters thereinafter focus on the explicit demonstration of how this particular system has been used to automate the calculations. It is important to note that the aim of the work was to prove the feasibility of the use of a Computer Algebra system and the scientific and educa-

[†] Software Environment for Numeric and Algebraic Computation

tional consequences rather than demonstrating the capabilities of this particular system.

2.3 REDUCE : An algebraic language

REDUCE[48] is a general-purpose Computer Algebra system, the oldest in the second grouping mentioned in section 2.1. It is the most widely used system in the UK and Europe. It is available for a wide range of machines although it has fairly large memory requirements (typically 1 MByte) to solve large problems. It has originally been developed by A.C. Hearn at the RAND Corporation in the USA and is continuously upgraded by Hearn and a number of other contributors throughout Europe and the USA. Specialized packages have been written by the user world throughout the years. They are loaded into REDUCE to provide extra facilities for difficult mathematical problems such as analytic integration of expression with square roots, calculus of differential geometry, determination of symmetries of partial differential equations ... etc. The version 3.3 (from July 1987) was used for the work in this thesis, although newer versions have appeared since. The machine on which it was run is an Amdhal 5860 mainframe running under the MTS† operating system.

REDUCE comprises all the general facilities mentioned in section 2.1 which are common to the modern algebraic languages. It has few built-in functions but makes provision for easily adding new rules and functions. It is written in the object-orientated language LISP[49] and it is possible to add new definitions at LISP level to cater for special needs. It is therefore a very open system which allows the user to have control at both higher and system levels. An overview of the system is now given, so as to gain familiarity with the syntax and the features used in this work. In all examples, *User input* will refer to what the user types in and *REDUCE response* to what the system prints subsequently on the screen. The REDUCE code itself will be typeset in `typewriter`.

2.3.1 Description of REDUCE

REDUCE is designed to be an interactive system, although it can be run in batch mode. In interactive mode, once the system is started, the user is prompted

† Michigan Terminal System

to enter a query. A query can be a declaration statement, an algebraic expression or a combination of both. In return, REDUCE responds by informative statements, algebraic expressions or another prompt.

Declarative statements include defining an array, a user function, a procedure, an output file ... etc. Single variables do not need to be declared. Variables in REDUCE are global in scope. Once variables have been defined they can be accessed anywhere in the program, including procedures. They can, however, be made local if desired.

Algebraic manipulation is automatically done by REDUCE according to internal rules. The user has some control over the way REDUCE manipulates expressions internally through switches which are one of its characteristic features. These switches enable the control of various aspects of the system such as the ordering of variables in expressions, the expansion of polynomials, the printing of expressions ... etc.

Only a subset of the facilities provided by REDUCE to carry out algebraic manipulation has been used in this work, therefore the presentation of the system will essentially focus on these facilities.

As most of the algebraic calculation involved in this work is concerned with polynomials, the manipulation of these quantities is of prime importance. REDUCE enables us to perform operations such as simplification and expansion of polynomials, analytical integration and diffrentiation of functions including polynomials. The following examples show how REDUCE expands polynomials, stores the result into variables and performs differentiation and integration.

user input	`P := (X+Y+Z)**2;`
REDUCE response	$\mathtt{P := X^2+ 2XY + 2XZ+ Y^2 + 2YZ + Z^2}$
user input	`D := DF(P,X);`
REDUCE response	`D := 2(X+Y+Z)`
user input	`INT(D,Y);`
REDUCE response	`Y(2X+Y+2Z)`

As can be seen in the previous examples, `DF(P,X)` performs the differentiation of the function P with respect to X, INT(D,Y) performs the integration of the function D with respect to Y. At the second line, the polynomial has been expanded, which corresponds to a default status of a switch (called EXP). If no expansion is wanted, the switch should be turned off (OFF EXP).

In the examples above, the variables X, Y and Z do not have a numerical value. They represent themselves and are as defined in mathematics. This is the basic tool which enables algebraic manipulation on the computer.

Another important feature of REDUCE is its ability to manipulate symbolic matrices. First, the difference between arrays, as defined in numerical languages, and matrices must be noted. A matrix can be globally manipulated to obtain, for example, its determinant and to perform addition or multiplication with another matrix. An array can only be manipulated element by element. Replacing the array with a matrix allows the user to perform matrix algebra more efficiently. Unlike ordinary variables, matrices, and also arrays, do not represent themselves and are initialised by default to zero when declared. They can, however, contain algebraic expressions. Matrices are of versatile use as the examples below illustrate, where a two-by-two matrix and a vector are defined and multiplied together.

```
user input           A := MAT( (a11,a12),(a21,a22) );
user input           V := MAT( (v1),(v2) );
user input           W := A*V;
REDUCE response      W(1,1)  := a11*v1 + a12*v2
                     W(2,1)  := a21*v1 + a22*v2
```

In the examples above, the quantities a11,a12, ... v1,v2 are symbolic variables. The result of the multiplication of the matrix A by the vector V is stored in the quantity W, which REDUCE recognises as being a vector and automatically evaluates its dimensions. The full range of matrix manipulation in REDUCE includes addition, multiplication, division, inversion, raising to a power, transposition, calculation of the determinant, the eigenvalues, the trace ... etc.

Yet another feature of REDUCE is its ability to perform analytical substitutions and pattern matching. The following examples illustrate how substitution and pattern matching work (the value of P is as defined in previous examples):

user input	`X := 3;  Y := U;  P;`
REDUCE response	`X := 3`
	`Y := U`
	$P := U^2 + 2UZ + 6U + Z^2 + 6Z + 9$

Unlike other Computer Algebra systems, REDUCE does not allow algebraic simplification rules to be applied selectively to an expression but applies all currently active rules until no further changes can be made to the expression. So a user input of the form:

user input	`X := Y ; Y := X ;`

will cause the system to crash because REDUCE would try to substitute X by Y, then Y by X and again X by Y, indefinitely. Nevertheless, it is possible to restrict the scope of a substitution by using the REDUCE command SUB which performes substitution within an expression and returns the resulting substituted expression. The original expression is, however, left unchanged. This is illustrated in the following examples:

user input	`INI := a*log(x)**2 + b;`
user input	`RES := SUB(log(x)=v,INI);`
REDUCE response	$RES := a*V^2 + b$
user input	`INI;`
REDUCE response	$INI := a*log(x)^2 + b;$

Since there are few built-in functions, REDUCE enables the user to easily define rules which are applied in a similar way and at the same time as the internal rules. A rule to transform trigonometrical expressions can therefore be defined as follows:

user input	`FOR ALL A,B LET SIN(A)*COS(B)= 1/2*(SIN(A+B)+SIN(A-B));`
user input	`SIN(X)*COS(Y) ;`
REDUCE response	$\frac{SIN(X+Y)+SIN(X-Y)}{2}$

As well as its symbolic capabilities, REDUCE provides all the usual control statements, including:

- Assignment (`:=`)
- Group statement (`BEGIN...END or <<...>>`)
- Loop (`FOR NB:=1:N DO`)
- Condition (`IF ... THEN ... ELSE`)
- Jump (`GO TO`)

REDUCE also performes exact arithmetic calculations. For example 2/7 remains as a rational and is not transformed into a real. This means that no truncation errors are introduced in the calculations which avoids the problems encountered in numerical languages about precision of calculations.

A program in REDUCE is built in the same way as in other languages. Tasks can be divided into sub-tasks performed by procedures which are called in turn. A program can be entered interactively or from a file (batch mode). There are also facilities to produce pretty printing expressions (natural mathematical notations).

2.3.2 The translator of code: GENTRAN

An important feature of REDUCE not yet discussed is its ability to produce FORTRAN code. This can be done using the `FORT` option in the language itself, but more comprehensive facilities are available through an additional package called GENTRAN[50]. GENTRAN can produce FORTRAN, C and RATFOR (RATional FORtran) code. GENTRAN is used within REDUCE (same syntax) to translate and generate numerical code.

The translator of code takes a REDUCE expression, statement or procedure and translates it into code in the target language. This mainly involves a change in the syntax. The code generator recognizes part of the statement or expression as needing some evaluation before translation. This is done by handing to REDUCE the pieces of code to evaluate and then translating the resulting expressions. The following examples illustrate how GENTRAN translates a loop statement and how the generation of code is achieved using the GENTRAN statement EVAL (the value of P is as defined before).

Examples:

```
user input          GENTRAN << FOR I:=1:N DO V(I):=0 >>;
REDUCE response           DO 25001 I=1,N
                          V(I) = 0.0
                    25001 CONTINUE
user input          GENTRAN << RES := EVAL(P) >>;
REDUCE response           RES = U^2 + 2.000000E0*U*Z + 6.000000E0*U +
                         Z^2 + 6.000000E0*Z + 9.000000E0
user input          GENTRAN << RES := P >>;
REDUCE response           RES = P
```

It is interesting to notice that in the examples above the executable statements in FORTRAN are aligned at column seven which is compulsory in FORTRAN. When a line extends beyong column seventy two, GENTRAN automatically truncates it and insert the remaining of the line onto the next line and makes provision for a continuation character to be place in column six.

Tools are provided by GENTRAN to insert type declarations (statement DECLARE) and comments (statement LITERAL). The type declarations are automatically inserted before any executable statement when the switch GENDECS is turned on (ON GENDECS). GENTRAN is an external package of REDUCE which needs to be loaded before being used (LOAD GENTRAN). CR!* represents a carriage return and TAB!* a FORTRAN tabulation (seven columns). The next example shows the production of a complete FORTRAN program, ready to be compiled.

user input

```
LOAD GENTRAN;

ON GENDECS;

GENTRAN

<<

    LITERAL "C test program ",CR!*;

    FOR I:=1:N DO V(I):=0;

    RES := EVAL(P);

    DECLARE ≪ V(!*) :  DIMENSION;

                  X,Y,Z,V, RES :REAL ≫;

    LITERAL TAB!*,"STOP",CR!*,TAB!*, "END"

>>;

END ;
```

REDUCE response

```
        REAL X,Y,Z,V(*),RES

 C test program

        DO 25001 I=1,N

        V(I) = 0.0

 25001  CONTINUE

        RES = U^2 + 2.000000E0*U*Z + 6.000000E0*U + Z^2 +

      . 6.000000E0*Z + 9.000000E0

        STOP

        END
```

References

1. Loos R., 'Introduction', *Computer Algebra : Symbolic and Algebraic Computation, Computing supplementum 4*, Springer-Verlag, pp 1–10, 1982.
2. Stein D., *Ada, A Life and a Legacy*, The MIT Press, 1985.
3. Dubbey J.M., *The mathematical work of Charles Babbage*, Cambridge University Press, 1978.
4. Kahrimanian H.G. and Nolan J., *Analytic Differentiation by a Digital Computer*, MA thesis, Temple Univ. Phil., PA. and Math. Dept., M.I.T Cambridge, Mass., 1953.

5. Brown W.S., 'The ALPAK System for Nonnumerical Algebra on a Digital Computer — I: Polynomials in Several Variables and Truncated Power Series with Polynomial Coefficients', *Bell Systems Truncated Journal*, **42**, pp 2081–2119, 1963.

6. Bond E. *et al*, 'FORMAC an Experimental Formula Manipulation Compiler', *Proceedings of the 19th ACM Conference*, pp K2.1-1–K2.1-18, 1964.

7. Davenport J.H., Siret Y. and Tournier E., *Computer Algebra : Systems and Algorithms for Algebraic Computation*, p vii, Academic Press, 1988.

8. Wooff C. and Hodgkinson D., *muMATH : A Microcomputer Algebra System*, Academic Press, pp 2–4, 1987.

9. van Hulzen J.A., 'Computer Algebra Systems', *Computer Algebra : Symbolic and Algebraic Computation, Computing supplementum 4*, edited by Buchberger B., Collins G.E. and Loos R., pp 221–243, Springer-Verlag, 1982.

10. REDUCE is available from the RAND Corporation, 1700 Main Street, Santa Monica, CA, 90406-2138, USA.

11. Macsyma-Symbolics Ltd, St. John's Court, Easton St, High Wycombe, Bucks, HP11 1JX, UK.

12. Scratchpad is available from IBM Research Division, T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, Ny 10598, USA.

13. SMP-Inference Corporation, 5300 West Central Building, Los Angeles, CA 90045, USA.

14. Mathematica is available from Wolfram Research Inc., PO Box 6059, Champaign, 61821, USA.

15. Waterloo Maple Software Inc., 160 Columbia Street, W., Waterloo, Ontario, Canada, N2L 3L3.

16. MuMath is available from Soft Warehouse Inc., 3615 Harding Avenue, Suite 505, Honolulu, Hawaii 96816, USA.

17. Derive is available from Soft Warehouse Inc., 3615 Harding Avenue, Suite 505, Honolulu, Hawaii 96816, USA.

18. Harper D., 'A Guide to Computer Algebra Systems', *Computer Algebra Support Project*, University of Liverpool, P.O Box 147, Liverpool, L69 3BX, fourth edition, March 1990.

19. Computer Algebra Support Officer, Computing Laboratory, University of Liverpool, P.O. Box 147, Liverpool, L69 3BX. Tel: 051–794 3755. Email: Algebra@ uk.ac.liverpool.

20. SIGSAM–ACM, Special Interest Group in Symbolic and Algebraic Manipulation, 11 Wesr 42nd. St., NY 10036, U.S.A.

21. Rayna G., *REDUCE — Software for Algebraic Computation*, Springer, 1987.

22. Maeder R., *Programming for Mathematica*. Addison-Wesley, 1989.

23. van der Waerden B.L.. *Modern Algebra*, Frederick Ungar, 1953.

24. Aho, Hopcroft and Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

25. Buchberger B., Collins G.E. and Loos R., *Computer Algebra : Symbolic and Algebraic Computation, Computing supplementum 4*, Springer-Verlag, 1982.

26. Sims C.S., *Abstract Algebra : A Computational Approach*, Wiley, 1984.

27. Calmet J., 'Computer Algebra Applications', *Computer Algebra : Symbolic and Algebraic Computation, Computing supplementum 4*, edited by Buchberger B., Collins G.E. and Loos R., pp 245–258, Springer-Verlag, 1982.

28. Hosack J.M. 'A guide to Computer Algebra Systems', *The College Mathematics Journal*, 17,5, pp 434–441, 1986.

29. Hodgkinson D., 'The use of Computer Algebra in teaching', *IUSC Workshop on Algebraic Computing*, University of Liverpool, 4–5 July 1989.

30. Levi I.M., 'Symbolic Algebra by Computer — Applications to Structural Mechanics', *AIAA, 12th Structure, Structural Dynamics and Materials Conference*, Anaheim, California, 19–21 April, 1971.

31. Pedersen P. and Megahed M.M., 'Axisymmetric Element Analysis using Analytical Computing', *Computers and Structures*, **5**, pp 241–247, 1975.

32. Pedersen P., 'On Computer-Aided Analytic Element Analysis and the Similarities of Tetrahedron Elements', *International Journal for Numerical Methods in Engineering*, **11**, pp 61–622, 1977.

33. Ladefoged T., 'Triangular Ring Element with Analytic Expressions for Stiffness and Mass Matrix', *Computer Methods in Applied Mechanics and Engineering*, **67**, pp 171–187, North-Holland, 1988.

34. Jensen J. and Niordson F., 'Symbolic and Algebraic Manipulation Languages and their Applications in Mechanics', *Structural Mechanics Software Series*, Volume 1, Editors Perrone N. and Pilkey W., University Press of Virginia, Charlottesville.

35. Cohen J., 'Symbolic and Numerical Computer Analysis of the Combined Local and Overall Buckling of Rectangular Thin-Walled Columns', *Computer Methods in Applied Mechanics and Engineering*, **7**, Jan. 1976.

36. Noor A.K and Andersen C.M., 'Computerized Symbolic Manipulation in Nonlinear Finite Element Analysis', *Computers and Structures*, **13**, pp 379–403, June 1981.

37. Crespo Da Silva and Marcela R.M., 'The Role of Computerized Symbolic Manipulation in Rotorcraft Dynamics Analysis', *Computer and Mathematics with Applications*, **12A**, Iss 1., pp 161–172, 1986.

38. Anonymous referee's comments from the International Journal for Numerical Methods in Engineering.

39. SCAFI'91, *Studies in Computer Algebra for Industry*, 10-11 December 1991, Computer Algebra Amsterdam (CAN), Amsterdam, The Netherlands.

40. Lee X.G. and Dasgupta G., 'Analysis of Structural Variability with Computer Algebra', *Journal of Engineering Mechanics-ASCE*, **114**, Iss 1, pp 161–171, 1988.

41. Bardnell N.S., 'The application of Symbolic Computing to the Hierarchical Finite Element Method', *International Journal for Numerical Methods in Engineering*, **28**, Iss 5, pp 1181–1204, 1989.

42. Grigorev F.N and Kistlerov V.L, 'Computer Algebra Methods used in Analysing the Stability of Linear Dynamic Systems', *Automation and Remote Control USSR*, **50**, Iss 7, pp 925–988. 1989.

43. Nishioka T. and Takemoto Y., 'Moving Finite Element Method Aided by Computerized Symbolic Manipulation and its Application to Dynamic Fracture Simulation', *JSME International Journal Series I-solid Mechanics Strength of Materials*, **32**, Iss 3, pp 403–410, 1989.

44. Ioakimidis N.I, 'Symbolic Computation — A Powerful Method for the Solution of Crack Problems in Fracture Mechanics', *International Journal of Fracture*, **43**, Iss 3, pp &39–&42, 1990.

45. Yagawa G., Ye G.W and Yoshimura S., 'A Numerical Integration Scheme for Finite Element Method based on Symbolic Manipulation', *International Journal for Numerical Methods in Engineering*, **29**, Iss 7, pp 1539–1549, 1990.

46. SENAC – A Software Environment for Numeric and Algebraic Computation, developed by the Mathematical Software Team, University of Waikato, New Zealand, distributed in Europe by the University of London Computer Centre, 20 Guilford Street, London, WC1N 1DZ.

47. SENAC workshop, University of London Computer Centre, 23rd October 1991.

48. Hearn A.C., *REDUCE User's Manual*, RAND Publication CP78, Rev. 7/87, July 1987.

49. Winston P.H. and Horn B.K.P., *LISP*, Addison-Wesley publishing Company, 1981.

50. Gates B.L., *GENTRAN user's Manual, REDUCE version*, Information Sciences Department, The RAND Corporation, P.O box 2138, 1700 Main Street, Santa Monica, CA 90406-2138, U.S.A.

Chapter III

Automatic generation of shape functions

3.1 Introduction

In finite element analysis, the shape functions are used to interpolate field variables within elements as explained in chapter 1. Most are based on polynomials. C_0 continuous shape functions are those in which the field variable is continuous between adjoining elements but not the derivative of the field variable. The theory for different element shape functions is available in many standard text books, for example, Zienkiewicz[1].

Most of the text books do not, however, give the derivatives of the shape functions, and leave it to the user to expand the polynomials, form the derivatives and simplify the resulting expressions. Fortran code is available for some shape functions in the NAG library, for example. While many users have their own tried and tested shape function routines, it is useful to have access to a comprehensive library of routines. With interest turning to p-type adaptive finite element analysis, higher order shape functions may be required and they are tedious to code and check.

The work of Wang *et al*[2,3,4] covers similar ground to the present work. Wang and his co-workers have developed an interactive system called FINGER (FINite element code GEneratoR). This system allows the user to develop FORTRAN code for isoparametric elements to various levels of sophistication. At the lowest level shape functions are generated which can then be used to construct the finite element strain matrix, the material properties matrix (where this depends on non-linear constitutive relations), and the integrand of the system matrix.

The final objective of integrating this last item to produce the stiffness matrix can be achieved in certain hybrid-mixed formulations[3] but in general it is not possible because the integrand is a rational function of the local element coordinates. In their early work Wang *et al*[2] used a polynomial approximation of the integrand by assuming that the jacobian of the element could be approximated by its value at the centre of the element. This approximation may be useful for low order elements suffering from little deformation, but it is unlikely to satisfactorily replace the full isoparametric formulation. Wang's later work does not mention this approximation[4].

Kidger[5] has also developed a REDUCE program to compute the shape functions for a 14-node brick element. It is a serendipity element where extra nodes at the mid-faces have been chosen so that the shape functions are complete up to the second order. Five choices of polynomial can be made and some have a particularly good accuracy, which makes them a viable alternative to the traditional 20-node brick element. Such a program could form an extension to the program developed here. Smith[6] has also done some work on this subject.

In this chapter a method for economically generating shape functions for a wide variety of problems will be discussed. The use of REDUCE to generate two dimensional C_0 continuous shape functions was given in an earlier paper[7]. The aim of this work is to describe the formation of all the main two and three dimensional C_0 continuous shape function routines, developed from the original REDUCE program but exploiting GENTRAN to generate more efficient FORTRAN 77 code. A paper containing most of the material in this chapter has been published[8].

Both the shape functions and their derivatives have been derived. Computationally efficient expressions have been obtained using GENTRAN, which also generates FORTRAN code. GENTRAN is a tool developed by Wang and van Hulzen[2] as part of the FINGER system. It can be used with VAXIMA[4], MACSYMA and REDUCE.

A feature of the Computer Algebra approach is its great flexibility and generality. For example, some users prefer to generate the shape functions for triangular elements retaining all three area coordinates as variables, even though they are not independent. In using REDUCE it is simple to change from the two variable form to the three variable form. Again some users prefer to explicitly evaluate numerically the shape functions and their derivatives at quadrature points and store them as values, in large arrays. It is easy to adapt the REDUCE code to do this also.

In the next sections the following typesetting conventions will be used. All mathematical formulæ will be typeset in *mathfont* while the corresponding REDUCE code will be typeset in `typewriter`.

3.2 Algorithms implemented

Working with shape functions is common in finite element analysis using isoparametric elements. Designing a program to calculate shape functions involves first hand manipulations then coding into a numerical language. When using a Computer Algebra system the same steps are followed but all the actual calculation and coding are done by the machine. We will now explain how the shape functions are developed. Several types of elements have been considered. They are Lagrangian and serendipity quadrilaterals and cubes, triangles and tetrahedra, Lagrangian and serendipity triangular prisms.

The general form of the shape functions for these elements can be found for example in Zienkiewicz[1]. All the shape functions are built up from the one dimensional Lagrange polynomials for interpolating through points in one dimension.

Lagrange polynomials, of degree n are defined as:

$$\mathrm{L}_i^n(\xi) = \frac{(\xi-\xi_0)(\xi-\xi_1)\dots(\xi-\xi_{i-1})(\xi-\xi_{i+1})\dots(\xi-\xi_n)}{(\xi_i-\xi_0)(\xi_i-\xi_1)\dots(\xi_i-\xi_{i-1})(\xi_i-\xi_{i+1})\dots(\xi_i-\xi_n),} \tag{3.1}$$

where $i = 0,\dots n$, ξ is the normalized coordinate in the range $[-1,+1]$ and the element has $n+1$ nodes.

The shape functions depend upon the cardinality condition of the Lagrange polynomials,

$$\begin{aligned} \mathrm{L}_i^n(\xi_j) &= 1 \qquad \text{if} \quad i=j \\ \mathrm{L}_i^n(\xi_j) &= 0 \qquad \text{if} \quad i\neq j. \end{aligned} \tag{3.2}$$

The formation of the shape functions using REDUCE is described next for each element considered. The cartesian coordinates over the standard square (cube) are ξ, η (and ζ). The barycentric coordinates over the standard triangle (tetrahedron) are L_1, L_2, L_3 (and L_4). They are called local coordinates (see Figure 3.1). The

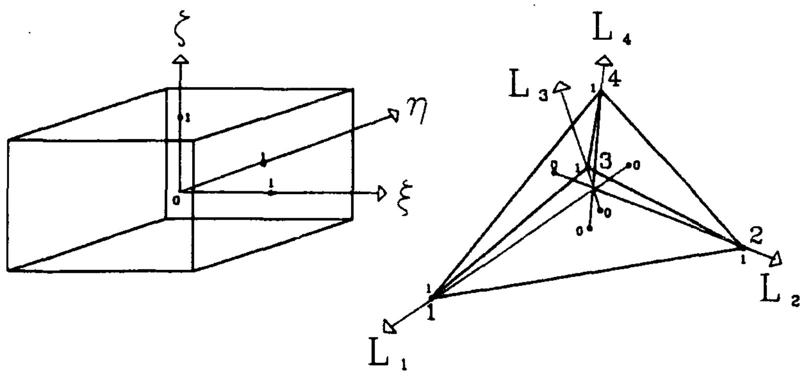


Figure 3.1 :*Local coordinate systems*

shape functions can be used to map the local coordinates to the global coordinates, ie the actual coordinates of the nodes in the mesh.

3.2.1 Lagrangian element

The shape functions for the Lagrangian element are obtained as follows:

- 2 dimensions:

$$SF_{node(i,j)}(\xi,\eta) = \mathrm{L}_i^n(\xi)\mathrm{L}_j^n(\eta). \tag{3.3}$$

- 3 dimensions:

$$SF_{node(i,j,k)}(\xi,\eta,\zeta) = \mathrm{L}_i^n(\xi)\mathrm{L}_j^n(\eta)\mathrm{L}_k^n(\zeta). \tag{3.4}$$

where ξ,η,ζ are the local coordinates.

i,j,k denote the position of the node in the standard cube (see below).

$node(i,j,k)$ is the node number of the node whose position in the standard cube is (i,j,k).

i,j,k are defined as follows :

$$\begin{aligned} i &= \frac{n}{2}(\xi+1) & -1 \leq \xi \leq 1 &\longrightarrow 0 \leq i \leq n \\ j &= \frac{n}{2}(\eta+1) & -1 \leq \eta \leq 1 &\longrightarrow 0 \leq j \leq n \\ k &= \frac{n}{2}(\zeta+1) & -1 \leq \zeta \leq 1 &\longrightarrow 0 \leq k \leq n. \end{aligned} \tag{3.5}$$

The REDUCE program carries out the calculations as described by the formulæ (3.1), (3.3) and (3.4). First the one dimensional normalized coordinates ξ_0, ξ_1, ... ξ_n are calculated and stored.

$$\xi_i = 2 * \frac{i}{n} - 1 \qquad \text{where} \qquad i = 0, \dots n. \tag{3.6}$$

The REDUCE code looks like this (using `XI` for ξ):

```
FOR I:=0:N DO
<< XI(I) := 2*I/N - 1 >>;
```

It is interesting to note that in the code above `N` does not need to hold a numerical value. The REDUCE code looks very much like any numerical language code, which makes it easy to use for people familiar with such languages.

The one dimensional Lagrange polynomials are then calculated in terms of a generic coordinate `VAR1` and stored. In other words, the Lagrange polynomials are calculated for any variable of direction `VAR1`, which can then be replaced by the relevant real variable ξ, η or ζ. The REDUCE code corresponding to equation (3.1) is given by:

```
FOR I:=0:N DO
<<  L(I) := 1;
    FOR J:=0:N DO
    <<  IF I NEQ J THEN
        << L(I) := L(I)*(VAR1-XI(J))/(XI(I)-XI(J)) >>
    >>
>>;
```

The REDUCE code, again, looks much like a numerical language code. The difference appears, however, when the values contined in the array `L` are printed which the REDUCE code shown below carries out:

```
user input               N := 2; L(0); L(1); L(2);
REDUCE response          N := 2
                         L(0) := (VAR1-1)*VAR1/2
                         L(1) := -(VAR1+1)*(VAR1-1)
                         L(2) := (VAR1+1)*VAR1/2
```

Let us now compare the expressions for L(0), L(1) and L(2) with the expressions for the one dimensional Lagrange polynomials usually derived by hand, shown below equation (3.7):

$$\begin{aligned} L_0^2(\xi) &= \frac{(\xi-1)*\xi}{2} \\ L_1^2(\xi) &= -(\xi-1)*(\xi+1) \\ L_2^2(\xi) &= \frac{(\xi+1)*\xi}{2.} \end{aligned} \tag{3.7}$$

The REDUCE expressions obviously match the equation (3.7) if we substitute VAR1 by ξ in the REDUCE expressions. This can be done using the REDUCE command SUB. The corresponding REDUCE code is given below, where ξ is denoted by XI, and the result is stored in the array SF (SF stands for Shape Function).

```
FOR I:=0:N DO
<< SF(I) := SUB (VAR1=XI, L(I)) >>;
```

As explained in chapter 2, in the section 'Description of REDUCE', the substitution above is applied locally to L(I) without affecting the value of L(I) in memory, therefore allowing further substitutions if necessary. This property will be used in the calculation of the two and three dimensional shape functions.

The two dimensional shape functions are obtained by multiplying two one-dimensional Lagrange polynomials in the ξ and η directions. REDUCE uses the one dimensional Lagrange polynomials already calculated and substitutes VAR1 (operator SUB) by the actual coordinate. The code corresponding to equation (3.3) is given next (using XI for ξ and ET for η) :

```
FOR I:=0:N DO
<<  FOR J:=0:N DO
    << SF(NODE(I,J)) := SUB(VAR1=XI,L(I))*SUB(VAR1=ET,L(J)) >>
>>;
```

In three dimensions similar code stands where the term corresponding to the third direction ζ is introduced. The REDUCE code is therefore as follows:

```
FOR I:=0:N DO
<<  FOR J:=0:N DO
    <<  FOR K:=0:N DO
        << SF(NODE(I,J,K)) := SUB(VAR1=XI,L(I))*SUB(VAR1=ET,L(J))
                              *SUB(VAR1=ZE,L(K)) >>
    >>
>>;
```

The REDUCE programming language uses dynamic array storage, which enables us to leave `N` as an unassigned variable. Therefore REDUCE is able to generate Lagrange polynomials to any high order, eg `N = 50`, which far exceeds any practical limit as far as finite element analysis is concerned.

3.2.2 Serendipity element

A geometrical interpolation of serendipity elements is given by Zienkiewicz[1]. For example the two dimensional quadratic and cubic quadrilateral shape functions are defined below. $SF_{node(i,j)}(\xi,\eta)$ takes three different values depending on the position of the node in the quadrilateral (see Figure 3.2):

$$SF_{node(i,j)}(\xi,\eta) = \begin{cases} EH_{node(i,j)}(\xi,\eta) & \text{for mid-side nodes on edges 1 and 3} \\ EV_{node(i,j)}(\xi,\eta) & \text{for mid-side nodes on edges 2 and 4} \\ C_{node(i,j)}(\xi,\eta) & \text{for corner nodes,} \end{cases} \tag{3.8}$$

where

$$EH_{node(i,j)}(\xi,\eta) = \mathrm{L}_i^n(\xi)\mathrm{L}_j^1(\eta) \qquad i = 1,\ldots n-1 \quad \text{and} \quad j = 0,1$$

$$EV_{node(i,j)}(\xi,\eta) = \mathrm{L}_i^1(\xi)\mathrm{L}_j^n(\eta) \qquad i = 0,1 \quad \text{and} \quad j = 0,\ldots n-1$$

$$C_{node(i,j)}(\xi,\eta) = \mathrm{L}_i^1(\xi)\mathrm{L}_j^1(\eta) - P_{node(i,j)}(\xi,\zeta) - Q_{node(i,j)}(\xi,\eta)$$

$$i = 0,1 \quad \text{and} \quad j = 0,1$$

and

$$\begin{aligned} P_{node(i.j)}(\xi,\eta) &= \frac{1}{2}\sum_{\mu=1}^{n-1}(1+\xi_c\xi_\mu)EH_{node(\mu.j)}(\xi,\eta) \\ Q_{node(i.j)}(\xi,\eta) &= \frac{1}{2}\sum_{\mu=1}^{n-1}(1+\eta_c\eta_\mu)EV_{node(i.\mu)}(\xi,\eta). \end{aligned} \tag{3.9}$$

(ξ_c, η_c) are the local coordinates of the corner node, $n+1$ is the number of nodes along one edge of the element, and edges are as shown in Figure 3.2.

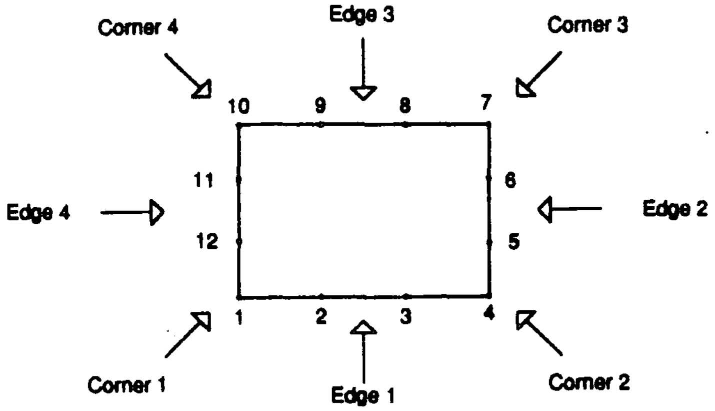


Figure 3.2 :*Cubic ,serendipity element 2D - definition of the edges*

EH corresponds to the nodes on the horizontal edges, except the corner nodes. It is composed of a $(n+1)^{th}$ order Lagrange polynomial in ξ direction and a linear Lagrange polynomial in η direction as there are $n+1$ nodes in the ξ direction and only 2 nodes in the η direction. Similarly, EV corresponds to the nodes on the vertical edges, except the corner nodes.

C is the shape function for all four corner nodes. It is formed from a bilinear function in ξ and η, from which two polynomials are subtracted. The two polynomials are weighted sums of EH and EV along the ξ and η directions. They ensure that C is equal to 1 at the corner and zero at all other points of the element. The first part of C (bilinear function) gives 1 at the corner, zero at the other corners and some non zero values at the mid-side nodes along the edges. P and Q modify C so that its value at the mid-side nodes is zero.

The calculation in three dimensions is similar. The mid-side nodes are calculated by multiplying two linear Lagrange polynomials by a $(n+1)^{th}$ order Lagrange polynomial. The corner nodes are calculated by assigning a trilinear polynomial and then subtracting the sums of weighted parts of the values of the shape functions at the mid-side nodes.

REDUCE handles the calculations as described above. The mid-side node shape functions EH and EV are constructed with suitable order Lagrange polynomials in ξ and η (and ζ in 3D) directions, using the operator SUB. The corresponding REDUCE code for two dimensional shape functions is given below. N corresponds to the degree of the Lagrange polynomial, therefore L(N,I) stores the Lagrange polynomial L_i^n. LC2D is an array containing the number of the nodes of the element as shown in Figure 3.2.

```
INDJ := 0;
FOR J:=0:1 DO
<<  FOR I:=1:N-1 DO
    COMMENT Calculation of EH;
    <<  NBNODE := LC2D (I,INDJ);
        F(NBNODE) := SUB(VAR1=XI,L(N,I))*SUB(VAR=ET,L(1,J));
    COMMENT Calculation of EV;
        NBNODE := LC2D (INDJ,I);
        F(NBNODE) := SUB(VAR1=XI,L(1,J))*SUB(VAT=ET,L(N,I))
    >>;
    INDJ := N
>>;
```

The corners are initially assigned with the bilinear (or trilinear in 3D) function using the one dimensional Lagrange polynomials. The two polynomials P and Q are not actually calculated. The program is such that as soon as one shape function for a mid-side node is obtained its weighted value is subtracted from the bilinear function. The REDUCE code for the corner nodes in two dimensions is as shown below. First the assignment with the bilinear function is carried out and then the modification by the P polynomial is shown. The modification by the Q, being similar, has been omitted.

```
COMMENT assignment with the bilinear function;
INDJ := 0;
FOR J:=0:1 DO
<<  INDI := 0
    FOR I:=0:1 DO
    <<  NVERT := LC2D (INDI,INDJ);
        F(NVERT) := SUB(VAR1=XI,L(1,I))*SUB(VAR=ET,L(1,J));
        INDI := N
    >>;
    INDJ := N
>>;
COMMENT modification with the P polynomial;
INDJ := 0;
FOR J:=0:1 DO
<<  FOR I:=1:N-1 DO
    COMMENT calculation of EH;
    <<  NBNODE := LC2D (I,INDJ);
        F(NBNODE) := SUB(VAR1=XI,L(N,I))*SUB(VAR=ET,L(1,J));
        FOR L:=0:N STEP N+1
        COMMENT calculation of the scaling factor (1+XIcXIm)/2;
        <<  SCALE := 1- ABS (L-I)/N;
            NVERT := LC2D(L,INDJ);
            F(NVERT) := F(NVERT) -SCALE*F(NBNODE)
        >>
    >>;
    INDJ := N
>>;
```

Extension to quartic and higher order serendipity polynomials requires the introduction of mid-face nodes. Although straight forward in principle it has not been done here, as such elements are not widely used.

3.2.3 Triangular and tetrahedral elements

The method given by Zienkiewicz[1] for the construction of shape functions for triangular and tetrahedral elements is followed here.

A point in a triangular element is defined by its barycentric coordinates L_1, L_2, L_3. A node can be denoted by three integers i,j,k which correspond to the transformed barycentric coordinates in the range $[0,n]$, where $n+1$ is the number of nodes along one edge of the element.

$$\begin{aligned} &i = nL_1 && 0 \leq L_1 \leq 1 \longrightarrow 0 \leq i \leq n \\ &j = nL_2 && 0 \leq L_2 \leq 1 \longrightarrow 0 \leq j \leq n \\ &k = nL_3 && 0 \leq L_3 \leq 1 \longrightarrow 0 \leq k \leq n \\ &i + j + k = n. && \end{aligned} \tag{3.10}$$

The shape functions can then be constructed by the multiplication of an $(i+1)^{th}$ order polynomial in the L_1 direction by a $(j+1)^{th}$ order polynomial in the L_2 direction and by a $(k+1)^{th}$ order polynomial in the L_3 direction. The first order polynomial is set to 1. The formula for the shape function is:

$$SF_{node(i,j,k)}(L_1, L_2, L_3) = \mathrm{L}_i^i(L_1)\mathrm{L}_j^j(L_2)\mathrm{L}_k^k(L_3), \tag{3.11}$$

where

$$\mathrm{L}_i^i(L_1) = \frac{(L_1 - L_{1(0)})(L_1 - L_{1(1)}) \ldots (L_1 - L_{1(i-1)})}{(L_{1(i)} - L_{1(0)})(L_{1(i)} - L_{1(1)}) \ldots (L_{1(i)} - L_{1(i-1)})}. \tag{3.12}$$

$L_{p(q)}$ denotes the L_p barycentric coordinate (p=1, 2 or 3) of the node at position q (q=0, 1 ... n) in the L_p direction, the first position being one the opposite edge to the corner p and the last position being at corner p (see Figure 3.3). Similar formulæ stand for $\mathrm{L}_j^j(L_2)$ and $\mathrm{L}_k^k(L_3)$.

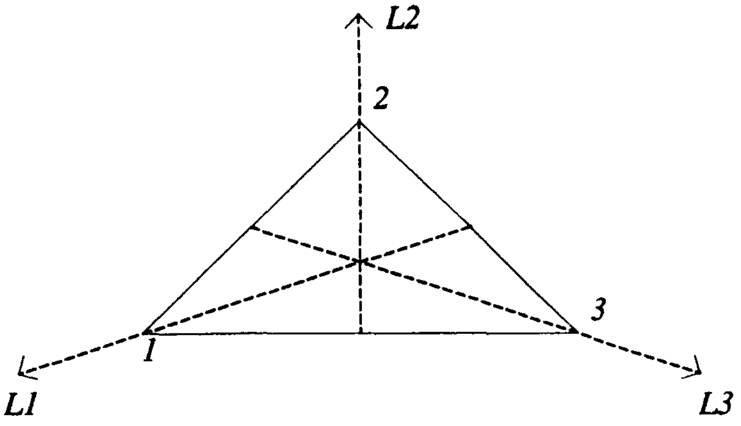


Figure 3.3 : *Coordinate definition for triangular elements*

As there can be several nodes in position q in the L_p direction the actual node used in the formula (3.12) is undetermined. However, as all nodes in position q have the same L_p coordinate the choice of the node is not important as the result will be the same in all cases. For example, a triangle (123) with 4 nodes along one edge has the following shape functions (see Figure 3.4):

- Node 1 (i,j,k)=(3,0,0)

$$\begin{aligned}
SF_1(L_1,L_2,L_3) &= 4^{th} \text{ order in } L_1 \;*\; 1^{st} \text{ order in } L_2 \;*\; 1^{st} \text{ order in } L_3\\
&= \frac{(L_1 - L_{1(10)})(L_1 - L_{1(8)})(L_1 - L_{1(5)})}{(L_{1(1)} - L_{1(10)})(L_{1(1)} - L_{1(8)})(L_{1(5)} - L_{1(5)})} * 1 * 1\\
&= \frac{(L_1 - 0)(L_1 - \frac{1}{3})(L_1 - \frac{2}{3})}{(1-0)(1-\frac{1}{3})(1-\frac{2}{3})}\\
&= \frac{L_1}{2}(3L_1 - 1)(3L_1 - 2).
\end{aligned}$$

- Node 2 (i,j,k)=(2,1,0)

$$\begin{aligned}
SF_2(L_1,L_2,L_3) &= 3^{rd} \text{ order in } L_1 \;*\; 2^{nd} \text{ order in } L_2 \;*\; 1^{st} \text{ order in } L_3\\
&= \frac{(L_1 - L_{1(9)})(L_1 - L_{1(6)})}{(L_{1(2)} - L_{1(9)})(L_{1(2)} - L_{1(6)})} * \frac{(L_2 - L_{2(5)})}{(L_{2(2)} - L_{2(5)})} * 1\\
&= \frac{9L_1}{2}(3L_1 - 1)L_2.
\end{aligned}$$

- Node 6 (i,j,k)=(1,1,1)

$$\begin{aligned}
SF_6(L_1,L_2,L_3) &= 2^{nd} \text{ order in } L_1 \;*\; 2^{nd} \text{ order in } L_2 \;*\; 2^{nd} \text{ order in } L_3\\
&= \frac{(L_1 - L_{1(9)})}{(L_{1(6)} - L_{1(9)})} * \frac{(L_2 - L_{2(8)})}{(L_{2(6)} - L_{2(8)})} * \frac{(L_3 - L_{3(2)})}{(L_{3(6)} - L_{3(2)})}\\
&= 27L_1L_2L_3.
\end{aligned}$$

The method can easily be extended to the 3 dimensional case. The 3 dimensional barycentric coordinate system L_1, L_2, L_3, L_4 is used. i,j,k,l are defined as

Node numbers	Barycentric coordinates		
	L_1	L_2	L_3
1	1	0	0
2	2/3	1/3	0
3	1/3	2/3	0
4	0	1	0
5	2/3	0	1/3
6	1/3	1/3	1/3
7	0	2/3	1/3
8	1/3	0	2/3
9	0	1/3	2/3
10	0	0	1

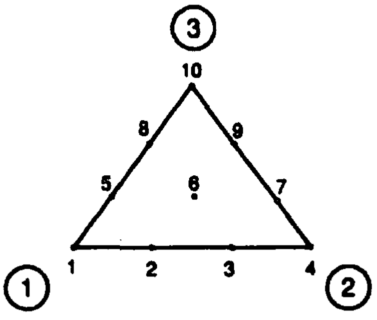


Figure 3.4: *Cubic triangular element*

follows:

$$\begin{aligned} i &= nL_1 & 0 \leq L_1 \leq 1 &\longrightarrow 0 \leq i \leq n \\ j &= nL_2 & 0 \leq L_2 \leq 1 &\longrightarrow 0 \leq j \leq n \\ k &= nL_3 & 0 \leq L_3 \leq 1 &\longrightarrow 0 \leq k \leq n \\ l &= nL_4 & 0 \leq L_4 \leq 1 &\longrightarrow 0 \leq l \leq n \\ i &+ j + k + l = n. \end{aligned} \tag{3.13}$$

The shape functions for tetrahedral elements are developed extending the formulation of the triangular element.

$$SF_{node(i,j,k.l)}(L_1, L_2, L_3, L_4) = \mathrm{L}_i^i(L_1)\mathrm{L}_j^j(L_2)\mathrm{L}_k^k(L_3)\mathrm{L}_l^l(L_4). \tag{3.14}$$

The REDUCE program calculates the shape functions in a similar way to the Lagrangian element except that, contrary to the case of the quadrilateral and cubic elements, where the polynomials are of the same order, here all lower order polynomials are used.

First, the normalized coordinates, varying between zero and 1, are calculated and stored. Then the one dimensional Lagrange polynomials for the triangle, given equation (3.12), are calculated in terms of a generic coordinate VAR2 and stored. The REDUCE code for these polynomials is given below:

```
FOR I:=0:N DO
<<  TL(I) := 1;
    IF I >= 1 THEN
    <<  FOR J:=0:I-1 DO
        << TL(I) := TL(I)*(VAR2-TXI(J))/(TXI(I)-TXI(J)) >>
    >>
>>;
```

where TXI represents the triangular one dimensional coordinate and TL stores the Lagrange polynomials for the triangle.

The calculation of the shape functions, in equations (3.11) and (3.14), is done by multiplying together the Lagrange polynomials for the triangle in L_1, L_2 and L_3 (and L_4 in 3D) using the operator SUB to substitute VAR2 by the actual barycentric coordinates. The REDUCE code for three dimensional elements is given next. In two dimensions similar code stands where the variable (B4) and the loop (FOR B4:=1:N) related to the third dimension are removed. LC3D holds the node numbers for the triangle expressed in the barycentric coordinate system denoted in the program B1, B2, B3 and B4. Since the barycentric coordinates are not independent from each other only three of them are needed, therefore it has been chosen that L_2, L_3 and L_4 will be retained in three dimensions (see section 3.3 for further details about this subject).

```
FOR B4:=0:N-1 DO
<<  LIMITB3 := N - B4;
    FOR B3:=0:LIMITB3 DO
    <<  LIMITB2 := LIMITB3 - B3;
        FOR B2:=0:LIMITB2 DO
        <<  B1 := LIMITB2 - B2;
```

```
                          NBNODE := LC3D(B2,B3,B4);
                          F(NBNODE) := SUB(VAR2=L1,TL(B1))*SUB(VAR2=L2,TL(B2))*
                                       SUB(VAR2=L3,TL(B3))*SUB(VAR2=L4,TL(B4))
                      >>
                  >>
              >>;
```

3.2.4 Triangular prisms

The triangular prism is defined by a local coordinate system which is made up of barycentric and cartesian coordinates, L_1, L_2, L_3 on the triangular faces and ζ in the perpendicular direction. The elements can be of Lagrangian or of serendipity type.

Lagrangian prism

The shape functions are derived as a product of triangular and Lagrangian one dimensional polynomials :

$$SF_{node(i,j,k,l)}(L_1, L_2, L_3, \zeta) = \mathrm{L}_i^i(L_1)\mathrm{L}_j^j(L_2)\mathrm{L}_k^k(L_3)\mathrm{L}_l^n(\zeta), \tag{3.15}$$

where $n+1$ is the number of nodes along one edge of the element, i, j and k are the transformed barycentric coordinates in the range $[0,n]$, l denotes the position of the node in the ζ direction, $l=0$ on the bottom triangular plan, $l=$n on the top triangular plan.

REDUCE handles these expressions in a similar manner to those described in the sections on Lagrangian and triangular elements.

Serendipity prism

The shape functions are obtained by combining the methods for the serendipity and triangular elements. The quadratic and cubic shape functions are obtained as follows :

$$SF_{node(i,j,k,l)}(L_1, L_2, L_3, \zeta) = \begin{cases} EH_{node(i,j,k,l)}(L_1, L_2, L_3, \zeta) & \text{for triangular faces} \\ EV_{node(i,j,k,l)}(L_1, L_2, L_3, \zeta) & \text{for rectangular faces} \\ C_{node(i,j,k,l)}(L_1, L_2, L_3, \zeta) & \text{for corner nodes,} \end{cases} \tag{3.16}$$

where

$$\begin{aligned}
EH_{node(i,j,k,l)}(L_1, L_2, L_3, \zeta) &= \mathrm{L}_i^i(L_1)\mathrm{L}_j^j(L2)\mathrm{L}_k^k(L3)\mathrm{L}_l^1(\zeta) & l = 0,1 \\
EV_{node(i,j,k,l)}(L_1, L_2, L_3, \zeta) &= \mathrm{L}_1^1(L_1)\mathrm{L}_1^1(L2)\mathrm{L}_1^1(L3)\mathrm{L}_l^n(\zeta) & l = 1, \dots n-1 \\
C_{node(i,j,k,l)}(L_1, L_2, L_3, \zeta) &= \mathrm{L}_1^1(L_1)\mathrm{L}_1^1(L2)\mathrm{L}_1^1(L3)\mathrm{L}_l^1(\zeta) & l = 0,1 \\
&\quad - R_{node(i,j,k,l)}(L_1, L_2, L_3, \zeta),
\end{aligned}$$

and

$$R_{node(i,j,k,l)}(L_1, L_2, L_3, \zeta) = \frac{1}{2}\sum_{\mu=1}^{n-1}(1 - \zeta_c\zeta_\mu)EV_{node(i,j,k,\mu)}(L_1, L_2, L_3, \zeta). \quad (3.17)$$

n+1 is the number of nodes along one edge of the element, i, j and k are the transformed barycentric coordinates in the range [0,n] and l denotes the position of the node on the vertical edges.

The geometrical interpretation and the handling of the calculations by REDUCE of these formulæ are similar to the serendipity and triangular elements.

3.3 Structure of the program

The program is composed of two series of modules which correspond to the analytical calculation with REDUCE and the obtention of the FORTRAN code. The first series of modules is concerned with the input of the user's options, the numbering of the element nodes and the analytical calculation of the shape functions. The second series of modules focuses on the generation of the FORTRAN code for the shape functions including the optimisation process and the name convention used for the FORTRAN subroutines generated.

These modules are organised within a main program unit. In the following sections, each of these series of modules will be examined in turn.

3.3.1 The REDUCE code

The main program unit is in charge of inputting user's choices, setting up initializations and calling the relevant procedures. The program can produce shape function FORTRAN code routines according to the user's choices. These choices are listed below:

- Generating all shape functions for given minimum and maximum orders
- Generating all shape functions of a given family and for given minimum and maximum orders
- Generating shape functions for a given family, dimension and order

This allows flexibility in the generation of shape functions and minimizes the intervention of the user.

It would normally be expected that these options were input interactively using read/write statements from and to the screen. REDUCE does not provide read statements, mainly because it is by nature an interactive system. When a program is run in batch mode, REDUCE reads the file where the program is stored and processes each set of instructions in turn. So if the parameters defining the choices above are put into a file and the file name is given to REDUCE, it will read in the parameters as if a read statement was used.

The user is therefore expected to edit the option file (called `SFOPT.R`), to write in the choices and then to run the program, which will automatically reads in the contents of `SFOPT.R`. The REDUCE command `'IN "SFOPT.R"'` is used for that purpose. The following variables are used to store the user's choices:

`OPT` : Contains the option chosen (1, 2 or 3)

`FAMILY` : Family chosen if option 2 or 3 is selected
(`L` = Lagrangian, `S` = Serendipity, `T` = Triangle,
`LP` = Lagrangian Prism, `SP` = Serendipity Prism)

`ELEMENT` : Contains the dimension of the space chosen if option 3 is selected (`D2` = 2 dimensions, `D3` = 3 dimensions)

`MIN` : Minimum number of nodes along one edge

`UPL` : Maximum number of nodes along one edge

`OUTPUT` : Selects which output is required. The user may want to generate FORTRAN code (`FO`) or examine the REDUCE form (`RE`) or both (`FORE`)

The initializations carried out by the main program unit include setting up switches, defining two output files–one for the shape function in REDUCE form, another one for the shape function in FORTRAN form–declaring the working arrays, calculating the normalized coordinates, calculating the one dimensional Lagrange polynomials and the one dimensional Lagrange polynomials for the triangle.

The program then processes the user's choices to call the relevant procedures. This consists of first calling the routines to number the nodes of the element whose shape functions are to be calculated and then calling the routine which carries out the analytical calculations.

The numbering of the element nodes uses the following conventions (see Figure 3.5). The Lagrangian element has its nodes numbered firstly along ξ, secondly along η and thirdly along ζ. The serendipity element is numbered going round the edges in 2 dimensions and up one layer after the other in 3 dimensions. The triangular element has its nodes numbered in the same way as the Lagrangian element. Both prism elements use the previous conventions.

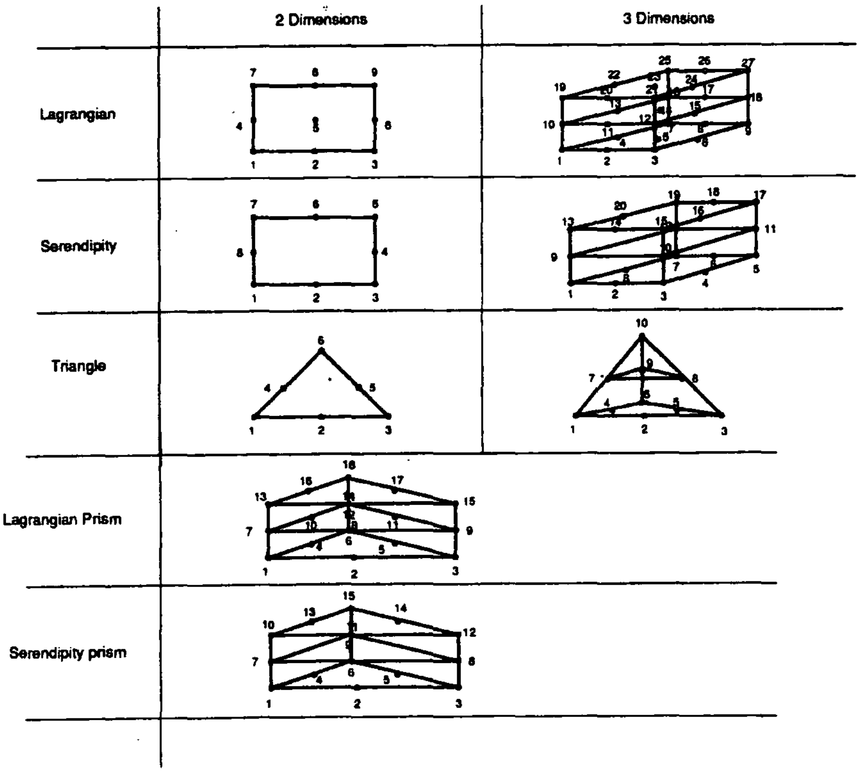


Figure 3.5 : *Node numbering conventions*

To each element corresponds one REDUCE procedure which performs the numbering. There is a test which determines whether a 2 or 3 dimensional numbering is needed. These procedures file the node numbers in the arrays `LC2D` and `LC3D`.

`LC2D(degree,col,co2)` : Contains the number of the node whose coordinates are col and co2 in 2 dimensions. '`degree`' corresponds to the number of nodes along one edge minus 1 (ie the degree of the polynomial).

`LC3D(degree,col,co2,co3)` : Contains the number of the node whose coordinates are col, co2 and co3 in 3 dimensions. '`degree`' corresponds .to the number of nodes along one edge minus 1 (ie the degree of the polynomial).

The coordinates are those described in the algorithms in the previous section. For the triangle only 2 coordinates are retained so that the arrays `LC2D` and `LC3D` can be used. The choice here is arbitrary.

The calculation of the shape functions is done as described in the previous section. There is a different REDUCE procedure corresponding to each element. Tests are made in each procedure to determine whether a two or three dimensional calculation is needed.

The program has been used to generate Lagrangian elements up to cubic, serendipity elements up to cubic, triangular and tetrahedral elements up to quartic, Lagrangian triangular prism elements up to cubic and serendipity triangular prism elements up to cubic. In the next section, the generation of the FORTRAN code will be explained and some sample of the generated FORTRAN code will also be given.

3.3.2 The FORTRAN code

The generation of FORTRAN code is done according to the option selected by the user in `SFOPT.R` file.

When the FORTRAN output (`FO` or `FORE`) is selected the REDUCE expressions are translated into FORTRAN code using the package GENTRAN and are written into a file. To form a proper FORTRAN subroutine additional code must be added to the translated shape functions.

The SUBROUTINE statement with the name of the subroutine and the arguments list are first generated (using GENTRAN command LITERAL and EVAL). Then a header of comments explaining what the subroutine does is written to the file. The type declarations are automatically generated by GENTRAN (command DECLARE). The shape functions are then translated from REDUCE followed by the shape function derivatives which are first calculated by REDUCE and then translated by GENTRAN. The two final FORTRAN statements RETURN and END are then written to the file. This produces a proper FORTRAN subroutine which calculates shape functions and derivatives for a given element.

An effort has been made to produce an efficient FORTRAN code from the REDUCE algebraic expressions for the shape functions. Indeed, the FORTRAN expressions for the shape functions produced by GENTRAN are not always optimized for numerical calculations. Here is an example:

```
SF(1)  =  (XI-1.0)*(ET-1.0)/4.0
SF(2)  =  -(XI+1.0)*(ET-1.0)/4.0
SF(3)  =  -(XI-1.0)*(ET+1.0)/4.0
SF(4)  =  (XI+1.0)*(ET+1.0)/4.0
```

This is not very good code because some expressions are calculated several times, which is time consuming. A better way of writting the same set of expressions would be:

```
T1     =  XI-1.0
T2     =  XI+1.0
T3     =  ET-1.0
T4     =  ET+1.0
SF(1)  =  T1*T3/4.0
SF(2)  =  -T2*T3/4.0
SF(3)  =  -T1*T4/4.0
SF(4)  =  T2*T4/4.0
```

This uses a bit more memory but could save a lot of time in cases where one of the Ti is used many times. REDUCE provides a command (STRUCT) which extracts common sub-expressions out of a series of expressions. This command can be applied to the shape functions which form a series of polynomial expressions. The common sub-expressions can then be stored for future use. The temporary variables T1, T2 ... etc have then to be automatically generated.

GENTRAN provides tools to automatically generate variable names using a radical (letters) followed by a number which is incremented each time a new variable is generated (command TEMPVAR). The REDUCE code to produce the optimized form of the shape functions given previously is shown below. The shape functions are stored in the array F and the common sub-expressions are stored in an array called COMSUB. The temporary variable names are stored in the array TVAR.

```
COMMENT extract common sub-expressions;
STRUCT (F,COMSUB);
COMMENT calculate the number of common sub-expressions;
NBCOMSUB := LENGTH (COMSUB);
COMMENT generate the temporary variable names T1, T2 ...;
FOR I:=1:NBCOMSUB DO
<<  TVAR(I) := TEMPVAR;
    COMMENT replace the sub-expression by the temporary variable in the
             shape functions;
    FOR J:=1:NBNODE
    <<  IF MAINVAR( DEN( F(J)/COMSUB(I) )) = 0 THEN
        <<  F(J) := F(J)*TVAR(I)/COMSUB(I) >>;
    >>
>> ;
COMMENT translation into FORTRAN;
FOR I:=1:NBCOMSUB DO
    <<  GENTRAN << EVAL(TVAR(I)) := EVAL(COMSUB(I)) >>
>>;
FOR I:=1:NBNODE DO
    <<  GENTRAN << SF(EVAL(I)) := EVAL(F(I)) >>
>>;
```

In the REDUCE code above `MAINVAR` is a command which returns the leading variable in the expression. If the expression is a constante `MAINVAR` returns zero. `DEN` extracts the denominator of a rational expression. The test to find whether a sub-expression appears in a shape function consists of dividing the shape function by the sub-expression and checking that the result is a polynomial, that is to say the denominator is a constant.

The developers of GENTRAN plan that an optimisation process will be included in the future so that any FORTRAN output generated by GENTRAN will be optimised. This is worthwhile for higher order elements where the size of the generated code has been dramatically reduced.

A convention for naming the FORTRAN subroutines generated by the REDUCE program has been adopted. An element is determined by the spatial dimension, the type (Lagrangian, Serendipity... etc) and the number of nodes along one edge of the element. There are as many subroutines as elements. The name of the subroutine tells which element it is. It is formed of 5 digits which are:

`SFdtn` where `SF` stands for Shape Function
`d` is the dimension (1,2 or 3)
`t` is the type of the element (`L`(agrangian), `S`(erendipity), `T`(riangle), `P`(lagrangian Prism), `Q`(serendipity prism))
`n` is the number of nodes along one edge of the element

In FORTRAN form the shape functions are output as an array `SF(a)` where `a` is the node number. The calculation of the shape function derivatives is done just before the output to the file (whether it is in FORTRAN or REDUCE form). So there is no need to store them in the REDUCE program. The differentiation with respect to each coordinate is carried out and the result is output in FORTRAN form as an array `SFDL(a,b)` where `a`=1,2 or 3 specifies with respect to which coordinate the differentiation is carried out (ξ, η, ζ, L_1, L_2, L_3 or L_4) and `b` is the node number.

A problem arises for the triangle in the choice of the coordinates taken into account for the derivatives since there are 3 coordinates in 2 dimensions and 4 coordinates in 3 dimensions. It is necessary for the rest of the finite element analysis to obtain the derivatives of the shape functions with respect to 2 coordinates in 2 dimensions and 3 coordinates in 3 dimensions. One of the barycentric coordinate has to be eliminated.

The choice is not arbitrary. It is motivated by the fact that a positive jacobian is needed for the transformation of local coordinates to global coordinates in the next step of the finite element analysis. This has led us to retain L_2 and L_3 in 2 dimensions and L_2, L_3 and L_4 in 3 dimensions. It is actually quite simple for the user to modify the code so as to select other choices here.

Finally, an example of an automatically generated and optimised FORTRAN subroutine is given in Figure 3.6.

```
      SUBROUTINE  SF2S4 (XI, ET, SF, SFDL, ISFDL)
C ***SHAPE FUNCTION SUBROUTINE
C (c) Christine Barbier , 1989
C -----------------------------------------------------------------
C PURPOSE :
C          Forms element shape function and derivative
C
C ARGUMENTS IN :
C
C          XI     : First co-ordinate
C          ET     : Second co-ordinate
C          ISFDL  : 1st dimension of shape function derivative array
C
C ARGUMENTS OUT :
C
C          SF     : Shape function array
C          SFDL   : Array of shape function derivatives with respect to
C                   local co-ordinates
C
C *******************************************************************
C
      INTEGER ISFDL
      DOUBLE PRECISION SF(*),SFDL(ISFDL,*),XI,ET
C
C*** Form the element shape functions
C
      DOUBLE PRECISION T1,T13,T12,T11,T10,T9,T2,T 7,T8,T14,T6,T4,T5,T17,
     . T16,T3,T15
      T1=9.000000E0*ET**2+9.000000E0*XI**2-1.000000E1
      T2=ET-1.000000E0
      T3=XI-1.000000E0
      T4=3.000000E0*XI-1.000000E0
      T5=XI+1.000000E0
      T6=3.000000E0*XI+1.000000E0
      T7=3.000000E0*ET-1.000000E0
      T8=ET+1.000000E0
      T9=3.000000E0*ET+1.000000E0
      T10=9.000000E0*ET**2+2.700000E1*XI**2-(1.800000E1*XI)-1.000000E1
      T11=9.000000E0*XI**2-(2.000000E0*XI)-3.000000E0
      T12=9.000000E0*XI**2+2.000000E0*XI-3.000000E0
      T13=9.000000E0*ET**2+2.700000E1*XI**2+1.800000E1*XI-1.000000E1
      T14=2.700000E1*ET**2-(1.800000E1*ET)+9.000000E0*XI**2-1.000000E1
      T15=9.000000E0*ET**2-(2.000000E0*ET)-3.000000E0
      T16=9.000000E0*ET**2+2.000000E0*ET-3.000000E0
      T17=2.700000E1*ET**2+1.800000E1*ET+9.000000E0*XI**2-1.000000E1
      SF(1)=T2*T3*T1/3.200000E1
      SF(2)=-(9.000000E0*T5*T2*T3*T4)/3.200000E1
      SF(3)=9.000000E0*T5*T2*T3*T6/3.200000E1
      SF(4)=-(T5*T2*T1)/3.200000E1
      SF(5)=9.000000E0*T5*T8*T2*T7/3.200000E1
      SF(6)=-(9.000000E0*T5*T8*T2*T9)/3.200000E1
      SF(7)=T5*T8*T1/3.200000E1
      SF(8)=-(9.000000E0*T5*T8*T3*T6)/3.200000E1
      SF(9)=9.000000E0*T5*T8*T3*T4/3.200000E1
      SF(10)=-(T8*T3*T1)/3.200000E1
      SF(11)=9.000000E0*T8*T2*T3*T9/3.200000E1
      SF(12)=-(9.000000E0*T8*T2*T3*T7)/3.200000E1
C
C***Form the shape function derivatives
C
      SFDL(1,1)=T2*T10/3.200000E1
      SFDL(1,2)=-(9.000000E0*T11*T2)/3.200000E1
      SFDL(1,3)=9.000000E0*T2*T12/3.200000E1
      SFDL(1,4)=-(T13*T2)/3.200000E1
      SFDL(1,5)=9.000000E0*T8*T2*T7/3.200000E1
      SFDL(1,6)=-(9.000000E0*T8*T2*T9)/3.200000E1
      SFDL(1,7)=T13*T8/3.200000E1
      SFDL(1,8)=-(9.000000E0*T8*T12)/3.200000E1
      SFDL(1,9)=9.000000E0*T11*T8/3.200000E1
      SFDL(1,10)=-(T8*T10)/3.200000E1
      SFDL(1,11)=9.000000E0*T8*T2*T9/3.200000E1
      SFDL(1,12)=-(9.000000E0*T8*T2*T7)/3.200000E1
      SFDL(2,1)=T3*T14/3.200000E1
      SFDL(2,2)=-(9.000000E0*T5*T3*T4)/3.200000E1
      SFDL(2,3)=9.000000E0*T5*T3*T6/3.200000E1
      SFDL(2,4)=-(T5*T14)/3.200000E1
      SFDL(2,5)=9.000000E0*T5*T15/3.200000E1
      SFDL(2,6)=-(9.000000E0*T5*T16)/3.200000E1
      SFDL(2,7)=T5*T17/3.200000E1
      SFDL(2,8)=-(9.000000E0*T5*T3*T6)/3.200000E1
      SFDL(2,9)=9.000000E0*T5*T3*T4/3.200000E1
      SFDL(2,10)=-(T3*T17)/3.200000E1
      SFDL(2,11)=9.000000E0*T3*T16/3.200000E1
      SFDL(2,12)=-(9.000000E0*T3*T15)/3.200000E1
C
```

```
      RETURN
      END
      SUBROUTINE  SF3S2 (XI, ET, ZE, SF, SFDL, ISFDL)
C ***SHAPE FUNCTION SUBROUTINE
C (c) Christine Barbier , 1989
C ----------------------------------------------------------------
C PURPOSE :
C          Forms element shape function and derivative
C
C ARGUMENTS IN :
C
C          XI     : First co-ordinate
C          ET     : Second co-ordinate
C          ZE     : Third co-ordinate
C          ISFDL  : 1st dimension of shape function derivative array
C
C ARGUMENTS OUT :
C
C          SF     : Shape function array
C          SFDL   : Array of shape function derivatives with respect to
C                   local co-ordinates
C
C ********************************************************************
C
      INTEGER ISFDL
      DOUBLE PRECISION SF(*),SFDL(ISFDL,*),XI,ET,ZE
C
C*** Form the element shape functions
C
      DOUBLE PRECISION T1,T6,T2,T4,T3,T5
      T1=ZE-1.000000E0
      T2=ET-1.000000E0
      T3=XI-1.000000E0
      T4=XI+1.000000E0
      T5=ET+1.000000E0
      T6=ZE+1.000000E0
      SF(1)=-(T2*T3*T1)/8.000000E0
      SF(2)=T2*T1*T4/8.000000E0
      SF(3)=-(T5*T1*T4)/8.000000E0
      SF(4)=T5*T3*T1/8.000000E0
      SF(5)=T2*T3*T6/8.000000E0
      SF(6)=-(T2*T6*T4)/8.000000E0
      SF(7)=T5*T6*T4/8.000000E0
      SF(8)=-(T5*T3*T6)/8.000000E0
C
C***Form the shape function derivatives
C
      SFDL(1,1)=-(T2*T1)/8.000000E0
      SFDL(1,2)=T2*T1/8.000000E0
      SFDL(1,3)=-(T5*T1)/8.000000E0
      SFDL(1,4)=T5*T1/8.000000E0
      SFDL(1,5)=T2*T6/8.000000E0
      SFDL(1,6)=-(T2*T6)/8.000000E0
      SFDL(1,7)=T5*T6/8.000000E0
      SFDL(1,8)=-(T5*T6)/8.000000E0
      SFDL(2,1)=-(T3*T1)/8.000000E0
      SFDL(2,2)=T1*T4/8.000000E0
      SFDL(2,3)=-(T1*T4)/8.000000E0
      SFDL(2,4)=T3*T1/8.000000E0
      SFDL(2,5)=T3*T6/8.000000E0
      SFDL(2,6)=-(T6*T4)/8.000000E0
      SFDL(2,7)=T6*T4/8.000000E0
      SFDL(2,8)=-(T3*T6)/8.000000E0
      SFDL(3,1)=-(T2*T3)/8.000000E0
      SFDL(3,2)=T2*T4/8.000000E0
      SFDL(3,3)=-(T5*T4)/8.000000E0
      SFDL(3,4)=T5*T3/8.000000E0
      SFDL(3,5)=T2*T3/8.000000E0
      SFDL(3,6)=-(T2*T4)/8.000000E0
      SFDL(3,7)=T5*T4/8.000000E0
      SFDL(3,8)=-(T5*T3)/8.000000E0
C
      RETURN
      END
```

Figure 3.6 : *FORTRAN generated code*

3.4 Tests, performance and conclusions

3.4.1 Tests

To check the validity of the FORTRAN code produced by GENTRAN, several tests were carried out on the shape functions and their derivatives. Sources of errors have been searched for in both the REDUCE and FORTRAN codes.

Firstly, the REDUCE code may contain errors. Simple tests have been carried out at this stage. The *REDUCE responses* for the low order elements have been printed out and checked against calculations from other sources. Secondly, the REDUCE/GENTRAN packages can contain faults, although it is less likely than programming mistakes. Problems can arise at the translation stage. This has been encountered and the problem was related to loss of parentheses in the denominator of rational polynomials. This problem has been reported.

The first test checks that the cardinality property of the shape functions is respected. The tests have been carried out up to cubic elements although they could be easily extended to higher orders. For each element the sum of the shape functions at each node should be equal to 1. Equation (3.18) gives the form of the test for a quadrilateral Lagrangian element :

$$\sum_{\xi=-1}^{1} \sum_{\eta=-1}^{1} SF_{node(i,j)}(\xi,\eta) = 1 \qquad for \quad i = 0,\dots n \quad , \quad j = 0,\dots n. \tag{3.18}$$

where $n+1$ is the number of nodes along one edge of the element.

At each node the sum of the shape functions derivatives with respect to the k^{th} coordinate should be equal to 0. Equation (3.19) gives the expression of the test for a quadrilateral Lagrangian element :

$$\sum_{\xi=-1}^{1} \sum_{\eta=-1}^{1} SFDk_{node(i,j)}(\xi,\eta) = 1 \qquad for \quad i = 0,\dots n \ , \ j = 0,\dots n \ , \ k = 1,2, \tag{3.19}$$

where $SFDk$ represents the derivative with respect to the k^{th} coordinate (ξ, η).

3.4.2 Performance

A interesting question was whether it was better to create the shape functions as products of one dimensional shape functions or to expand them completely as it is done in the generated FORTRAN routines produced by the REDUCE program.

To determine which is better, FORTRAN software for all shape functions has been written directly by hand. This software uses the one dimensional Lagrange polynomials, the one dimensional Lagrange polynomials for the triangle and their derivatives produced by REDUCE and translated by GENTRAN. The FORTRAN code written by hand reflects the REDUCE code in the way it works. The two and three dimensional shape functions and their derivatives have been calculated by forming products of the one dimensional Lagrange polynomials and their derivatives. The numerical results obtained with the above software have been checked to be identical to those obtained with the REDUCE translated code.

The performance of the REDUCE translated code have been evaluated in absolute value, in comparison with the software directly coded by hand in FORTRAN and the NAG library routines. The results for the quadratic and cubic elements are given in Table 3.1 and 3.2. The times have been obtained on a mainframe Amdahl 5860 computer running under MTS† operating system. The IBM FORTRANVS V.1.3.2 FORTRAN compiler has been used. Timings have been obtained without using the optimiser of the FORTRAN compiler.

The results in Tables 3.1 and 3.2 show that in most cases the REDUCE translated code is faster than the FORTRAN direct code and the NAG library.

Quadratic elements	2 dimensions			3 dimensions		
	FD	NG	RT	FD	NG	RT
Lagrangian	80	/	29	263	/	137
Serendipity	147	40	24	449	401	95
Triangle	90	112	14	196	268	26

All times are in μs, RT : Reduce translated, NG : NAG library routines,FD : Fortran Directly written by hand.

Table 3.1 : *Performance of REDUCE translated code for quadratic elements*

The REDUCE translated code contains only assignment statements and the time is largely spent doing arithmetic operations. It is likely to run faster than the code written directly which contains calls to procedures, loops ... etc in addition to assignments and arithmetic operations. The best speed-ups have been obtained for the serendipity and triangular elements where the REDUCE translated code runs on average 75% faster than the direct FORTRAN. The Lagrangian elements give average speed-up for quadratic elements but quite poor speed-up for cubic elements. Indeed, the expanded expressions for the Lagrangian elements are complicated because the polynomials used are of higher order than for the serendipity and triangular elements.

† Michigan Terminal System

Cubic elements	2 dimensions			3 dimensions		
	FD	NG	RT	FD	NG	RT
Lagrangian	107	/	91	546	/	546
Serendipity	217	85	56	770	710	176
Triangle	115	231	47	333	598	101

All times are in μs, RT : Reduce translated, NG : NAG library routines,FD : Fortran Directly written by hand.

Table 3.2 :*Performance of REDUCE translated code for cubic elements*

The REDUCE translated code for serendipity elements compares well with the NAG library routines. The times for the NAG routines for the triangle are to be considered with caution as the calculations are carried out differently. The REDUCE translated code and the direct FORTRAN code use the barycentric coordinates in the calculations and as arguments in the subroutine statement. The NAG library uses the cartesian coordinates in the subroutine statement and the barycentric coordinates in the calculations which implies extra calculation to transform the cartesian coordinates into barycentric coordinates. Lagrangian shape functions are not available in the NAG library.

3.4.3 Conclusions

An alternative method for generating numerical code for shape functions has been presented which uses Computer Algebra. It is comprehensive and accurate because most algebraic manipulations are carried out by the machine and FORTRAN code is generated automatically from the resulting expressions. The shape function derivatives are also provided. The FORTRAN code obtained is optimized and its performance compares well with other methods and existing libraries.

In the next chapter the case of infinite elements will be studied. The interpolation functions for these elements can be automatically derived using similar methodology to that presented in this chapter.

References

1. Zienkiewicz O.C., *The Finite Element Method*, 3rd edn, McGraw-Hill, 1977.
2. Wang P.S., Chang T.Y.P and van Hulzen J.A., 'Code generation and optimization for finite element analysis', *Proc. EUROSAM '84, London*, pp 237-247, 9-11 July 1984.
3. Wang P.S., Tan H., Saleeb A. and Chang T.Y., 'Code generation for hybrid mixed mode formulation in finite element analysis', *ACM SYMSAC'86 Conference*, University of Waterloo, Canada, 21-23 July.
4. Wang P.S., 'FINGER : a symbolic system for automatic generation of numerical programs in finite element analysis', *Journal for Symbolic Computation*, **2**, pp 305-316, 1986.
5. Kidger D.J, 'The 14 node brick element', *Proceedings of the second annual Robert J. Melosh Medal Paper competition*, Duke University, N.C., March 1990, to be published in a special issue of *Finite Elements in ,Analysis and Design.*
6. Smith I.M., 'Are there any new elements', *The finite element method in the 1990's*, Editors Oñate E., Periaux J. and Samuelsson A., Cimne Barcelona, pp 109-118, 1991.
7. Bettess P. and Bettess J.A., 'Automatic Generation of Shape Function Routines', Paper S20 in *Numerical Techniques for Engineering and Design*, Proceedings of the International Conference on Numerical Methods in Engineering: Theory and Applications, NUMETA '87, Swansea, Vol. II, Matinus Nijhoff, Dordrecht, 1987.
8. Barbier C., Clark P.J, Bettess P. and Bettess J.A, 'Automatic generation of shape functions for finite element analysis using REDUCE', *Engineering Computation*, **7**, pp 349-358, Dec. 1990.

Chapter IV

Automatic generation of mapping functions for infinite elements

4.1 Introduction

As is now known, infinite elements can be used to extend the finite element method to unbounded domain problems, for example behaviour of dam foundations and flow past aerofoils. A number of papers have been published on this subject[1,2,3,4]. One particularly effective technique is the mapped infinite element.

The first explicitly stated mapping was by Beer and Meek[5] who used a mapping which included a term of the form $1/(1-\xi)$ which maps a finite ξ domain onto an infinite x domain. Their mapping was in two sections, linear and non-linear. The second part of the mapping is similar to that proposed later by Zienkiewicz[6]. They also used a standard Gauss-Legendre numerical integration over the finite ξ domain. The Pissanetzky[7,8] approach is similar, but he carried out the integration in the infinite domain, and so had to modify the Gauss-Legendre abscissæ and weights. However, the mapping method has the benefit of retaining the finite element quadrature rule.

The Zienkiewicz approach leads to a clarification and simplification of the class of methods introduced by Beer and Meek and Pissanetsky. The form in which the Zienkiewicz mapping was originally given has been simplified and systematized by Marques and Owen[9], who worked out and tabulated the mapping functions for a large range of commonly used infinite elements. The simplification was also proposed by Kumar[10].

This chapter focuses on the automatic generation of these mapping functions using Computer Algebra. The method used is similar to that presented in the previous chapter on shape functions. Most of the material in this chapter is contained in a paper submitted for publication [11]. The background information is derived from a draft chapter of a book being written by Peter Bettess, who I acknowledge here.

4.2 Mapped infinite element

The main characteristic of the Zienkiewicz method for deriving mapped infinite elements is the mapping used for the shape functions and the one for the numerical integration, usually Gauss-Legendre, are identical. This has the advantage that the original Gauss-Legendre abscissæ and weights are retained. The only change needed to a finite element routine to make the element infinite is a new computation of the Jacobian matrix. The theory for the one dimensional elements will be briefly derived next.

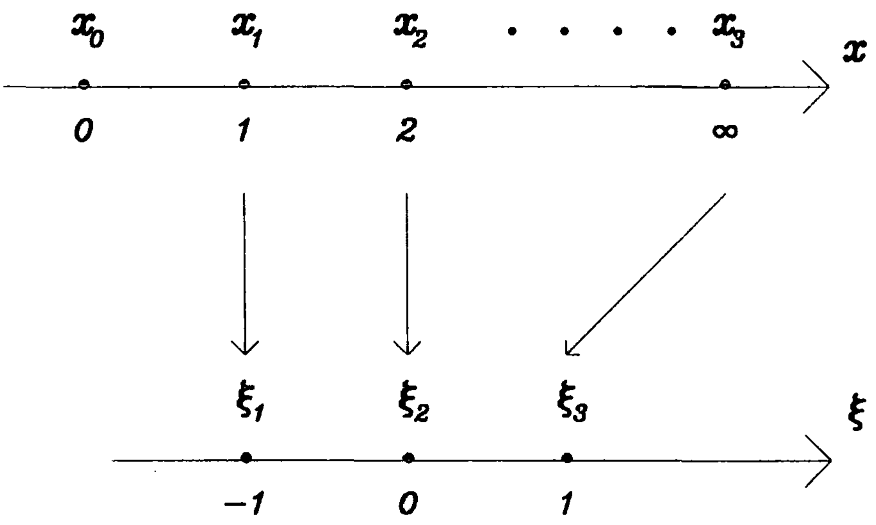


Figure 4.1 : *Infinite element mapping*

Consider first the geometry of the one-dimensional problem as shown in Figure 4.1. The element extends from point x_1 through x_2 to x_3, which is at infinity. x_0 is taken to be the 'pole' of the radial behaviour. This element is to be mapped onto the finite domain $-1 < \xi < 1$. A suitable mapping expression is:

$$x = \tilde{N}_0(\xi)x_0 + \tilde{N}_2(\xi)x_2, \tag{4.1}$$

where

$$\begin{aligned}\tilde{N}_0(\xi) &= \frac{-\xi}{1-\xi}\\ \tilde{N}_2(\xi) &= 1 + \frac{\xi}{1-\xi}\,.\end{aligned} \tag{4.2}$$

Indeed,

$$\begin{aligned}&At \quad \xi = 1, && x = \frac{\xi}{(1-\xi)}(x_2 - x_0) + x_2 = x_3 = \infty\\ &\xi = 0, && x = x_2\\ &\xi = -1, && x = (x_0 + x_2)/2 = x_1.\end{aligned} \tag{4.3}$$

The point at $\xi = -1$ is to correspond to the point x_1, which is now defined to be midway between x_0 and x_2. It implies that the inner half of the infinite element, from the 'pole' to the inner boundary of the infinite element, has the same extent as in the finite domain.

A mapping between the finite and infinite domains has now been established. The next step is to see into what form polynomials in the finite ξ domain are transformed in the unbounded x domain. Consider a polynomial, P,

$$P = \alpha_0 + \alpha_1\xi + \alpha_2\xi^2 + \alpha_3\xi^3 + \ldots, \tag{4.4}$$

which is typical of those used in finite element methods. The ξ to x mapping already given in equation (4.1) can be written:

$$x = x_0 + \frac{2a}{(1-\xi)}\,, \tag{4.5}$$

where $a = x_2 - x_1 = x_1 - x_0$. Its inverse is:

$$\xi = 1 - \frac{2a}{(x - x_0)}\,, \tag{4.6}$$

and where $r = x - x_0$, these can be written as

$$r = \frac{2a}{1-\xi} \quad \text{and} \quad \xi = 1 - \frac{2a}{r}\,. \tag{4.7}$$

On substitution into the general polynomial, P, a new polynomial in inverse powers of r is obtained:

$$P = \beta_0 + \frac{\beta_1}{r} + \frac{\beta_2}{r^2} + \frac{\beta_3}{r^3} + \ldots, \tag{4.8}$$

where the β_i's can be determined from a and the α's. If the polynomial is required to decay to zero at infinity then $\beta_0 = 0$.

It can be seen from equation (4.7) that there is a strict relation between ξ and r, and this should be adhered to when placing the nodes of the infinite element in the radial direction. Specific values are given in Table 4.1. For example, when using the quadratic element, if the first node is at a distance a from the 'pole' of the problem, in order to obtain the appropriate mapping, the mid-side node must be at a distance $2a$. If the nodes are put at other positions, the results will not necessarily be wrong, but may be unpredictable. Certainly, the polynomial in ξ will not map into a form like equation (4.8).

ξ	-1	-1/2	-1/3	0	1/3	1/2	1
r	a	$4a/3$	$3a/2$	$2a$	$3a$	$4a$	∞

Table 4.1 : *Relation between ξ and r, for mapped infinite elements*

Many exterior potential problems have solutions in the form of equation (4.8) and the advantage of this mapping is that they can be modelled using ordinary finite element polynomials. Any degree of accuracy can be obtained by adding extra terms to the series equation (4.8). The point x_0 is seen to be the pole of the expansion of P. The advantage of this approach is that the finite element domain is used for the definition of the shape functions and for the numerical integration. No changes need to be made to the element shape function routines, or to the integration abscissæ and weights. The only alteration needed is that the Jacobian matrix is calculated using the mapping, equation (4.1), and not using derivatives of shape functions.

In some respects it is more convenient to relate the mapping to the element nodes. This can be achieved simply by changing the mapping functions, as was done by Marques and Owen[9]. A similar procedure was suggested by Kumar[10].

In the Marques and Owen formulation $\tilde{N}_0$ and $\tilde{N}_2$ are replaced by mapping functions M_1 and M_2 so that:

$$x = M_1 x_1 + M_2 x_2. \tag{4.9}$$

It is easy to work out the forms of these functions. The mapping function $\tilde{N}_0(\xi)$ has the value 1/2 at $\xi = -1$ and 0 at $\xi = 0$ and tends to $-\infty$ as ξ tends to 1. We seek a mapping function M_1 which will behave in the same way at $\xi = 0$ and $\xi = 1$, but will be 1 when $\xi = -1$. Clearly the correct expression for M_1 is $2 \times \tilde{N}_0(\xi)$. N_2 can be evolved in a similar fashion. Since $\tilde{N}_2 = 1/2$ when $\xi = -1$ and $\tilde{N}_2 = 1$ when $\xi = 0$, while tending to ∞ when ξ tends to 1, a suitable choice for M_2 is $\tilde{N}_2(\xi) - \tilde{N}_0(\xi)$. The new mapping functions are shown in Table 4.2. The mapping functions for the 'last' node, the node at infinity, are not given, because they are not generally needed. They are also difficult to conceive of and define.

Mapping Function			$\xi =$	-1	0	1
M_1	$2 \times \tilde{N}_0(\xi)$	$-2\xi/1-\xi$		1	0	$-\infty$
M_2	$\tilde{N}_2(\xi) - \tilde{N}_0(\xi)$	$(1+\xi)/(1-\xi)$		0	1	∞

Table 4.2 : *Infinite element mapping functions*

Now consider the standard Lagrange type finite element shape functions for a one dimensional quadratic element (see Figure 4.2). The three nodes are conventionally placed at $\xi = -1$, $\xi = 0$ and $\xi = 1$. The shape functions can be written as follows:

$$L_1(\xi) = (\frac{\xi-\xi_2}{\xi_1-\xi_2})(\frac{\xi-\xi_3}{\xi_1-\xi_3}) = (\frac{\xi-0}{-1-0})(\frac{\xi-1}{-1-1}) = -\xi(\frac{1-\xi}{2}), \tag{4.10}$$

$$L_2(\xi) = (\frac{\xi - \xi_1}{\xi_2 - \xi_1})(\frac{\xi - \xi_3}{\xi_2 - \xi_3}) = (\frac{\xi + 1}{0 + 1})(\frac{\xi - 1}{0 - 1}) = (1 + \xi)(1 - \xi). \qquad (4.11)$$

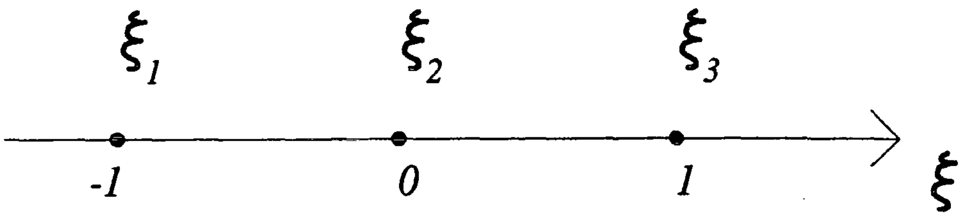


Figure 4.2 : *One dimensional quadratic Lagrange element*

On comparing the two sets of shape and mapping functions, we note that the only difference is in the terms relating to $\xi = 1$, that is the terms at infinity. This is demonstrated in Table 4.3.

Node Number, i	ξ_i	Quadratic Finite Element Parent Shape Functions P_i	Quadratic Infinite Element Mapping Functions M_i
1	-1	$-\xi \times (1-\xi)/2$	$-\xi \times 2/(1-\xi)$
2	0	$(1+\xi) \times (1-\xi)$	$(1+\xi) \times 1/(1-\xi)$

Table 4.3 : *Comparison of infinite and finite element functions*

The term corresponding to the node at infinity is inverted. This immediately shows the possibility of generating an open-ended set of infinite element mapping functions. As will be seen it is possible to generate sets of mapping functions, both Lagrange and serendipity, for all square and cube finite element parent shapes to any desired order, just as for finite elements. The necessary processes will now be explained.

4.3 Multi-dimensional mapping functions

In two and three dimensions, the most usual case is to have an element which extends to infinity in one direction and is finite in the other directions. More infrequently, one needs an element which extends to infinity in several directions. Mapped shape functions can be derived for both Lagrange and serendipity elements. The Lagrangian element is the simplest and will be dealt with first.

4.3.1 Lagrange mapping functions

Let us denote L_i^n the Lagrange polynomials and M_i^n the mapped Lagrange polynomials. L is given by:

$$\mathrm{L}_i^n(\xi) = \frac{(\xi - \xi_0)(\xi - \xi_1)....(\xi - \xi_{i-1})(\xi - \xi_{i+1})....(\xi - \xi_n)}{(\xi_i - \xi_0)(\xi_i - \xi_1)....(\xi_i - \xi_{i-1})(\xi_i - \xi_{i+1})....(\xi_i - \xi_n)}, \tag{4.12}$$

where $i = 0,...n$, ξ is the normalized co-ordinate in the range [-1,+1] and the element has $n + 1$ nodes. M is calculated as described in section 4.2 (see Table 4.3). The corresponding equation is given below:

$$\mathrm{L}_i^n(\xi) = \frac{(\xi - \xi_0)(\xi - \xi_1)....(\xi - \xi_{i-1})(\xi - \xi_{i+1})....(\xi_i - \xi_n)}{(\xi_i - \xi_0)(\xi_i - \xi_1)....(\xi_i - \xi_{i-1})(\xi_i - \xi_{i+1})....(\xi - \xi_n)}. \tag{4.13}$$

The mapping functions for a Lagrangian element are derived next. Let consider the general case of a q dimensional element. The element is supposed to have p infinite directions $dir_1, dir_2, ...dir_p$ out of q directions and n nodes along one edge. The equation for the mapping function can be expressed as follows:

$$\begin{aligned} MF_{node(l_1,...l_p,...l_q)}&(dir_1, ...dir_p, ...dir_q) \\ &= M_{l_1}^n(dir_1) * ...M_{l_p}^n(dir_p) * L_{l_{p+1}}^n(dir_{p+1}) * ...L_{l_q}^n(dir_q) \\ &= \prod_{j=1}^{p} M_{l_j}^n(dir_j) * \prod_{j=p+1}^{q} L_{l_j}^n(dir_j), \end{aligned} \tag{4.14}$$

where $l_1,...l_q$ denote the position of the node in the element (see below equation (4.15)) and $node(l_1, ...l_q)$ gives the number of the node at that position. The l_j are related to the dir_j as follow:

$$l_j = \frac{n}{2}(dir_j + 1) - 1 \leq dir_j \leq 1 \longrightarrow 0 \leq l_j \leq n. \tag{4.15}$$

In practice $p \leq q \leq 3$.

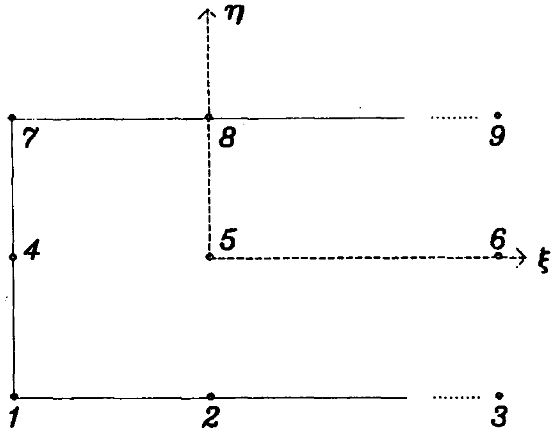


Figure 4.3 : *Three nodes Lagrangian element*

For example, the mapping functions of a two dimensional Lagrangian element with three nodes along one edge and extending to infinity in the ξ direction (Figure 4.3) are as follows:

$$MF_1 = \frac{\eta(1-\eta)\xi}{(1-\xi)}$$
$$MF_2 = \frac{-\eta(1-\eta)(1+\xi)}{2(1-\xi)}$$
$$MF_4 = \frac{-2\xi(1+\eta)(1-\eta)}{(1-\xi)}$$
$$MF_5 = \frac{(1-\eta^2)(1+\xi)}{(1-\xi)}$$
$$MF_7 = \frac{-\xi\eta(1+\eta)}{(1-\xi)}$$
$$MF_8 = \frac{\eta(1+\eta)(1+\xi)}{2(1-\xi)} .$$

The mapping functions above can be easily calculated using the algebraic language REDUCE. First the algebraic expressions for the Lagrange polynomials and the mapped Lagrange polynomials are obtained according to formulæ(4.12) and (4.13). the REDUCE code is given next:

```
FOR I:=0:N DO
<<  L(I) := 1;
    M(I) := 1;
    FOR J:=0:N DO
    <<  IF I NEQ J THEN
        << L(I) := L(I)*(VAR1-XI(J))/(XI(I)-XI(J)) >>;
           IF J=/ N THEN
               M(I) := M(I)*(VAR1-XI(J))/(XI(I)-XI(J))
           ELSE
               M(I) := M(I)*(XI(I)-XI(J))/(VAR1-XI(J))
           ENDIF
        >>
    >>
>>;
```

The mapping functions are then constructed as described in equation (4.14). They are made up of Lagrange and mapped Lagrange polynomials depending on whether the direction is finite or infinite. The REDUCE program includes a test on the finitude of each direction and selects the appropriate polynomial for that direction. The mapping functions corresponding to the nodes at infinity are set to zero. The REDUCE code is shown below for the three dimensional mapping functions:

```
COORD(1) := XI; COORD(2) := ET; COORD(3) := ZE;
FOR K:=0:N DO
<<  IND(3) := K;
    FOR J:=0:N DO
    <<  IND(2) := J;
        FOR I:=0:N DO
        <<  IND(1) := I;
            NBNODE := LC3D(I,J,K);
            MF(NBNODE) := 1;
            FOR DIR:=1:3 DO
            <<  IF INFI(DIR) = 0 THEN
                    MF(NBNODE) := MF(NBNODE)*SUB(VAR1=COORD(DIR),L(IND(DIR))
                  ELSE
                  << IF IND(DIR) =/ N THEN
                          MF(NBNODE) := MF(NBNODE)*SUB(VAR1=COORD(DIR),M(IND(DIR))
                      ELSE
                          MF(NBNODE) := 0 >>;
              >>
         >>
     >>
>>;
```

The array COORD stores the cartesian co-ordinate system ($dir_1, dir_2, \ldots dir_p$ in equation (4.14)) while the array IND stores the position of the node in the element ($l_1, l_2, \ldots l_p$ in equation (4.15)).

Using REDUCE, the algebraic expressions obtained are differentiated with respect to the local variables ξ, η and ζ to obtain the mapping function derivatives, needed for the calculation of the Jacobian matrix. These expressions are then translated into FORTRAN using GENTRAN. This method enables us to automatically produce compilable FORTRAN code with a high confidence in its correctness.

4.3.2 Serendipity mapping functions

As there is a rational procedure for deriving the shape functions for the serendipity element, which is clearly described by Zienkiewicz[12], a precisely analogous procedure can be followed for the infinite mapped elements. A formula is now given for the two dimensional quadratic and cubic elements with one infinite direction.

Let us denote ξ and η the two directions, ξ being the infinite direction and η the finite one. As for the Lagrange mapping functions L will represent the Lagrange polynomials and M the mapped Lagrange polynomials. The mapping functions take three different values depending on the position of the node in the quadrilateral (see Figure 4.4):

$$MF_{node(i,j)}(\xi,\eta) = \begin{cases} EH_{node(i,j)}(\xi,\eta) & \text{for mid-side nodes on edges 1 and 3} \\ EV_{node(i,j)}(\xi,\eta) & \text{for mid-side nodes on edges 2 and 4} \\ C_{node(i,j)}(\xi,\eta) & \text{for corner nodes,} \end{cases} \tag{4.16}$$

where

$$EH_{node(i,j)}(\xi,\eta) = \mathrm{M}_i^n(\xi)\mathrm{L}_j^1(\eta) \qquad i = 1,...n-1 \quad and \quad j = 0,1$$

$$EV_{node(i,j)}(\xi,\eta) = \mathrm{M}_i^1(\xi)\mathrm{L}_j^n(\eta) \qquad i = 0,1 \quad and \quad j = 0,...n-1$$

$$C_{node(i,j)}(\xi,\eta) = \mathrm{M}_i^1(\xi)\mathrm{L}_j^1(\eta) - T_{node(i,j)}(\xi,\eta) - U_{node(i,j)}(\xi,\eta)$$
$$i = 0,1 \quad and \quad j = 0,1,$$

and

$$\begin{aligned} T_{node(i,j)}(\xi,\eta) &= \sum_{\mu=1}^{n-1} \frac{2}{(1+\xi_c\xi_\mu)} EH_{node(\mu,j)}(\xi,\eta) \\ U_{node(i,j)}(\xi,\eta) &= \sum_{\mu=1}^{n-1} \frac{(1+\eta_c\eta_\mu)}{2} EV_{node(i,\mu)}(\xi,\eta). \end{aligned} \tag{4.17}$$

(ξ_c,η_c) are the local co-ordinates of the corner nodes, n+1 is the number of nodes along one edge of the element, and edges are as shown in Figure 4.4.

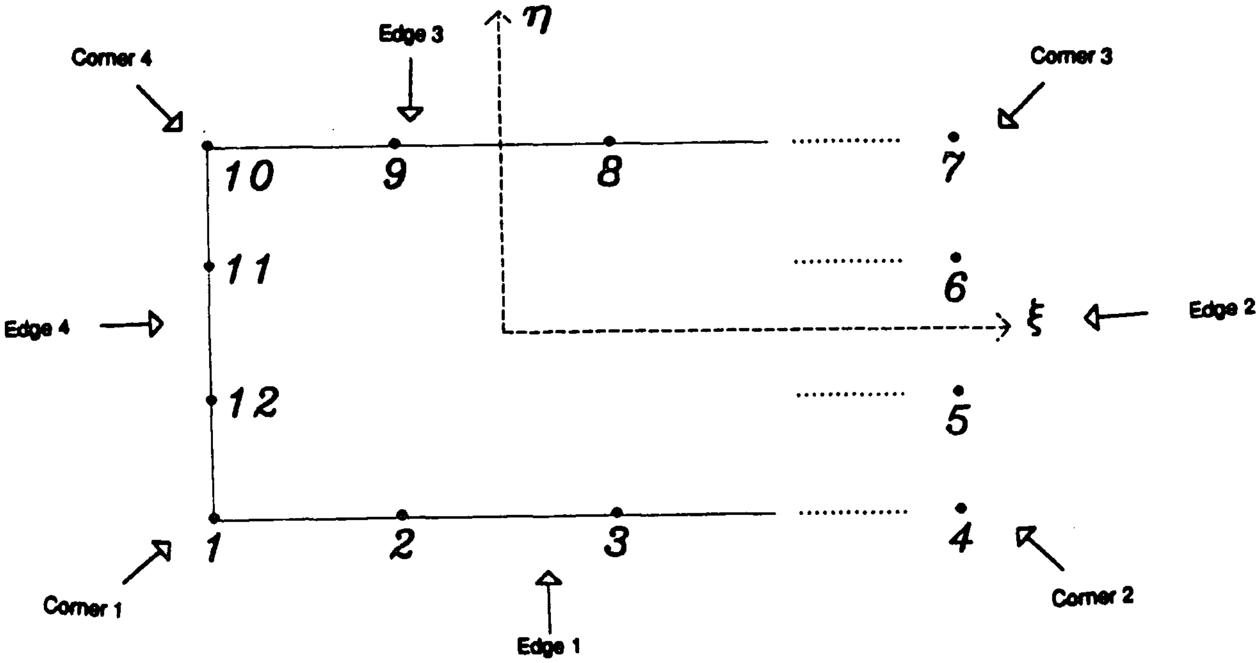


Figure 4.4 : *2D Cubic serendipity element - definition of the edges*

EH corresponds to the nodes on the horizontal edges, except the corner nodes. It is composed of a $(n+1)^{th}$ order mapped Lagrange polynomial in ξ direction and a linear Lagrange polynomial in η direction as there are n+1 nodes in the ξ direction and only two nodes in the η direction. Similarly, EV corresponds to the nodes on the vertical edges, except the corner nodes.

C is the shape function for all four corner nodes. It is formed from a mapped linear function in ξ and a linear function in η, from which two polynomials are subtracted. The two polynomials are weighted sums of EH and EV along the ξ and η directions. They ensure that C is equal to 1 at the corner and zero at all other finite points of the element. The first part of C gives 1 at the corner, zero at the other finite corners and some non zero values at the finite mid-side nodes along the edges. T and U modify C so that its value at the finite mid-side nodes is zero.

When the η direction is infinite and the ξ direction is finite, the formula for the mapping functions can be obtained from equation (4.17) by using M in place of L in the η direction, L in place of M in the ξ direction and inverting the scaling factors in the sums for T and U, that is to say $(1+\xi_c\xi_\mu)/2$ for T and $2/\ (1+\eta_c\eta_\mu)$ for U. When both directions are infinite we need to use M everywhere in the formula (4.17) and inverted scaling factors in T and U which are $2/(1+\xi_c\xi_u)$ and $2/(1+\eta_c\eta_u)$.

The REDUCE code is similar to that described in the previous chapter for the shape functions with the addition of a test for choosing between the Lagrange polynomials and the mapped Lagrange polynomials. The complete REDUCE program to calculate the two dimensional serendipity mapping functions is given in appendix A.

The procedure for three dimensions is similar. Mapped Lagrange polynomials are used in the infinite directions and ordinary Lagrange polynomials in the finite directions. Extension to quartic and higher order Serendipity polynomials requires the introduction of mid-face nodes. Although simple in principle it has not been done here, as such elements are not widely used.

The automatic generation of FORTRAN code from the REDUCE analytical expressions is carried out in a very similar way to the one used for the shape functions in the previous chapter. The optimisation process is also the same. The complete REDUCE program to generate the FORTRAN code is given in appendix B while two examples of FORTRAN mapping functions routines are given in appendix C.

It is now useful to explain what name convention has been adopted for the FORTRAN mapping functions routines. This is shown next:

Midtn where M stands for Mapping Function
i indicates which direction is infinite
i = 1 , x infinite (y and z finite)
i = 2 , y infinite (x and z finite)
i = 3 , z infinite (x and y finite)
i = 4 , x and y infinite (z finite)
i = 5 , y and z infinite (x finite)
i = 6 , x and z infinite (y finite)
i = 7 , x, y and z infinite
d is the dimension (1,2 or 3)
t is the type of the element (L (agrangian),
S (erendipity).
n is the number of nodes along one edge of the element

For similar reasons to the one explained in the previous chapter it has been necessary to carry out tests on the FORTRAN mapping functions routines to ensure that they were correct.

At first, the REDUCE analytical expressions for the mapping functions for lower order elements have been checked against other sources. The tests on the FORTRAN routines are more complicated than in the case of the shape functions.

These tests involve taking linear combination of the mapping functions for the nodes at finite distance and checking that the result is as expected. The mapping functions for the nodes at finite distance are all multiplied by the concerresponding value of r, from Table 4.1. The constant a is taken to be unity. Thus for the 8-node serendipity element, M_1, M_7 and M_8 are multiplied by one and M_2 and M_6 by two. If the mapping functions are then summed, we should recover the linear mapping functions, which is in this case $2/(1\text{-}\xi)$. A similar operation on the ξ derivatives should yield $2/(1\text{-}\xi)^2$ and on the η derivatives zero when the element extends to infinity only in the η direction. Similar tests are valid for mapping functions extending to infinity in more than one direction.

4.4 Conclusions

A simple method for calculating mapping functions for the infinite elements, using the Zienkiewicz method, has been described. The automatic generation of numerical code for the mapping functions using an algebraic language has been presented. The generated FORTRAN code is comprehensive, reliable and optimized. The mapping function derivatives are also provided.

The next chapter ends the series of investigation of the application of Computer Algebra to finite element method through explaning the generation of particular element matrices.

References

1. Ungless R.L., *An infinite finite element*, M ASc Thesis, University of British Columbia, 1973.
2. Anderson D.L. and Ungless R.L., 'Infinite finite elements', *Int. Symp. Innovative Num. Anal. Appl. Eng. Sci.*, France, 1977.
3. Lynn P.P. and Hadid H.A., 'Infinite elements with $1/r^n$ type decay', *Int. J. Num. Meth. Eng.*, **17(3)**, pp 347-355, 1981.
4. Zienkiewicz O.C., Emson C. and Bettess P., 'A novel boundary infinite element', *Int.J. Num. Meth. Eng.*, **19**, pp 393–404, 1983.
5. Beer G. and Meek J.L., 'Infinite domain elements', *Int. J. Num. Meth. Eng.*, **17(1)**, pp 43–52, 1981.
6. Zienkiewicz O.C., Bettess P., Chiam T.C. and Emson C., 'Numerical methods for unbounded field problems and a new infinite element formulation', *ASME. AMD*, **46**, pp 115–148, New York, 1981.
7. Pissanetzky S., 'An infinite element and a formula for numerical quadrature over an infinite interval', *Int. J. Num. Meth. Eng.*, **19**, pp 913–928, 1983.
8. Pissanetzky S., 'A Simple Infinite Element', *COMPEL*, Boole Press, to be published.

9. Marques J.M.M.C. and Owen D.R.J., 'Infinite elements in quasi-static materially non-linear problems', *Computers and Structures*, to be published.

10. Kumar P. discussion of Ref. 2–27, *Int. J. Num. Meth. Eng.*, **20**, pp 1173–1174, 1984.

11. Barbier C., Bettess P. and Bettess J.A, 'Automatic Generation of Mapping Functions for Infinite Elements using REDUCE', submitted for publication in the *Journal of Symbolic Computation*

12. Zienkiewicz O.C, *The Finite Element Method*, 3rd edn, McGraw-Hill, 1977.

Chapter V

Automatic generation of bending element matrices

5.1 Introduction

In finite element analysis the approximation solution is defined by the nodal values and the shape functions. The shape functions are usually based on polynomials. A common way of obtaining these functions is by using Lagrange polynomials as seen in chapter 3. The functions derived are then C_0 continuous which is sufficient in many cases. Sometimes, though, it is necessary to use shape functions with higher order of continuity particularly in plate and beam bending problems and in streamfunction models of viscous flow problems. Other types of polynomial are then used.

In this chapter we are concerned with the generation of shape functions based on Hermite polynomials using a Computer Algebra package. The Hermite polynomials used are the Hermite interpolation polynomials, as distinct from the Hermite orthogonal polynomials which are quite different. These shape functions are then used to generate element mass, geometric stiffness and stiffness matrices. These matrices are well known and can be found in textbooks related to structural analysis[1,2,3,4]. Since the original paper on the topic[5] showed computer generated coefficients, the idea is not new. Rather the idea is to show, in an educational sense, how simply standard element matrices can be generated using Computer Algebra.

As an example, this chapter demonstrates how much easier and simpler it is to derive the mass, geometric stiffness and stiffness matrices using the Computer Algebra system REDUCE. The REDUCE program incorporates the calculation of the Hermite shape functions, the formation of the integrand for the element matrices and the analytical integration of these integrands, which is probably the most difficult and tedious task to carry out by hand, despite the fact that these integrands are polynomials. FORTRAN code is also automatically produced from the symbolic expressions. The method is flexible as it allows us to generate matrices for any size of element in one and two dimensions, with an obvious extension to three dimensions. A paper has been published[6] on this work.

In the following sections the equations will be established and the use of Computer Algebra to carry out the calculations will be explained. The full REDUCE code and FORTRAN routines are provided in appendices D, E and F. This and similar exercises could profitably be used in finite element courses.

5.2 Formation of the Hermite shape functions

The theory for the one dimensional elements will be briefly repeated here for the convenience of the reader. The interpolation polynomials used here are such that at each point i of co-ordinates x_i, the values of the interpolated function $f(x)$ are continuous as well as its derivative $f'(x)$. This can be expressed as follows (see Figure 5.1):

$$f(x) = \sum_{i=1}^{n}(H_i(x)f_i + h_i(x)f_i'), \tag{5.1}$$

where

$$\begin{aligned}
&f_i = f(x = x_i) \\
&f_i' = \frac{df(x)}{dx}(x = x_i) \\
&H_i(x) = 1 \quad \text{and} \quad H_i'(x) = 0 \quad \text{when} \quad x = x_i \\
&H_i(x) = 0 \quad \text{and} \quad H_i'(x) = 0 \quad \text{when} \quad x = x_j \quad j \neq i \\
&h_i(x) = 0 \quad \text{and} \quad h_i'(x) = 1 \quad \text{when} \quad x = x_i \\
&h_i(x) = 0 \quad \text{and} \quad h_i'(x) = 0 \quad \text{when} \quad x = x_j \quad j \neq i,
\end{aligned} \tag{5.2}$$

and i=1,2...n.

H_i and h_i are the Hermite interpolation polynomials where H ensures that the values of the interpolated function f are continuous and h ensures that the

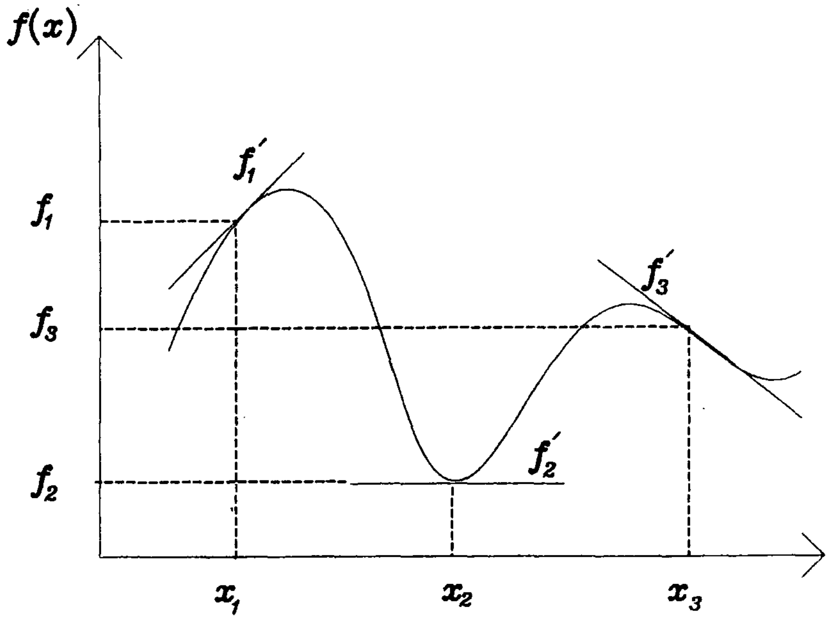


Figure 5.1 : *Interpolation using Hermite polynomials*

derivatives of f are continuous. The expressions for H and h are recalled below:

$$\begin{aligned}
H_i(x) &= (a_i x + b_i) \prod_{\substack{j=1 \\ j\neq i}}^{n} \left(\frac{x - x_j}{x_i - x_j}\right)^2 \\
a_i &= -2 \sum_{\substack{j=1 \\ j\neq i}}^{n} \frac{1}{(x_i - x_j)} \\
b_i &= 1 - a_i x_i \\
h_i(x) &= (x - x_i) \prod_{\substack{j=1 \\ j\neq i}}^{n} \left(\frac{x - x_j}{x_i - x_j}\right)^2 .
\end{aligned} \tag{5.3}$$

It is easy to check that the above polynomials exhibit the properties given in equation (5.2).

The language REDUCE can easily be used to generate the analytical expressions corresponding to equation (5.3). This can be seen in the REDUCE program in appendix D at lines 91–100, where XX(I) represents the node's co-ordinate x_i. X is a variable standing for itself, which means that it does not hold a numerical value. a_i, b_i are represented in the program by AA and BB. Both H and h are stored in the same vector HERM where the arrangement is that $H_1, h_1, H_2, h_2, \ldots H_n, h_n$ are stored sequentially in HERM. An example of the values of H and h for a one dimensional linear element calculated by the REDUCE program is next given below:

```
               3       2       3                      2                 2
              L  - 3*L*X  + 2*X                 X*(L  - 2*L*X + X )
HERM (1) = --------------------          HERM (2) = --------------------
                        3                                    2
                       L                                    L
              2                                      2
              X *(3*L - 2*X)                         X *(L - X)
HERM (3) = ---------------               HERM (4) =  - ------------
                      3                                     2
                     L                                     L
```

The two dimensional shape functions are obtained by multiplying together two one dimensional Hermite polynomials, each related to one of the two variables x or y. The continuity of both the values of a two dimensional function $g(x, y)$ and its derivatives with respect to x, y and xy has to be achieved. Let us call the two dimensional shape functions SF. To ensure continuity as expressed above the following equations hold:

$$\begin{aligned} SF_m(x,y) &= H_i(x)H_j(y) \\ \frac{\partial SF_m}{\partial x}(x,y) &= h_i(x)H_j(y) \\ \frac{\partial SF_m}{\partial y}(x,y) &= H_i(x)h_j(y) \\ \frac{\partial^2 SF_m}{\partial x \partial y}(x,y) &= h_i(x)h_j(y), \end{aligned} \tag{5.4}$$

where (i,j) refers to the node at position i,j in the element as shown in Figure 5.2. m is the number of the node whose position is i,j. The definition of SF above enables us to write the interpolated expression for a function $g(x, y)$ as follows:

$$g(x,y) = \sum_{m=1}^{n*n} (SF_m g_m + \frac{\partial SF_m}{\partial x}\frac{\partial g_m}{\partial x} + \frac{\partial SF_m}{\partial y}\frac{\partial g_m}{\partial y} + \frac{\partial^2 SF_m}{\partial x \partial y}\frac{\partial^2 g_m}{\partial x \partial y}), \tag{5.5}$$

where n is the number of nodes in each dimension, g_m is the value of the function g at point m of position i,j in the element, $\partial g_m/\partial x$, $\partial g_m/\partial y$, $\partial^2 g_m/\partial x \partial y$ are the values of the derivatives of g at point m.

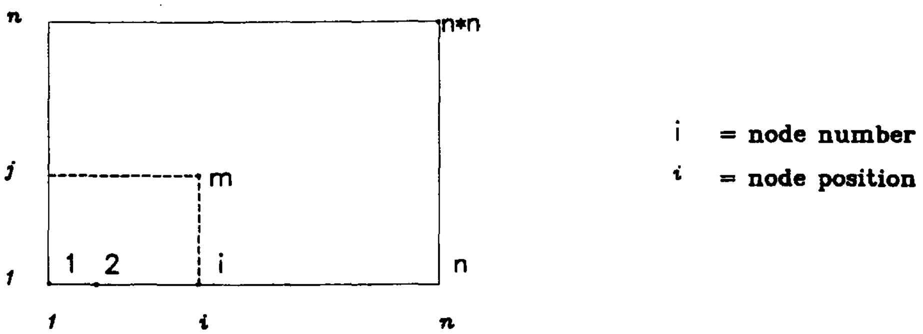


Figure 5.2 : *Position and numbering of the two dimensional element nodes*

The REDUCE code for two dimensional elements can be found in appendix D at lines 107–118. The REDUCE command SUB locally substitutes the variable X by Y in HERMXY, which stores the two dimensional shape functions SF, but does not affect their value in memory, thus allowing further substitutions to be carried out.

Although three dimensional C_1 shape functions are rarely used they can be found as follows. All three variables x, y and z and their corresponding derivatives ($\partial/\partial x$, $\partial/\partial y$, $\partial/\partial z$, $\partial^2/\partial x\partial y$, $\partial^2/\partial y\partial z$, $\partial^2/\partial x\partial z$, $\partial^3/\partial x\partial y\partial z$) are involved. For example, SF and the derivative $\partial^2/\partial x\partial y$ are defined as follows:

$$\begin{aligned} SF_m(x,y,z) &= H_i(x)H_j(y)H_k(z) \\ \frac{\partial^2 SF_m}{\partial x\partial y}(x,y,z) &= h_i(x)h_j(y)H_k(z). \end{aligned} \tag{5.6}$$

The function H is used when the variable x, y or z does not appear in the derivative and h is used otherwise. The REDUCE code, although not given here, would be very similar to that for the two dimensional case, where an extra term SUB(X=Z,SUB(L=C,HERM(1,K4))) would be multiplied to the expression for HERMXY.

5.3 Formation of the bending element matrices

Using the shape functions established previously, element mass, geometric stiffness and stiffness matrices can be calculated. These three matrices can be derived for one, two or three dimensional cases. In this section we will only consider the one and two dimensional equations as only these two have been implemented. The three dimensional case is easy to obtain too but there are few practical applications for this case.

First the matrix equations in the one dimensional case are established. The element considered is shown in Figure 5.3.

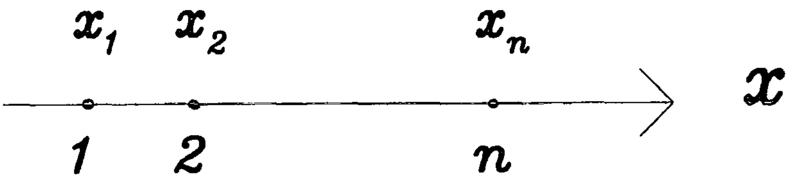


Figure 5.3 : *One dimensional element*

The mass (**M**), geometric stiffness (**G**) and stiffness (**K**) equations are defined as follows[4]:

$$\begin{aligned} \mathbf{M} &= \int_0^a \rho (f(x))^2 dx \\ \mathbf{G} &= \int_0^a \sigma \left(\frac{df(x)}{dx} \right)^2 dx \\ \mathbf{K} &= \int_0^a EI \left(\frac{d^2 f(x)}{dx^2} \right)^2 dx, \end{aligned} \tag{5.7}$$

where ρ, σ and EI are constants related to physical properties of the element. Replacing $f(x)$ by its discretised interpolated expression from equation (5.1) we obtain the matrix formulation. Let us introduce a vector **N** storing all shape functions as follows:

$$\begin{aligned} \mathbf{N}(2i-1) &= H_i \\ \mathbf{N}(2i) &= h_i \qquad 1 \leq i \leq n, \end{aligned} \tag{5.8}$$

where n is the total number of nodes of the one dimensional element. This is shown Figure 5.4. The element matrices can then be expressed as a function of the vector **N** as follows:

$$\begin{aligned} \mathbf{ELM} &= \int_0^a \rho \mathbf{N}^t \mathbf{N} dx \\ \mathbf{ELG} &= \int_0^a \sigma \left(\frac{d\mathbf{N}}{dx} \right)^t \left(\frac{d\mathbf{N}}{dx} \right) dx \\ \mathbf{ELK} &= \int_0^a EI \left(\frac{d^2\mathbf{N}}{dx^2} \right)^t \left(\frac{d^2\mathbf{N}}{dx^2} \right) dx. \end{aligned} \tag{5.9}$$

H_1	h_1	H_2	h_2		H_n	h_n

Figure 5.4 : *Vector* **N**

The REDUCE code which achieves this is given at lines 101–105 and lines 130,141 and 151 of the appendix D. The vector **N** is represented in the REDUCE program by `HERM`. The first and second derivatives of **N** with respect to x are stored in the program as `DHERM` and `D2HERM` (lines 101–105 of appendix D). They are calculated using the REDUCE operator `DF(f,var)` which calculates the partial derivative of the function `f` with respect to the variable `var`.

The integrands of the integrals shown in equation (5.9) are derived as depicted in equation (5.9), which is shown in the appendix D at lines 130, 141 and 151. The REDUCE operator `TP` takes a symbolic matrix as argument and returns it transposed. The multiplication sign refers to matrix multiplication, although the same sign is used for scalar multiplication. The analytical integration is carried out using the operator `INT2` which is defined in the appendix D at lines 79–80. It is striking that the extensive expansion of polynomials and their integration, which is so tedious for the human, are carried out with very few instructions, and with no mistakes. The integrands of the element matrices obtained at lines 130, 141 and 151 are then integrated using this user defined operator (lines 131-132, 142–143 and 152–153 of appendix D).

In two dimensions, the mass, geometric stiffness and stiffness equations become:

$$\begin{aligned}
\mathbf{M} &= \int_0^a \int_0^b \rho(g(x,y))^2 dxdy \\
\mathbf{G} &= \int_0^a \int_0^b (\nabla(G_{x,y}g(x,y)))^2 dxdy \qquad (5.10) \\
\mathbf{K} &= \int_0^a \int_0^b (\nabla^2(D_{x,y}g(x,y)))^2 dxdy,
\end{aligned}$$

where ∇ is the gradient functional ($\nabla g = (\partial g/\partial x,\ \partial g/\partial y)$), ∇^2 is the Laplace operator, $G_{x,y}$ and $D_{x,y}$ are matrices of constants related to the directions x and y. A matrix form can then be derived from equation (5.10) by replacing $g(x,y)$ by its discretised interpolation form. In two dimensions the vector $\mathbf{N}$ of shape functions becomes:

$$\begin{aligned} \mathbf{N}(4k-3) &= H_iH_j \\ \mathbf{N}(4k-2) &= h_iH_j \\ \mathbf{N}(4k-1) &= H_ih_j \\ \mathbf{N}(4k) &= h_ih_j, \end{aligned} \tag{5.11}$$

where (i,j) are the positions of the nodes in the element, $1 \leq k \leq n$ and n is the number of nodes in each dimension. The expressions for the element matrices are therefore as follows:

$$\begin{aligned} \mathbf{ELM} &= \int_0^a \int_0^b \rho \mathbf{N}^t \mathbf{N} dx dy \\ \mathbf{ELG} &= \int_0^a \int_0^b \left\{ \begin{array}{c} \frac{\partial \mathbf{N}}{\partial x} \\ \frac{\partial \mathbf{N}}{\partial y} \end{array} \right\}^t [G] \left\{ \begin{array}{c} \frac{\partial \mathbf{N}}{\partial x} \\ \frac{\partial \mathbf{N}}{\partial y} \end{array} \right\} dx dy \\ \mathbf{ELK} &= \int_0^a \int_0^b \left\{ \begin{array}{c} \frac{\partial^2 \mathbf{N}}{\partial x^2} \\ \frac{\partial^2 \mathbf{N}}{\partial y^2} \\ \frac{\partial^2 \mathbf{N}}{\partial x \partial y} \end{array} \right\}^t [D] \left\{ \begin{array}{c} \frac{\partial^2 \mathbf{N}}{\partial x^2} \\ \frac{\partial^2 \mathbf{N}}{\partial y^2} \\ \frac{\partial^2 \mathbf{N}}{\partial x \partial y} \end{array} \right\} dx dy, \end{aligned} \tag{5.12}$$

where

$$\mathbf{G} = \begin{bmatrix} \sigma_x & \tau_{xy} \\ \tau_{xy} & \sigma_y \end{bmatrix} \qquad \mathbf{D} = \begin{bmatrix} D_x & D_1 & 0 \\ D_1 & D_y & 0 \\ 0 & 0 & D_{xy} \end{bmatrix},$$

and σ_x, σ_y, τ_{xy}, D_x, D_y, D_{xy}, D_1 are constants. They respectively represent the direct stress resultant in x direction, the direct stress resultant in y direction, the shear stress resultant in x and y directions, the flexural rigidity in x direction, the flexural rigidity in y direction, the Poisson ratio effect and the torsional rigidity.

The REDUCE code for the two dimensional elements is very similar to that for the one dimensional elements where both variables x and y appear and the differentiation has to be carried out with respect to these two variables. This is shown in appendix D at the following lines: 119–122 for the calculation of derivatives of N, represented by HERMXY in the program, 135–137, 145–147 and 155–157 for the calculation of the element matrices and their integration.

5.4 The REDUCE program

The REDUCE program is organised in two parts: procedures which generate the FORTRAN code and a main program which carries out the calculations to obtain the analytical expressions for the element matrices, as shown in previous sections.

The FORTRAN code is obtained using a REDUCE feature (switch ON FORT) which enables the translation of REDUCE expressions into FORTRAN code. This is done automatically so that provided the REDUCE code is error free the FORTRAN code will also be error free. This is a very easy and comprehensive way for obtaining code for the element matrices.

A listing of the REDUCE procedures which translate the REDUCE analytical expressions for the element matrices into FORTRAN is given in appendix D at lines 4–66. The procedure WRITEEL1 (lines 4–24) generates FORTRAN code for the one dimensional matrices and WRITEEL2 (lines 27–50) produces FORTRAN code for the two dimensional matrices.

The FORTRAN code obtained for two dimensional elements appeared to be very large which lead us to consider optimising the code. The first obvious optimisation arises from the fact that the element matrices are symmetric therefore only half of each matrix needs to be generated. A small FORTRAN routine can be used to fill in the other half of the matrix so it is complete. This optimisation process has been included in the REDUCE code so that only half of the analytical expressions for the matrices are actually translated into FORTRAN.

The second optimisation consists of avoiding multiple calculation of the same intermediate expressions. For example in the calculation for **ELM**, expressions like ab, ab^2, a^2b^2, a^2b^3 ... etc arise several times in the FORTRAN code (up to 32 times for a^2b^2). Similarly, expressions like $a^2\sigma_y$ and a^2D_y appear up to 136 times in the code for **ELG** and **ELK**. It is more efficient to calculate these expressions once at the beginning of the program, assign them to intermediate variables and use these intermediate variables subsequently in the program.

Using the version 3.3 of REDUCE from 1987, there is no easy way to detect such repetition of intermediate expressions in the analytical expressions in the REDUCE code itself. Therefore the optimisation has been carried out by hand in the FORTRAN code using a text editor. The FORTRAN routines generated for the one and two dimensional stiffness matrices is given in appendix F. Nevertheless, the latest version of REDUCE from July 1991, which has just been announced, claims to contain an optimisor of FORTRAN code which would avoid the post-editing mentioned above.

The program as it stands in appendix D is theoretically capable of generating the bending element matrices for any order of element. In practice, though, as the element matrices become larger, the system tends to run out of memory. This has been experienced for two dimensional quadratic elements. The program was run on the mainframe computer of the University of Durham, an Amdhal 5860, where REDUCE can use up to 1Mbyte of memory.

Therefore, it has been necessary to modify the program so that the bending element matrices are not actually stored in the REDUCE program, but each element of these matrices is evaluated in turn. The principle of the new program is similar to that of the old program except that the matrix multiplication which was carried out automatically to obtain the integrand of the element matrices is replaced by the explicit coding of this multiplication using loops. The full modified program is shown in appendix E. This modification implies that a lot less memory is required and the code could run on a smaller machine. Although it might take up to several minutes to compute the FORTRAN routines for quadratic and higher order two dimensional elements, this is only done once, thus it is not of prime concern.

5.5 Tests and conclusions

When FORTRAN code is automatically produced using a Computer Algebra system it is necessary to check that the code obtained is correct. The checks can be carried out by hand on the analytical expressions obtained. It can be tedious if no external source of results is available and some of the calculations have to be carried out by hand, which defeats in a way the purpose of using a Computer Algebra system.

Another method is to automatically test the FORTRAN code by checking that it complies with some known physical or mathematical properties. Usually a combination of both methods leads to a very secure FORTRAN code.

The mass, geometric stiffness and stiffness bending element matrices for one and two dimensional linear elements have been checked by hand against published results[7,8,9].

The matrix **ELK** has also been checked using the property that a general rigid body motion should give zero nodal forces and moments. The corresponding equations are as follows:

$$\begin{array}{ll} \text{1 dimension} & \mathbf{x} = (\alpha, \beta, \alpha + \beta * a, \beta) \\ \text{2 dimensions} & \mathbf{x} = (\alpha, \beta, \gamma, 0, \\ & \quad \alpha + \beta * a, \beta, \gamma, 0, \\ & \quad \alpha + \gamma * b, \beta, \gamma, 0, \\ & \quad \alpha + \beta * a + \gamma * b, \beta, \gamma, 0), \end{array} \tag{5.13}$$

where a and b are the dimensions of the element in the x and y directions. Using the vector $\mathbf{x}$ defined in equation (5.13) should then lead to the following result:

$$\mathbf{ELK} * \mathbf{x} = 0. \tag{5.14}$$

This has successfully been checked.

Computer Algebra has been used to automatically generate known expressions for element matrices used in mass, geometric stiffness and stiffness problems. The FORTRAN code obtained is reliable and has been optimised. The advantage of the Computer Algebra approach is that it is an easy and comprehensive way for obtaining code for these element matrices and if necessary new matrices for higher order elements can also be derived using the REDUCE program developed.

In using Computer Algebra, the work of forming element matrices is greatly simplified and the possibility of errors is reduced. It allows students to generate their own matrices without a lot of tedious algebra. This makes the teaching process more interesting and enjoyable for the student. He or she can concentrate on the underlying theory and not the tedious and error prone algebraic manipulations. New formulations and speculative elements can be explored quickly and easily. The conciseness of the REDUCE code, particularly after the formatting instructions are removed, in comparison with the resulting FORTRAN is very striking.

This chapter concludes the first part of the thesis concerning the use of Computer Algebra in finite element analysis. Its main aim was to demonstrate the feasability of such methods and their potential use in this branch of engineering, opening prospects for other fields using similar algebra. A practical application of this methodology will be presented in part 3 where Computer Algebra is used to generate element matrices for non-linear studies using finite element analysis.

References

1. McMinn S.J., *Matrices for structural analysis*, 1962.
2. Livesley R.K., *Matrix Methods of Structural Analysis*, Pergamon press, 1964.
3. Godden W.G., *Numerical Analysis of Beam and Column Structures*, Prentice-Hall Inc.,1965.
4. Przemieniecki J.S., *Theory of matrix structural analysis*, McGraw-Hill Book Company, 1968.
5. Bogner F.K, Fox R.L and Schmit L.A., 'The generation of interelement–compatible stiffness and mass matrices by the use of interpolation formulae', *Proceedings of the Conference on Matrix Methods in Structural Mechanics*, Air Force Institute of Technology, Wright Patterson A.F Base, Ohio, USA, October 1965.
6. Barbier C., 'Automatic generation of bending element matrices for finite element method using REDUCE', to appear in *Engineering Computation.*
7. Ref 4., pp 81, 297 and 391.
8. Ref 5., Addendum p 441 and Table 6 pp 430–431.
9. Clark. P.J., 'The Vibration of a Stiffened Panel: Analysis by means of an Orthotropic Plate Conforming Finite Element', *MSc Marine Technology*, The University of Newcastle-upon-Tyne, September 1982.

Part II

Parallel Solvers

Chapter VI

Introduction to parallel processing

'Parallel processing' is a term found increasingly in journals and conferences. It appears to be the key in the search for more processing power. Although this concept is usually thought of as new, parallel processing has been around for some time, not always, however, successfully surpassing the performance of conventional computers of their time.

Parallel processing is an attractive option in number crunching applications, such as the finite element method, to gain more power at little cost . Several aspects of the finite element method are suitable for efficient implementation on parallel machines including the formation of the element matrices and the solving of linear/nonlinear systems of equations. In this part, the implementation of parallel solvers for systems of linear equations is analysed. The next part is concerned with the parallel formation of element matrices.

Since parallel processing has only recently become widely available there are no standards available either in machines and software or in vocabulary and terms used to describe such systems. Therefore, it seems appropriate to define more precisely what is meant by parallel processing and what systems are available, including the ones used in this work.

6.1 Brief history

The concept of parallel processing has been around since the early days of computing. Menabrea[1] (1842) wrote in his 'Sketch of the analytical engine invented by Charles Babbage' that 'when a long series of identical computations is to be performed ... , the machine can be brought into play so as to give several results at the same time ... '.

When the first specification for computers was designed by John von Neumann in 1947, the model was serial which meant that only one thing was happening at a time, therefore making the specification easy to understand and implement. Nevertheless, the idea of parallelism was already there but unfeasable to implement with the technology of the time. Several factors contributed to bring the idea of parallel processing to life in the late seventies[2], when these machines were primilarily built for study as things in themselves. The main factor was the advent of the VLSI technology which meant that the cost of computers was not any more an exponential function of the cost of the individual elements like at the time when the vacuum tubes were used. The second factor was the development of programming techniques such as time sharing, semaphores which, being well understood, could be used directly in a parallel computer prototype.

One of the first practical implementations of a parallel machine was 'the first supercomputer'[3], the ILLIAC IV composed of 64 processors and designed in 1967. It was commercially produced and used for several years by NASA[4] in the 70's. A number of research projects on parallel computers resulted in machines which did not turn into commercial products. The examples of the Cm* (Computer Module, 1978) and WRM (Wire Routine Machine, 1983) are discussed in Almasi and Gottlieb[4]. The first machine consisted of 50 16-bit processors and the latter machine had 64 8-bit processors.

These early multiprocessor systems still suffered from underpowered hardware with heavy overheads for communications and memory access. At the same time serial machines were developing fast as the hardware improved, and the new technology parallel machines had not only to keep up with the serial machines but also had to surpass them.

In the eighties, parallel computers intended for use rather than for study started to appear. Nevertheless, they were generally regarded as an academic curiosity whose natural environment was the research laboratory[2]. Among these early commercially produced machines are the ICL–DAP and the Cosmic-cube.

There are several reasons for moving to parallel processing on a commercial basis nowadays. Parallel computers tend to be more cost effective than the serial machines, as ten small VLSI chips cost less than one big one. The other and probably main reason is the physical limitation that information cannot travel faster than the speed of the light. One way of overcoming this limitation is by reducing the distance that the information has to travel, which is achieved by new technologies, like the use of GAs, but is limited by quantum mechanics. The other way is to move more information at once, which is parallelism.

Today, parallelism is being used to produce increasingly powerful and cost effective machines. The questions are then how many and how big should the processors be and how should they be organised. This is discussed in the following sections.

Although the term 'serial machines' is used nowadays to designate machines using the von Neumann model of computers, the hardware architecture of these machines is very often parallel with operations within the machine being executed concurrently, usually hidden from the user.

This is the case of the CRAY-1[5], one of the first widely available 'Supercomputers', which was based on a pipeline architecture where only one processor was used but the ALU and CPU within this processor were replicated, connected in a pipeline form and ran concurrently. Another type of parallelism found in single processor machines is the multifunction architecture where some function units are replicated (like the co-processor for floating point operations) and run in parallel under an expanded control unit. An example is the 8086 chip from Intel with its floating point co-processor 8087 chip found in Personal Computers. Parallelism is also present when machines are connected through communication networks (distributed processing).

6.2 Definitions

Before starting a more detailed review of parallel processing it is useful to define what is meant by parallel processing in the light of the discussion in the previous section. A general definition is given in Almasi and Gottlieb[6]: it is 'a large collection of processing elements that can contribute and cooperate to solve large problems fast'.

As this definition includes the kind of parallelism found in single processor architecture (pipelined, multifunction) as well as the parallelism found in distributed systems (networks), for this work it is necessary to narrow down the scope of the definition by adding that only processing elements composed of a whole CPU and included in a single physical machine will be considered. This can be denoted as a multiprocessor machine.

Another term which appears along with this kind of machine is process or task. This is a part of a job which can be carried out on a multiprocessor machine with several processes working concurrently to contribute to the end result. In other words, a process is to software what a processor is to hardware.

The problem to be solved is split up into tasks or processes. Processes are then assigned to processors for execution. Where more than one process is assigned to one processor, each process may run either serially (in a von Neumann fashion) or concurrently. In the latter case, techniques like time sharing may be used to emulate parallelism.

6.3 Classification of multiprocessor machines

The size, number and inter-relationship of processors and processes (hardware and software) are used to classify the type of machine available and the kind of parallelism which can be achieved for a particular problem. A technical vocabulary has been developed to define the quantities used in classification. First let us consider the machines.

The coupling of a multiprocessor machine defines how much hardware is shared between the different processors present in the machine. Two types of machines currently exist: loose coupling and tight coupling machines which respectively correspond to what is called distributed memory (not to be confused with distributed processing) and shared memory multiprocessor machines. The shared resource is the memory and the amount of coupling is measured by how much memory two processors in a machine can both freely access.

Another parameter which is used in the classification of machines is the grain of the machine. This denotes the relationship between the number of processors and the size of each processor. Fine grain machines and coarse grain machines are available. Coarse grain machines are composed of fewer more powerful processors whilst the fine grain machines consist of a larger number of simpler processors.

In a similar way one can define grain and coupling of software processes. The grain of a particular problem is the average size of the processes which make up that problem. The coupling is the amount of data shared between processes. Finally, the degree of parallelism is defined as the number of processes which make up the problem, and the the level of parallelism defines whether the parallelism occurs at procedure level, expression level, instruction level, bit level ... etc.

The concepts introduced above enable us to differentiate between multiprocessor machines. More general classification schemes enable us to identify all possible machines whether serial, parallel, single or multiprocessor. The current most widespread classification is that of Flynn[7] which, interestingly, was defined in 1972 well before fast and efficient multiprocessor machines became commercially available. It relies upon the concept of 'stream', which may be a stream of instruction or a stream of data. The classification divides computers in four groups as shown in Figure 6.1.

	Single Instruction	Multiple instruction
Single data	SISD (von Neumann model)	MISD (pipeline systems)
Multiple data	SIMD (array processors)	MIMD (general multiprocessors)

SISD	:	Single Instruction stream Single Data stream
MISD	:	Multiple Instruction stream Single Data stream
SIMD	:	Single Instruction stream Multiple Data stream
MIMD	:	Multiple Instruction stream Multiple Data stream

Figure 6.1: *Flynn's taxonomy*

This classification fails, however, to distinguish between the different types of MIMD machines. In order to improve the Flynn classification, different schemes have been proposed but are less in use.

Sharp [8] proposed the scheme shown in Figure 6.2 where the MIMD category is divided into two sub-classes. Other classification have been designed by Kuck[9], Treleaven[10] and Gajski[11]. They are shown in Figure 6.2[12]. A totally different approach has been followed by Shore[13], where machines are classified according to their organisation from four abstract basic parts – the control unit, the processor unit, the data memory and the instruction memory. This classification is shown in Figure 6.3[14].

	Single processor	Multiple processors
Scalar data	SES (Flynn's SISD)	MES (Flynn's MIMD)
Array data	SEA (Flynn's SIMD)	MEA (Flynn's MIMD)

SES	:	Scalar data executing on single processor
SEA	:	Array data executing on single processor
MES	:	Scalar data executing on multiple processors
MEA	:	Array data executing on Multiple processors

Sharp's taxonomy

	Execution Stream number and type			
n Stream d type	Single, Scalar	Single, Array	Multiple, Scalar	Multiple, Array
calar	SISSES	SISSEA		
rray		SIASEA		
Scalar			MISMES	MISMEA
Array				

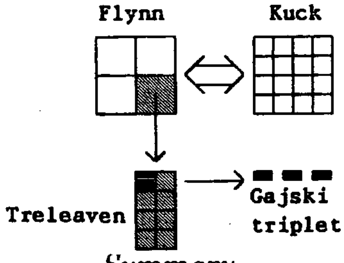
	DATA MECHANISM	
CONTROL MECHANISM	SHARED MEMORY	PRIVATE MEMORY (message passing
control driven	von Neumann	communicating processes
pattern driven	logic	actors
demand driven	graph reduction	string reduction
data driven	dataflow I–structure	dataflow tokens

less explicit control

Kuck's taxonomy *Treleaven's taxonomy*

	Task	Process	Instruction
Serial			
Parallel			

Gajski and Pier's taxonomy



Summary

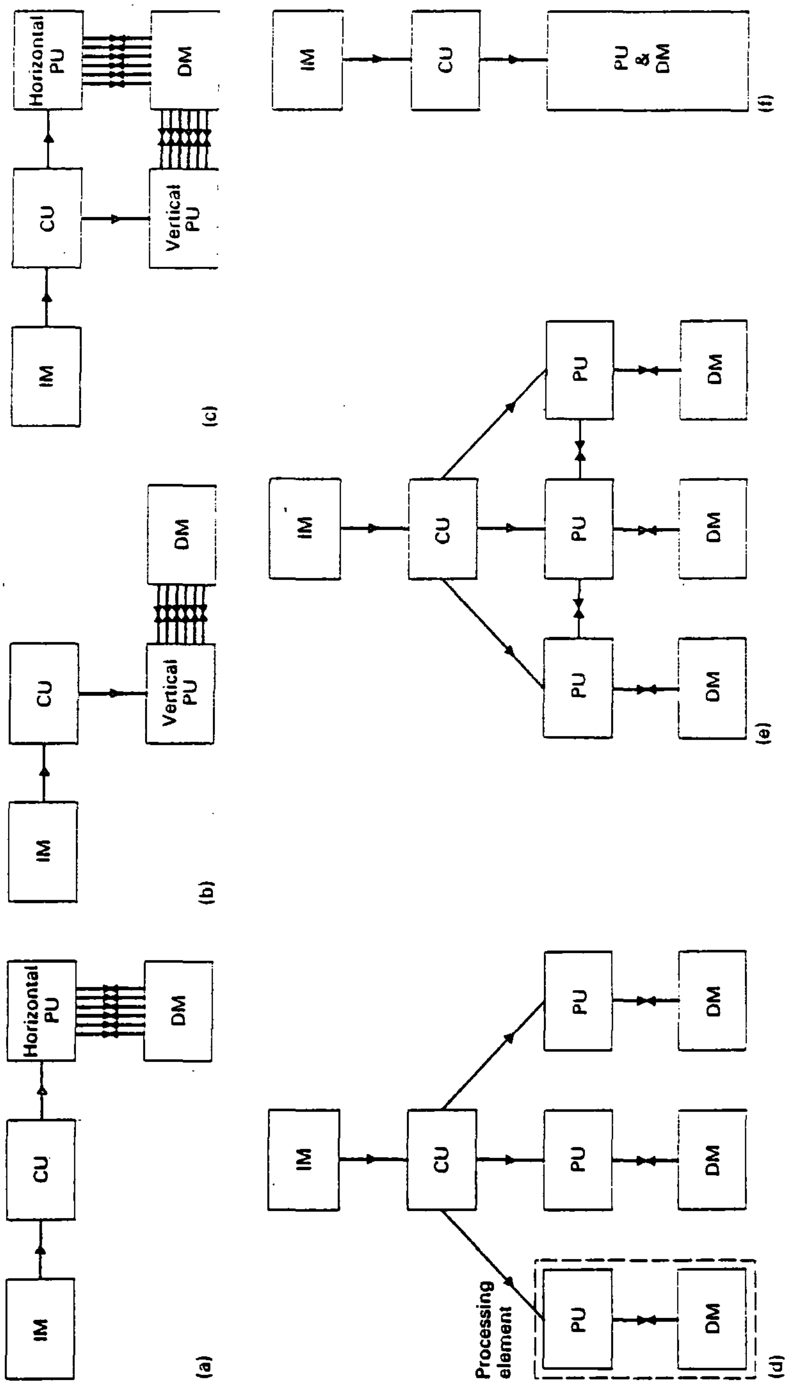
Figure 6.2: *Sharp's , Kuck's, Treleaven's and Gajski and Pier's taxonomies*

6.4 Multiprocessor machines : Old and New

Although not complete, the Flynn taxonomy being the most popular, it is used next for the description of a few typical machines. Several books have a comprehensive description of parallel machines available[15,8,16,4]. An up-to-date survey of all parallel machines available in 1991 along with a description of the hardware and the software has recently been published[2]. Latest developments are found in computing journals[17,18]. There are currently available on a commercial basis several types of machines: SIMD and MIMD.

The SIMD can be of pipeline or vector type. The pipeline machines are simply a series of processors arranged in a pipeline form with the data passing along the pipeline and each processor receiving a different treatment. The speed of such a structure is limited by the speed of the slowest processor. The structure attains its maximum performance when the pipeline is full.

The vector and array processor machines are composed of a large number of simpler processors (fine grain) which execute the same instruction, normally with a set of hardware connections between the processors which enables the data to circulate among the processors in a predefined way (loose coupling). All the processors are synchronised by a clock. The machine is then rated at the speed of the slowest processor.



Shore's classification of computer architecture according to the connections of logical components. (a) Type I, word-serial and bit-parallel; (b) Type II, word-parallel and bit-serial; (c) Type III, orthogonal; (d) Type IV, unconnected array; (e) Type V, connected array; (f) Type VI, logic-in-memory array. IM = instruction memory; CU = control unit; PU = processing unit; DM = data memory.

Figure 6.3: *Shore's taxonomy*

Examples of the above machines are given in Table 6.1.

There is also a special case of SIMD machine which is described next. The

OLD (70's)	NEW (80's)	VERY NEW (late 80's, 90's)
UNIVAC	CYBER 205	1) *Mid-range machines*
ILLIAC IV	CRAY X-MP	Convex (4 proc)
IBM Stretch	IBM 3090	Alliant (28 proc)
IBM 7090	ICL DAP	FPS (8–28 proc)
IBM 701	NEC SX	HP Apollo DN10000 (4 proc)
CDC 6600		
WARP (sytolic)		2) *Large scale machines*
Purdue (systolic)		Connection Machine (65356 1-bit proc)
		CRAY Y-MP
		MASPAR MP-1 (1 to 16000 proc)

Table 6.1: *Examples of SIMD machines*

term systolic † machine originally designated a special purpose architecture. It consisted of a series of simple processors, called processing elements, which were regularly interconnected so that there was links with the neighbours only and synchronisation of the execution of all instructions by the processing elements was achieved through a clock[19]. They 'pump' data synchronously to give regular data to the network. They are a combination of pipeline and array architectures. A successor to the systolic machine is the wavefront machine[19] which has the same connectivity as the systolic arrays but is data-flow driven, so therefore does not require a synchronous clock.

Nowadays, the word systolic has a more general meaning refering more to a programming technique than to a hardware architecture[20]. This programming method, called systolic design, transforms algorithm descriptions that do not specify concurrency or communication into functions that distribute the program's operations over time and space. These functions can then be refined further and translated into a description for either fabrication of a VLSI chip, as it used to be, or more recently into a distributed program for execution on a multiprocessor machine which was not originally designed for systolic style programming.

† Systolic derives from the word systole which is a biological term designating the heart contraction rhythm to pump blood

The systolic design mainly consists of, given a serial specification for a serial algorithm, designing a restricted serial specification which can then be directly mapped onto a systolic specification. The attractive aspect of this concept is that this design can be automated and a wide range of problems can be suitably solved using this technique, such as numerical analysis, signal or image processing, graph theory ... etc. With the increasing possibilities of the new computer architectures, some of the restrictions on systolic specification can be relaxed, such as the enforcement of connection to neighbours only can be extended to constant distance for all connections. The main idea of regular array has to be kept, though. The essence of a systolic design is to minimize both the time it takes to run the program and the number of processors necessary to run the program.

Two types of MIMD machines are found: shared (tightly coupled) and distributed (loosely coupled) memory machines. Examples are given in Table 6.2. The performance of the most recent machines are over the GFlop (10^9 Floating Point Instructions per Second) mark.

The shared memory machines use common memory to exchange information between processors with memory protection implemented both in hardware and via the operating system. The distributed memory machines use a message passing mechanism to transmit information between processors via hardware links. This is illustrated in Figure 6.4.

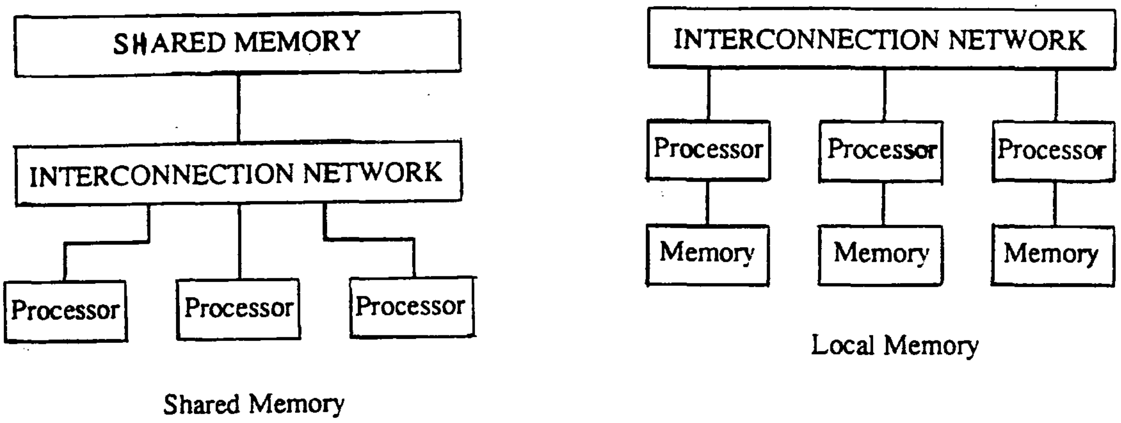


Figure 6.4: *Communication in shared and distributed memory environments*

The connections between processors of distributed machines can take various

SHARED MEMORY		DISTRIBUTED MEMORY	
OLD (80's)	NEW (late 80's, 90's)	OLD (80's)	NEW (late 80's, 90's)
CMU	Encore (Multimax)	Dado project	1) *Mid-range machines*
C.mmp	Pyramid	PAX (japanese)	Caltech Cosmic Cube
Cm*	Sequent	WRM	iPSC
Denelor HEP	DEC 6000,9000,5000		NCUBE-2
NYU Ultracomputer	TC 2000		Intel Hypercube
BBN Butterfly	Stardent		Parsys
IBM RP3			INMOS Transputer
UI Cedar			Transtech boards
CHopp			
			2) *Large scale systems*
			Meiko Computing surface
			Tnode (France)
			Intel Delta
			(announced Nov 90)

Table 6.2: *Examples of MIMD machines*

geometric forms–linear, ring, star, tree, cube ... etc as illustrated in Figure 6.5. The connectivity chosen depends on the nature of the parallelism found in a particular application. Research is still going on in this area.

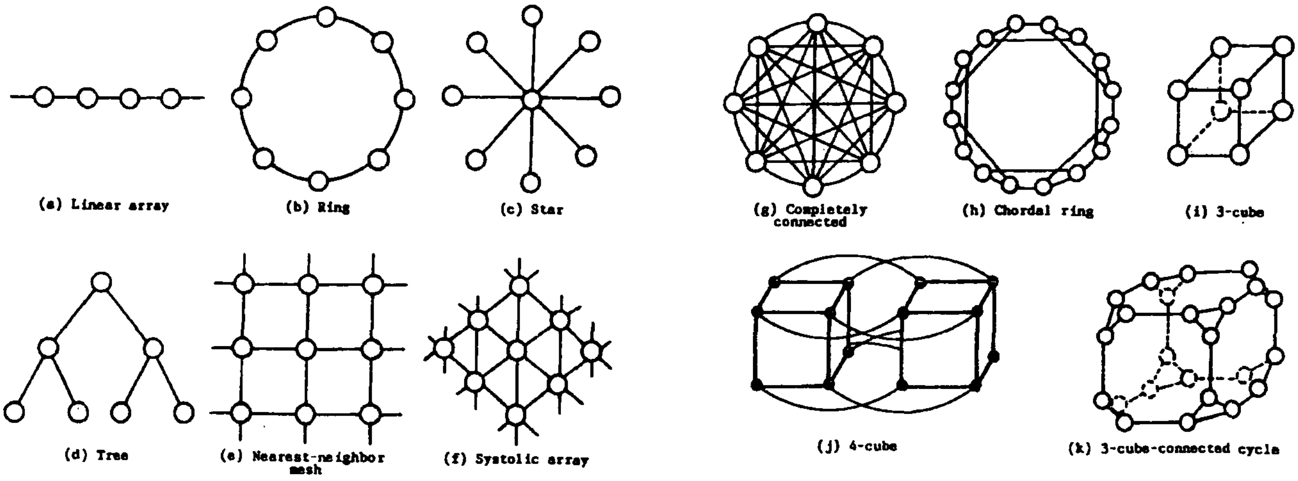


Figure 6.5: *Connections between processors*

Future machines seem to be moving towards hybrid architectures such as SIMD+MIMD or towards a totally different approach to software analysis like the data flow model described in detail by Sharp[21].

6.5 Software survey

One of the main argument against parallel processing is the lack and non-standard nature of the software available and the difficulty to program parallel machines. This argument only applies to programmers in general, not those, like scientific programmers in leading edge fields, who have been used to programming in assembler languages to get optimum performance out of their machine therefore accepting the everchanging nature of computers and computer languages.

To overcome the problems caused by parallelism various approaches have been followed[2]. The approach in which the parallelism of the machine is completly hidden to the user through the use of parallelising compilers can be quite popular for those who are not interested in parallel processing but want more performance out of their machine. These compilers take standard serial code, for example FORTRAN code, and automatically search for parallelism and produce a parallel executable code. This means that the parallel machine appears to be a serial machine to the user with increased computing power. Such compilers are already available for the shared memory MIMD machines like Convex and Alliant, and Meiko is developing a similar compiler for its distributed memory MIMD machine[22].

Another approach is through tools which are developed to allow programmers to reuse what they know about concurrent programming on conventional machines such as semaphores. This is the case for the MIMD shared memory machines like the Encore machines, where languages like FORTRAN have been enhanced with a 'parallel' construct together with a semaphore mechanism[23]. This is easy enough to use but ensuring that memory contengency is in all cases properly resolved by the program implies some effort on the programmer's side, though it is not a problem if the programmer is already familiar with these concepts.

Finally, the more involving approach is when programmers are made to write parallel programs explicitly using various software tools. These tools can range from more or less sophisticated function calls made within serial programming languages to entirely new programming languages for describing parallel concepts. Here the degree of commitment of the programmer to learn new concepts is variable.

So far the most popular approach[2] has been to provide the user with the ability to create and place processes on processors which can communicate with one another through function calls. The functions are provided by the manufacturer in the form of a set of tools, or library, with a range of capabilities. Examples of such libraries of parallel utilities, or toolsets, are given next.

The 3L parallel FORTRAN[24] or C libraries for the Transputer contain very basic facilities to send and receive messages between processes running in parallel. It is very dependent on the hardware of the machine on which it is run as process numbers and communication names have to be specified according to the particular Transputer network used.

The CS Tools[25] toolset is more general in the sense that it provides a layer of abstraction between the hardware and the processes, so the variation of the number of processors used or the change in the mapping process-processor dos not require the program to be changed and recomplied. It also allows mixed processor architectures to be connected together like, for example, a network of Suns and Transputers.

At the top of the range of these utilities can be found systems like Express[26]. The degree of abstraction from the hardware is higher, therefore the degree of portability between different machines and processors is superior, although it still restricts itself to MIMD distributed memory machines. The facilities provided are also more comprehensive than in CS Tools. They include broadcast of messages, global synchronisation, exchange of messages and automatic split up of grid-based applications, such as image processing, onto processors, standardised access to input/output and graphics routines, debugger, tools to evaluate performance without interfering with the execution of the program ... etc.

Another approach is to create a 'half language' which is well suited for implementing communication between parallel processes and can be tied up with conventional languages like C or FORTRAN for executing the calculations. This produces an hybrid-mixed language whose aim is to both reuse existing sequential code and provide a standard well-suited and portable communication language. Two examples of such languages are LINDA and STRAND. A brief specification of the capabilities of both systems is given next.

LINDA[27] is solely a communication paradigm which supports interprocess communication, shared data structures and process creation. It is therefore portable on a wide range of multiprocessor machines, whether MIMD shared or distributed memory machines. It claims to be simple to use and scalable, which means that the number of processors on which the program actually runs can be changed without altering the program. Debugging tools are provided.

LINDA is based on the concept of *tuples* and *tuple space*, which are respectively objects and object store. A tuple is comprised of fields. These fields can, for example, be an integer variable, which either have an actual value – they have been assigned a value – or have a formal value – they stand for themselves. A pattern matching mechanism operates on the tuple space so that two tuples in the tuple space can be associated and their fields matched so that formal value fields from one tuple are assigned to the values of the actual value fields of another tuple.

This pattern matching mechanism is used to carry out to the communication between different processes. This is illustrated in Figure 6.6.

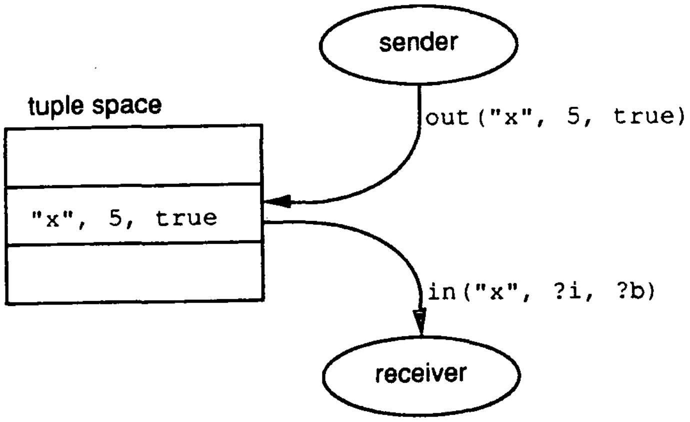


Figure 6.6: *Tuples and tuple space for communication between processors*

The process which sends a message places the contents of the message in the tuple which is itself inserted in the tuple space (operation 'out'). The receiving process looks in the tuple space for a matching tuple. If this matching tuple is present , the message is copied accross using the matching mechanism and it is removed from the tuple space. If not, the receiver process is blocked until a matching tuple appears in the tuple space. The communication is thus synchronised.

LINDA is based on ideas taken from logical languages such as object-orientated languages. One drawback, though, is that being a high level communication mechanism, overheads are introduced which makes it not suitable for applications like real time monitoring.

STRAND[28] follows very much the same ideas as LINDA in the sense that it is a logic language based on ,pattern matching mechanism, whose syntax resembles very much PROLOG syntax. It is more general, however, as a whole program can be written in STRAND, which supports arithmetic operations, comparisons, recursivity, input/output procedures ... etc. It is therefore a concurrent logic programming language. One important feature is its foreign language interface which enables the incorporation of modules written in sequential languages like C and FORTRAN. This makes it, like LINDA, an hybrid language which permits the reuse of existing serial codes. It can thus be used as a harness to sequential languages. It claims to be portable accross the whole range of parallel machines, from SIMD to MIMD computers.

Finally, for those who enjoy programming in parallel, or more realistically those who need to fully control calculations and communications, there are several dedicated parallel languages. OCCAM and ADA are two good examples of such languages. They are both based on a formal method called Communicating Sequential Processes[29] (CSP) developed by Hoare. OCCAM is virtually the implementation of this method whereas ADA implements a modified version of it. The CSP method is briefly described next and is followed by an overview of where ADA implementation differs from it.

CSP unifies the concepts of synchronisation and communication. Processes in CSP synchronise and communicate by means of input and output statements based on a rendezvous mechanism. A rendezvous is established when one task is ready to execute an input statement and the second task is ready to execute the corresponding output statement. If either task is not ready, then the other task is forced to wait. Communication in CSP is therefore explicit, requiring no shared variables and CSP is therefore suitable for MIMD distributed memory machines.

The language OCCAM[30] has been developed for the Transputer (see section thereafter) and follows exactly the CSP concepts. ADA, on the other side, has adapted the CSP concepts to produce a language containing both communication mechanism and programming facilities designed to tackle real problems, mainly in the field of real time applications.

The main differences between CSP and ADA are listed below[31]. A rendezvous in CSP is an unidirectional communication. In ADA, the rendezvous mechanism, which is implemented by çalls to functions, enables exchange of information between both processes, thus realising a bi-directional communication. When a communication is established in CSP parameters are copied accross between the two processes, then each process resumes its own task. In ADA, when the rendezvous is established the process which is accepting the communication can execute statements before exchanging information with the process which initiated the communication. This enables the implementation of drivers for interface with the hardware, such as printer drivers. Additional facilities include time-outs on communication, which enables the process initiating the communication to stop waiting for the other process to be ready after a given time and retry later. This is useful for implementing error handling mechanism.

6.6 Applications for parallel computers

The applications that can benefit from parallel processing are obviously those which need a lot of computing power. This includes graphics, scientific and engineering fields such as flow dynamics, particle behaviour, weather prediction and seismic modeling, VLSI design, Artificial Intelligence ... etc.

Some economic factors must also be taken into account in the advantages for using multiprocessor machines. Nowadays these machines are capable of reaching and surpassing the performance of the traditional supercomputers like the CRAY at a fraction of the cost. This, in theory, suggests that multiprocessor machines should put out of business the more costly machines. In practice, however, one of the reasons that these expensive machines still sell is the availability of a wide range of application software.

This is why machines like the CRAY Y–MP, which contains a smaller number of powerful processors, are still very popular because large amount of software can be recycled for them. On the smaller scale, machines based on classical microprocessors like the Intel 80386 which provide mainframe performance at minicomputer cost are also popular because much of the existing software can be re-used.

Most multiprocessor machines are difficult to program and porting software from serial machines to parallel machines is expensive and not always efficient because of overheads introduced by the communication necessary to exchange information between processes. In this sense, the shared memory MIMD machines are easier to use than the distributed memory machines because the communication is implicit through the use of common memory. Therefore, most of the techniques developed for multitasking computers such as semaphores can be used directly. One drawback, though, is that they cannot be scaled up indefinitely as the access to memory eventually creates a bottleneck.

A typical example of the economic strength of the traditional supercomputer machines is the Met Office in charge of weather forecasting and of monitoring climatic changes, which in early 1990 bought a new CRAY Y-MP (1.4 GFlop) to improve its forecasting[32]. The European Centre for Medium Range Weather Forecasting is in the process of installing a similar computer[32]. At the same time, computer scientists are excited at the possibility of analysing data related to weather forecasting in real time when 300 GFlop machines are be available[32]. The possibility and ease of programming such a machine is, however, still questionable.

These giant multiprocessor machines are seen by some as a step backward because the influence of the hardware on the software is great compared to the serial model. The implications are that parallel software is not really portable between machines of different architectures, for example between shared and distributed memory machines. Some high level languages have been developed to address this issue, like LINDA and STRAND, discussed before, where communication and parallelism are expressed as abstract models.

The trend for the future is not clear. Some people firmly believe that parallel machines should be able to run existing serial codes faster than on existing serial machines and compilers which automatically parallelise serial codes are being developed to address this demand. The parallelism is then hidden as it was in single processor machines and the user need not worry about it.

Another approach which follows the same kind of ideas is to hide distributed memory by emulating shared memory on top of it, using a switching network, as this is easier to deal with. An example of such a machine is the BBN Butterfly with its most recent version, the TC2000[33].

Others are in favour of a totally different approach to computing where the von Neumann model is forgotten and programs would not be instruction driven any more. A step in this direction has been taken by object-orientated languages like STRAND. Functional and data-flow approaches seem the trend for the future. These models assume parallelism and introduce serial processing only when necessary.

Among all the machines and systems described in the previous sections, the work carried out in this part of the thesis focuses on the use of a particular MIMD distributed memory machine based on the Transputer. The application area is engineering with the development of solvers for large systems of linear equations as they occur in the finite element method.

6.7 Transputers

The Transputer[34] is a VLSI chip which combines processing, memory and connection links on a single physical chip[35]. The first Transputer to be commercially available appeared in 1985. It was the T414 32-bit Transputer. It had 2KBytes of on-chip memory and four 10 Mbits/second serial links.

These links are a special feature of all members of the Transputer family as they enable Transputers to be connected together in a flexible way. There is an on-chip link manager which looks after the exchange of information with other Transputers enabling the communication to be concurrent with the calculations. Although each Transputer possesses four links which enable the construction of various shapes of Transputer network, there is a limitation on the complexity of possible connections. For example, with four links, a maximun of five Transputers can be arranged in a completely connected network as shown previously in Figure 6.5.

A more powerful Transputer, the T800, was introduced in 1987. It has 4 KBytes of on-chip memory and a floating point co-processor but still four links, running at 20 Mbits/second. Its internal architecture is shown in Figure 6.7.

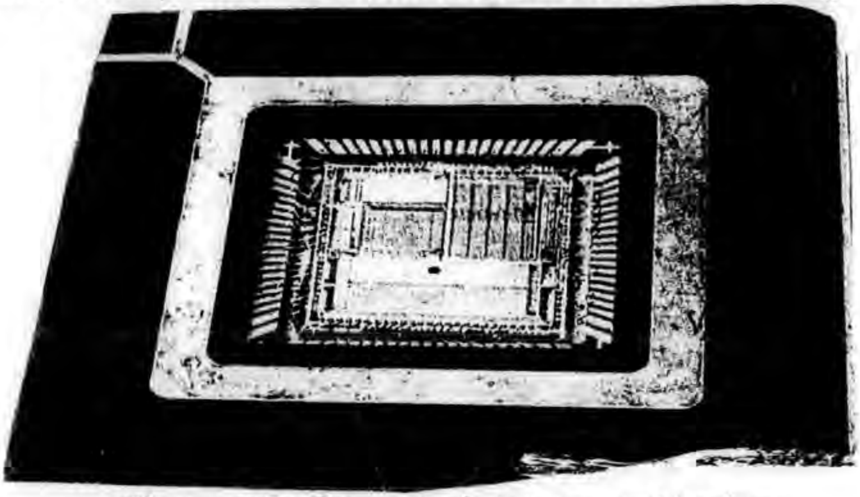


Figure 6.7: *Transputer's internaı architecture*

This is the type of Transputer used for the work carried out in this thesis. At the time of writing this thesis, a newer version of the Transputer family was launched, the T9000. It has increased power, achieving 200 Mips and 25 MFlops peak performance at 50 MHz. It is based on a superscalar architecture with a 32-bit integer processor and a 64-bit floating point unit. The on-chip memory has been increased to 16 Kbytes. It is believed that it has the right balance of computing and communications[36] and is expected to fare particularly well in the embedded applications market.

Transputers can be programmed with many conventional languages: FORTRAN, Pascal, C, Prolog ... etc. These languages have been enhanced with tools to build parallel programs. A special purpose language, OCCAM, was developed based on the formal method CSP The Transputer, in terms of design, implements in hardware the philosophy of CSP. OCCAM and the Transputer were developed at the same time

The Transputer Development System[37] helps to develop OCCAM programs, providing a special editor based on folds, a compiler, a linker and a means of configuring the Transputer. A number of other operating systems have been developed for Transputer-based systems including Trollius, MeikOS and Helios, all described by Harp[19].

Transputers on their own do not constitute a usable machine as they do not provide access to external devices like keyboards, screens and files. Most of the time a network of Transputers is attached to a 'host machine', for example a PC or a workstation. A file server runs on the host machine which executes, for example, file access for the network of Transputers.

A Transputer-based machine can come in two forms. The first form consists of Transputers connected to a host machine which runs the operating system. Processes are loaded onto the processors and executed by a command given on the host. While the parallel system is running, the host acts as a fileserver. In the second form, the operating system runs on one or more of the parallel processors. In this case, the host machine acts solely as a fileserver.

An example of the first case is a PC with a INMOS[34] or a Transtech[38] add-on board where the user accesses the network of Transputers via the PC command line. An example of the second case is the Meiko Computing Surface[39], based on the MeikOS operating system (Meiko's implementation of UNIX), such as the one in the Edinburgh Parallel Computing Centre[40], where the user actually logs on to the Transputers themselves and never has to deal directly with the file server machine. The operating system runs, however, solely on one Transputer of the network. Another operating system, Helios, is a fully parallel implementation of UNIX which runs accross the network of Transputers.

The machines used for this work are of the two types. The machines involved were a PC with a Transtech board, a Sun SPARCstation remotely logged on to a Meiko Computing Surface and the Edinburgh Concurrent Supercomputer. A brief description of each machine and their corresponding software is given next.

The first machine is an IBM PC AT with a TMB08[41] Transtech board containing three T800 Transputers, one with 8 MBytes of memory and the other

UNIV DUNELM

two with 2 MBytes of memory. The idea behind this selection of memory is that one Transputer, called the root, would store all the relevant data for the program while the other two Transputers would carry out calculations on subsets of the data, therefore needing less memory.

The programs for this machine have been developed using the 3L parallel FORTRAN[24] which is standard FORTRAN enhanced with some non-standard features (such as identifier names longer than six characters) and with a library of routines enabling communication between processes and processors. This is the function calls approach to parallelism described in section 6.5.

The second system is composed of a SPARCstation 1, running version 4.0.3 of Sun's SunOS operating system and a Meiko Computing Surface comprising one local host MK014 board, two MK060 boards with four T800 Transputers with 2 MBytes of memory each, the whole being contained in a M10 cabinet. Programs have been written in Meiko FORTRAN which is similar to that of 3L except the library of routines for communication is a general purpose ready-made communication harness, CS Tools, more powerful than the 3L routines for communication. The SPARCstation was used to develop the program as CS Tools enables us to run concurrent processes on one processor, here the Sun processor[42]. The program had to be recompiled to run on the Transputers as the machine language is different for each chip.

The third system used is the Edinburgh Parallel Computer Center (EPCC)'s Computing Surface which is based on Meikos's Computing Surface. It is a multi-user machine, consisting of domains of Transputers, each with its own local memory, and interconnected by programmabe switch chips. This machine is part of the EPCC's pool of parallel machines which also includes an AMT DAP, a Meiko i860 facility and a Parsytec machine. Details about the EPCC can be found in its Anual Report and Project Directory[43]. As this machine is in effect a standalone Transputer-based machine, it is worth giving a more detailed description, which is presented in the following paragraphs.

The Computing Surface comprises over 430 Transputers organised in groups of fixed sizes. It revolves around a spine of T414 Transputers which handles all the data transfers in the machine. Connected to the spine are file servers, terminals and these groups of Transputers called domains.

Each domain is composed of a seat Transputer, a T800, and a number of slave Transputers, also T800, all having a minimum of 4Mbyte of memory each. The seat Transputer is connected to the file servers and outside world via its corresponding T414 on the spine.

The entire machine is managed by a global operating system, called M^2VCS, which is in charge of allocating resources to the user, handling communication accross the spine and dividing the machine into a number of domains. Each domain runs on the seat Transputer a UNIX like operating system called MeikOS. The editing and compiling of the user's programs are carried out under MeikOS on the seat Transputer while the program itself runs on the network of slave Transputers. The domains thus appear like a private, single user computing surface, very much like powerful workstations. MeikOS also manages the fileservers which are Hewlett-Packard disks.

The partition of the machine into domains is such that small domains are available for testing and debugging purposes and larger domains are reserved for high performance runs. The machine can be accessed remotely via the national academic network JANET, which was the route used for the work in this part. Finally, the software supported on the Computing Surface comprises FORTRAN, C, OCCAM and CS Tools.

The programs were initially developed on the PC then adapted to run on the Meiko machines. The difference between the two versions is in the communication aspect of the program. On the PC the communication involves establishing and numbering every communication between two processes whereas on the Meiko the actual communication is carried out by CS Tools and the user refers to it by abstract names which means that the number of processes and processors can easily be changed without recompiling the programs. More details about the use of CS Tools in the program will be given in subsequent sections.

Having set the scene on parallel processing and Transputers the next section will be devoted to the definition of the problem and the derivation of the parallel algorithms.

References

1. Menabrea L.F, *Sketch of the analytical engine invented by Charles Babbage*, ESP. Bibliothèque Universelle de Genève, 82, 1842.
2. Trew A. and Wilson G., *Past, Present, Parallel: A Survey of Available Parallel Computing Systems*, Springer-Verlag, 1991.
3. Hord R.M., *The Illiac-IV: The first Supercomputer*, Computer Science Press, 1982.
4. Almasi G.S. and Gottlieb A., *Highly Parallel Computing*, The Benjamin/Cummings Publishing Company Inc, p27, 1989.
5. CRAY-1 system (1976) described in Ref 3, p311 & Hockney R.W and Jesshope C.R, *Parallel Computers*, Adam Hilger Ltd, 1981.
6. Ref 4, p5.
7. Flynn M.F, 'Some Computer organisations and their effectiveness', *IEEE Trans. Compt. C-21*, pp 948–960, 1972.
8. Sharp J.A, *An introduction to Distributed and Parallel Processing*, Blackwell Scientific publications, p37, 1987.
9. Kuck D.J, *The structure of Computers and Computations*, John Wiley publishing, 1978.
10. Treleaven P., 'Control-Driven Data-Driven and Demand-Driven Computer Architecture (abstract)', *Parallel Computing 2*, 1985.
11. Gajski D.D. and Peir J., 'Comparison of five multiprocessor systems', *Parallel Computing 2*, pp 265–282, 1985.
12. Ref 4, pp 112–114.
13. Shore J.E., 'Second thoughts on parallel processing', *Computers and Electrical Engineering*, **1**, (1), pp 95–109, 1973.
14. Ref 8, p35.
15. Kuhn R.H. and Padua D.A, *Tutorial on Parallel Processing*, IEEE Computer Society publication, 1981.
16. Hockney R.W. and Jesshope C.R, *Parallel Computers 2: Architecture, programming and Algorithms*, ed. Adam Hilger, 1988.
17. *Computing*, weekly publication, VNU House, 32–34 Broadwick street, London, W1A 2HG, UK.
18. *Computer Weekly*, weekly publication, Quadrant House, The Quadrant, Sutton, Surrey, SM2 5AS, UK.
19. Harp G., *Transputer Applications*, Pitman ed., p5, 1989.
20. Lengauer C. 'Systolic Design', *Edinburgh Parallel Computing Centre annual Seminar: Abstracts*, 23rd September 1991.
21. Ref 8, pp 139–169.

22. Neesham C., 'Hi-tech wheathercocks help to save the planet', *Computing*, pp 16–17, 19 july 1990.

23. Encore Computer Corporation, PO Box 409148, Fort Lauderdale, Florida, 33340–9148, USA.

24. *Parallel FORTRAN user guide*, 3L Ltd, Peel House, Ladywell, Livingston, EH54 6AG, UK, 1988.

25. *CS Tools, A technical overview* and *A programmer's introduction to SUN - CS Tools*, Meiko Limited (Ref 30).

26. Ref 2., pp 292–296.

27. Leler W., 'Linda meets UNIX', *Application of Transputers, Proceedings of the First International Conference on Application of Transputers*, Liverpool, UK, 23–25 August 1989, and Ref 2., pp 304–308.

28. Foster I. and Taylor S., *STRAND: New Concepts in Parallel Programming*, Prentice Hall.

29. Hoare C.A.R., 'Communication Sequential Process', *Communication of ACM*, **21**, (8), August 1978.

30. INMOS Limited, *OCCAM 2 Reference Manual*, Prentice Hall, 1988.

31. Gehani N., *ADA, Concurrent Programming*, Prentice Hall, 1984.

32. Abate T., 'Engines power business towards the chequered flag', *Computing*, pp 22–23, 13 december 1990.

33. Ref 2., pp 64–75.

34. The Transputer is manufactured by INMOS Limited, 1000 Aztec West, Almondsbury, Bristol, BS12 4SQ, UK.

35. Ref 19, pp 11–30.

36. SERC/DTI Transputer Initiative Mailshot, *INMOS News Release*, p 17, May 1991.

37. *Transputer Developement System, second edition*, INMOS Ltd, Prentice Hall, 1990.

38. Transtech Devices Limited, Unit 17, Wye industrial estate, London road, High Wycombe, Bucks., HP11 1LH, UK.

39. Meiko computing surface, available from Meiko Limited, 650 Aztec west, Bristol, BS12 4SD, UK.

40. Edinburgh Parallel Computing Centre, the King's Building, Mayfield road, Edinburgh, EH9 3JZ, UK.

41. *TMB08 installation and user manual*, available form Transtech Devices Limited (Ref 29).

42. *Computing Surface, CS Tools for SunOS documentation*, edition 83 - 009 A00 - 02.02, two volumes, available from Meiko Limited (Ref 30).

43. Edinburgh Parallel Computing Centre, *Annual Report 1990–91 and Project Directory*, available from Ref. 41.

Chapter VII

Parallel solvers

.1 Introduction

In such fields as Engineering and Science the need for solving systems of linear equations often arises as a consequence of the discretising sets of partial differential equations, using for example finite element or finite difference methods. Such equations model a vast multitude of physical phenomena.

There is always a need to solve large systems of equations and the size of the systems which are actually attempted are greatly dependent on the speed with which they can be solved. Therefore, much effort has been put into reducing the time taken for such solutions by the construction of efficient algorithms, especially where special features of the system can be exploited, such as symmetry, sparsity, positive definiteness and so on. One way to speed up the solution is to use a computer with a faster chip (as one becomes available) without the need to alter the software.

With the advent of parallel computers the necessity arose not simply to port the relevant software to these new machines but to devise new algorithms to exploit the inherent parallelism of the problem. Further difficulties existed, and still exist, because of the greatly varying architecture of the different parallel computers available. These differences manifesting themselves in the forms of variation in the access to data, change in the number of processors and hence complexity, and programming languages.

Research on parallel solvers has followed the developments and improvements of parallel machines. The amount of work done in this field is too large to be reviewed here since there is a wide range of both solvers and parallel machines. For references, we direct the reader to two recent comprehensive surveys by Bertsekas[1] on the iterative class of solvers and Gallivan[2] on the direct class of solvers for dense systems.

Our prime interest in this work is in the direct class of solvers based on the LU decomposition applied to dense or locally dense matrices. Therefore, the survey focuses on this class of solvers, although it also seems interesting to mention two special methods for solving sparse systems, as they involve new concepts also relevant to dense problems.

7.1.1 Survey

The first method is concerned with the use of a data flow model of computer and was developed to solve sparse systems in parallel using the LU decomposition technique followed by forward and backward substitutions[3,4].

The algorithm for the sparse LU decomposition consists of two steps: a divide operation involving a division and an update operation involving a multiplication and a subtraction. A data flow diagram for these three operations is devised where the arithmetic operations on the operands are only executed when the operands become available.

The algorithm is implemented on the MIT TTDA[†] machine which is a fine grain distributed memory computer. It is composed of N identical processing elements whose structure is very simple, N identical storage elements which are a special type of memory suited for storing array-like data structures and a communication network which links processing elements and storage elements. The network is arranged in a N-cube configuration which is composed of exactly N nodes, each of which has exactly $\log_2 N$ links to the neighbouring nodes.

The scheduling of the processors and the optimal configuration for the minimal completion time are derived from the data flow model. The point in describing this work is to show that data flow computers are now used in practical applications and that they are a potential successor to the traditional instruction driven models, whether serial or parallel.

The second method is based on a large-scale MIMD machine, the $(SM)^2$–II[‡]. containing thousands of microprocessors[5]. It is a machine dedicated to scientific

[†] Tagged Token Dataflow Architecture

[‡] Sparse Matrix solving machine

calculations and is intended to be a back-end processor. For that reason, a static approach is implemented and no multi-user or multi-task services are provided.

The structure of the machine can be described as a series of clusters connected via a simple bus. Each cluster is itself composed of a limited number of processors, each with private memory, and connected via a simple bus. Although the memory is physically distributed each cluster has global shared logical addresses. It is a mixture of shared and distributed memory configurations.

A small special operating system controls the machine and ensures that the global addresses are converted to the local addresses of each processor. A C-like language is used to program the machine. The communication between processes is achieved through static channels and shared variables are not allowed. The language is therefore based on a message passing strategy while the architecture is organised around buses and global addressing mechanisms. It is an interesting mixture of concepts which demonstrates that hybrid-mixed strategies can be viable for certain type of problems.

The paper discusses the advantages and drawbacks of shared memory, distributed memory and data flow approaches for massively parallel machines and explains why the hybrid-mixed strategy was used in the context of solving sparse systems of linear equations. The authors claim that for this particular problem they can obtain 'almost the same performance level as data flow machines with a more cost-effective structure'[5].

The aim in discussing this special-purpose machine was to show how a parallel successor to the simple math co-processor might look like and that there is a need for this type of machines which would be plugged into the back of a general-purpose parallel computer, in much the same way as graphics cards of FFT chips are plugged into serial computers nowadays.

More in the line of the work carried out in this chapter, Geist and Romine[6] have published a study of two possible strategies for the LU factorisation on distributed memory machines: the row-wise and column-wise distribution of the matrix with partial pivoting combined with respectively dynamic load balancing and pipelining of operations executed at each step. Their analysis demonstrates that both solutions are acceptable and achieve high rates of efficiencies. Lin and Zhang[7] have proposed an algorithm for linear triangular systems suitable for both shared and distributed memory machines which can efficiently be used for the backward and forward substitutions for the LU solver. Concepts similar to those described in the two previous papers have been used by Farhat and Wilson[8] in solving specific systems arising from finite differences and finite elements in engineering problems.

7.1.2 Overview of the chapter

The work in this chapter has focused on the development of a set of routines for solving symmetric and unsymmetric systems of linear equations. The prime interest has been in solving dense or locally dense systems, that is to say banded systems, leaving out the solution of sparse systems, which requires very different storage scheme and algorithms. The main reason for this choice is that these solvers have been developed for use in finite element methods for which the systems encountered are mostly banded. A special storage scheme has hence been adopted to take advantage of this structure and some control over the unknowns of the system has also been included. This is discussed in greater details in the next section.

The work described in this chapter is a result of a collaboration with Ian Applegarth, of the University of Newcastle-upon-Tyne. Starting from the same basic equations, Ian has developed algorithms for an Encore Multimax shared memory computer while a version for the Transputer-based distributed machines was implemented as part of this work. It seems very interesting to see how the parallel algorithms, their implementation and the performance vary between the two machines. Therefore, together with a detailed description of the distributed memory version of the solvers, a brief overview of the shared memory version will be given.

The method used in this chapter follows the work carried out by Farhat and Wilson[8] on the solution of symmetric systems of linear equations in parallel. The implementation of the symmetric solver uses exactly the same algorithms as those described by Farhat and Wilson although they did not give details of their communication scheme which had therefore to be developed independently. The unsymmetric solver is an extension of the symmetric method for which the full algorithms have been derived and implemented.

The algorithms have been developed in FORTRAN using double precision arithmetic. As far as possible standard FORTRAN F77 has been used. Inevitably however, the language has to be extended to include parallel operations. These are carried out using the 3L FORTRAN and the CS Tools libraries of parallel utilities for the Transputer-based machines and the Encore FORTRAN extension for the shared memory machine.

This chapter is organised in five sections. Firstly, the full algorithms with the storage scheme and control over the unknowns are given for the serial implementation. The algorithms for this serial version have been coded up by Ian. This has enabled us to perform comparison tests between serial and parallel implementations in order to evaluate the efficiency of the parallel solvers.

Secondly, the algorithms for the parallel solution are derived for the symmetric and unsymmetric solvers for the distributed and shared memory machines. Thirdly, the implementation on the various machines used for this work is described. The performance evaluation of the algorithms is then explained and a description of the tests carried out is given. Finally, graphs of comparative timings of the various solvers along with the analysis of their meaning are discussed and conclusive remarks are made. A paper has been written[9] on this work.

7.2 The serial approach

This section concentrates on explaining what are the underlying equations and the corresponding serial algorithms. The problem considered is that of solving a system of symmetric or unsymmetric linear equations which can be denoted in a matrix form as follows:

$$Ax = b, \tag{7.1}$$

where A is a $n \times n$ real symmetric or unsymmetric matrix, x is a vector of length n which is the unknown of the problem and b is a vector of length n which is the right hand side of the system or known of the problem.

The theory for the symmetric case is a simplified version of the unsymmetric case, therefore easier to understand. When the programs were developed, the parallel algorithms for the symmetric solver were first investigated as it was the easier case, and the unsymmetric system was then considered. Nevertheless, in this chapter the unsymmetric case will be explained first followed by a presentation of the simplification of the algorithms in the symmetric case.

The method used to solve the system of linear equations is a direct method based on the LU decomposition followed by forward and backward substitutions. This is a well established method which is described in many textbooks on numerical analysis[10,11].

The idea is to decompose the matrix A as a product of a lower triangular matrix L and an upper triangular matrix U. This decomposition is possible because any matrix with non-zero terms on the diagonal can be written as a product of a lower and an upper triangular matrix in an infinity of ways[12].

The choice of a particular decomposition has lead to various methods. The Cholesky decomposition corresponds to the case when the decomposition is made unique by imposing that all the diagonal terms of L are equal to all the diagonal terms of U ($l_{ii} = u_{ii}$). The Crout reduction imposes that all the diagonal terms of U are equal to 1 ($u_{ii} = 1$) and the Doolittle decomposition assumes that all the diagonal terms of L are equal to 1 ($l_{ii} = 1$). The technique used here is the Doolittle method.

The corresponding equation is:

$$A = LU, \tag{7.2}$$

where L is a lower triangular matrix with unit diagonal terms and U is an upper triangular matrix. .In the case when the matrix A is symmetric, equation (7.2) takes the following simplified form:

$$A = LDL^t, \tag{7.3}$$

where D is a diagonal matrix. Identifying equations (7.2) and (7.3) in the symmetric case leads to:

$$U = DL^t. \tag{7.4}$$

Therefore, a simple relation exists between U and L in the symmetric case and only one of the two matrices needs to be calculated. The choice is arbitrary. In the unsymmetric case both L and U have to be evaluated. The formulæ for both cases are derived next.

7.2.1 LU decomposition

The decomposition of A into the LU form is shown in Figure 7.1.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 & 0 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 & 0 & 0 & 0 \\ l_{41} & l_{42} & l_{43} & 0 & 0 & 0 & 0 \\ & & & & & & \\ l_{i1} & l_{i2} & l_{i3} & l_{i,i-1} & 1 & 0 & 0 \\ & & & & & & \\ l_{n1} & l_{n2} & l_{n3} & l_{n,i-1} & l_{ni} & l_{n,n-1} & 1 \end{pmatrix} * \begin{pmatrix} u_{11} & u_{12} & u_{13} & u_{14} & u_{1i} & u_{1n} \\ 0 & u_{22} & u_{23} & u_{24} & u_{2i} & u_{2n} \\ 0 & 0 & u_{33} & u_{34} & u_{3i} & u_{3n} \\ 0 & 0 & 0 & u_{44} & u_{4i} & u_{4n} \\ & & & & & \\ 0 & 0 & 0 & 0 & u_{ii} & u_{in} \\ & & & & & \\ 0 & 0 & 0 & 0 & 0 & u_{nn} \end{pmatrix}$$

$$= \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{1i} & a_{1n} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{2i} & a_{2n} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{3i} & a_{3n} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{4i} & a_{4n} \\ & & & & & \\ a_{i1} & a_{i2} & a_{i3} & a_{i4} & a_{ii} & a_{in} \\ & & & & & \\ a_{n1} & a_{n2} & a_{n3} & a_{n4} & a_{ni} & a_{nn} \end{pmatrix}.$$

Figure 7.1: *LU decomposition*

It is important to stress that while Figure 7.1 shows a dense matrix, which is a matrix whose elements are all non-zero *a priori*, it is equally valid for a banded matrix where the zero elements simplify some of the calculations. This is taken into account in the algorithms for which the special storage scheme adopted is described in a later section.

The unsymmetric case is considered next. The calculation proceeds such that the first row of U and the first column of L are derived first, then the second row of U and the second column of L are obtained and so on. The calculation therefore involves n steps, for a $n \times n$ matrix, where each step produces one row of U and one column of L. The equations for step 1, 2 and i are explicitly derived below followed by the general formula for the step i.

Calculation of row 1 of U

$$\begin{aligned}
u_{11} &= a_{11}\\
u_{12} &= a_{12}\\
u_{13} &= a_{13}\\
u_{14} &= a_{14}\\
u_{1i} &= a_{1i}\\
u_{1n} &= a_{1n}
\end{aligned}$$

Calculation of column 1 of L

$$\begin{aligned}
l_{21}u_{11} &= a_{21} \iff l_{21} = a_{21}/u_{11}\\
l_{31}u_{11} &= a_{31} \iff l_{31} = a_{31}/u_{11}\\
l_{41}u_{11} &= a_{41} \iff l_{41} = a_{41}/u_{11}\\
l_{i1}u_{11} &= a_{i1} \iff l_{i1} = a_{i1}/u_{11}\\
l_{n1}u_{11} &= a_{n1} \iff l_{n1} = a_{n1}/u_{11}
\end{aligned} \tag{7.5}$$

Calculation of row 2 of U

$$\begin{aligned}
l_{21}u_{12} + u_{22} &= a_{22} \iff u_{22} = a_{22} - l_{21}u_{12}\\
l_{21}u_{13} + u_{23} &= a_{23} \iff u_{23} = a_{23} - l_{21}u_{13}\\
l_{21}u_{14} + u_{24} &= a_{24} \iff u_{24} = a_{24} - l_{21}u_{14}\\
l_{21}u_{1i} + u_{2i} &= a_{2i} \iff u_{2i} = a_{2i} - l_{21}u_{1i}\\
l_{21}u_{1n} + u_{2n} &= a_{2n} \iff u_{2n} = a_{2n} - l_{21}u_{1n}
\end{aligned}$$

Calculation of column 2 of L

$$\begin{aligned}
l_{31}u_{12}) + l_{32}u_{22} &= a_{32} \iff l_{32} = (a_{32} - l_{31}u_{12})/u_{22}\\
l_{41}u_{12}) + l_{42}u_{22} &= a_{42} \iff l_{42} = (a_{42} - l_{41}u_{12})/u_{22}\\
l_{i1}u_{12}) + l_{i2}u_{22} &= a_{i2} \iff l_{i2} = (a_{i2} - l_{i1}u_{12})/u_{22}\\
l_{n1}u_{12}) + l_{n2}u_{22} &= a_{n2} \iff l_{n2} = (a_{n2} - l_{n1}u_{12})/u_{22}
\end{aligned} \tag{7.6}$$

Calculation of row i of U

$$\begin{aligned}
&l_{i1}u_{1i} + l_{i2}u_{2i} + l_{i3}u_{3i} + \ldots l_{i,i-1}u_{i-1,i} + u_{ii} = a_{ii}\\
&\qquad \iff u_{ii} = a_{ii} - (l_{i1}u_{1i} + l_{i2}u_{2i} + l_{i3}u_{3i} + \ldots l_{i,i-1}u_{i-1,i})\\
&l_{i1}u_{1i} + l_{i2}u_{2i} + l_{i3}u_{3i} + \ldots l_{i+1,i-1}u_{i-1,i+1} + u_{i,i+1} = a_{i,i+1}\\
&\qquad \iff u_{i,i+1} = a_{i,i+1} - (l_{i1}u_{1i} + l_{i2}u_{2i} + l_{i3}u_{3i} + \ldots l_{i+1,i-1}u_{i-1,i+1})\\
&l_{i1}u_{1i} + l_{i2}u_{2i} + l_{i3}u_{3i} + \ldots l_{n,i-1}u_{i-1,n} + u_{in} = a_{in}\\
&\qquad \iff u_{in} = a_{in} - (l_{i1}u_{1i} + l_{i2}u_{2i} + l_{i3}u_{3i} + \ldots l_{n,i-1}u_{i-1,n})
\end{aligned}$$

Calculation of column i of L

$$\begin{aligned}
&l_{i+1.1}u_{1i} + l_{i+1,2}u_{2i} + l_{i+1,3}u_{3i} + \ldots l_{i+1,i-1}u_{i-1,i} + l_{i+1,i}u_{ii} = a_{i+1,i}\\
&\qquad \iff l_{i+1,i} = a_{i+1,i} - (l_{i+1,1}u_{1i} + l_{i+1,2}u_{2i} + l_{i+1.3}u_{3i} + \ldots l_{i+1,i-1}u_{i-1,i})/u_{ii}\\
&l_{n.1}u_{1n} + l_{n2}u_{2n} + l_{n3}u_{3n} + \ldots l_{n,i-1}u_{i-1,n} + l_{ni}u_{ii} = a_{ni}\\
&\qquad \iff l_{ni} = a_{ni} - (l_{n1}u_{1n} + l_{n2}u_{2n} + l_{n3}u_{3n} + \ldots l_{n,i-1}u_{i-1,n})/u_{ii}
\end{aligned} \tag{7.7}$$

The general formulæfor obtaining the elements u_{ij} of U and l_{ji} of L at step i are therefore:

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj} \qquad j \geq i \tag{7.8a}$$

$$l_{ji} = (a_{ji} - \sum_{k=1}^{i-1} l_{jk}u_{ki})/u_{ii} \qquad j > i. \tag{7.8b}$$

One interesting feature of these equations is that u_{ii} is the only element of U calculated at step i that is required to calculate the whole of the column i of L.

Therefore, the evaluation of this diagonal term is performed first then each element u_{ij} and l_{ji} are calculated in turn. The corresponding algorithms is:

```
for $i = 1$ to $n$ do
    calculate $u_{ii}$
    for $j = i + 1$ to $n$ do
        calculate $u_{ij}$
        calculate $l_{ji}$
    end for
end for
```

Figure 7.2: *Algorithm for the unsymmetric LU decomposition (1)*

Similar equations hold for the symmetric matrix. The two matrices L and U being dependent, only one of them need to be evaluated and U is chosen as the independent matrix. The elements of L are therefore expressed as a function of the elements of U.

The dependence formula can be derived using a proof by induction. The formula for the first step can be derived from equation (7.5) using the property that $a_{ij} = a_{ji}$.

$$\begin{aligned} l_{21} &= \frac{a_{21}}{u_{11}} = \frac{a_{12}}{u_{11}} = \frac{u_{12}}{u_{11}} \quad & l_{31} &= \frac{a_{31}}{u_{11}} = \frac{a_{13}}{u_{11}} = \frac{u_{13}}{u_{11}} \quad & l_{41} &= \frac{a_{41}}{u_{11}} = \frac{a_{14}}{u_{11}} = \frac{u_{14}}{u_{11}} \\ l_{i1} &= \frac{a_{i1}}{u_{11}} = \frac{a_{1i}}{u_{11}} = \frac{u_{1i}}{u_{11}} \quad & l_{n1} &= \frac{a_{n1}}{u_{11}} = \frac{a_{1n}}{u_{11}} = \frac{u_{1n}}{u_{11}}. \end{aligned} \tag{7.9}$$

Assuming that for all steps from 1 to $i-1$ the relation $l_{ji} = u_{ij}/u_{ii}$ holds and using equation (7.8) leads to:

$$\begin{aligned} l_{ji} &= (a_{ji} - \sum_{k=1}^{i-1} l_{jk}u_{ki})/u_{ii} \\ &= (u_{ij} + \sum_{k=1}^{i-1} l_{ik}u_{kj} - \sum_{k=1}^{i-1} l_{jk}u_{ki})/u_{ii} \\ &= (u_{ij} + \sum_{k=1}^{i-1} (\frac{u_{ki}}{u_{kk}})u_{kj} - \sum_{k=1}^{i-1} (\frac{u_{kj}}{u_{kk}})u_{ki})/u_{ii} \\ &= (u_{ij} + \sum_{k=1}^{i-1} (\frac{u_{ki}u_{kj}}{u_{kk}} - \frac{u_{ki}u_{kj}}{u_{kk}}))/u_{ii} \\ &= \frac{u_{ij}}{u_{ii}}. \end{aligned} \tag{7.10}$$

Therefore the relation between l and u is:

$$l_{ji} = u_{ij}/u_{ii}, \tag{7.11}$$

and the formula for the LU decomposition in the symmetric case is:

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} \frac{u_{ki}}{u_{kk}} u_{kj} \qquad j \geq i. \tag{7.12}$$

An examination of the equation above shows that the division by u_{kk} does not need to be carried out for every u_{ij}. This equation is related to the calculation of the row i of U where i is fixed for that step and j varies. This means that the elements u_{ki}, which are contained in the column i standing above u_{ii} are used without modification for the calculation of each u_{ij} where $i < j < n$, whereas the elements u_{kj} are different for each u_{ij}. This is illustrated in Figure 7.3.

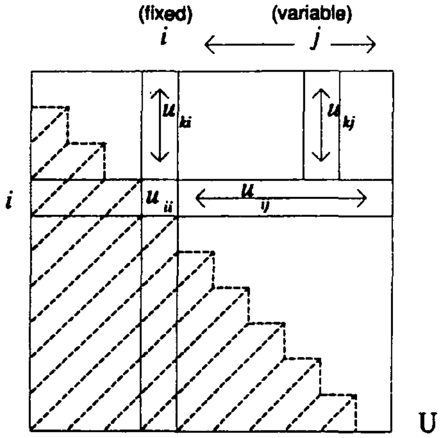


Figure 7.3: *location of terms u_{ki} and u_{kj} in U*

Therefore if, when calculating u_{ii}, the terms u_{ki}/u_{kk} are stored , they can be used for the calculation of all the other elements of the row i of U. This can be expressed in vector notation as follows:

$$\begin{aligned}\mathbf{v}_i &= \left(\frac{u_{1i}}{u_{11}}, \frac{u_{2i}}{u_{22}}, \frac{u_{3i}}{u_{33}}, \dots \frac{u_{i-1,i}}{u_{i-1,i-1}}\right) \\ \mathbf{w}_j &= (u_{1j}, u_{2j}, u_{3j}, \dots u_{i-1,j}) \\ u_{ij} &= a_{ij} - \mathbf{v}_i \cdot \mathbf{w}_j,\end{aligned} \tag{7.13}$$

where · represents the dot product of vectors. The advantages of this formulation is that $\mathbf{v}_i$ only needs to be formed once for each step, therefore saving time by reducing the number of divisions to carry out. The algorithm corresponding to equation (7.13) is therefore:

```
for $i$=1 to $n$ do
   form $\mathbf{v}_i$
   calculate the dot product $\mathbf{v}_i \cdot \mathbf{w}_i$
   for $j=i+1$ to $n$ do
      calculate the dot product $\mathbf{v}_i \cdot \mathbf{w}_j$
      $u_{ij} = a_{ij} - \mathbf{v}_i \cdot \mathbf{w}_j$
   end for
end for
```

Figure 7.4: *Algorithm for the symmetric LU decomposition*

The algorithm for the unsymmetric LU decomposition can also be expressed in terms of the vectors $\mathbf{v}_i$ and $\mathbf{w}_j$ where

$$\begin{aligned} \mathbf{v}_j &= (l_{i1}, l_{i2}, l_{i3}, \ldots l_{i,i-1}) \\ l_{ji} &= \frac{(a_{ji} - \mathbf{v}_j \cdot \mathbf{w}_i)}{u_{ii}}. \end{aligned} \tag{7.14}$$

The corresponding algorithm is given next:

```
for $i$=1 to $n$ do
   form $\mathbf{v}_i$ and $\mathbf{w}_i$
   calculate the dot product $\mathbf{v}_i \cdot \mathbf{w}_i$
   $u_{ii} = a_{ii} - \mathbf{v}_i \cdot \mathbf{w}_i$
   for $j=i+1$ to $n$ do
      calculate the dot product $\mathbf{v}_i \cdot \mathbf{w}_j$ and $\mathbf{v}_j \cdot \mathbf{w}_i$
      $u_{ij} = a_{ij} - \mathbf{v}_i \cdot \mathbf{w}_j$
      $u_{ji} = (a_{ji} - \mathbf{v}_j \cdot \mathbf{w}_i)/u_{ii}$
   end for
end for
```

Figure 7.5: *Algorithm for the unsymmetric LU decomposition (2)*

Obviously, in this case no arithmetic operations are avoided but the fact of copying the ith column of U and the ith row of L into respectively $\mathbf{v}_i$ and $\mathbf{w}_i$ means that they are locally available which can save time on access to memory

7.2.2 Forward and backward substitutions

When the matrix A is decomposed in its LU form, the equation (7.1) which describes the system of linear equation can be reformulated as follows:

$$LUx = b, \tag{7.15}$$

which can be divided in two separate calculations as shown below:

$$Ly = b \tag{7.16a}$$

$$Ux = y. \tag{7.16b}$$

This involves solving two triangular systems of equations which correspond to forward and backward substitutions. These substitutions consist of a series of steps which each produces one of the elements of the solution using arithmetic operations and which have to be executed in a set sequence. The term forward denotes the sequence corresponding to evaluating in turn the first to the last element of the solution, whereas backward refers to the reverse order. Equation (7.16a) is therefore a forward substitution and equation (7.16b) a backward substitution.

The equations for the substitutions are the same for both symmetric and unsymmetric solvers, except that in the symmetric case the elements of L have to be evaluated in function of the elements of U as described in equation (7.11) prior to calculations.

The forward substitution is illustrated in Figure 7.6. The first few steps are given next.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 & 0 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 & 0 & 0 & 0 \\ l_{41} & l_{42} & l_{43} & 0 & 0 & 0 & 0 \\ l_{i1} & l_{i2} & l_{i3} & l_{i,i-1} & 1 & 0 & 0 \\ l_{n1} & l_{n2} & l_{n3} & l_{n,i-1} & l_{ni} & l_{n.n-1} & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_i \\ y_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_i \\ b_n \end{pmatrix}$$

Figure 7.6: *Forward substitution*

$$\begin{array}{ll}
y_1 = b_1 & \\
l_{21}y_1 + y_2 = b_2 & \Longleftrightarrow y_2 = b_2 - l_{21}y_1 \\
l_{31}y_1 + l_{32}y_2 + y_3 = b_3 & \Longleftrightarrow y_3 = b_3 - (l_{31}y_1 + l_{32}y_2) \\
l_{41}y_1 + l_{42}y_2 + l_{43}y_3 + y_4 = b_4 & \Longleftrightarrow y_4 = b_4 - (l_{41}y_1 + l_{42}y_2 + l_{43}y_3) \\
l_{i1}y_1 + l_{i2}y_2 + l_{i3}y_3 + \ldots l_{i,i-1}y_{i-1} + y_i = b_i & \Longleftrightarrow y_i = b_i - (l_{i1}y_1 + l_{i2}y_2 + l_{i3}y_3 + \ldots l_{i,i-1}y_{i-1}) \\
l_{n1}y_1 + l_{n2}y_2 + l_{n3}y_3 + \ldots l_{n,n-1}y_{n-1} + y_n = b_n & \Longleftrightarrow y_n = b_n - (l_{n1}y_1 + l_{n2}y_2 + l_{n3}y_3 + \ldots l_{n,n-1}y_{n-1})
\end{array} \tag{7.17}$$

The general formula is therefore:

$$y_i = b_i - \sum_{k=1}^{i-1} l_{ik}y_k. \tag{7.18}$$

The formula above is not optimal for the computer implementation as for each new y_i all the y's calculated at previous steps have to be retrieved from memory. It is more efficient to carry out the calculation 'vertically' in the matrix, which means that when a new value y_i is obtained all the other y_j for $j > i$ below y_i are updated such as:

$$y_j = y_j - l_{ji}y_i. \tag{7.19}$$

The initial value of y_j is b_j. This means that the calculation runs in n steps, where at each step i a partial value for y_j, $j > i$, is computed and y_i is obtained. The examination of the formulæ in (7.17) shows that when b_i has been used once, it is never used again in the calculation, therefore its storage space is overwritten by the y_j values. The corresponding algorithm is therefore:

```
for $j = 1$ to $n - 1$ do
  for $i = j + 1$ to $n$ do
    $b_j = b_j - l_{ji}y_i$
  end for
end for
```

Figure 7.7: *Algorithm for the forward substitution*

The algorithm implicitly assumes that b_1 remains unchanged which is equivalent to saying $y_1 = b_1$ in equation (7.17).

The equation and the algorithm are similar in the symmetric case except that l_{ji} has to be replaced by u_{ij}/u_{ii}, since the matrix L is not stored.

The backward substitution is evolved similarly. The formulæ for both symmetric and unsymmetric solvers are the same. The first few steps are given in Figure 7.8.

$$\begin{pmatrix} u_{11} & u_{12} & u_{1i} & u_{1,n-2} & u_{1,n-1} & u_{1n} \\ \\ 0 & 0 & u_{ii} & u_{i,n-2} & u_{i,n-1} & u_{in} \\ \\ 0 & 0 & 0 & u_{n-2,n-2} & u_{n-2,n-1} & u_{n-2,n} \\ 0 & 0 & 0 & 0 & u_{n-1,n-1} & u_{n-1,n} \\ 0 & 0 & 0 & 0 & 0 & u_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ \\ x_i \\ \\ x_{n-2} \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ \\ y_i \\ \\ y_{n-2} \\ y_{n-1} \\ y_n \end{pmatrix}$$

Figure 7.8: *Backward substitution*

$$\begin{aligned}
&u_{nn}x_n = y_n && \iff x_n = y_n/u_{nn} \\
&u_{n-1,n-1}x_{n-1} + u_{n-1.n}x_n = y_{n-1} && \iff x_{n-1} = (y_{n-1} - u_{n-1,n}x_n)/u_{n-1,n-1} \\
&u_{n-2,n-2}x_{n-2} + u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n = y_{n-2} && \\
& && \iff x_{n-2} = (y_{n-2} - (u_{n-2,n}x_n + u_{n-2,n-1}x_{n-1}))/u_{n-2,n-2} \\
&u_{ii}x_i + \ldots u_{i,n-2}x_{n-2} + u_{i,n-1}x_{n-1} + u_{in}x_n = y_i && \\
& && \iff x_i = (y_i - (u_{i,n}x_n + u_{i,n-1}x_{n-1} + \ldots u_{i,i+1}x_{i+1}))/u_{ii} \\
&u_{11}x_1 + \ldots u_{1,n-2}x_{n-2} + u_{1,n-1}x_{n-1} + u_{1n}x_n = y_1 && \\
& && \iff x_1 = (y_1 - (u_{1n}x_n + u_{1,n-1}x_{n-1} + \ldots u_{12}x_2))/u_{ii}
\end{aligned} \tag{7.20}$$

In the same way as for the forward substitution, the general formula can be expressed either in a 'horizontal' fashion where the values of x_i is obtained from all previous values x_j, $j > i$, or in a 'vertical' manner where partial values for all x_i are calculated as new values become available. For similar reasons to those explained for the forward substitution the vertical approach is more efficient. The corresponding algorithm is given in Figure 7.9, where the storage space for y_i is reused, avoiding the storage of x_i in a separate vector.

```
$y_n = y_n/u_{nn}$
for $j = n$ to 1 step $-1$ do
  for $i = 1$ to $j - 1$ do
    $x_i = (x_i - u_{ij}x_j)/u_{ii}$
  end for
end for
```

Figure 7.9: *Algorithm for the backward substitution*

7.2.3 Storage scheme

In order to take advantage of the particular features of the matrices arising from finite element problems, a special-purpose storage scheme has been adopted. It is known as 'profile' or 'skyline' storage and was originally introduced by Jenning[13]. It is well known and widely adopted in engineering fields, particularly for problems solved by finite element method. This form of storage is explicitly stated next.

The matrix A is stored as two separate matrices: one which stores the elements of A below the main diagonal and the other stores elements which lie on or above the main diagonal. These two matrices (which are lower and upper triangular matrices) will be referred to as the L' and U' matrices respectively. This is shown in Figure 7.10.

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ l'_{21} & 0 & 0 & 0 & 0 \\ l'_{31} & l'_{32} & 0 & 0 & 0 \\ l'_{41} & l'_{42} & l'_{43} & 0 & 0 \\ l'_{51} & l'_{52} & l'_{53} & l'_{54} & 0 \end{pmatrix} + \begin{pmatrix} u'_{11} & u'_{12} & u'_{13} & u'_{14} & u'_{15} \\ 0 & u'_{22} & u'_{23} & u'_{24} & u'_{25} \\ 0 & 0 & u'_{33} & u'_{34} & u'_{35} \\ 0 & 0 & 0 & u'_{44} & u'_{45} \\ 0 & 0 & 0 & 0 & u'_{55} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{pmatrix}$$

Figure 7.10: *Storage of A as upper and lower triangular matrices*

The system matrix A can therefore be denoted:

$$L' + U' = A. \tag{7.21}$$

The diagonal elements of U' are set equal to those of A, so this implies that each diagonal term in L' must equal zero. Therefore, we do not need to store these zeros in the computer memory. The storage for these zeros is, however, retained in order to keep the storage schemes for both L' and U' identical which unifies the procedures to access elements in each matrix. It is obvious that in the symmetric case the matrix L' is not stored at all and this problem does not arise.

From now on, we will use the notation U' and L' to represent both the above matrices and their form of storage in the computer memory. The matrices will be denoted with round brackets () and their corresponding storage with square brackets [].

The storage for each L' and U' matrices is perhaps best explained by the means of an example. Consider the upper triangular matrix as shown in Figure 7.11.

$$\begin{pmatrix} u'_{11} & u'_{12} & 0 & u'_{14} & 0 & 0 \\ 0 & u'_{22} & u'_{23} & u'_{24} & 0 & 0 \\ 0 & 0 & u'_{33} & 0 & u'_{35} & 0 \\ 0 & 0 & 0 & u'_{44} & u'_{45} & u'_{46} \\ 0 & 0 & 0 & 0 & u'_{55} & u'_{56} \\ 0 & 0 & 0 & 0 & 0 & u'_{66} \end{pmatrix}$$

Figure 7.11: *Upper triangular matrix U'*

This matrix can be stored as a single vector of consecutive columns in the form of

$$U' = [u'_{11}, u'_{12}, u'_{22}, u'_{23}, u'_{33}, u'_{14}, u'_{24}, 0, u'_{44}, u'_{35}, u'_{45}, u'_{55}, u'_{46}, u'_{56}, u'_{66}].$$

To reduce memory requirements only the elements from the first non-zero term to the diagonal term in each the column are stored. An additional piece of information is also needed to complete this storage scheme. This is the steering vector which contains the position in the vector of the diagonal terms in the matrix, and for the above example has the form

$$Su' = [1, 3, 5, 9, 12, 15],$$

where Su' stands for *Steering vector for the matrix U'.*

The lower triangular matrix is stored similarly to the U' matrix , but this time by consecutive rows .The storage for the matrix L' is shown in Figure 7.12.

$$\begin{pmatrix} l'_{11} & 0 & 0 & 0 & 0 & 0 \\ l'_{21} & l'_{22} & 0 & 0 & 0 & 0 \\ 0 & 0 & l'_{33} & 0 & 0 & 0 \\ l'_{41} & 0 & l'_{43} & l'_{44} & 0 & 0 \\ 0 & l'_{52} & l'_{53} & l'_{54} & l'_{55} & 0 \\ 0 & 0 & l'_{63} & l'_{64} & l'_{65} & l'_{66} \end{pmatrix}.$$

$$L' = [l'_{11}, l'_{21}, l'_{22}, l'_{33}, l'_{41}, 0, l'_{43}, l'_{44}, l'_{52}, l'_{53}, l'_{54}, l'_{55}, l'_{63}, l'_{64}, l'_{65}, l'_{66}]$$
$$Sl' = [1, 3, 4, 8, 12, 16].$$

Figure 7.12: *Storage of the lower triangular matrix L'*

Finally, a working example of the storage of the whole system matrix A is given next, in Figure 7.13.

$$A = \begin{pmatrix} 1. & 3. & 7. & 0. & 5. & 0. \\ 9. & 4. & 9. & 3. & 7. & 0. \\ 8. & 1. & 8. & 1. & 1. & 0. \\ 0. & 3. & 2. & 6. & 6. & 1. \\ 0. & 0. & 0. & 7. & 2. & 9. \\ 0. & 2. & 4. & 8. & 1. & 3. \end{pmatrix} = U' + L',$$

where

$$U' = [1.,3.,4.,7.,9.,8.,3.,1.,6.,5.,7.,1.,6.,2.,1.,9.,3.]$$

$$L' = [0.,9.,0.,8.,1.,0.,3.,2.,0.,7.,0.,2.,4.,8.,1.,0.]$$

$$Su' = [1,3,6,9,14,17]$$

$$Sl' = [1,3,6,9,11,16].$$

Figure 7.13: *Storage of an unsymmetric matrix A*

The above storage takes advantage of any bandedness of the matrix to reduce memory requirements. Another advantage is that the calculation for these zero elements are not carried out therefore saving on the number of operations to compute.

The examination of equations (7.8) shows that after any element of A, a_{ij} is used once, it never again appears in the equations. This enables us to reuse the storage space for L' and U' which are overwritten by L and U during the decomposition.

7.2.4 Fixing the unknowns

In engineering problems, it is sometimes convenient to be able to fix some of the unknowns x_i to a value known before starting the calculations. This corresponds to the application of boundary conditions in real problems. For example, when solving a stress problem using finite element method a set of linear equations arises where the x_i are the displacements of the nodes and the b_i are the forces. Physically some of the nodes must not be allowed to move (for example attached to a wall). The x_i for these nodes are then known and the b_i, which are the reaction forces, are unknown a priori.

A profile matrix solver containing built-in constraints was devised by Bettess and Bettess[14]. Their method of implementing the constraint facility is used here. The principle of constraining unknowns and the corresponding equations are given next.

In the equation for the solvers, $Ax = b$, all the elements of b are normally known and the elements of x are unknown. When one element of x, x_i, is assigned a value *a priori*, it is said to be fixed. When this happens, it is not necessary to perform the calculations corresponding to obtaining this x_i. As we will see in the example below, this is equivalent to ignoring the row i and column i of A, therefore not performing the step i in the calculations, and modifying the vector b. Consider the simple system composed of three linear equations as shown below:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3. \end{aligned} \tag{7.22}$$

If x_2 is fixed to be a known value v, the equation (7.22) can be restated as:

$$\begin{aligned} a_{11}x_1 + a_{13}x_3 &= b_1 - a_{12}v = b'_1 \\ a_{31}x_1 + a_{33}x_3 &= b_3 - a_{32}v = b'_3 \\ a_{21}x_1 + a_{22}v + a_{23}x_3 &= b_2, \end{aligned} \tag{7.23}$$

where x_1, x_3 and b_2 are the unknowns and b'_1, b'_3 and v are known. When solving the system for the unknowns x_1 and x_3 the third equation can be ignored. The system is therefore reduced to two equations with a modified right hand side vector $\mathbf{b}'$.

The generalisation to a $n \times n$ system with p fixed x_i is straightforward. The columns and rows corresponding to the fixed values are removed from the matrix A, which is achieved by retaining their storage and ignoring them in the algorithms, and the vector b is altered according to the following equation:

$$b_i = b_i - \sum_{k \in \Omega} a_{ik}x_k, \tag{7.24}$$

where the x_k are the fixed values and Ω is a series of length p containing the numbers of the fixed unknowns.

The implementation of this technique is carried out through the use of a vector **fix** of dimension n. It is a vector whose elements point to the elements of x which are fixed. The corresponding equations are:

$$\begin{aligned} \mathbf{fix}(i) &= 0 \qquad \text{if } x_i \text{ is free} \\ \mathbf{fix}(i) &= 1 \qquad \text{if } x_i \text{ is fixed.} \end{aligned} \tag{7.25}$$

The algorithm which performs the modification of b uses this vector. It works differently from equation (7.24) as for each fixed value x_k all the right hand sides b_i are modified. The values of the b_i corresponding to a fixed x_i are left unchanged. The algorithm is given below:

```
for $k = 1$ to $n$ do
  if fix($k$) equals 1
    for $i = 1$ to $n$ do
      if fix($i$) equals 0
        $b_i = b_i - a_{ik}x_k$
      end if
    end for
  end if
end for
```

Figure 7.14: *Algorithm for the modification of the right hand side*

The algorithms previously given for the LU decomposition, the forward and backward substitutions have also to be modified to take into account fixed values. They are given below:

```
for $i = 1$ to $n$ do
  if fix($i$) equals 0
    calculate $u_{ii}$
    for $j = i + 1$ to $n$ do
      if fix($j$) equals 0
        calculate $u_{ij}$
        calculate $l_{ji}$
      end if
    end for
  end if
end for
```

a: Unsymmetric LU decomposition

```
for $j = 1$ to $n - 1$ do
  if fix($j$) equals 0
    for $i = j + 1$ to $n$ do
      if fix($i$) equals 0
        $b_j = b_j - l_{ji}y_i$
      end if
    end for
  end if
end for
```

b: Forward substitution

```
  $y_n = y_n/u_{nn}$
  for $j = n$ to 1 step $-1$ do
    if fix($j$) equals 0
      for $i = 1$ to $j - 1$ do
        if fix($i$) equals 0
      $x_i = (x_i - u_{ij}x_j)/u_{ii}$
    end if
  end for
end if
  end for
```

c: Backward substitution

Figure 7.15: *Serial algorithms with constraints*

The computer programs implemented for this serial approach are a direct coding of the algorithms and storage scheme presented previously. They have been developed in standard FORTRAN F77. This version of the solvers is subsequently used for comparison with the parallel version and for performance evaluation.

7.3 Algorithms for the parallel solution

The serial approach described in the previous section can be parallelised in different ways. Before explaining which technique was used in this work, an review of the various methods used in parallelising the solvers is given which is mainly based on the work done by Geist and Romine[6]. They investigated the parallelisation of the LU decomposition including the technique of pivoting and its parallel implementation. A quick introduction to the pivoting scheme is given next, followed by a review of Geist and Romine work.

7.3.1 Survey

The direct class of solvers[10] is based on the Gauss elimination technique where the equations of the system are linearly combined and exchanged in order to obtain an upper triangular system by successive elimination of the variables between the equations which can then be solved with a backward substitution. An alternative method, the Gauss-Jordan method, is to incorporate the backward substitution in the process so that a diagonal system is obtained for which the solution is immediate.

A problem arising in this class of solvers is that some pivots, which are the coefficient in front of the variable to be eliminated might be zero. This happens when either the matrix is singular, in which case no unique solution exists, or when the order of the equations is unsuitable. In the latter case, the pivoting technique is used to resolve the zero pivot and also to improve the accuracy[6].

Various types of pivoting scheme exist. The partial pivoting corresponds to the case when only rows are exchanged. One approach, the maximal column pivoting, is to find the maximum value of the elements of the column below the pivot and use that value as the pivot. The scaled-column pivoting follows a similar method but before finding the maximum pivot all rows are scaled with the largest coefficient of that row. The total pivoting, on the other hand, implies pivoting both rows and columns. At each step, the maximum value of all rows and columns below the pivot is found and used as the pivot by bringing it in place through column and row pivoting.

When the LU decomposition method is used to solve the system, normally only partial pivoting of the type maximal column pivoting is used as it is the only one easily incorporated in the LU decomposition. The parallel algorithms investigated by Geist and Romine concern the LU decomposition with maximal column pivoting. They have considered two schemes: distribution of the matrix by rows and distribution by column.

In the distribution by rows, at each step finding the pivot involves communication among all processors because the pivot column is scattered among the processors. The way communication has been carried out by the authors is related to the particular machine they used, the Intel iPSC, for which a tree configuration is efficiently implemented.

Each leaf node of the tree calculates its local maximum and passes it up to the parent node which compares it with its local maximum and passes up the largest of the two. When the pivot is found, the processor which stores the pivot acts as the leading processor, therefore avoiding the communication needed in exchanging the rows. This is what the authors call 'dynamic pivoting'.

The rows of the matrix are initially distributed in an arbitrary order among the processors. For the first p rows of the matrix, p being the number of processors, the order of the processors which contain the pivot is different for each system solved and is determined dynamically.

A process can only contain one of the first p pivots, therefore when more than one pivot is stored in the process, actual communication to exchange the rows has to take place. The row is exchanged with a process which does not already contain a pivot from the first p rows. There might be more than one process in this situation so the choice is made by selecting the nearest neighbour which does not already store a pivot. Depending on the system of equations and its order, pivots might be stored on p different processors, which is the best case when no communication is needed, or all pivots might be stored in only one process, which is the worst case when $(p - 1)$ communications have to take place.

In practice, the initial distribution of the rows is even so that a distribution between those two extreme situations is generated. The algorithm produces a mapping of rows to processors which is unique and different for each system. This procedure described for the first p rows is carried out again for the next packet of p rows and so on.

In the distribution by columns, the pivoting is much easier as the whole pivot column is stored on one processor only. The initial distribution of the columns is in a wrapped fashion therefore fixes the mapping of columns processors and is optimal. The problem in this case is the bottleneck introduced by the serial calculation of the pivot, where the determination of the maximum value and calculation of the portion of the matrix attributed to that processor are carried out while the other processors are idle waiting for the pivot. A solution, which the authors call 'pipelining', consists of sending the pivot column to all the other processors before calculation of the rest of the matrix carries on for that step.

In the serial algorithms described in the previous section, no mention has been made of pivoting. This is mainly due to the particular type of matrices found in finite element problems. Often these matrices are positive definite, which means that no diagonal term is zero, therefore the LU decomposition always works properly. Even when the matrices are not positive definite, they are thought of being well-behaved, that is to say diagonally dominant, therefore most finite element system solvers do not implement pivoting.

Another reason is related to the storage scheme used, the profile storage, which makes pivoting harder to implement. The solvers developed in this chapter do not therefore contain any pivoting scheme. Nevertheless, it might be that for some non-linear problems the matrices encountered are not so well-behaved and that pivoting may be necessary for improving the accuracy.

The method followed for the parallel implementation is based on the column-wise distribution of the matrix as described by Geist and Romine. Therefore, it would be possible to implement a pivoting scheme without modifying the communication aspect of the program, although the calculations and length of the messages sent would have to be altered.

The remainder of this section is devoted to the description of the parallel solvers for both symmetric and unsymmetric matrices, with first an explanation of the distribution of the data on the processors.

A word on terminology is necessary at this stage. A processor corresponds to a physical chip while a process denotes a software piece of work. Although the storage of data is physically done in the memory of the processors, expressions like 'storage of data on a process' will be used. This is because the processes composing a program are unique whereas their mapping onto the processors can vary. Therefore data which can only be accessed by one process might physically be adjacent in the memory of the processor to the private data allocated to another process, in the case when both processes are mapped onto the same processor.

7.3.2 Parallelisation of the solvers

The parallelisation of the solvers is based on the identification within the serial algorithms of the operations which can be executed independently. Within a series of calculations, when each calculations is independent from one another, each piece can be computed by a different processor concurrently, and the speed-up obtained can ideally be a linear function of the number of processors used.

The independent operations in the solvers considered here are the calculation of the elements of the ith row of U and ith column of L. This means that the parallel algorithms will be composed of n steps, for a $n \times n$ matrix, where at each step a series of calculations is carried out concurrently. Because the results obtained at each step are used in the following step, a global synchronisation of all processors after each step has to take place so that one process does not start a new iteration using the wrong data.

The independence of the calculations at each step can be seen from equation (7.8) which is recalled below:

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj} \qquad j \geq i \qquad (recall\ 7.8a)$$

$$l_{ji} = (a_{ji} - \sum_{k=1}^{i-1} l_{jk}u_{ki})/u_{jj} \qquad j > i. \qquad (recall\ 7.8b)$$

At step i, the elements u_{ij} and l_{ji}, $j > i$, are evaluated after u_{ii} is computed. Therefore, if u_{ii} is first calculated all u_{ij} and l_{ji} can be calculated independently as none of them require values from the other for that step since the index k varies from 1 to $i-1$. None of the l elements from step i are used in the formula (7.8a) for u and none of the u elements from step i are used in the formula (7.8b) for l. The only values used in the formula are those which have been obtained in the previous steps.

A first specification of the full parallel algorithms is to take the serial algorithms given in the previous section and insert the generic term 'in parallel' to show the independent operations. Very often, independent operations happen within a loop, where each step in the loop can be carried out at the same time as another step in the loop. This is the easiest way of identifying parallelism in serial program which is used by compilers for automatic production fo parallel code. The independent operations for the solvers are located in the inner loop which is executed in parallel. This is shown next:

```
for $i = 1$ to $n$ do
  calculate $u_{ii}$
  in parallel
    for $j = i + 1$ to $n$ do
      calculate $u_{ij}$
      calculate $l_{ji}$
    end for
  end parallel
end for
```

Figure 7.16: *First unsymmetric LU decomposition parallel algorithm*

A similar algorithm stands for the symmetric LU decomposition. The forward and backward substitutionş algorithms are parallelised in the same way, as shown next:

```
for $j = 1$ to $n - 1$ do
  in parallel
    for $i = j + 1$ to $n$ do
      $b_j = b_j - l_{ji}y_i$
    end for
  end parallel
end for
```

a: Forward substitution

```
$y_n = y_n/u_{nn}$
  for $j = n$ to 1 step $-1$ do
    in parallel
      for $i = 1$ to $j - 1$ do
      $x_i = (x_i - u_{ij}x_j)/u_{ii}$
    end for
  end parallel
end for
```

b: Backward substitution

Figure 7.17: *First substitution parallel algorithms*

The parallel algorithms as shown above implicitly assume the free access to all data and ignore any memory contingency. There are two ways in which these algorithms can be modified to account for these problems. One is to assume shared memory environment and the other is to suppose distributed memory environment. Therefore, from now the algorithms for the two implementation will diverge.

Since the work undertaken in this chapter is concerned with the implementation in a distributed memory context, this will be described next in detail. The shared memory implementation is underlined at the end of this section. From now on, the word 'process' will be used in relation to memory storage as explained previously.

In the distributed memory configuration, each process has its private memory which cannot be directly accessed by any other processes. The exchange of data between the processes is achieved through a mechanism of communication where one process has to submit its request for data to the process storing it in its private memory which in turn fetches the data and sends it to the requesting process.

A crude way to implement the parallel solvers in a distributed communication environment would be to store the whole matrix A and the vector b in the private memory of each process to give full access to all data. Nevertheless, after the first step the process which holds the up-to-date values of L and U for that step still needs to communicate these new values to all the other processes so they hold the correct version of the matrices.

An efficient way of implementing the parallel algorithms is to distribute data so that the matrix A and the vector b are evenly spread across the processes. Attention must be paid to load balancing issue when choosing the distribution scheme. Load balancing means that one process should not have too much work while others are idle. This corresponds in our case to the fact that all processes should have at each step roughly the same number of elements of L and U to calculate.

A well-known method is the 'wrapping scheme' also used by Geist and Romine and Farhat and Wilson. For example, in the symmetric case, this corresponds to the distribution where each process stored every p-alternate column of U, where p is the total number of processes. In the unsymmetric case, U is wrapped onto the processes by columns and L by rows. This is shown in Figure 7.18.

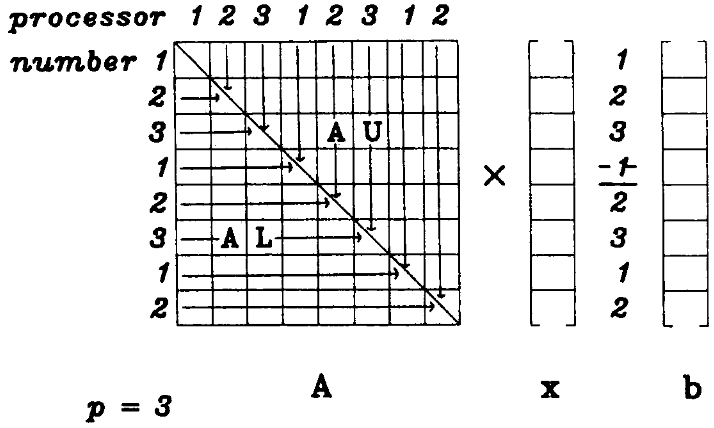


Figure 7.18: *Wrapped distribution scheme*

The full parallel algorithms for the LU solvers are derived next in the unsymmetric case. When the data is distributed each process needs to know whether it stores the pivot row and column. The pivot row and column, which are denoted 'active row' and 'active column' following the terminology of Farhat and Wilson, are the row i of L and the column i of U for the iteration i. This is shown in Figure 7.19.

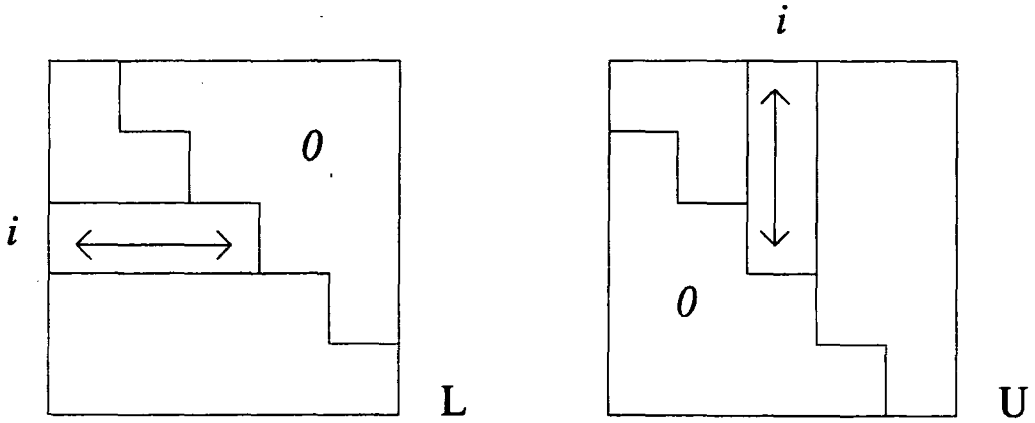


Figure 7.19: *Elements involved at step i*

The process which stores the active column and row can therefore evaluate the diagonal term u_{ii} since it only needs the element of this row and column to perform the calculation. Obtaining u_{ii} is a prerequisite for all the other processes to calculate their share of the row i of U and the column i of L. Therefore, the process which calculates u_{ii}, denoted the 'active process', has to send it to all the other processes together with the active row and column also needed by the other processes. This can be seen in equation (7.8) where l_{ik} and u_{ki} are the elements of the active row and column. This is illustrated in Figure 7.19.

One interesting aspect of this algorithm is that the forward substitution can be performed at the same time as the LU decomposition. Indeed, referring back to equation (7.18) recalled next

$$y_i = b_i - \sum_{k=1}^{i-1} l_{ik} y_k, \qquad (\textit{recall}\ 7.18)$$

the terms of L involved, l_{ji}, are those being calculated during one step in the LU decomposition since l_{ji} is the ith active column of L shown in Figure 7.19. Therefore, when a process has obtained one element of column i of L, this element can straight away be used to compute a new value of y_i. This assumes that each process has access to all up-to-date values of y_i which means that these values have to be communicated between the processes, which can be done at the same time as the communication of the active row and column.

The combined algorithm for the LU decomposition and forward substitution is given below:

```
for $i = 1$ to $ncol$ do
  calculate $l_{i1} = u_{i1}/u_{11}$
  calculate $y_i = y_i - l_{i1}y_1$
end for
for $i = 2$ to $n$ do
  active process = process storing column $i$ and row $i$
  if this process is active process then
    Calculate $u_{ii}$
    send column $i$, row $i$ and $y_i$ to all other processes
  else
    receive and store column $i$, row $i$ and $y_i$
  end if
  for $j$ = start to $ncol$ do
    calculate $u_{ij}$
    calculate $l_{ji}$
    calculate $y_j = y_j - y_i l_{ji}$
  end for
end for
```

Figure 7.20: *Pseudo code for the LU decomposition and forward substitution*

Since the forward substitution cannot be performed independently from the LU decomposition, only systems with a single right hand side can be solved. If systems with multiple right hand sides had to be solved, a modification of the algorithm would be necessary so as to separate the forward substitution from the LU decomposition. This is easy to do as the structure of the algorithm for the LU decomposition and the forward substitution are similar. The communication scheme would be unchanged and only the calculation corresponding to the LU decomposition would have to be removed.

The main problem with the implementation of a parallel backward substitution lies in the distribution of data scheme adopted where L is distributed by rows and U by columns. This means that the parallel algorithm shown in Figure 7.17 cannot be directly implemented because the operations shown as being executed in parallel would be executed serially by one process. Indeed, the calculations carried out in parallel correspond to the evaluation of the terms $u_{ij}x_j$ where j is fixed and i varies, therefore involving the column j of U which is stored on one process only.

It would be possible to implement the backward substitution in parallel as it stands but the amount of communication compared to the calculation would impose too much overheads. At each step in the backward substitution the whole column j of U and the term x_j would have to be distributed to all the other processors as shown in Figure 7.21.

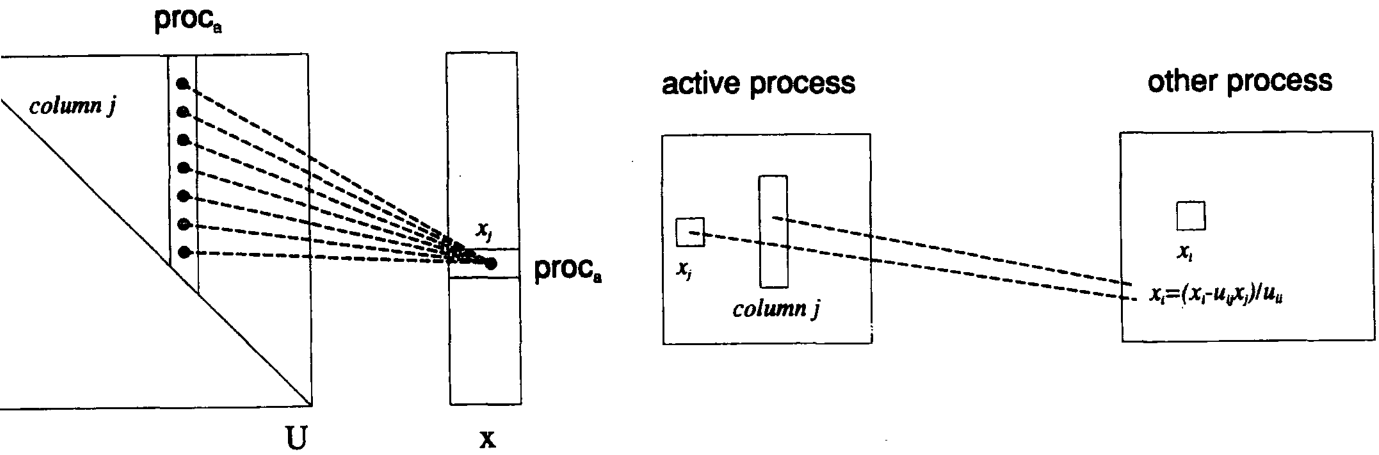


Figure 7.21: *parallel backward substitution (1)*

A more efficient way of parallelising the backward substitution is to design a scheme which runs across the columns of U rather than across the rows of U as explained before. This corresponds to a direct parallelisation of the original formula in equation (7.20) where all the products $u_{ij}x_j$ for i fixed and j variable are executed concurrently and the results are accumulated by the active process which also carries out the division.

This scheme works at its maximum efficiency only when step p is reached, where p is the total number of processes. In the iterations before step p, there are

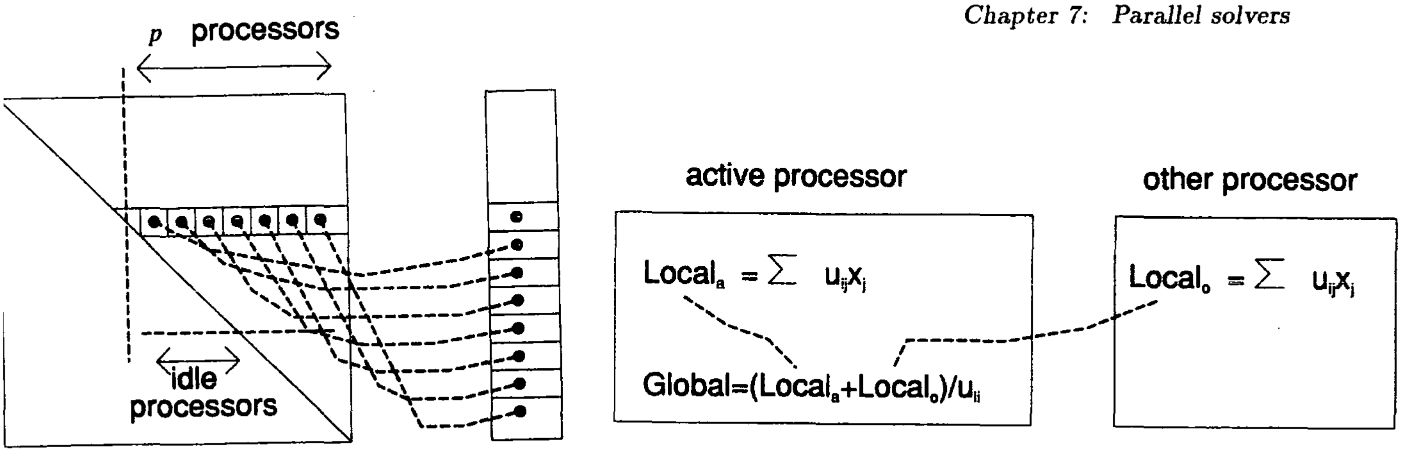

p processors
idle processors
active processor
Local$_a$ = Σ u$_{ij}$x$_j$
Global=(Local$_a$+Local$_o$)/u$_{ii}$
other processor
Local$_o$ = Σ u$_{ij}$x$_j$

Figure 7.22: *parallel backward substitution (2)*

less products to be carried out than processes, which means that some processes will be idle during that time. This is shown in Figure 7.22.

The parallel algorithm is given next:

```
calculate $y_n = y_n/u_{nn}$
for $i = n - 1$ to 1 do
  active process = process storing element $i$ of $y$
  if this process is active process then
    Calculate products $u_{ij}x_j$ from own data and accumulate
    receive products from other processes and add to own
    $y_i = (y_i + \text{products})/u_{ii}$
  else
    calculate products $u_{ij}x_j$ from own data and accumulate
    send to active process total products
  end if
end for
```

Figure 7.23: *Parallel algorithm for the backward substitution*

In the algorithms presented previously, the keywords **send** and **receive** have been used to express the communication between the processes. The actual implementation of these functions is machine dependent and will be discussed in the next section. The remaining this section concentrates on briefly describing the shared memory implementation of the solvers.

7.3.3 Shared memory implementation

The shared memory multiprocessor computer used is the Encore Multimax containing 14 processors. The chip for this machine is the NS32532 with a clock speed of 30 MHz. For 14 processors the Encore has an integer rating of 8.5 Vax Mips, for floating point operations it achieves a sustained 0.3 Mflops. All processors are connected to the same memory via a high speed bus, known as the Nanobus, which has a rate of data transfer of 100 Mbytes/sec. The particular Multimax used in this work has 96 Mbytes of real memory.

The main aspect of the Encore Parallel FORTRAN exploited here is the parallel **do** loop. The parallel **do** loop executes iterations of the loop in parallel. It can be seen from the algorithms given in Figures 7.17 and 7.18 that this is exactly what we require to execute the algorithms in parallel. The program for the Encore machine looks very much like the algorithms (Figure 7.17 and 7.18) where the inner loops statements **for** $i=$ ... **to** *neq* **do** are replaced by **doall** (i=1:*neq*) and the **end** by **end doall**.

Each iteration of the parallel **do** loop is then executed in parallel. It means that if, for example, the program is run on three processors, the first three iterations of the loop will be executed concurrently by processors 1, 2 and 3. Then, as soon as one of the processors has finished its work it is assigned to execute the next iteration. This is very much a 'processor farm' type configuration where the iterations are feed to whichever processor has finished its work first.

The implementation of the parallel algorithms based on executing the body of the inner loops in parallel is therefore straightforward on the Encore machine. Practically, the program can be first developed and debugged in a serial form as far as possible and when it is working the **do** loops are replaced by the parallel **do** loops and all the other alterations to the serial code are straight forward.

7.4 Communication schemes

The aim of this section is to outline the communication schemes adopted for each distributed memory machine used in this work and to explain the problems encountered and the solutions devised. Although three different machines were used, a Transtech board in a PC, a Meiko Computing Surface and the EPCC Computing Surfaces, only two different schemes are needed as both the Meiko and the EPCC Computing Surface are fully compatible and the software can be readily ported between the two machines.

The communication model has been implemented with the 3L FORTRAN and the CS Tools library of utilities. As briefly mentioned in the previous chapter, the 3L FORTRAN provides very basic utilities for message passing implementation whereas CS Tools enables a higher level and more abstract development of communication schemes.

In order to fully describe the communication model used, it is necessary to explain in more detail the structure of the program.

The structure for both symmetric and unsymmetric solvers is identical. Only the actual calculation carried out changes. The programs are composed of three types of processes, all running in parallel. A 'main' process is in charge of generating the matrix A and vector b, calling the relevant solver, distributing the data across the network and collecting the results. This is basically a test program whose functionalities will be described in the next section. A series of 'slave' processes is assigned with the task of performing the LU decomposition, the forward and backward substitutions. Finally, a series of 'communication' processes carries out the message passing needs of the program.

When using the 3L library the communication processes have to be written by hand whereas in the case of CS Tools these processes are supplied as part of the library. The details in both cases are given next.

The Transtech board used only contains three Transputers, therefore a small communication harness has been written which is not extendable to more than three processes. Nevertheless, the program has been written keeping in mind that more processes might become available later on. It is organised in modules where the communication is separate from the calculation and can be removed and replaced by new modules without affecting the overall structure of the program.

The reason for developing a communication harness limited to three Transputers is that there has been a lot of work done on these harnesses and it was felt unnecessary to duplicate work already done when an efficient commercial package could easily be used.

The communication in the program is composed of two parts: the initial distribution of the data to all the processes and the communication between processes at each iteration. Before the calculation can start, each slave process has to receive its part of the matrix A and vector b from the main process. The communication involved here can be seen in Figure 7.24.

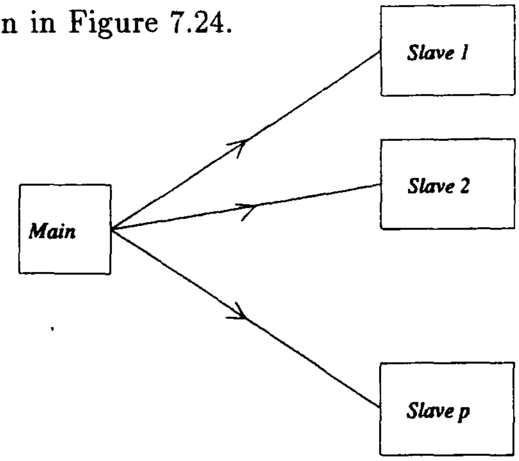


Figure 7.24: *Initial distribution of the data*

It is a succession of 'one to one' process communication, which means that a message is sent from one process to another independently. When a process has received its share of the data, it can start calculating. If it is process number one, it has to evaluate the first diagonal term and sent it to all the other processes together with the first active row and column. Then, at step two, process number two becomes active and initiates the communication, and so on. The communication involved here is a broadcast for which the active process changes at each step, as illustrated in Figure 7.25. This is due to the wrapping method used to initially distribute the data across the processes.

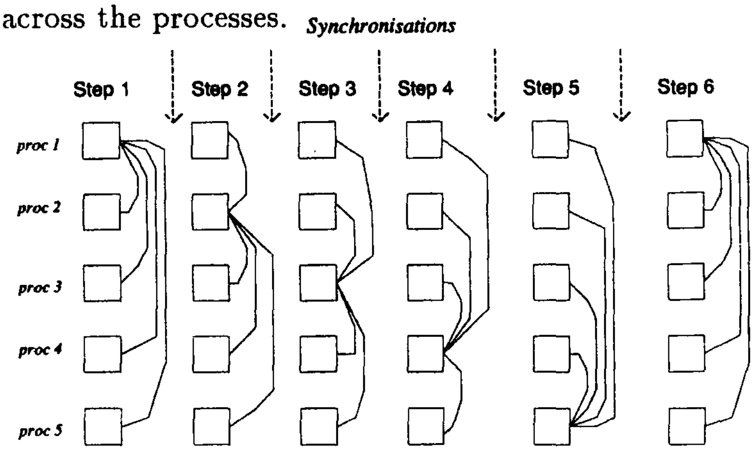


Figure 7.25: *Broadcast communication*

One important feature indicated in Figure 7.25 is the synchronisation between each broadcast. This corresponds to the main problem which has been experienced when implementing the communication algorithms. It is essential to prevent any process from starting the calculation for the next iteration before the broadcast for the current iteration is completed. If this is not enforced, messages can get out of synchronisation, which is a problem that has been experienced several times during the development of the programs.

7.4.1 Implementation with the 3L library

The synchronisation for this case has been achieved using a selective mechanism for inputting messages into processes. This means that a process is allowed to start the next iteration as soon as it has received the message for the current iteration but it cannot receive a message from the wrong process. This is illustrated in Figure 7.26:

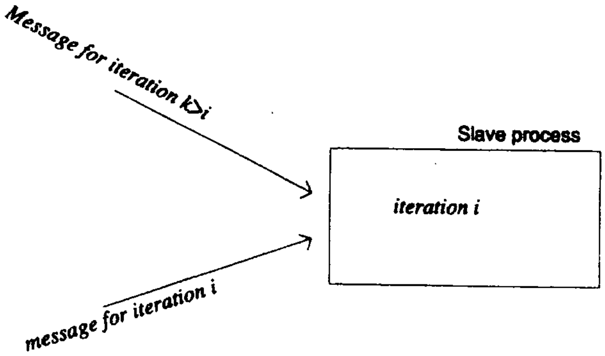


Figure 7.26: *Selective input of messages*

In Figure 7.26, if the message for iteration k $(k > i)$ arrives before the message for iteration i, it will be blocked until the message for iteration i has arrived. If $k > i + 1$, that message will remain blocked until iteration k in the calculation is reached.

The implementation of this scheme for three Transputers requires a knowledge of the connections between the processes. The three slave processes are fully connected together and the main process is only connected to the first slave as shown in Figure 7.27.

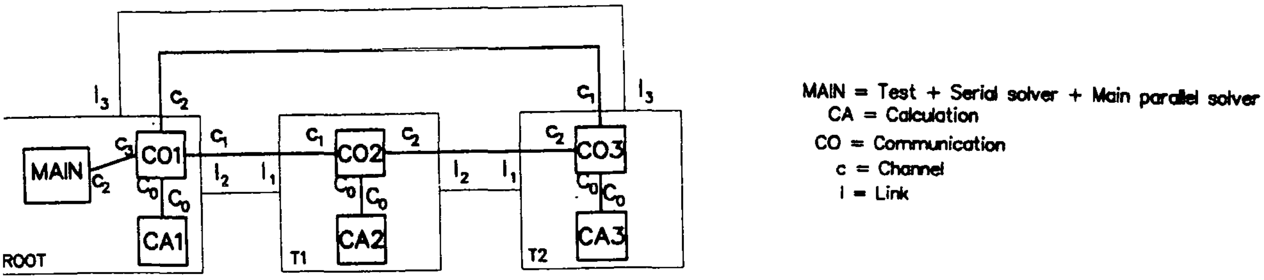


Figure 7.27: *Connections between processes*

The main process is not fully connected to all the slaves because a Transputer only has four links. Since, the main process has to be placed on the only Transputer which has access to the input/output facilities (screen , keyboard, files) for which two links are reserved for system activities, it only leaves two links free which is not enough for a full connection. The other two Transputers have all four links available to the user, but no input/output can directly be performed from them.

The communication processes run in parallel with the slave processes as shown in Figure 7.27. They are given the knowledge of which slave process they are associated with and which step in the calculations is currently processed. Knowing the number of the associated slave process, the order in which the messages should arrive to a communication process can be determined in advance. This is shown in Figure 7.28.

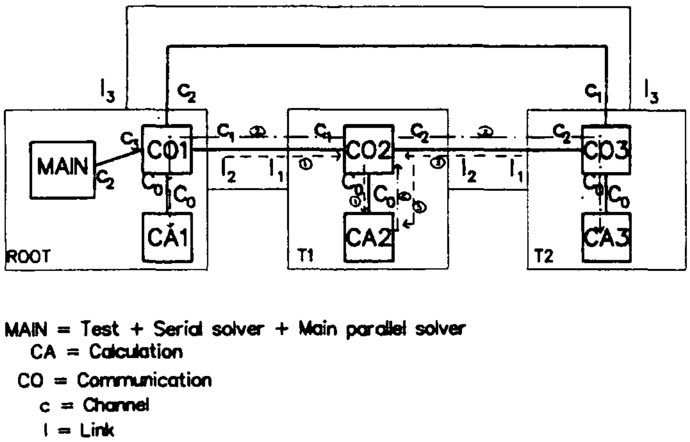


Figure 7.28: *Order of arrival and destination of messages*

This scheme ensures that the correct message arrives for each iteration. This is equivalent to the synchronisation shown in Figure 7.25. Depending on from where the message has arrived, the communication process is able to broadcast the message to the relevant processes. This is indicated by the dashed arrows for process two in Figure 7.28

The initial distribution of the data from the main process to the slave processes has been routed via the first communication process since the main process does not have access to all the other processes. This is shown in Figure 7.29.

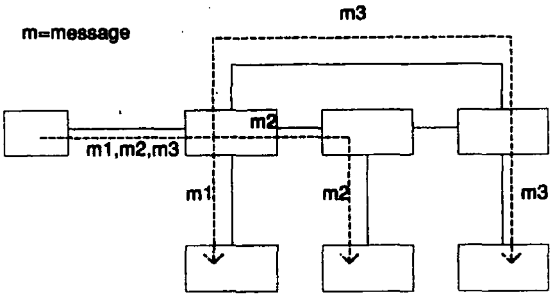


Figure 7.29: *Routage for the initial messages*

Finally, the full algorithms for the three communication processes, denoted *C1, C2 and C3* are given. *M* denotes the main process and *S1, S2 and S3* denote the slave processes.

Main	*C1*	*C2*	*C3*

sends message	**receive** message
	retrieve the number of the destination process
	if destination process **equals** 1
	  send to *S1*
	else if destination process **equals** 2
	  send to *C2*
	else
	  send to *C3*
	end if
		receive message
		send to *S2*
			receive message
			send to *S3*

a: initial distribution of system matrix A

(communication process number)	*C1*	*C2*	*C3*
For $i = 2$ **to** n **do**			
If i **equals** 2, 5, 8 ...			
Receive message from	*S1*	*C1*	*C1*
Send to	*C2 and C3*	*S2*	*S3*
else if i **equals** 3, 6, 9 ...			
Receive message from	*C2*	*S2*	*C2*
Send to	*S1*	*C1 and C3*	*S3*
else if i **equals** 4, 7, 10 ...			
Receive message from	*C3*	*C3*	*S3*
Send to	*S1*	*C2*	*C1 and C2*
end if			
	Send to main		
End for			

b: Broadcast at each step

Figure 7.30: *Algorithms for the communication with the 3L library*

The broadcast algorithm given previously shows the different behaviours depending on the number of the communication process. The '**send** to main' instruction executed by *C1* at the end corresponds to the fact that the results are collected by the main process at each step in the calculations rather than at the end of the calculations. This is a general feature of the implementation also appearing in the CS Tools version.

7.4.2 Implementation with CS Tools

When using CS Tools, the communication processes are provided by the library and are valid for an arbitrary number of processes. Nevertheless, CS Tools does not provide a broadcast function which has to be programmed specifically. The control over the way the communication was carried out using 3L FORTRAN is lost when using CS Tools therefore a new mechanism for synchronisation has been devised.

A process is attributed the task of synchronising all the other processes. At each step, the active process sends its message to the synchronisation process and then carries on with its own task. The synchronisation process distributes the message to all the other processes in ascending process number order, first to process one and last to process p (p processes in total). This is illustrated in Figure 7.31.

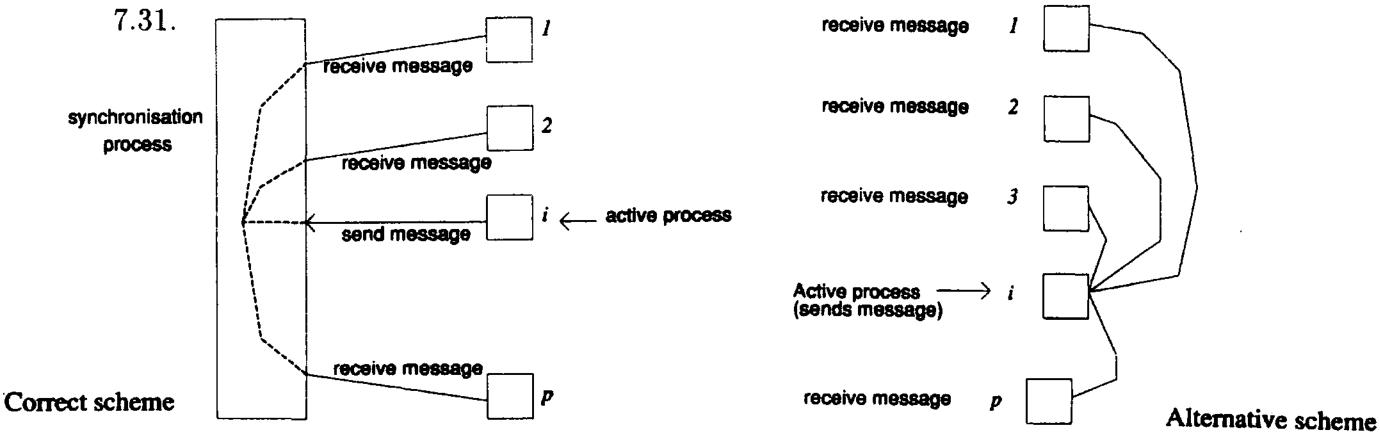


Figure 7.31: *Synchronisation with CS Tools*

The alternative scheme shown in Figure 7.31 was first attempted but proved to introduce synchronisation problems as two messages could arrive together in a process and their acceptance would then be arbitrary if their arrival was simultaneous.

The basic difference between the two schemes is that the active process in the correct scheme is not blocked by any other process except the synchronisation process. This means that if another process is slower because it has more work to do, it does not affect the active process. In the second scheme the active process can be blocked by any process therefore permitting another process to send a message before the active process has finished sending its message, inducing loss of synchronisation.

The synchronisation process should be dedicated to its task and should not perform any calculations. The main process is chosen as the synchronisation process because during the calculation it is idle, only collecting the results as they arrive. This process is, however, busy before the start of the calculations when it has to distribute the data among the processes.

7.5 Performance evaluation

This section concentrates on defining the various ways of assessing the efficiency of a parallel program and applies it to the evaluation of the performance of the solvers.

The evaluation of parallel programs is more difficult than for serial programs because several quantities can be measured. Moreover, the process of evaluation itself can influence the behaviour of a parallel program. This is caused by the time dependency of parallel programs which can behave in different ways when diagnostic writes are added because the scheduling of the program is then altered.

Various authors have discussed the problem of parallel system evaluation and we refer to reader to the work done by Filho[15] who discusses in detail this topic and other topics concerning the use of parallel solvers in the finite element method.

An interesting new experimental system for evaluation of parallel computing systems has recently appeared[16]. PAWS[†] provides an interactive user-friendly environment for analysis of existing, prototype and conceptual machine architectures running a common application. It is based on the transformation of high level source language into a single data dependency graph, following similar ideas to those found in data driven languages. This graph can then be mapped onto real or conceptual machines. The authors claim that it enables the comparison of existing machines as well as the evaluation of machines before building them.

7.5.1 Definitions

One of the basic concept in analysing parallel software is that of obtaining speed-up factors. This is the ratio of the time it takes for the program to run on one processor (T_1) to the time it takes to run on p processors (T_p). The efficiency can then be derived by dividing the speed-up factor by the total number of processors p:

$$\text{speedup} = \frac{T_1}{T_p} \qquad \text{Efficiency} = \frac{T_1}{T_p} * \frac{1}{p}. \tag{7.26}$$

This gives an indication of how efficiently the processors are being used. This efficiency can ideally be up to 100% ($T_p = T_1/p$) but in reality factors like communication and the proportion of the program which cannot be parallelised imply

[†] Parallel Assessment Window System

the following upper bound limit to the efficiency[15]:

$$\text{Efficiency} \le \frac{t_s + m \times t_p}{(t_p + t_{oh}) \times \beta + t_s} \times \frac{1}{p}, \tag{7.27}$$

where

p is the total number of processors
m is the total number of processes
t_s is the time to run the serial part (non-parallelisable part)
t_p is the time to run the parallel part
t_{oh} is the time introduced by the communication overhead
$\beta = \text{int}\ (m/p)$ is the grain size (int is the nearest whole integer)

In some respects this definition is unrealistic as it would be pointless to construct a parallel program with ensuing overhead just to be run on one processor as a serial program.

This bring us to define another definition of efficiency, which will be denoted *Eff*$_2$. The efficiency obtained previously will be called *Eff*$_1$. The new efficiency consists of comparing the time taken by the most efficient serial method to the time taken by the parallel program. It is run on one processor and the time is denoted T_{serial}. The new efficiency is then:

$$\mathit{Eff}_2 = \frac{T_{serial}}{T_p} * \frac{1}{p}. \tag{7.28}$$

In the case of the solvers there are two possible definitions for T_p. A parallel solver can be called from within a serial code or from within a parallel code. In the first case the user does not want to know about parallelism and the solver is just seen as being faster. In the second case, the users are aware of parallelism and is ready to incorporate in their own parallel program any data distribution or communication necessary for the correct use of the solver.

This distinction between the two parallel solvers only applies in the case of implementation on distributed memory machines. The difference between the two comes from the data distribution which is different in each case. When the solver is used as part of a serial program the matrix A and the vector b have to be initially distributed among the processes and at the end the results must be brought back to the process where the serial program runs. In the second case, when the solver is run as part of a parallel program, it is assumed that the data is already distributed among the processes and that it is available at the beginning of the calculations.

When the solver is called from a serial program it is possible to leave the backward substitution serial as the whole matrix A is stored on one processor. This has been done on the recommendations of Farhat and Wilson[8] who indicate that due to the data structure used the backward substitution is not worth parallelising. When the solver is called from a parallel program the backward substitution had to be parallelised since the matrix A is not available as a whole for execution of a serial backward substitution.

For distributed memory machines, where the communication between processes takes time, T_p in the first case takes into account the time for distributing the data before the calculations start whereas in the second case it is not included. This distinction does not exist on shared memory machines as the data is always available for all processes at any time, therefore avoiding the need for communication.

The code for both types of solvers, called within serial and parallel programs, has been developed on the distributed memory machine. The parallel execution time in the serial case is denoted T'_p and in the parallel case T_p. A series of tests have been run to evaluate the various efficiencies.

The tests consist of generating a matrix A and a solution vector x, obtaining the vector b as $b = Ax$, solving for A and b and comparing the solution to the initial x. Two types of matrices have been considered: densed and banded. The former gives an optimal efficiency, as at each step in the algorithm as much calculation as possible is carried out, whereas the latter gives a more realistic efficiency, as this is the type of problem which may arise in practical engineering applications. The actual values of the elements of A and x are randomly generated. For these tests, no unknown element of b has been fixed.

7.5.2 Description of the tests

The tests have been carried out on dense matrices mainly. The tests are numerous as all three efficiencies have to be evaluated. Some of the tests have been restricted because of the limited amount of memory available for each processor. When running fully parallelised code the data is distributed across the memory of all the Transputers. On the machine used, each Transputer has a minimum of 4Mbytes of memory which means that the total memory available is quite large and it increases as processors are added.

As explained before the evaluation of the efficiency involves obtaining the time to solve the problem for a serial version and a parallel version running on one processor. The maximum memory available with any one Transputer on the machine used is 16Mbytes of memory which means that double precision matrices larger than 1300×1300 can not be stored in the local memory of one processor. This has limited the range of tests which can be carried out on the Meiko machine.

In order to estimate the running time for problems with more than 1300 unknowns the serial time has been extrapolated from the values obtained for the number of unknowns ranging from 100 to 1300 to a number of unknowns up to 4800. A cubic polynomial has been fitted with very good precision through the data available and has been used to deduce the values above 1300.

Efficiencies have been obtained for fixed numbers of processors of 4, 8, 15, 32 and 64 and fixed number of unknowns of 1000. The three efficiencies Eff_1, Eff_2 and Eff_3 correspond to Eff_1, Eff_2 for call within a serial program and Eff_2 for call within a parallel program. Some speed-ups are also given which are denoted S_1, S_2 and S_3 following similar notation to the efficiencies.

7.5.3 Results and conclusions

The tests have been limited by memory availability. When numerical results are missing in the tables and graphs it is because the available memory has been exhausted.

There are various ways of evaluating the efficiencies of the solvers which influence the results. The first variation is related to the way the program is mapped onto the Transputers.

In previous sections, pseudo codes for performing the solution of the system have been presented. All these pseudo codes are duplicated to run on each processor. A 'main' process is in charge of testing the solver by creating the matrix A and vector x, distributing the data, collecting the results and performing the timings. In addition to the various implementations of the solver depending on whether the main program using the solver is assumed to be serial or parallel, the calculation of each efficiency can be altered by the mapping chosen.

Denoting 'main' the test process and 'slave' the solution processes two types of mapping are possible, which are shown in Figure 7.32.

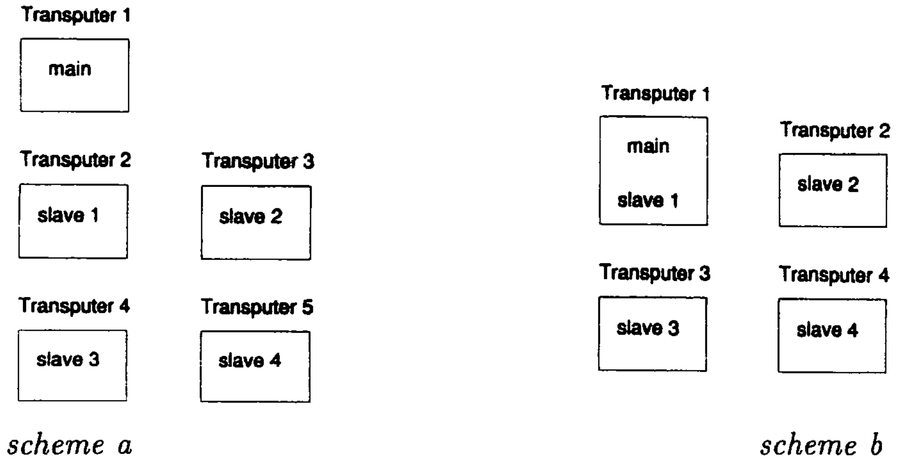


Figure 7.32: *Mappings for 4 slave processes*

When calculating the efficiency in the case of the scheme a the speed-up should be divided by 5, which is the number of physical Transputers used whereas in the case of the scheme b the speed-up is divided by 4 since only 4 Transputers are used. An example of the two ways of evaluating the efficiency is given in Table 1.

Although all the figures indicate good performance, the difference in the efficiencies between the schemes a and b is striking. The mapping of type b is more efficient and is used in all subsequent timings.

Before presenting the results obtained for both the symmetric and unsymmetric solvers, it seems interesting to examine the results obtained by the Farhat and

	T_1(s)	T_p(s)	S	*eff*(%)
scheme *a*	3040	785	3.87	77
scheme *b*	3040	786	3.86	96

Times for 4 slave processes and 1000×1000 dense unsymmetric matrix.

Table 1: *Comparison of the different mappings*

Wilson[8] symmetric solver which has also been implemented on the Meiko machine. These are given in Table 2.

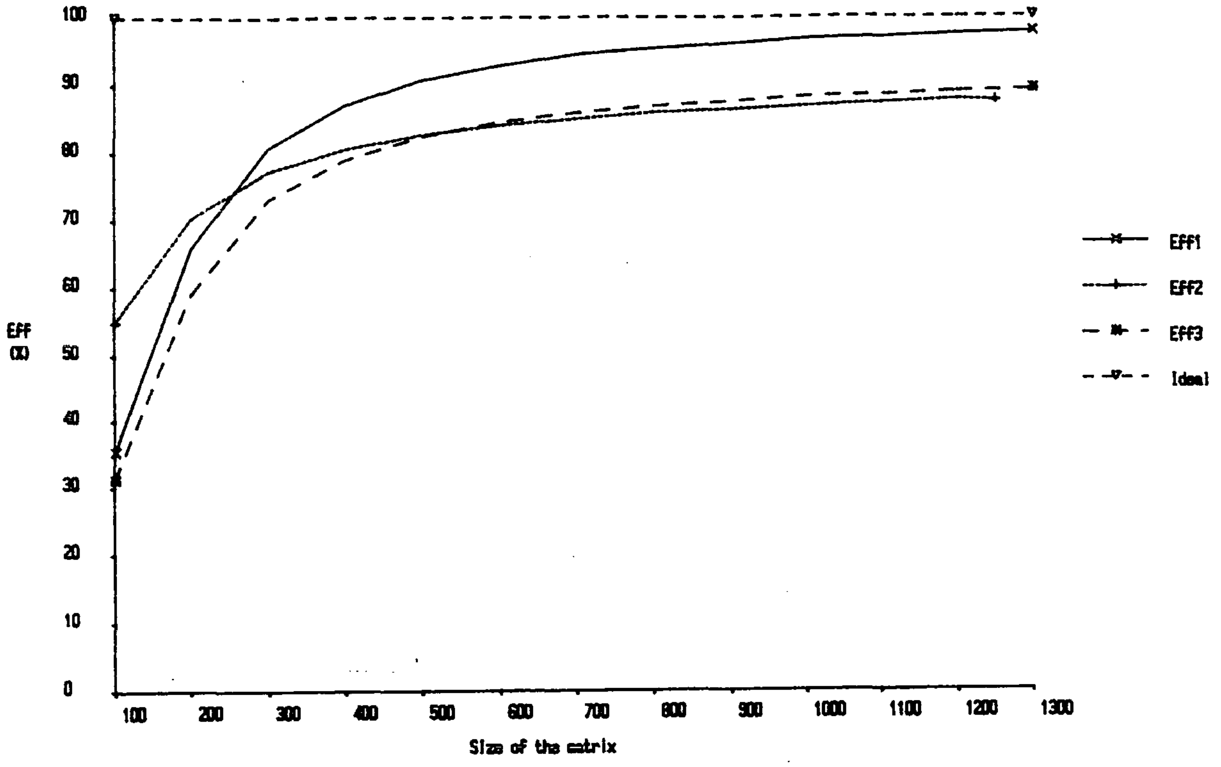
For 4 processors our solver fares slightly better which is due to the fact that the LU decomposition and the forward substitutions are performed together whereas Farhat and Wilson performed them separately. For higher number of processors the significant drop in performance of our solver is due to a poor configuration of the machine and also to a different hardware which may imply that communication is carried out faster on the IPSC machine used by Farhat and Wilson than on the Meiko machine.

matrix	4 processors		16 processors	15 processors
size	Far	our	Far	our
450	91	94	60	46
900	/	/	83	64

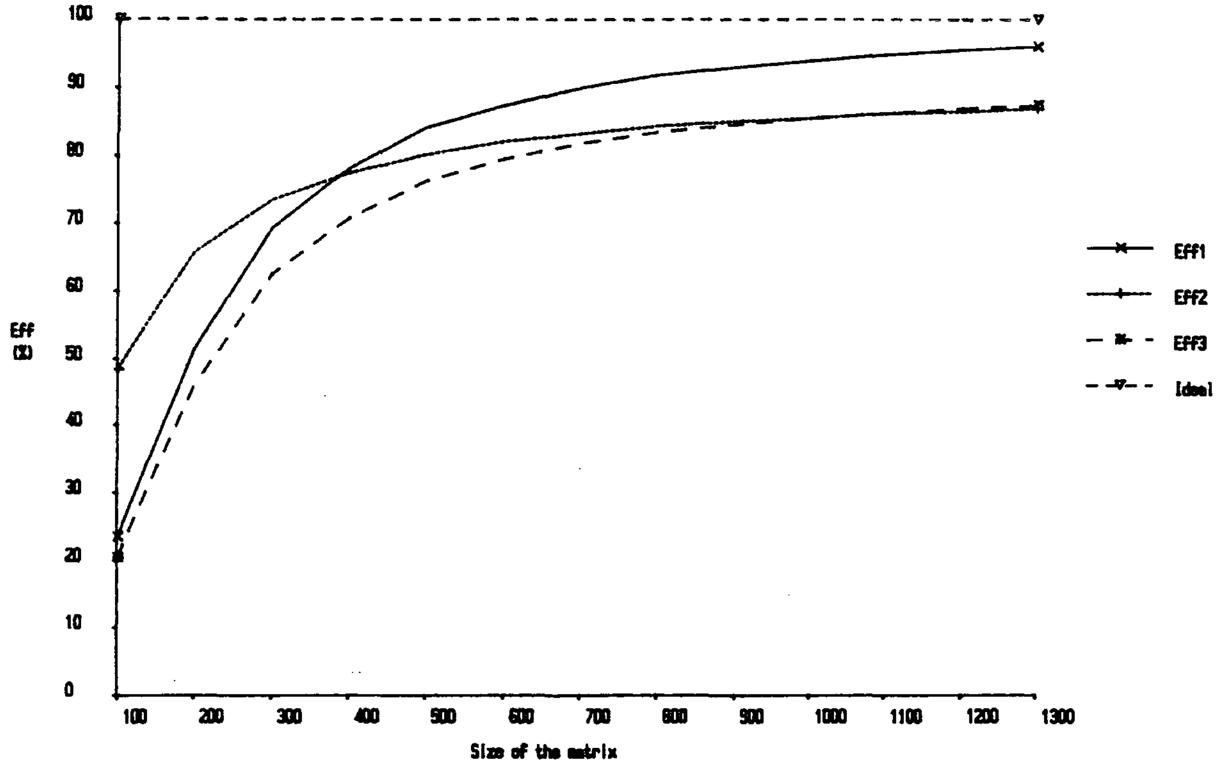
The matrices are symmetric and dense. 'Far' represents Farhat and Wilson results and 'our' represents the results obtained on the Meiko machine. The numbers represent eff_2 in percents.

Table 2: *Comparisons for the symmetric solver*

The results obtained are given in Figures 7.33 to 7.41. The results are next analysed.

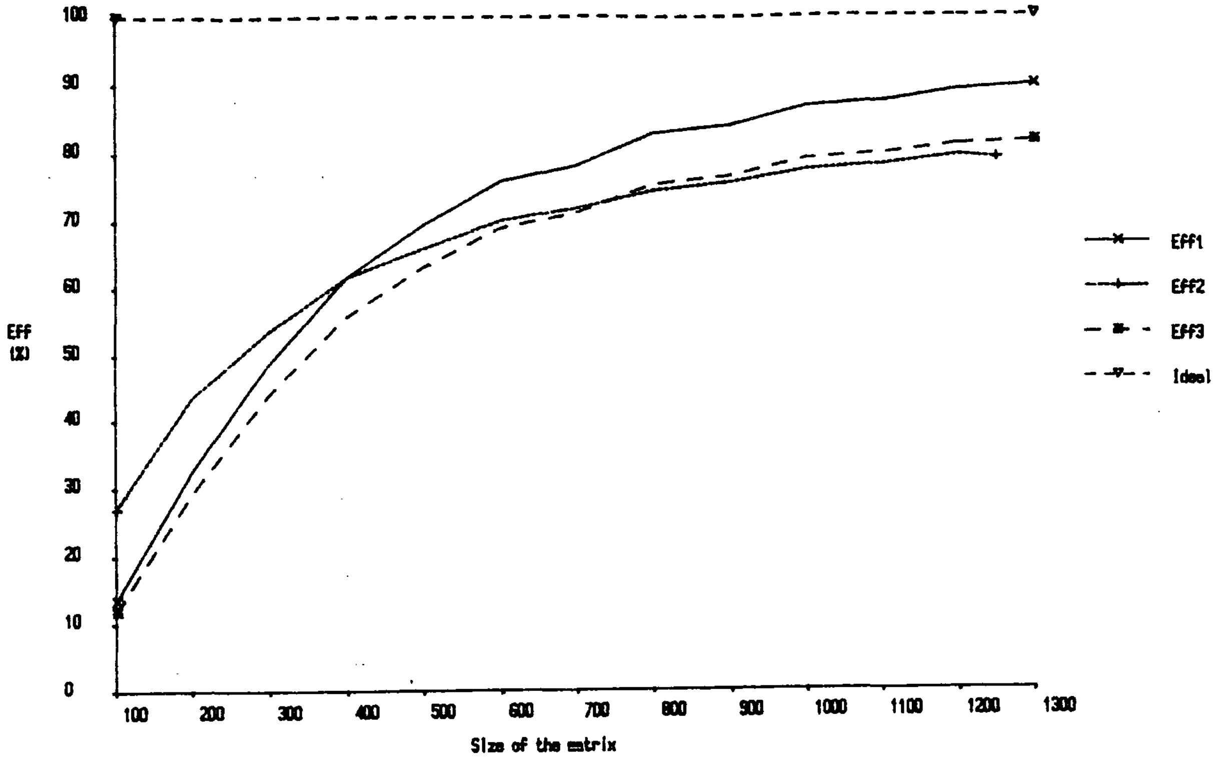


Efficiencies for unsymmetric matrices on 4 Transputers

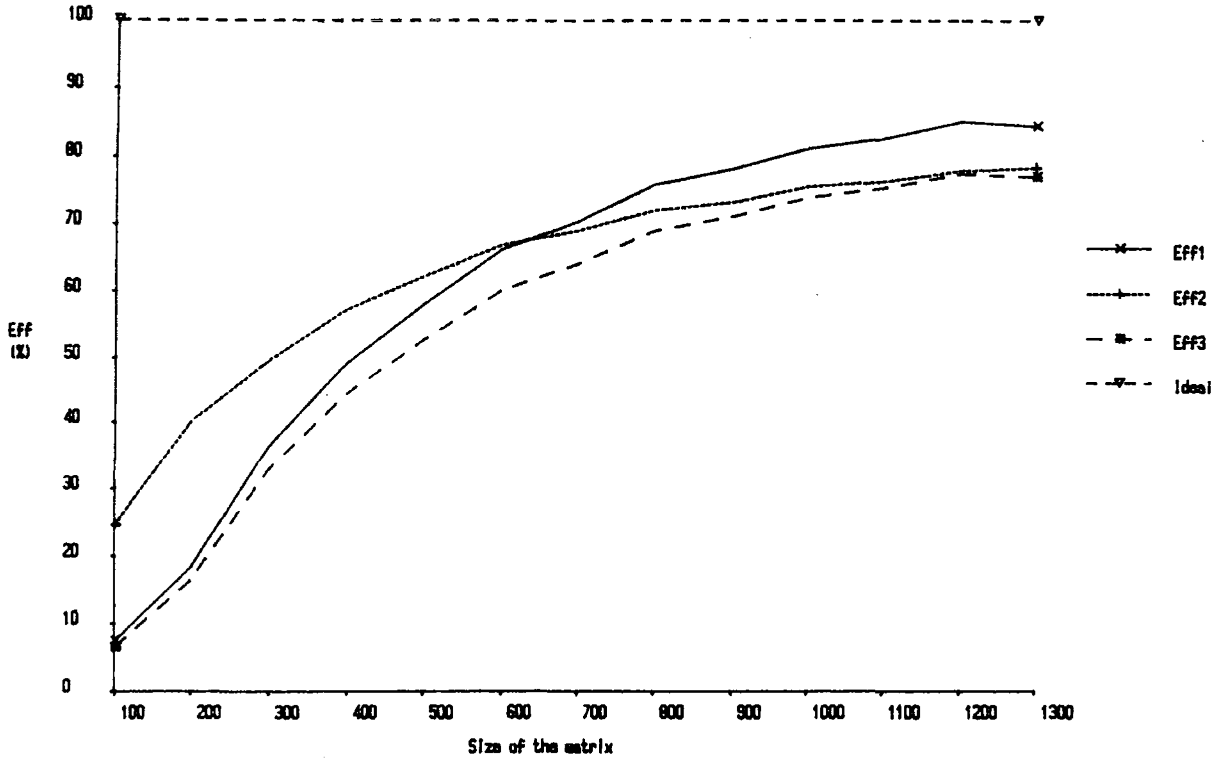


Efficiencies for symmetric matrices on 4 Transputers

Figure 7.33

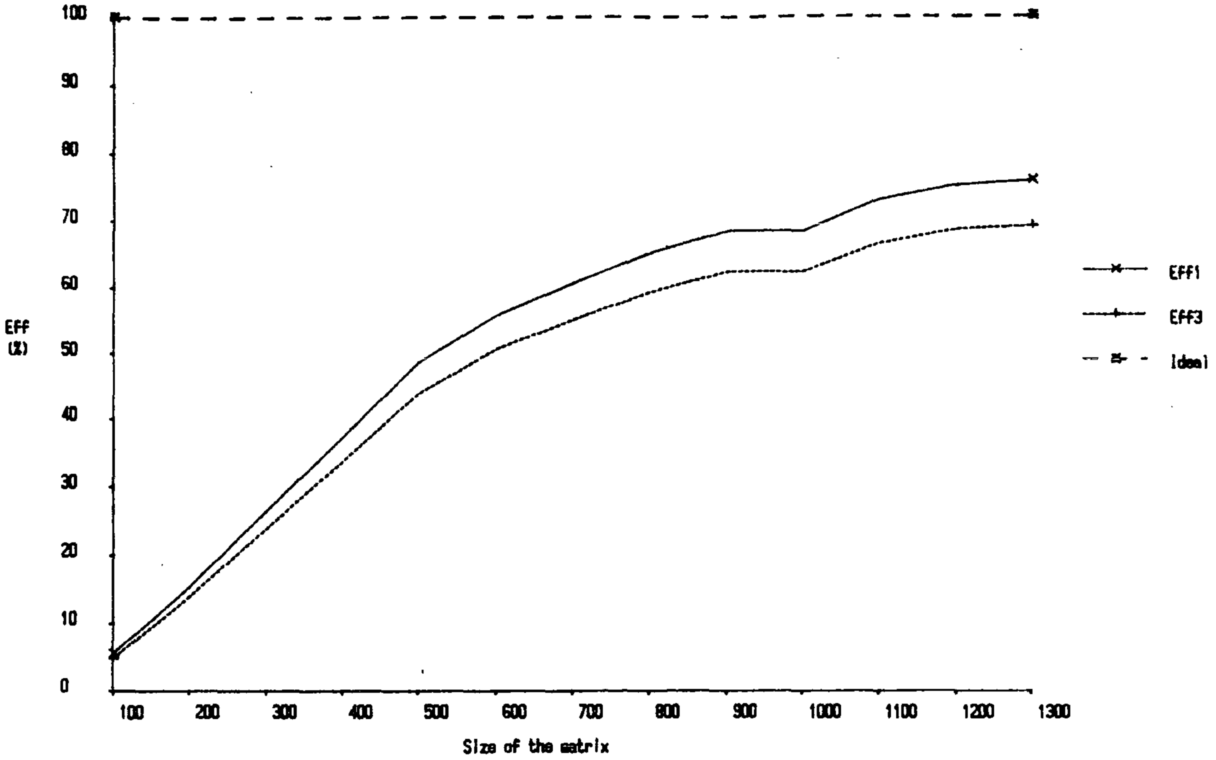


Efficiencies for unsymmetric matrices on 8 Transputers

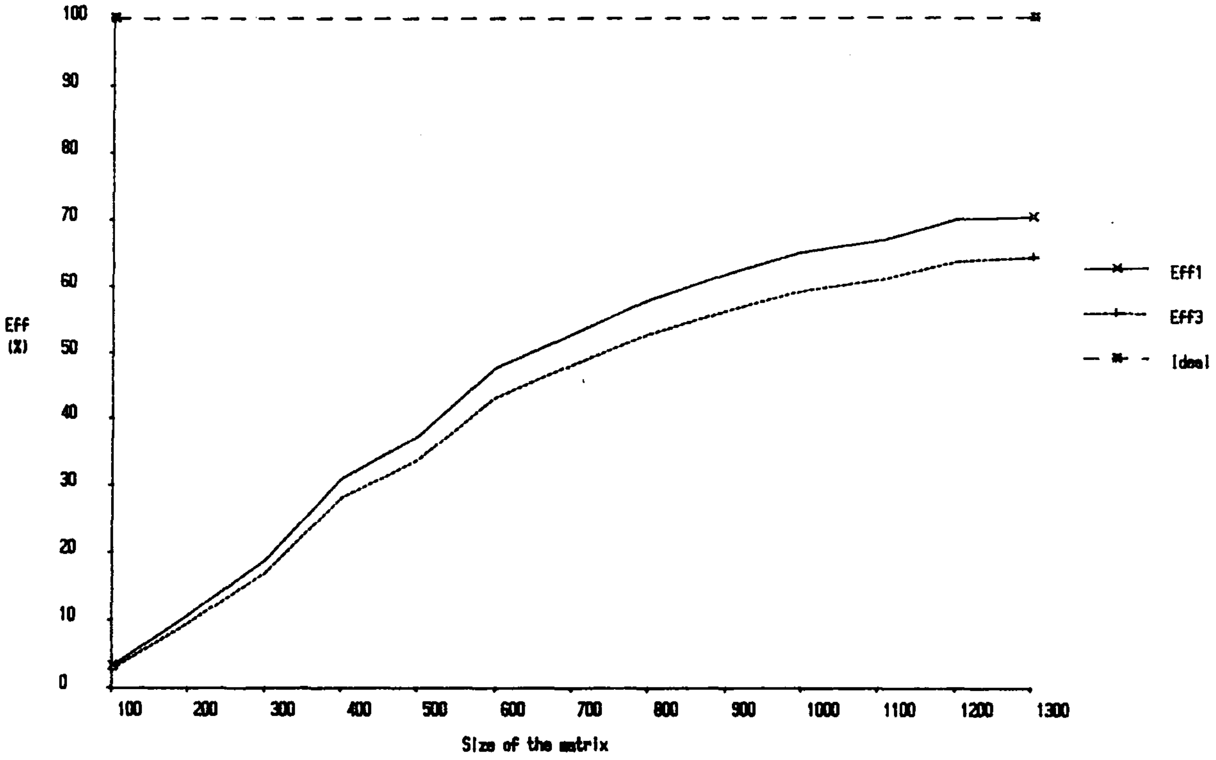


Efficiencies for symmetric matrices on 8 Transputers

Figure 7.34

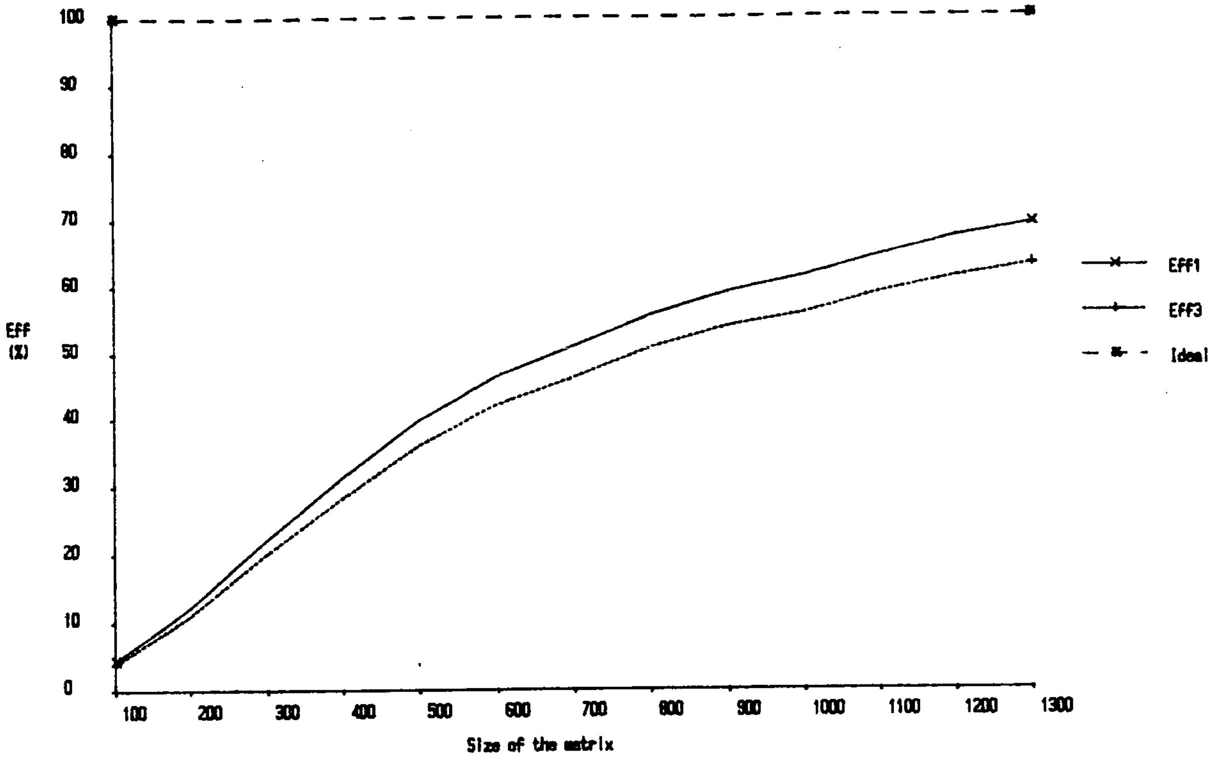


Efficiencies for unsymmetric matrices on 12 Transputers

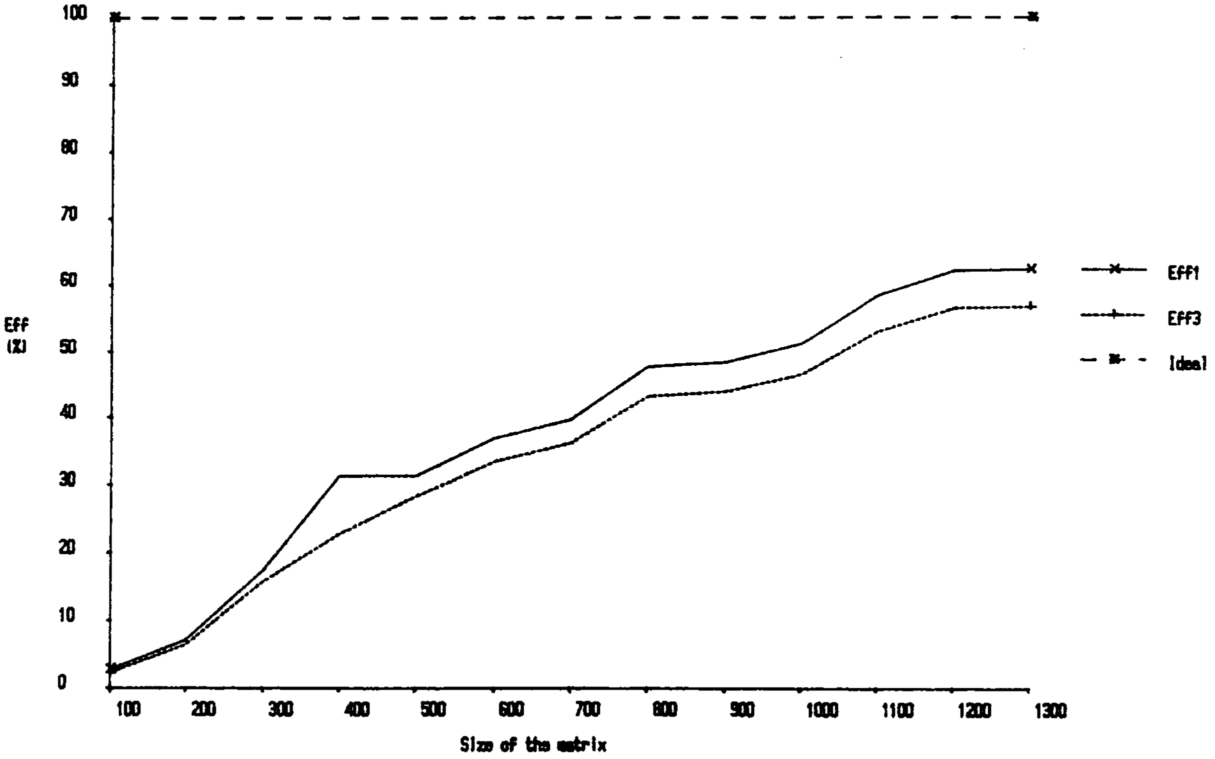


Efficiencies for symmetric matrices on 12 Transputers

Figure 7.35

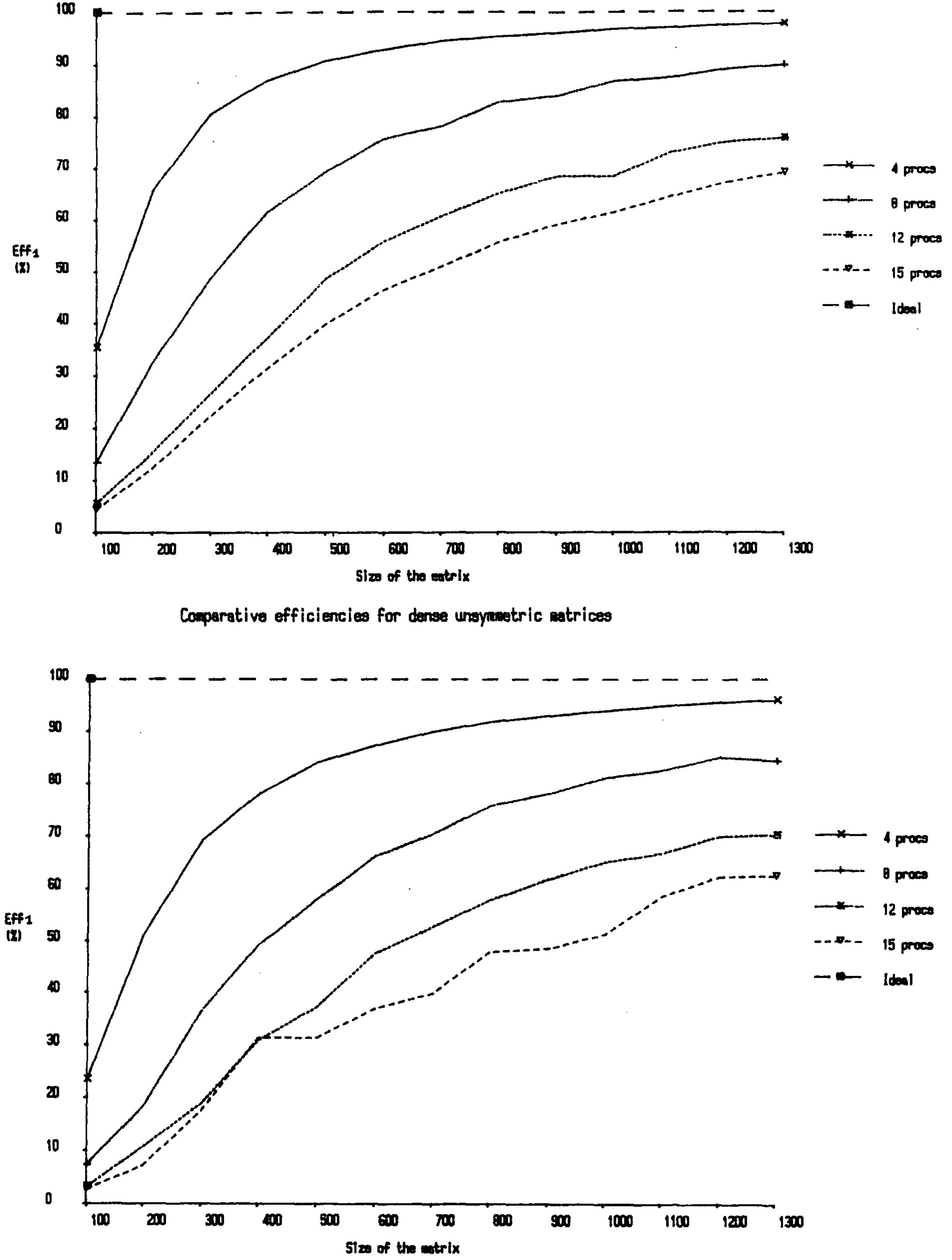


Efficiencies for unsymmetric matrices on 15 Transputers



Efficiencies for symmetric matrices on 15 Transputers

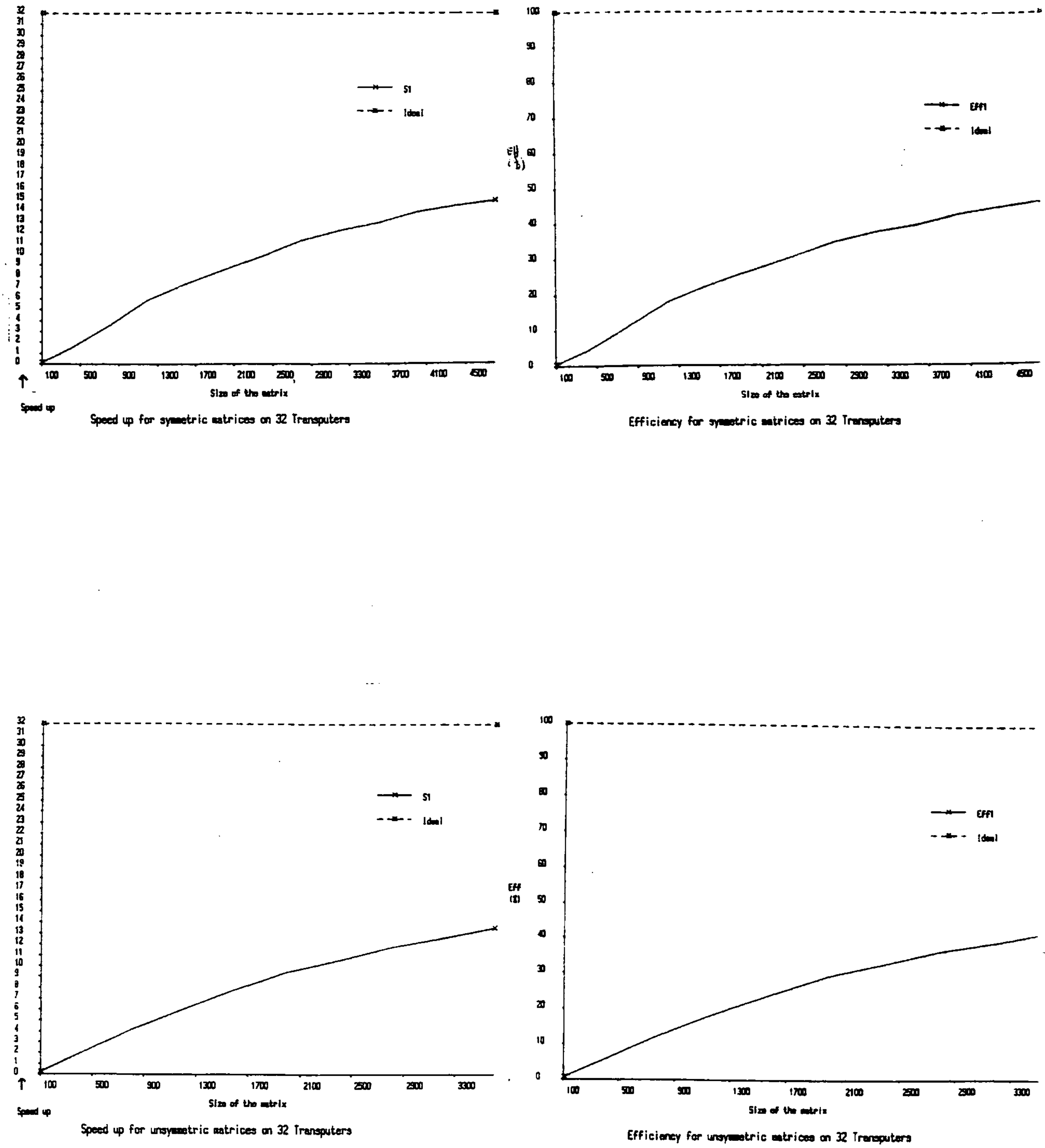
Figure 7.36



Comparative efficiencies for dense unsymmetric matrices

Comparative efficiencies for dense symmetric matrices

Figure 7.37



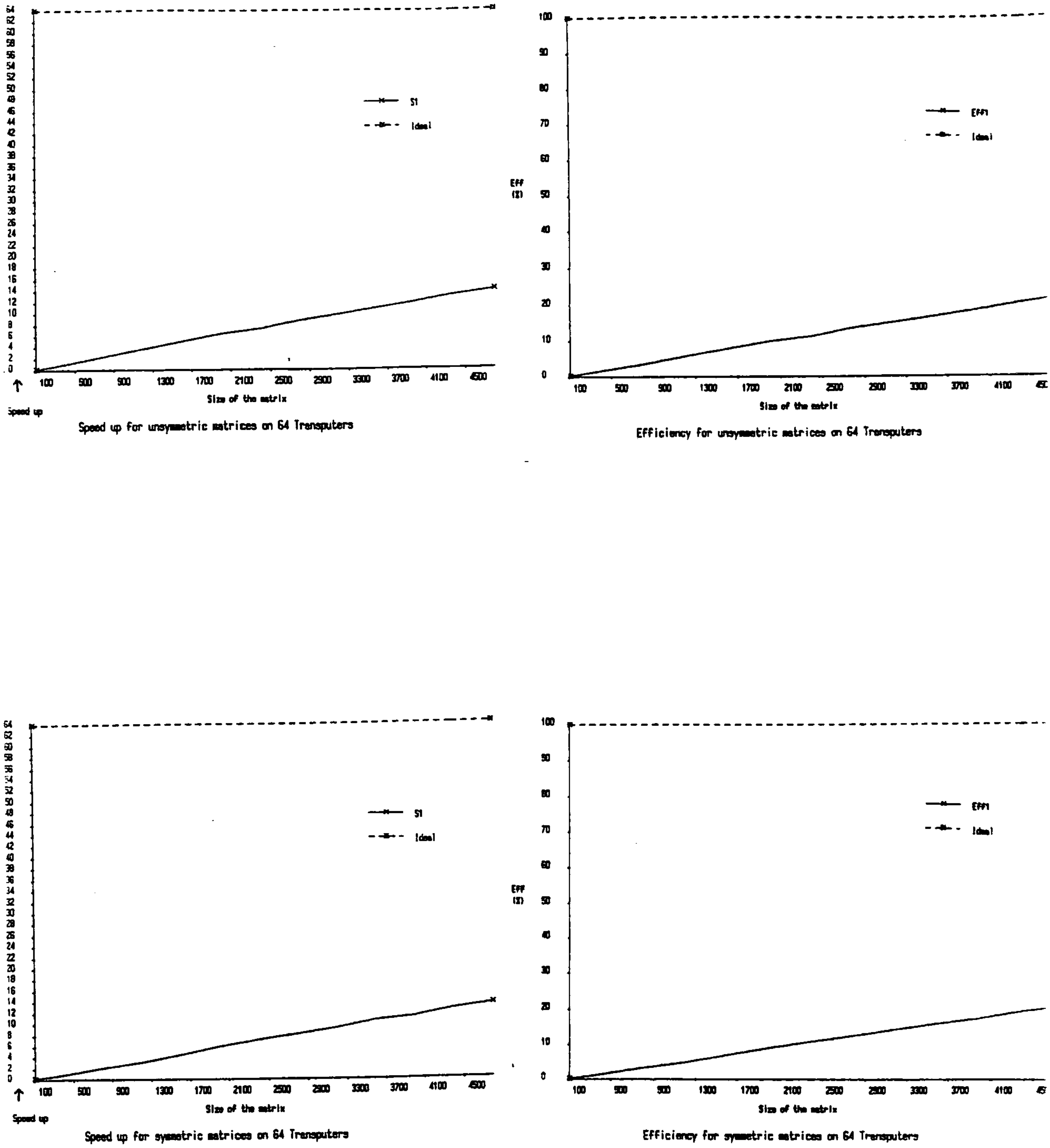
Speed up for symmetric matrices on 32 Transputers

Efficiency for symmetric matrices on 32 Transputers

Speed up for unsymmetric matrices on 32 Transputers

Efficiency for unsymmetric matrices on 32 Transputers

Figure 7.38



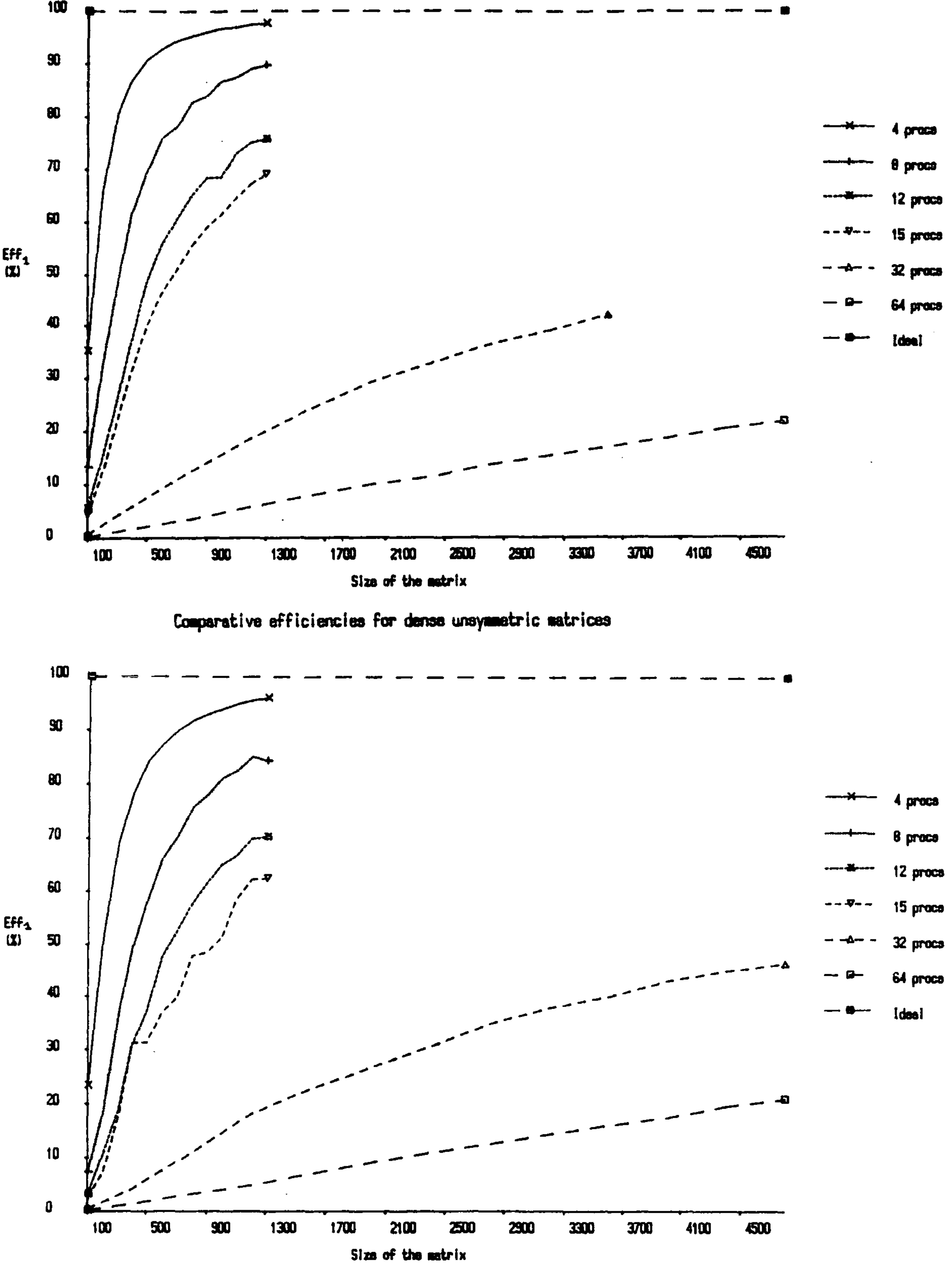
Speed up for unsymmetric matrices on 64 Transputers

Efficiency for unsymmetric matrices on 64 Transputers

Speed up for symmetric matrices on 64 Transputers

Efficiency for symmetric matrices on 64 Transputers

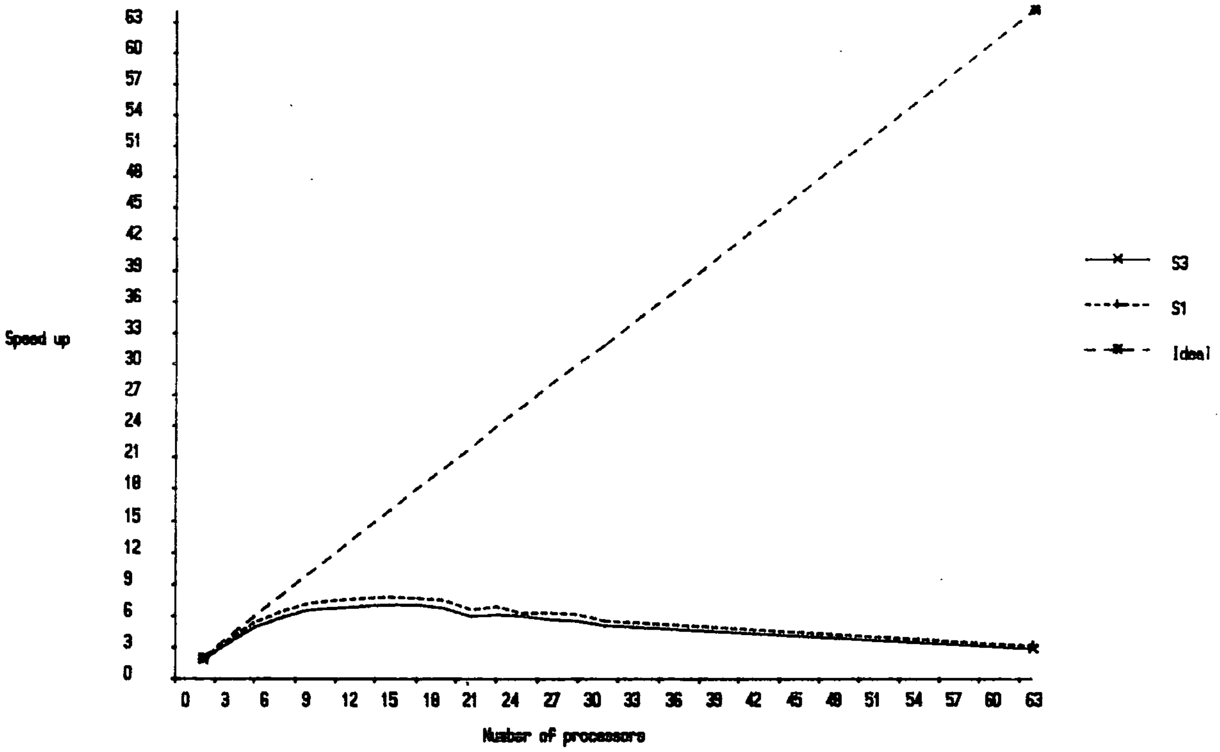
Figure 7.39



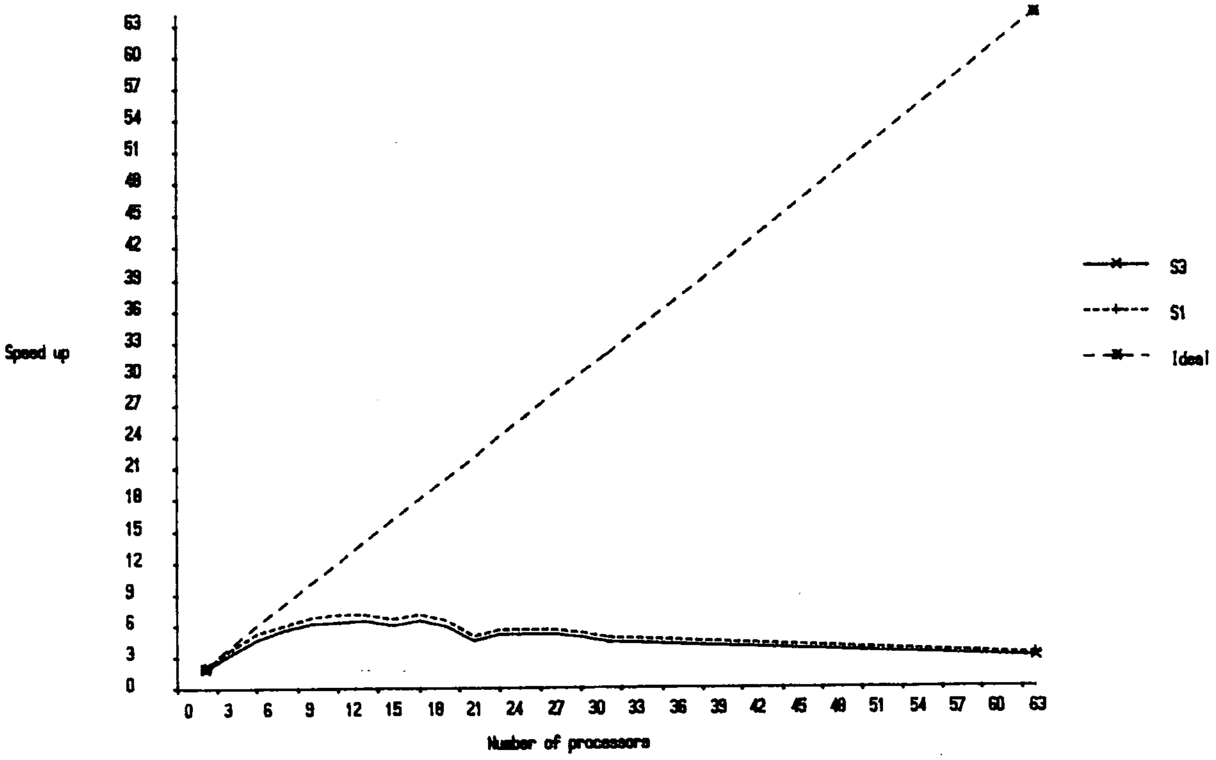
Comparative efficiencies for dense unsymmetric matrices

Comparative efficiencies for dense symmetric matrices

Figure 7.40



Speed ups for a dense 1000*1000 unsymmetric matrix on Transputers



Speed ups for a dense 1000*1000 symmetric matrix on Transputers

Figure 7.41

The comparison of the three efficiencies *eff*$_1$, *eff*$_2$ and *eff*$_3$ are given in Figures 7.33 to 7.37. The graph for *eff*$_1$ is always located above the graph for *eff*$_3$ which indicates that evaluating the solver using *eff*$_1$ is misleading since in reality when the user replaces the serial solver by the parallel solver the efficiency obtained (*eff*$_3$) is between 5 and 10 percent lower than predicted by *eff*$_1$.

The behaviour of *eff*$_2$ in Figure 7.33 and 7.34 indicates that for matrices large enough the parallel backward substitution becomes efficient whereas for smaller matrices the serial execution of the backward substitution is definitely faster. *eff*$_2$ is not available for 12 and 15 processors (Figures 7.35 and 7.36) because the Meiko machine used had some configuration problems.

Figure 7.37 shows the comparative efficiencies *eff*$_1$ from the previous 4 graphs. For the size of matrices tested, using 4 processors gives the best efficiencies. From the shape of the lines for 8, 12 and 15 processors and their relative position it can be concluded that using more processors is worth if the matrices are bigger than 1300×1300.

Figures 7.38 and 7.39 indicate similar behaviour. When the number of processors increases the ratio between the time spent on communication and the time spent on calculations increases which has the effect of reducing the efficiency. This is due to the fact that Transputers having only four links when a large number of them is used messages from one Transputer to another have to be routed via a number of other Transputers, which takes time. It means that the distance between two Transputers increases which is very penalising for large networks.

Figure 7.40 gathers all the results from the previous graphs. It indicates that for each matrix size there is an optimum number of Transputers to be used to obtained the best efficiency. Unfortunately is has not been possible to extend the lines for 4, 8, 12 and 15 processors to more than 1300 unknowns due to memory restrictions. What is suspected is that all the lines will eventually rise towards a 100/down when the amount of work to be carried out by each Transputer becomes too large.

Finally Figure 7.41 shows what is the optimal number of Transputers to be used for obtaining the solution of a 1000×1000 matrix the fastest possible in terms of real figures. This number is situated around 15 processors. This optimal number of Transputer does not make the best use of the processing power available (relatively poor efficiency) but gives the fastest solution time. It is suspected that as the matrix size increases the graph will be shifted rightwards thus increasing the optimal number of Transputers.

In all the graphs the unsymmetric solver shows better performance than the symmetric solver. This is expected since both solvers have exactly the same communication structure and the unsymmetric solver carries out more calculations than the symmetric solver. The proportion of calculation to communication is therefore more advantageous in the unsymmetric case than in the symmetric case.

Overall the solvers implemented work well on the Meiko machine and give the best efficiencies for a small number of Transputers. The algorithm used appears to be a coarse grain algorithm. Some work still needs to be done on the topic of optimal configuration of the Network of Transputer. It would also be useful if a few Transputers in the network would have an increased amount of memory which would enable us to run much larger problems.

The efficiencies for the shared memory Encore machine are very similar. The graph is given in Figure 7.42.

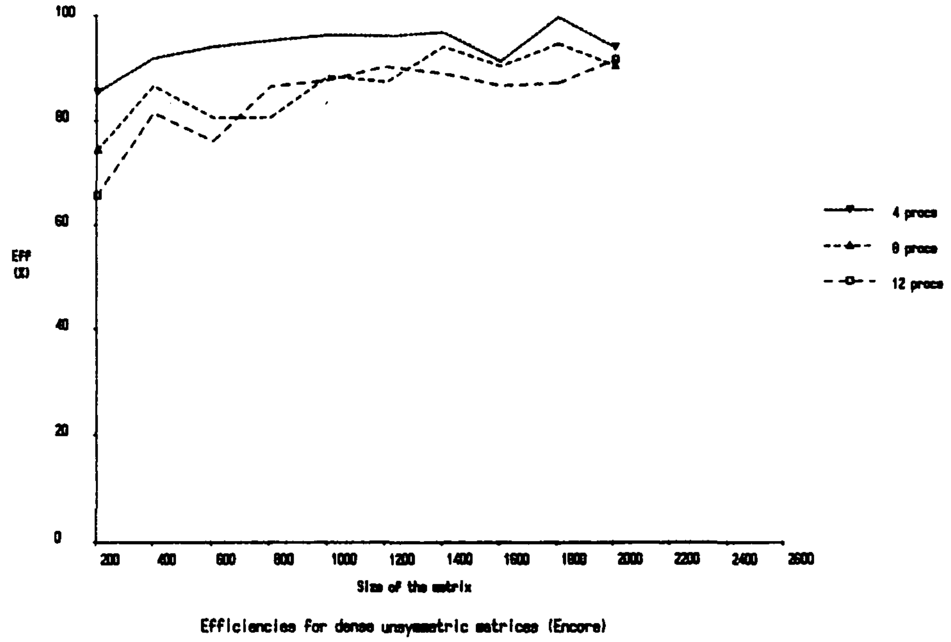


Figure 7.42: *Efficiencies on the Encore machine for unsymmetric matrices*

One point about this machine is that the efficiencies increase until all but one processors are used. When they are all used, the operating has to compete with the user's program which incures degradation in the performance.

The similarity of the results on the two machines confirms that the application is better suited for coarse grain parallelisation, at least for the size of problems attempted. For larger problems, the limitation on the Encore machine would come from a saturation of the communication bus and on the Meiko machine they would be brought by the small amount of memory available to each Transputer.

References

1. Bertsekas D.P. and Tsitsiklis J.N., 'Some Aspects of Parallel and Distributed Iterative Algorithms — A Survey', *Automatica*, **27**, Iss 1, pp 3-21, 1991.
2. Gallivan K.A., Plemmons R.J. and Sameh A.H., 'Parallel Algorithms for Dense Linear Algebra Computations', *SIAM Review*, **32**, Iss 1, pp 54–135, March 1990.
3. Yu D.C. and Wang H., 'A New Parallel LU Decomposition Method', *IEEE Transactions on Power Systems*, **5**, Iss 1, pp 303–310, February 1990.
4. Yu D.C. and Wang H., 'A New Approach to the Forward and Backward Substitutions of Parallel Solution of Sparse Linear Equations — Based on Dataflow Architecture', *IEEE Transactions on Power Systems*, **5**, Iss 2, pp 621–627, May 1990.
5. Amano H., Boku T. and Kudoh T., '$(SM)^2$-II: A Large-Scale Multiprocessor for Sparse Matrix Calculations', *IEEE Transactions on Computers*, **39**, Iss 7, pp 889–905, July 1990.
6. Geist G.A. and Romine C.H., 'LU Factorisation on Distributed-Memory Multiprocessors', *Proceedings of the third SIAM Conference on Parallel processing for Scientific Computing*, pp 15–18, Los Angeles, California, USA, December 1987.
7. Lin A. and Zhang H., 'A new Parallel Algorithm for Linear Triangular Systems', *Proceedings of the third SIAM Conference on Parallel processing for Scientific Computing*, pp 36–39, Los Angeles, California, USA, December 1987.
8. Farhat C. and Wilson E., 'A Parallel Active Column Equation Solver', *Computers and Structures*, **28**, Iss 2, pp 289–304, 1988.
9. Applegarth I. and Barbier C., 'A Parallel Equation Solver for Unsymmetric Systems of Linear Equations', to be submitted for publication.

10 Gerald C.F. and Wheatley P.O., *Applied Numerical Analysis*, third edition, Addison-Wesley Publishing Company, 1984.

11. Burden R.L. and Faires J.D., *Numerical Analysis*, third edition, Prindle, Weber and Schmidt publishers, 1985.

12. Ref. 9, p 100.

13. Jennings A., 'Solution of variable band-width partial differential equations', *Computer. J.*, **15**, p 446, 1971.

14. Bettess P. and Bettess J.A., 'A profile matrix solver with built-in constraint facility', *Engineering Computations*, **3**, Iss 3, pp 209–216, September 1986.

15. Filho J.S.R.A., 'The Use of Transputer Based Computers in Finite Element Calculations', *PhD Thesis*, Department of Civil Engineering, University College of Swansea, September 1989.

16. Pease D., Ghafoor A., Ahmad I., Andrews D.L., Foudil-Bey K., Karpinski T.E., Mikki M.A. and Zerrouki M., 'PAWS: A Performance Evaluation Tool for Parallel Computing Systems', *Computer*, **24**, Iss 1, pp 18–29, January 1991.

Part III

Study of a Free Surface Flow over Gated and Non-gated Spillways

Chapter VIII

Introduction to Free Surface Flows

.1 Introduction

The study of gravity flows with a free surface has attracted great interest in the past, mainly because of its importance in the design of spillways. In developing a spillway crest-profile engineers are concerned principally with the avoidance of negative pressures against the surface of the structure. A negative pressure on the spillway means that a cavitation phenomenon can appear on the concrete surface and incur damages to the spillway. This damage show itself as holes of variable diameter and depth in the concrete surface of the spillway.

It is extremely costly to repair such damage to the surfaces. The idea is thus to design a spillway crest-profile such that in normal conditions of use no negative pressures appear on the concrete bed. In abnormal conditions of use, however, such as flooding due to heavy rains, negative pressures may appear, incurring some costs for repairs which cannot always be avoided as a particular spillway shape can only guaranty a restricted range of trouble-free conditions of use.

Other factors must be taken into account in the design of a spillway crest-profile which include the hydraulic efficiency (coefficient of discharge), the constraints imposed by a particular site, the stability of the structure and the economics of the construction.

Several standard designs have been developed in the past century, all with the principle aim of avoiding negative pressures. Some were developed from a simple mathematical analysis of the trajectory of a fully aerated nappe[1], others from measurements on a series of physical models[2,3]. The latter had the advantage of including the effect of friction between the fluid and the spillway surface, a significant factor in terms of surface pressures.

In many instances it is not convenient to use a standard profile and one has to resort to the use of physical models to evaluate the particular crest geometry chosen. If the crest proves to be hydraulically unsatisfactory the model has to be modified and retested until the results are acceptable. Therefore the final design is obtained by trial and error. Using physical models this process can be time consuming and expensive. On the other hand a numerical model, even if its accuracy did not match that of the physical model, could look at a range of design fairly quickly and economically and the physical model would only be required for the final confirmation of the chosen design.

The problem is to predict numerically the position of the free surface in some two-dimensional flow, for example a spillway, over a weir or under a sluice or gate. It will be assumed that the flow is inviscid and irrotational. For many practical problems this is a reasonable assumption, although of course for some free surface flows, the hydraulic jump for example, it would not be, and the method described later would be inadmissible.

The difficulty of the problem is that both the discharge and the position of the free surface are unknown, and that the boundary condition on the free surface is nonlinear. In addition, the flow is subcritical in the upstream portion of the crest of the spillway, while it is supercritical in the downstream portion. Because of these difficulties, the earlier studies of the problem were mostly experimental.

8.1.1 Survey

The first attempt to numerically predict the position of the free surface was done by Southwell and Vaisey[4] using finite difference scheme. The first modelling of spillway flow was made by Cassidy[5]. By means of a relaxation technique iterating within the complex potential plane, the surface profiles and discharges coefficient for weir contours designed as profiles of a spillway were calculated. Slow convergence was encountered. From then, three different approaches to the problem have been followed, which are described by Bettess and Bettess[6]. They are as follows:

1. Fixing the element mesh and varying the element properties so as to model the position of the free surface.

2. Extending the finite element mesh from the bed to the free surface and moving the mesh to follow the free surface as iterations are performed.

3. Inverting the problem by using coordinates as dependent variables and using the streamfunction and the velocity potential as independent variables.

The method 1 has mainly been used for seepage or similar flows in which the kinetic energy of the flow is small. It is not of interest for the problems of flows over spillways considered here, where the kinetic energy of the flow can be high. The methods 2 and 3 are those which have been employed for solving the type of problems in which we are interested. Method 2 uses finite element methods while method 3 is based on relaxation techniques. They both, in their earlier forms, relied on the user to guess an initial discharge and iterate manually towards the discharge which gives the smoothest surface for the flow.

Early work on method 2 was carried out by Ikegawa and Washizu[7] who were the first to use a variable domain functional approach. They used the variational principle derived by Luke[8] which was stated in terms of the velocity potential. Although this principle was well suited to problems of water waves, for the two-dimensional steady flow problems it was more convenient to solve the problem in terms of the streamfunction. Ikegawa and Washizu rederived Luke's variational principle in terms of the streamfunction and used a Newton-Raphson type of iteration and a formulation by finite elements but incorrectly neglected some terms.

Betts[9] derived the correct variational formulation and used triangular linear elements in the finite element approximation. Further work using a similar approach was performed by Diersch *et al*[10] and Aitchison[11].

The method of Varoglu and Finn[12] is different in that it also automatically determined the discharge. They formulated the problem in terms of the hydraulic head H rather than the velocity potential or streamfunction. They solved the problem in the (x,H) coordinate system where x is the x coordinate of the nodes and transformed back the solution in the (x,y) plan. Although they initally applied their method to seepage problems they also successfully used it for steady flow over spillway.

Early work on method 3 was carried out by Cassidi[5], as mentioned earlier on, and Markland[13] who applied it to free flow over an overfall. It was subsequently applied to large amplitude waves by Williams[14].

The next generation of applications of methods 2 and 3 were based on the observation that the linear approximations used to model the different parameters and variables of the problem were too crude and that higher order approximations were needed to model the curved surfaces. They also tried to solve for both the streamfunction and the discharge.

The work done by Li *et al*[15] on the Finite Analytic Solution of Flows over Spillways considers a problem similar to the one in this chapter but they use a different technique to solve the governing equations which overcomes the problem of the simplistic linear approximation. They use a boundary fitted coordinate system which has the advantage of accurately model the curved boundary on the free surface.

The principle of the boundary fitted coordinate system is to consider the boundary of the problem as the new coordinate system. A mapping between the real domain of study and a simple rectangular domain is numerically generated as the solution of an elliptic system of partial differential equations with Dirichlet boundary conditions. The advantage is that complex boundary geometry can be expressed simply and with minimal error.

In this new system the equations are expressed and solved using the Finite Analytic method which has been developed to solve equations in this new system. This method is based on the decomposition of the new domain into small elements. Triangular elements are used here. In each of these elements the analytic equations are solved and the solutions are combined into a set of algebraic equations which approximate the governing equations in the whole domain. These are solved and the solution is transformed back to the original domain. The interesting aspect of this method is in the avoidance of the use of approximation functions, like in the finite element method, therefore eliminating the truncation errors.

The solution for the free surface shape and the value of the discharge is obtained through a double iteration scheme. From an initial guess of the free surface position and the discharge, an inner iteration solves for the free surface shape and at each step adjusts the shape of the surface according to the solution obtained. When the nodes on the surface do not vary more than 10^{-3} from one iteration to another, the shape of the surface obtained is considered satisfactory and a new value for the discharge is derived.

The outer iteration modifies the value of the discharge until a tolerance of 5×10^{-4} in the changes of the value is obtained. Finally, the pressure on the spillway are calculated. The authors claim that this method gives good results when comparing the output of the program to practical experiments.

Henderson *et al*[16] have developed a computer-aided spillway design based on very similar grounds to that used in this part. Their formulation is in term of a streamfunction and Laplace and Bernoulli equations with extra boundary conditions. They do not however use the finite element method to solve the equations but the Boundary element method.

This method, developed in the late 70's, is based on the discretisation into elements of only the boundary of the domain. The solution on the boundary enables the calculation of the solution inside the domain. The elements obtained with this technique are not independent, contrary to the finite element method, and therefore the matrices obtained contain fewer zeros. The dimensions of the matrices are, however, smaller than the dimensions for the finite element method and the discretisation of the free surface is much easier. The element used are still linear, though.

The Laplace equation is transformed into a line integral along the boundaries, using the second Green's theorem, and the Bernoulli equation is directly discretised. The equations are solved for the position of the free surface, the values of the streamfunction on the boundaries and the discharge. The model is non-dimensionalised so that the equations have an equal weight.

This approach is interesting because the free surface and the discharge are obtained at the same time. The set of nonlinear equations is solved using a successive quadratic programming algorithm and a finite difference gradient. The authors mention the use of higher order elements, but point out that a numerical derivation of the equations would then be necessary and the calculations involved are more costly. They do not mention Computer Algebra as an aid to derive the equations.

The effect of the number of elements in the inlet and outlet has been studied as well as the behaviour when enlarging the upstream and downstream sections. The authors qualify the results as being satisfactorily. They also used the method to derive a cavitation criterion.

Another method which seems popular in the solution of free surface flows is the use of a complex potential made up of the velocity potential Φ and the streamfunction ψ such as $f_{\cdot} = \Phi + i\psi$. The complex plane is mapped using various methods onto a ζ-plane where the flow equations are expressed and the problem is reduced to a complex analysis. All models use non-dimensionalised equations.

Dias *et al*[17] have used this method to find the flow over rectangular weirs. They use an hodograph variable to indirectly find the complex analytic function solution to the equations. This hodograph variable is expressed in terms of a truncated infinite series which constitutes the discretisation used. The nonlinear equations obtained are solved using a numerical Newton method. The results obtained are said to be in fair agreement with the real spillways.

King and Bloor[18] used the complex potential formulation to find the flow over an arbitrary bed topology. They reformulated the fluid equations using a generalised Schwarz-Christofell method to obtain their form in the ζ-plane. They then obtain a pair of coupled integral and integro-differential equations holding on the free surface and the bed. In the first place the equations are solved using a linearisation method then a full nonlinear solver, the hybrid Powell's method, is used to find the solution. The results obtained from the two methods are compared.

Forbes[19.20,21] also used the complex potential technique but he applied it to critical flows rather than steady state flows. Although this is slightly out of the scope of the work done here, some of the results obtained are interesting, particularly those concerning the use of the Newton method. Forbes has studied critical flows over a semi-circular obstruction where the Froude number is sought as part of the solution.

The nonlinear equations obtained are solved using the Newton method. For small semi-circle radius the Newton method gives a reasonable agreement with the real flow but for larger values the it completely fails. The author attributes this to a problem with numerical accuracy, which they found was connected to a clustering of the numerical grid points in the downstream part, although this is not sure. Forbes has also studied two-layer critical flows[20] using the same technique.

This survey highlight the fact that no definite method has been found to accurately solve the problem of free surface flows. New techniques are investigated to solve the governing equations which inevitably lead to nonlinear systems of equations which are solved numerically by a linear approximation or an iterative solver like the Newton method. Although most of the papers seem to obtain good results, the actual words used to qualify the accuracy are generally 'fair', 'acceptable' and 'satisfactory'.

8.1.2 Overview of the chapter

The work done in this part of the thesis follows the same basic ideas as the works mentioned in the survey section. The method chosen to solve the governing equation is the finite element method and the solver for the nonlinear equations is the Newton method. The novelty, though, is in the investigation of new tools to help improve the accuracy and open scope for more complicated formulations such as the use of higher order elements or more refined nonlinear solvers.

The work is based on a previous investigation of Bettess and Bettess[6] from which some of the material in this part is derived. They carried out an analysis of free surface flows using isoparametric finite elements. The equations were expressed using a variational principle for which the nonlinear inverse of the Jacobian matrix was developed in a Taylor series truncated after the second order. The Newton method was used to solve the resulting discretised set of nonlinear equations. They also used an iterative scheme to obtain the discharge which was based on the errors in the Bernoulli equation on the surface. A double iteration technique similar to that of Li[15] was employed to find both the free surface position and the discharge.

Starting from the same basic equations, the approach followed in this part of the thesis formulated the inverse of the Jacobian matrix exactly by obtaining its analytical expression using Computer Algebra. The integration of the element matrix is still carried out numerically. The idea in using Computer Algebra is to remove one level of approximation in the hope that the convergence problems encountered by the original investigators would partly be solved through an improvement of the accuracy.

An other aim of the investigation is to improve the running time of the programs through the use of parallel processing. The formation of the element matrices in parallel has been carried out and the efficiency has been evaluated.

This part of the thesis is organised in three chapters. The present chapter concentrates on stating the basic principles of free surface flows. The next chapter explains in details the dicretisation of the governing equations, the use of Computer Algebra to form the finite element matrices and the parallelisation of their numerical evaluation. Finally, the last chapter is devoted to the discussion of the tests carried out, the results obtained and the problems encountered together with the further investigation carried out to solve them.

8.2 The governing equations

The aim of the work is not to examine the leading equations of the problem but to concentrate on their coding on the conputer, including the analysis of the approximation method used to solve the problem. Therefore, the basic principles of fluid mechanics are assumed and the corresponding equations are given without special references or demonstrations. Nevertheless, where these equations have been used in a different form from the traditional statements, for example use of different conventions or coordinate systems, their expression is fully derived.

In this section, the basic equations governing the behaviour of a flow over a spillway are derived for the conventions adopted in this study. Their solution using a variational principle is given.

8.2.1 Laplace equation

The flow considered is assumed to be irrotational. Therefore the problem can be formulated in term of a streamfunction, denoted ψ. Other formulations can also be used including velocity potentials. Although the form of the equations obtained varies depending on the formulation chosen, the end results are the same.

The streamfunction measures the flux of the flow across a given section of the flow. The streamlines of the flow are the lines where ψ is constant. The value of ψ on a streamline is not unique as the streamline $\psi = 0$ may be assigned to any convenient streamline.

The streamfunction is related to the velocities of the fluid through the first order derivatives. In a cartesian coordinate system Oxy, let denote u the component of the velocity in the x direction and v the component in the y direction. The relation between the streamfunction and the velocity is then given by the following equations:

$$u = -\frac{\partial \psi}{\partial y} \qquad v = \frac{\partial \psi}{\partial x}. \tag{8.1}$$

The choice of the signs in front of the first order derivatives of ψ depends on a convention which is variable from author to author. Providing the same convention is used throughout the calculations, the choice of the signs is arbitrary. The equation (8.1) uses the convention of Massey[21].

The condition of irrotationality of the flow is given below:

$$\frac{\partial u}{\partial y} - \frac{\partial v}{\partial x} = 0. \tag{8.2}$$

When equations (8.1) and (8.2) are combined together, the Laplace equation is obtained:

$$\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2} = 0. \tag{8.3}$$

This equation applies throughout the volume of the liquid.

8.2.2 Bernoulli equation

For inviscid, incompressible and steady flows, the Bernoulli equation applies along each streamline. Free surface flows have a natural streamline along the surface, for which the Bernoulli equation holds:

$$\frac{P}{\rho} + \frac{V^2}{2} + gh = constant, \tag{8.4}$$

where P is the pressure, V is the velocity and h is the elevation above some convenient datum of a point along the surface of the flow. We will here take h as y and the convenient datum as the y origin of the coordinate system $0xy$. ρ is the density of the flow and is constant since the fluid is incompressible. g is the gravity acceleration. For free surface flows, the surface of the flow is at atmospheric pressure and P is zero. The Bernoulli equation is simplified as:

$$\frac{V^2}{2} + gy = constant. \tag{8.5}$$

The velocity V in the equation above can be expressed in function of the streamfunction ψ as shown below:

$$\begin{aligned} V^2 &= u^2 + v^2 \\ &= \left(-\frac{\partial\psi}{\partial y}\right)^2 + \left(\frac{\partial\psi}{\partial x}\right)^2 \\ &= \left(\frac{\partial\psi}{\partial x}\right)^2 + \left(\frac{\partial\psi}{\partial y}\right)^2. \end{aligned} \tag{8.6}$$

This can be further simplified when introducing the local coordinate system made up of the tangential and normal directions at point (x,y), called (w,n). This is shown in Figure 8.1.

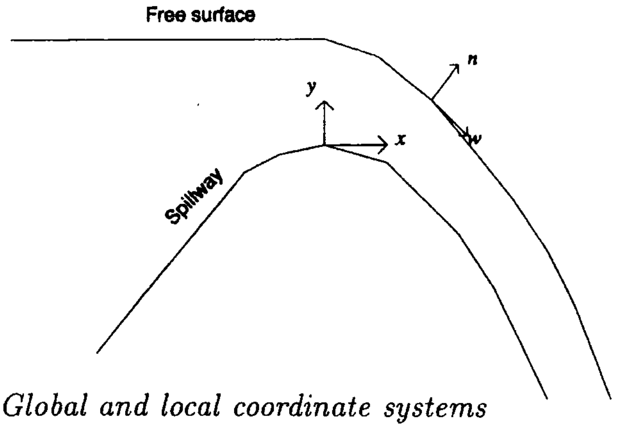


Figure 8.1: *Global and local coordinate systems*

Denoting the components of V in (w,n) as V_w and V_n, the following equation holds:

$$\begin{aligned} V^2 &= V_w^2 + V_n^2 \\ &= \left(-\frac{\partial\psi}{\partial n}\right)^2 + \left(\frac{\partial\psi}{\partial w}\right)^2 \\ &= \left(\frac{\partial\psi}{\partial w}\right)^2 + \left(\frac{\partial\psi}{\partial n}\right)^2 . \end{aligned} \tag{8.7}$$

The surface being a streamline, the velocity on the surface is tangantial to the surface, therefore

$$V_n = 0. \tag{8.8}$$

Combining equations (8.6), (8.7) and (8.8) leads to:

$$V^2 = \left(\frac{\partial\psi}{\partial x}\right)^2 + \left(\frac{\partial\psi}{\partial y}\right)^2 = \left(\frac{\partial\psi}{\partial n}\right)^2 . \tag{8.9}$$

The Bernoulli equation becomes:

$$\frac{1}{2}\left(\frac{\partial\psi}{\partial n}\right)^2 + gy = constant. \tag{8.10}$$

To determine the *constant*, more details about the geometry of the problem are needed. The domain of study is composed of a fixed geometry bed and a fluid domain Ω expressed in the coordinate system Oxy as shown in Figure 8.2.

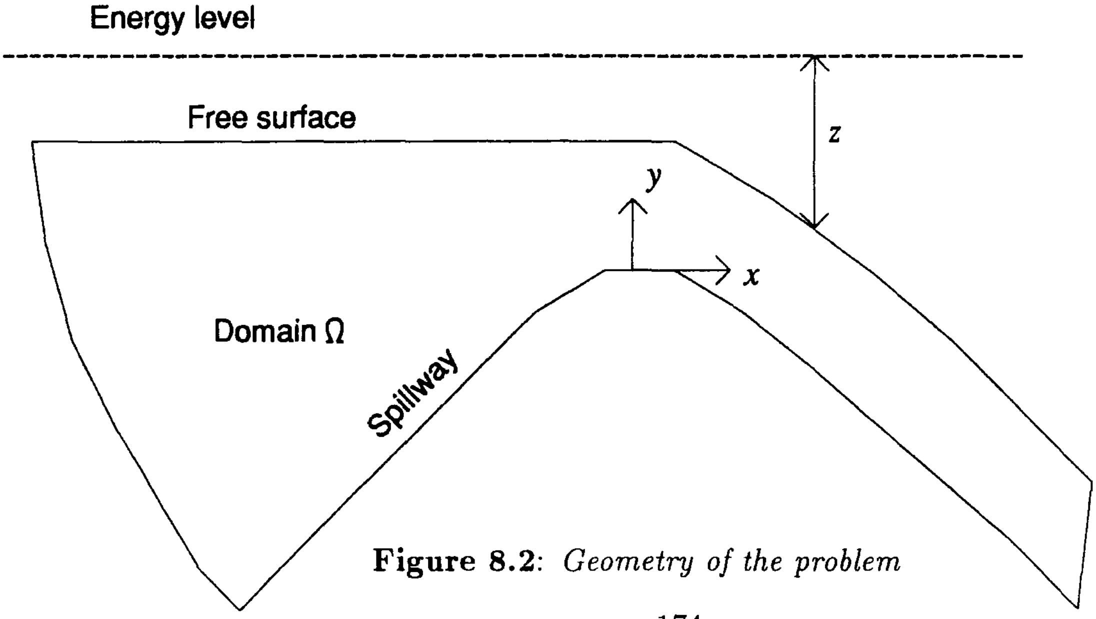


Figure 8.2: *Geometry of the problem*

E represent the energy level, which is the stagnation level far upstream. The Bernoulli equation (8.1) holds for all points on the surface, including those at infinite distance upstream for which the velocity is zero (stagnant water) and the level coincide with the energy level by definition. Therefore, for these points the Bernoulli equation is:

$$\frac{V^2}{2} + gy = gE = constant. \tag{8.11}$$

The final form of the Bernoulli equation is given below, where $z = E - y$ is taken as the distance between the energy level and the free surface:

$$\left(\frac{\partial\psi}{\partial n}\right)^2 = 2(constant - gy) = 2(gE - gy) = 2gz. \tag{8.12}$$

Another quantity in the problem is the discharge Q, which is defined as the rate of volume flow which enters the domain of study Ω. In the case of steady state flows, the rate going in is equal to the rate coming out. The formula for Q is given below:

$$Q = \int_{\Gamma_i} V d\Gamma, \tag{8.13}$$

where Γ_i is the boundary of the domain Ω at the inlet, as shown in Figure 8.3. A similar equation could be written at the outlet. In the special case when the velocities at the inlet are constant and Γ_i is a straight vertical boundary of length d this relation simplifies as:

$$Q = Vd. \tag{8.14}$$

8.2.3 Statement of the problem to solve

The problem to solve here is, given Q, determining the values of the stream-functions throughout the domain Ω coupled with finding the position of the free surface boundary Γ_s. The additional problem of finding the value of Q has also to be solved.

When studying a real situation, the knowns of the problem normally are the shape of the spillway and the stagnation level upstream. Under these given conditions, and assuming they are realistic, the fluid flows over the spillway in a unique manner imposing both the shape of the surface and the value of Q. Therefore, the governing equations of the problem should take into account this coupling.

Traditionally the two problems have been separated. The discharge Q is fixed to a value close to the real one, obtained by empirical formulæ, and the position of the surface is found. Some criteria for measuring the 'goodness' of the surface are then derived and a new value of Q is tried until the ideal match between the surface shape and the discharge is found.

This type of approach has been followed here, since there is some reference work on the subject. Later, a proposed alternative scheme will be explained where the new equations solve for both the surface shape and the discharge at the same time.

Ultimately, what is interesting to find out is the pressure distribution on the bed. This can be obtained through the Bernoulli equation applied to the streamline coinciding with the bed. This is stated below:

$$P = E - \frac{V^2}{2g} - y. \tag{8.15}$$

V being related to the steam function through equation (8.6), the values of the streamfunction on the bed have to be found.

The problem can then be stated as follows. The unknowns of the problem are the values of the streamfunctions on the bed and the position of the free surface. The problem is described by a set of two equations given below:

$$\begin{aligned} &\frac{\partial^2\psi}{\partial x^2} + \frac{\partial^2\psi}{\partial y^2} = 0 && \text{(Laplace equation)} \\ &\left(\frac{\partial\psi}{\partial n}\right)^2 = 2gz && \text{(Bernoulli equation).} \end{aligned} \tag{8.16}$$

A set of boundary conditions are added to these equations to specify a unique solution:

$$\begin{aligned} \psi &= 0 && \text{on the bed} \\ \psi &= Q && \text{on the surface.} \end{aligned} \tag{8.17}$$

8.2.4 Variational equations

The solution of these equations uses a functional formulation where the equations are derived as stationary points of an appropriately chosen integral. The functional was first derived by Luke[8] and is based on the calculus of variations as developed in Courant and Hilbert[23]. The functional is:

$$\begin{aligned} \Pi &= \Pi_1 + \Pi_2 \\ \Pi_1 &= \int\int_{\Omega} \frac{1}{2}\left[\left(\frac{\partial\psi}{\partial x}\right)^2 + \left(\frac{\partial\psi}{\partial y}\right)^2\right] d\Omega \\ \Pi_2 &\doteq -\frac{1}{2} g \int_0^L z^2(x) dx, \end{aligned} \tag{8.18}$$

where Ω is a variable domain. Figure 8.3 shows the variations of the domain Ω which are measured as displacements in the normal direction to the surface δn. Γ is the variable boundary of the domain.

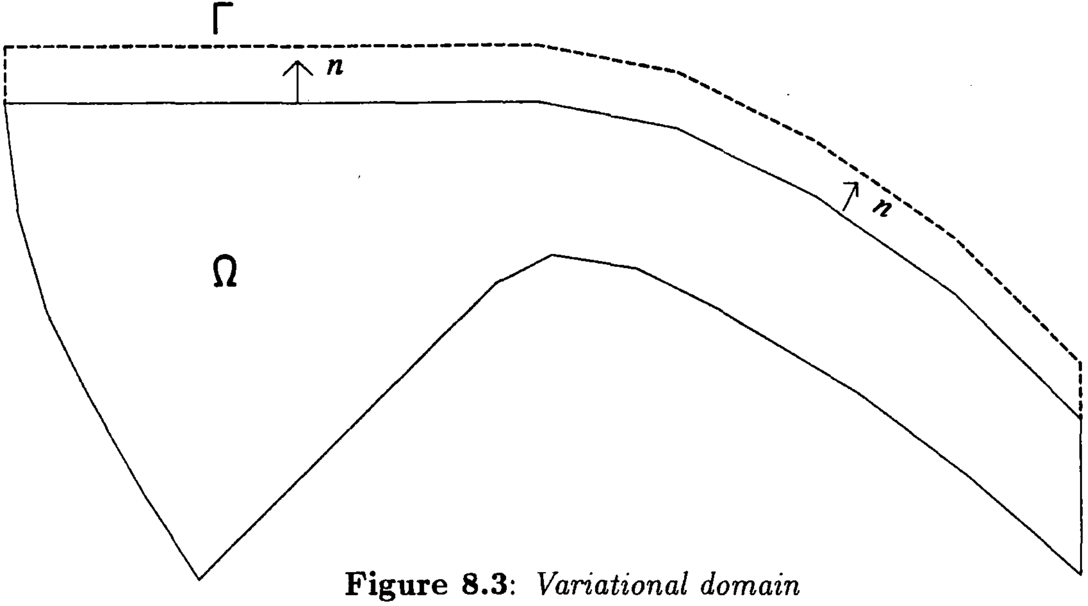


Figure 8.3: *Variational domain*

The term Π_1 relates to the Laplace equation which applies throughout the fluid, hence we call it the volume term. Π_2 is called the surface term as it is associated with the Bernoulli equation applying on the surface.

Finding the stationary points of Π is equivalent to solving both the Laplace equation and the Bernoulli equations given in (8.8). The proof of this statement is given in the following sections.

The proof is based on the formulæ related to variable bound integrals and their variation. Consider the integral G as follows:

$$G = \int\int_{\Omega} F(x, y, f(x, y), f_x(x, y), f_y(x, y))d\Omega, \tag{8.19}$$

where not only the function f is variable but also the limits of the domain Ω may be variable. The variation of G can then be written as follows[23]:

$$\delta G = \int\int_{\Omega} [F]_f \delta f dx dy + \int_{\Gamma} \left(F_{f_x} \frac{\partial x}{\partial n} + F_{f_y} \frac{\partial y}{\partial n} \right) \delta f ds + \int_{\Gamma} F \left(\delta x \frac{\partial x}{\partial n} + \delta y \frac{\partial y}{\partial n} \right) ds, \tag{8.20}$$

where

$$\begin{aligned}
f_x = \frac{\partial f}{\partial x} \qquad & F_{f_x} = \frac{\partial F}{\partial f_x} \\
f_y = \frac{\partial f}{\partial y} \qquad & F_{f_y} = \frac{\partial F}{\partial f_y} \\
\delta n &= \text{variation of the domain } \Omega \\
[F]_f &= \text{Euler functional derivative of F} \\
&= \frac{\partial F}{\partial f} - \left[\frac{\partial}{\partial x} \frac{\partial F}{\partial \frac{\partial f}{\partial x}} + \frac{\partial}{\partial y} \frac{\partial F}{\partial \frac{\partial f}{\partial y}} \right] \\
&= \frac{\partial F}{\partial f} - \left[\frac{\partial}{\partial x} \frac{\partial F}{\partial f_x} + \frac{\partial}{\partial y} \frac{\partial F}{\partial f_y} \right].
\end{aligned} \tag{8.21}$$

8.2.5 Variation of the volume term

The variation of the volume term Π_1 is first derived using the variation formula (8.20) for which f is the streamfunction ψ and the function F can be written as follows using the definitions in equation (8.21):

$$F = \frac{1}{2}(\psi_x^2 + \psi_y^2). \tag{8.22}$$

F is therefore a function of ψ_x and ψ_y only. The Euler functional derivative of F

is then:

$$\begin{aligned}[F_\psi] &= -\frac{1}{2}\left(\frac{\partial}{\partial x}\frac{\partial F}{\partial \psi_x} + \frac{\partial}{\partial y}\frac{\partial F}{\partial \psi_y}\right) \\ &= -\frac{1}{2}\left(\frac{\partial}{\partial x}2\psi_x + \frac{\partial}{\partial y}2\psi_y\right) \\ &= -\left(\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2}\right).\end{aligned} \tag{8.23}$$

The expressions for F_{ψ_x} and F_{ψ_y} are next given:

$$\begin{aligned}F_{\psi_x} &= \frac{\partial F}{\partial \psi_x} = \psi_x \\ F_{\psi_y} &= \frac{\partial F}{\partial \psi_y} = \psi_y.\end{aligned} \tag{8.24}$$

On the surface, the expression of F can be simplified, using the relation established equation (8.9):

$$\begin{aligned}F &= \frac{1}{2}(\psi_x^2 + \psi_y^2) \\ &= \frac{1}{2}\psi_n^2.\end{aligned} \tag{8.25}$$

Taking into account the results established in the equations (8.23), (8.24) and (8.25), the variation of Π_1 can be written as follows:

$$\begin{aligned}\delta\Pi_1 = -\int\int_\Omega \left(\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2}\right)\delta\psi dxdy + \int_\Gamma \left(\frac{\partial \psi}{\partial x}\frac{\partial x}{\partial n} + \frac{\partial \psi}{\partial y}\frac{\partial y}{\partial n}\right)\delta\psi ds \\ + \int_\Gamma \frac{1}{2}\psi_n\left(\delta x\frac{\partial x}{\partial n} + \delta y\frac{\partial y}{\partial n}\right).\end{aligned} \tag{8.26}$$

Further simplification can be brought in by examining the terms in the surface integrals. First consider the second integral of $\delta\Pi_1$. The chain rule can be applied so that:

$$\frac{\partial \psi}{\partial x}\frac{\partial x}{\partial n} + \frac{\partial \psi}{\partial y}\frac{\partial y}{\partial n} = \frac{\partial \psi}{\partial n}. \tag{8.27}$$

The variation of ψ along the surface is constrained as ψ is constant along that line. When the surface moves by a quantity δn along the outer normal n, there is a change in the discharge Q of δQ and a change in ψ of $\delta\psi$:

$$\begin{aligned}\psi_1 &= \psi_0 + \delta\psi \\ Q_1 &= Q_0 + \delta Q,\end{aligned} \tag{8.28}$$

where the indices 0 and 1 represent respectively the values of ψ and Q before and after the movement of the free surface. From equation (8.13), δQ is defined as:

$$\delta Q = V \delta n. \tag{8.29}$$

The velocity on the surface is tangential to the surface and can be expressed in the local (w,n) coordinate system as (see equations (8.7) and (8.8)):

$$V = -\frac{\partial \psi}{\partial n}. \tag{8.30}$$

The boundary conditions impose that ψ is equal to Q on the surface. This implies that:

$$\begin{aligned} \psi_1 &= Q_1 \\ \psi_0 &= Q_0. \end{aligned} \tag{8.31}$$

Combining equations (8.28), (8.29) and (8.30) leads to:

$$\delta\psi = \delta Q = V\delta n = -\frac{\partial \psi}{\partial n}\delta n. \tag{8.32}$$

The second integral in the variation of Π_1 thus becomes:

$$\begin{aligned} \int_\Gamma \left(\frac{\partial \psi}{\partial x}\frac{\partial x}{\partial n} + \frac{\partial \psi}{\partial y}\frac{\partial y}{\partial n} \right) \delta\psi ds &= \int_\Gamma \frac{\partial \psi}{\partial n}\delta\psi ds \\ &= -\int_\Gamma \left(\frac{\partial \psi}{\partial n} \right)^2 \delta n ds \\ &= -\int_\Gamma \psi_n \delta n ds. \end{aligned} \tag{8.33}$$

In the third integral of $\delta\Pi_1$, the expression in parenthesis can be simplified as follows:

$$\delta x \frac{\partial x}{\partial n} + \delta y \frac{\partial y}{\partial n} = \delta n. \tag{8.34}$$

This can be seen from geometric relations shown in Figure 8.4.

Adding all the terms together leads to the following result:

$$\begin{aligned} \delta\Pi_1 &= -\int\int_\Omega \left(\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2} \right) \delta\psi dxdy + \frac{1}{2}\int_\Gamma \psi_n^2 \delta n ds - \int_\Gamma \psi_n^2 \delta n ds \\ &= -\int\int_\Omega \left(\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2} \right) \delta\psi dxdy - \frac{1}{2}\int_\Gamma \psi_n^2 \delta n ds. \end{aligned} \tag{8.35}$$

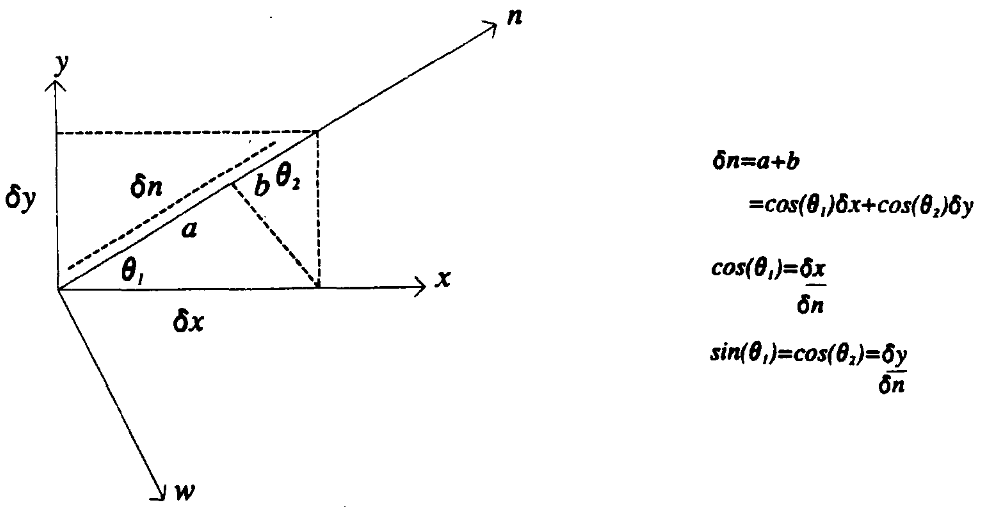


Figure 8.4: *Relation between local and global coordinates*

8.2.6 Variation of the surface term

Now, the variation of Π_2 has to be found. Π_2 is recalled below:

$$\Pi_2 = -\frac{1}{2}g\int_0^L z^2(x)dx. \qquad (recall\ 8.18)$$

The evaluation of the variation involves a change of coordinate system from x to s, where s is the curvilinear coordinate along the free surface, as shown in Figure 8.5

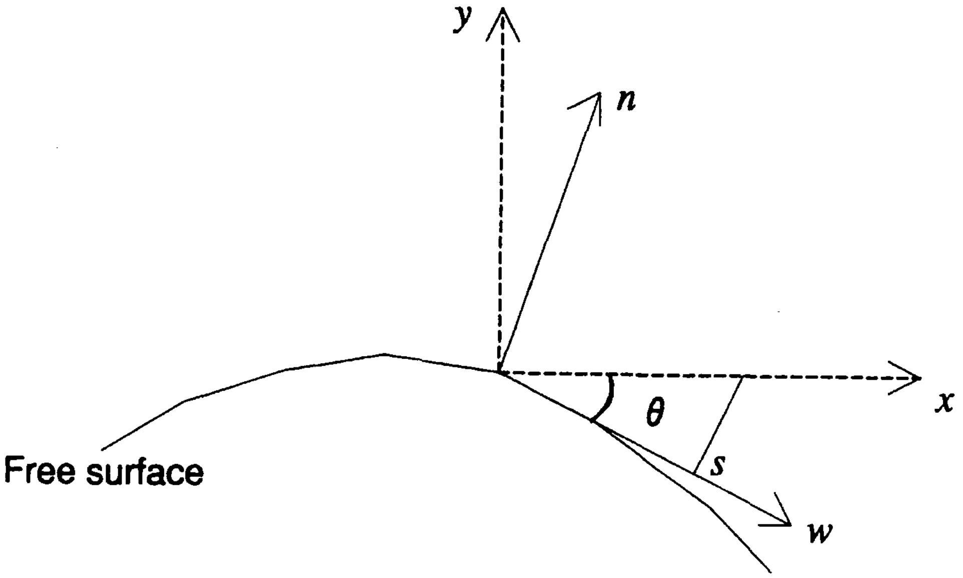


Figure 8.5: *Change of coordinate on the free surface*

The expression of z as a function of the displacement of the surface δn is:

$$z = z_0 - \delta n\cos(\theta), \tag{8.36}$$

where z_0 is the position before the surface moves and θ is the angle between the direction of movement of the surface, that is to say the normal to the surface, and the vertical direction y, as shown in figure 8.4. The integral Π_2 is transformed from the formulation in terms of the cartesian coordinate x to the formulation in terms of the curvilienar coordinate s, as shown in Figure 8.5. This is carried out in order to obtain a formula for the variation of Π_2 compatible with that obtained for the first part of the functional Π_1, whose line integrals were expressed in terms of s rather than x. The relation between the two coordinates, which can be seen in Figure 8.5, is:

$$dx = \frac{ds}{\cos\theta}. \tag{8.37}$$

Changing coordinate system in (8.37) and expressing z explicitly leads to:

$$\Pi_2 = -\frac{1}{2}g\int_\Gamma (z_0 - \delta n\cos(\theta))^2 \frac{ds}{\cos\theta}\delta n. \tag{8.38}$$

The integral Π_2 is a function of n only, therefore its variation is:

$$\begin{aligned}
\delta\Pi_2 &= \frac{\partial \Pi_2}{\partial n}\delta n \\
&= -\frac{1}{2}g\int_\Gamma \frac{\partial}{\partial n}\left((z_0 - \delta n\cos(\theta))^2\right)\frac{ds}{\cos\theta}\delta n \\
&= -\frac{1}{2}g\int_\Gamma 2(z_0 - \delta n\cos(\theta))(-\cos\theta)\frac{ds}{\cos\theta}\delta n \\
&= g\int_\Gamma (z_0 - \delta n\cos(\theta))ds\delta n.
\end{aligned} \tag{8.39}$$

The variation of the surface is therefore:

$$g\int_\Gamma z(s)\delta n ds. \tag{8.40}$$

8.2.7 Final results

Adding all terms of the variation of the functional Π together leads the final expression:

$$\delta\Pi = -\int\int_{\Omega}\left(\frac{\partial^2\psi}{\partial x^2}+\frac{\partial^2\psi}{\partial y^2}\right)\delta\psi dxdy + \int_{\Gamma}(-\frac{1}{2}\psi_n^2 + gz)\delta n ds. \qquad (8.41)$$

Setting $\delta\Pi$ to zero, which is equivalent to finding the stationary points of the functional, clearly leads to the Laplace and Bernoulli equations stated previously equation (8.16).

In the next chapter, the solution of finding the stationary points of Π is explained.

References

1. Craeger W.P., *Engineering of masonary dams*, John Wiley publisher, New York, 1929.
2. Davis C.V., *Handbook of applied hydraulics*, McGraw-Hill, New York, second edition, pp 259–263, 1952.
3. USBR, *Design of small dams*, US Government Printing Office, Washington, 1973.
4. Southwell R.V. and Vaisey G., 'Relaxation methods applied to engineering problems. XII. Fluid motions characterised by 'free' stream-lines', *Phil. Trans. Roy. Soc.*, Land, **240 A**, pp 117–160, 1946.
5. Cassidy J.J., 'Irrotational flow over spillways of finite height', *J. Engrg. Mech. Div., ASCE*, **91**, Iss 6, pp 155–173, 1965.
6. Bettess P. and Bettess J.A, 'Analysis of free surface flows using isoparametric finite elements', *International Journal for Numerical Methods in Engineering*, **19**, pp 1675–1689, 1983.
7. Ikegawa M. and Washizu K., 'Finite element method applied to analysis of flow over a spillway crest', *International Journal for Numerical Methods in Engineering*, **6**, pp 179–189, 1973.
8. Luke J.C, 'A variational principle for a fluid with a free surface', *Journal of Fluid Mechanics*, **27**, Iss 2, pp 395–397, 1967.
9. Betts P.L., 'A variational principle in terms of stream function for free-surface flows and its application to the finite element method', *Computers and fluids*, **7**, pp 145-153, 1979.
10. Diersch H. Schirmer A. and Busch K., 'Analysis of Flows with Initially Unknown Discharge', *Journal of the Hydraulic Division*, **103**, pp 213–232, 1977.

11. Aitchison J.M., 'A variable finite element method for the calculation of flow over a weir', *RL-79-069*, Rutherford Laboratory. 1979.

12. Varoglu E. and Finn W.D.L, 'Variable domain finite element analysis of free surface gravity flow', *Computers and fluids*, **6**, pp 103-114, 1978.

13. Markland E., 'Calculation of flow at a free overfall by relaxation method', *Proc. Inst. Civ. Engrs.*, **31**, pp 71–78, 1965.

14. Williams J.M., 'An integral equation method for the computation of progressive gravity waves of finite height', *HRS Report INT 136*, 1974.

15. Li W., Xie Q. and Chen C.J., 'Finite Analytic Solution of Flow over Spillways', *Journal of Engineering Mechanics*, **115**, Iss 12, pp 2635–2648, 1989.

16. Henderson H.C., Kok M. and De Koning W.L., 'Computer-aided spillway design using the boundary element method and non-linear programming', *International Journal of Numerical Methods in Fluids*, **13**, pp 625–641, 1991.

17. Dias F., Keller J.B. and Vanden-Broeck J., 'Flows over rectangular weirs', *Physics of Fluids*, **31**, Iss 8, pp 2071-2076, 1988.

18. King A.C. and Bloor M.I.G., 'Free-surface flow of a stream obstructed by an arbitrary bed topography', *Quaterly Journal of Mechanics and Applied Mathematics*, **43**, Iss 1., pp 87–106, 1990.

19. Forbes L.K., 'Critical free-surface flow over a semi-circular obstruction', *Journal of Engineering Mathematics*, **22**, pp 3–13, 1988.

20. Forbes L.K., 'Two-layer critical free-surface flow over a semi-circular obstruction', *Journal of Engineering Mathematics*, **23**, pp 325–342, 1989.

21. Forbes L.K., 'An Algorithm for 3-Dimensional Free-Surface Problems in Hydrodynamics', *Journal of Computational Physics*, **82**, pp330–347, 1989.

22. Massey B.S., *Mechanics of Fluids*, Second edition, Van Nostrand Reinhold Company, London, 1970.

23. Courant R. and Hilbert D., *Methods of Mathematical Physics*, Vol. 1, Wiley-Interscience, New-York, 1953.

Chapter IX

Finite element formulation

This chapter concentrates on describing how the governing equations of free surface flow have been solved. The finite element method has been used to find the position of the free surface and the values of the streamfunction. The discretisation of the equations leads to a system of nonlinear equations whose solution has been attempted using iterative methods. The two novel aspects of this work lie in the generation of the element matrices for the finite element formulation using the symbolic language REDUCE and in the parallelisation of their numerical evaluation.

This chapter is organised around the description of the finite element formulation with explanations of the discretisation models and the discretised form of the equations, the organisation of the computer code including the use of REDUCE and the parallelisation of the code. Nonlinear solvers are also considered.

This work is based on an earlier program, written in Algol 68 by Peter and Jackie Bettess[1], which has some common features with the computer code developed here in FORTRAN. Most of the routines dealing with inputting the data in to the program and generating the finite element mesh are a direct translation of the Algol code into FORTRAN code. The routines which perform the calculations to solve the problem are new. The graphics interface which displays the mesh and the streamlines of the flow is an amalgam of graphics routines developed by Noel Hardy, of the Department of Marine Technology in the University of Newcastle-upon-Tyne, of calculation routines for finding out the streamlines written by Peter Bettess and modified and other routines specially written for this work.

The full program for the analysis of the free surface flow is quite large and the problem of testing the code important, especially in view of the various instabilities of both the physical problem and the numerical approximation used. This has lead us to devote a separate chapter to this matter. This constitutes the next chapter.

9.1 Discretisation of the governing equation

The governing equation presented in the previous chapter is recalled below:

$$\begin{aligned}\Pi &= \Pi_1 + \Pi_2 \\ &= \frac{1}{2}\int\int_{\Omega}\left[\left(\frac{\partial\psi}{\partial x}\right)^2 + \left(\frac{\partial\psi}{\partial y}\right)^2\right] d\Omega - \frac{1}{2}g\int_0^L z^2(x)dx.\end{aligned} \qquad (recall\ 8.18)$$

The discretisation of this equation consists of replacing the continuous variables ψ and n by a set of approximation values $\psi_1, \psi_2, \ldots, n_1, n_2, \ldots$ taken at given points of coordinates $(x_1, y_1,), (x_2, y_2), \ldots.$ When using the finite element method, the discretisation points are the nodes which define the mesh of finite elements. The domain where the fluid is studied, Ω, is divided into a finite number of elements as shown in Figure 9.1.

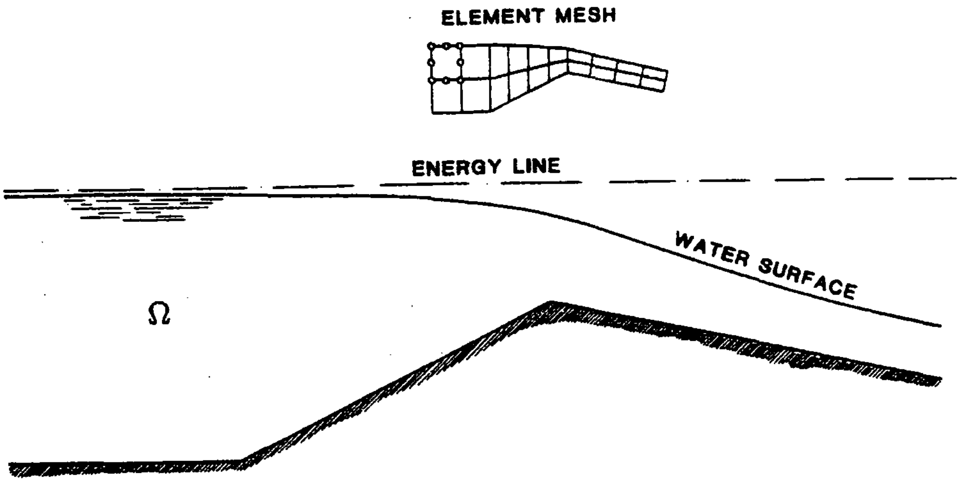


Figure 9.1: *Division of the domain into elements*

This can be expressed by the following equation:

$$\Omega = \sum_{i=1}^{totels} \Omega_i, \qquad (9.1)$$

where *totels* is the total number of elements composing the domain Ω and each Ω_i represents one element. The elements are made up of nodes which are the discretisation points. The continuous variables are replaced by their values at the nodes of the elements and interpolation functions are used to obtain values between the nodes.

The elements used in this work are the eight-noded serendipity elements and the interpolation functions are the quadratic shape functions associated with these elements. The formulæ for these shape functions have been derived in chapter 3. The FORTRAN code used in this chapter is the one generated by the REDUCE program presented in chapter 3.

9.1.1 Discretised form of the volume term

Consider first the volume term Π_1. It can be expressed in a matrix form by introducing the vector B defined as follows:

$$\mathbf{B} = \begin{pmatrix} \frac{\partial \psi}{\partial x} \\ \frac{\partial \psi}{\partial y} \end{pmatrix}. \tag{9.2}$$

The volume term then becomes:

$$\Pi_1 = \frac{1}{2} \int \int_{\Omega} \mathbf{B}^t \mathbf{B} d\Omega. \tag{9.3}$$

The summation property of integrals can be applied to the volume term to give the following equation:

$$\Pi_1 = \sum_{i=1}^{totels} \left(\frac{1}{2} \int \int_{\Omega_i} \mathbf{B}_i^t \mathbf{B}_i d\Omega_i \right), \tag{9.4}$$

where $\mathbf{B}_i$ is the vector $\mathbf{B}$ related to the element i.

The problem is narrowed down to finding the expression of the first order derivatives of the streamfunction in relation to the discretised variables. The basic relation between the approximation and its nodal values is expressed through the shape functions. Given the eight-noded serendipity element as shown in Figure

9.2, the relations are:

$$x = \sum_{i=1}^{8} N_i x_i$$
$$y = \sum_{i=1}^{8} N_i y_i \tag{9.5}$$
$$\psi = \sum_{i=1}^{8} N_i \psi_i,$$

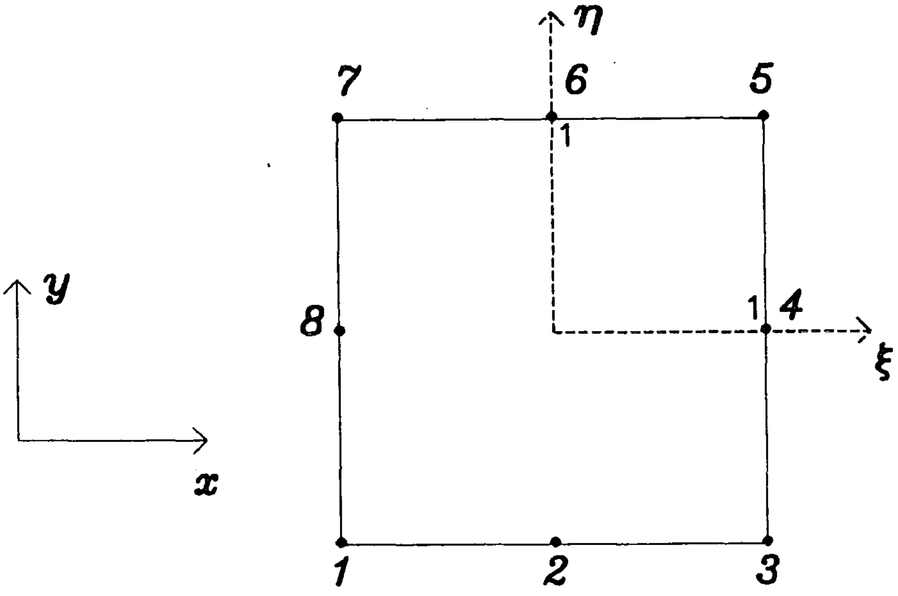


Figure 9.2: *Element and coordinate systems*

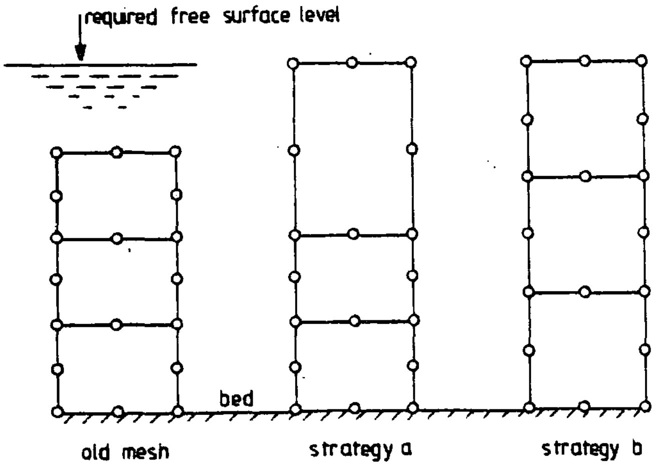
where N_i is the quadratic shape function associated with the node i. It is a function of the local coordinates ξ and η shown in Figure 9.2. The full definition of the shape functions and local and global coordinate systems can be found in chapter 3. Although the discussion in this section is based on the eight-noded serendipity element, it is quite general and the computer code has been developed so that other elements can be used such as higher order elements.

The relations in equation (9.5) assume that the domain defining the flow is fixed. In the case when the flow has a surface which is allowed to move freely, the relations defining x and y are modified to take into account the displacement of the surface nodes.

When the surface nodes move, two approaches can be followed:

1. The surface nodes move but the other nodes in the mesh are fixed (Figure 9.3 *a*)

2. The surface nodes move and the other nodes are moved proportionally, leaving the nodes on the bed unchanged (Figure 9.3 *b*)



a: movement of top nodes only. b: movement of all nodes.

Figure 9.3: *Movements due to the free surface*

The first strategy has the advantage of simplifying the calculations, as the relations in equation (9.5) remain unchanged but it can introduce distortions in the mesh which may lead to inaccurate solution or no solution at all if some of the elements 'flip over' as shown in Figure 9.4. The second strategy makes the finite element formulation more complicated but ensures a regular distribution of the nodes in the mesh which is more likely to lead to a correct solution.

The second strategy has been adopted here. When a surface node is displaced by a distance b all the nodes underneath it are moved proportionally as shown in Figure 9.5. The angle between the direction of displacement and the vertical coordinate y is measured by θ. The component of the displacement in the normal direction n is b_i. Let now consider the movements at element level. In equation (9.5) the values of the x_i and y_i have to be modified to take into account the top node displacements. The modification of the element node number 1,8 and 7 are related to b_1, node number 2 and 6 are related to b_2 and node number 3, 4 and 5

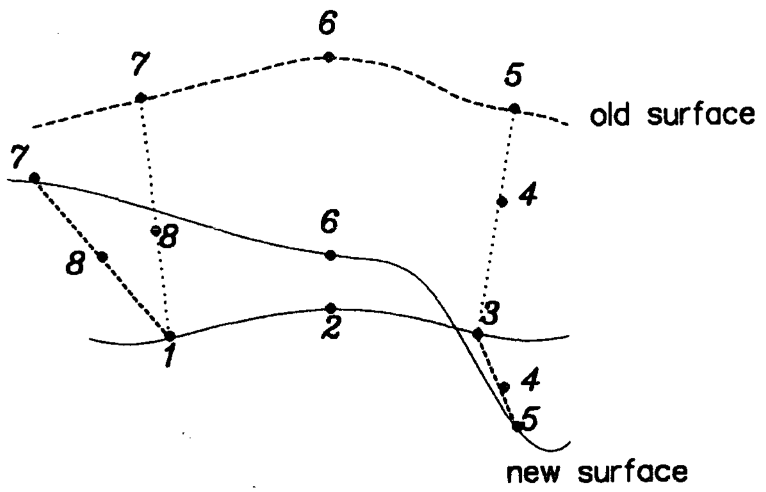


Figure 9.4: *Flip over of the elements*

are related to b_3 (see Figure 9.2 for element node numbers). Simple trigonometric consideration enables us to write:

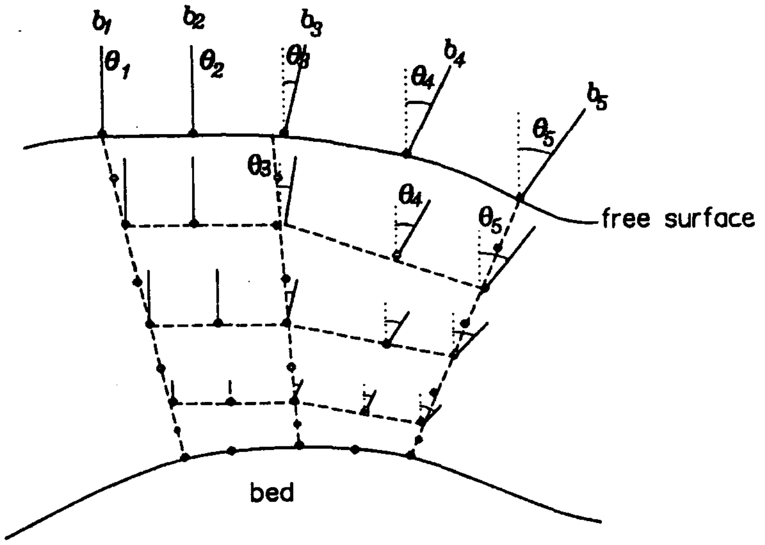


Figure 9.5: *Proportional displacement of the nodes in the mesh*

$$x'_i = x_i + \alpha_i \sin(\theta_j) b_j, \tag{9.6}$$

where x'_i is the new value of the x position for node i after taking into consideration the displacement of the corresponding surface node j. α_j is the proportional amount of displacement of node i in the element. An example of the values of α_i

α_9	1
α_8	0.875
α_7	0.75
α_6	0.625
α_5	0.50
α_4	0.375
α_3	0.25
α_2	0.125
α_1	0

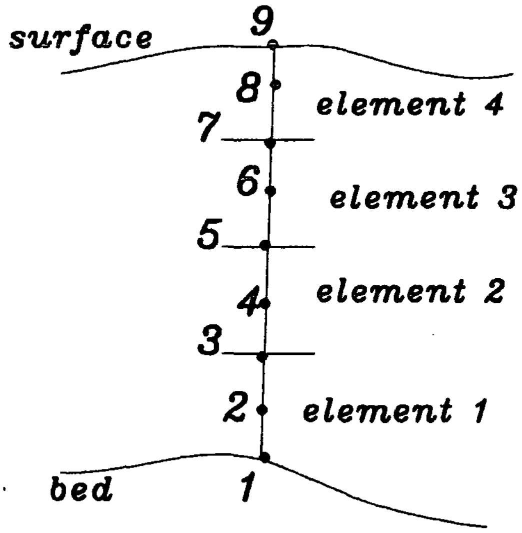


Figure 9.6: *Example of displacement values*

is given in Figure 9.6.

In equation (9.5), x_i is thus replaced by x'_i. A similar relation holds for y_i where β and $\cos(\theta)$ are used in place of α and $\sin(\theta)$. Equation (9.5) can then be reformulated in a matrix form as follows. Denoting by $\mathbf{N}$ the vector of shape functions such that:

$$\mathbf{N} = (\,N_1,\;\; N_2,\;\; N_3,\;\; N_4,\;\; N_5,\;\; N_6,\;\; N_7,\;\; N_8\,)\,, \tag{9.7}$$

and defining the three matrices representing the movement of the nodes as:

$$\mathbf{L}_1 = \begin{pmatrix} \alpha_1\sin(\theta_1) & \beta_1\cos(\theta_1) \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ \alpha_7\sin(\theta_1) & \beta_7\cos(\theta_1) \\ \alpha_8\sin(\theta_1) & \beta_8\cos(\theta_1) \end{pmatrix} \qquad \mathbf{L}_2 = \begin{pmatrix} 0 & 0 \\ \alpha_2\sin(\theta_2) & \beta_2\cos(\theta_2) \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ \alpha_6\sin(\theta_2) & \beta_6\cos(\theta_2) \\ 0 & 0 \\ 0 & 0 \end{pmatrix}$$

$$\mathbf{L_3} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ \alpha_3\sin(\theta_3) & \beta_3\cos(\theta_3) \\ \alpha_4\sin(\theta_3) & \beta_4\cos(\theta_3) \\ \alpha_5\sin(\theta_3) & \beta_5\cos(\theta_3) \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}, \tag{9.8}$$

the equation (9.5) can be expressed in matrix form as:

$$(x, y) = \mathbf{N}(\mathbf{x} + \mathbf{L_1}b_1 + \mathbf{L_2}b_2 + \mathbf{L_3}b_3), \tag{9.9}$$

where (x,y) is the vector of the continuous variables and $\mathbf{x}$ is the matrix of the discretised variables as defined below:

$$\mathbf{x} = \begin{pmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \\ x_4 & y_4 \\ x_5 & y_5 \\ x_6 & y_6 \\ x_7 & y_7 \\ x_8 & y_8 \end{pmatrix}. \tag{9.10}$$

The relation in equation (9.5) for ψ remains unchanged.

Coming back to the original aim of calculating $\mathbf{B}$, the first order derivatives of ψ need to be found. From equation (9.5), the following relations can be derived:

$$\begin{aligned} \frac{\partial\psi}{\partial x} &= \sum_{i=1}^{8} \frac{\partial N_i}{\partial x}\psi_i \\ \frac{\partial\psi}{\partial y} &= \sum_{i=1}^{8} \frac{\partial N_i}{\partial y}\psi_i, \end{aligned} \tag{9.11}$$

where the ψ_i are constant values and N_i is a function of x and y. More precisely, N_i is a direct function of ξ and η, the local coordinates and an indirect function of x and y. The chain rule is used to transform equation (9.11) as follows:

$$\begin{aligned}\frac{\partial N_i}{\partial x} &= \frac{\partial N_i}{\partial \xi}\frac{\partial \xi}{\partial x} + \frac{\partial N_i}{\partial \eta}\frac{\partial \eta}{\partial x}\\ \frac{\partial N_i}{\partial y} &= \frac{\partial N_i}{\partial \xi}\frac{\partial \xi}{\partial y} + \frac{\partial N_i}{\partial \eta}\frac{\partial \eta}{\partial y},\end{aligned} \tag{9.12}$$

the equations above can be reformulated in matrix terms as follows:

$$\begin{pmatrix}\frac{\partial N_i}{\partial x}\\ \frac{\partial N_i}{\partial y}\end{pmatrix} = \begin{pmatrix}\frac{\partial \xi}{\partial x} & \frac{\partial \eta}{\partial x}\\ \frac{\partial \xi}{\partial y} & \frac{\partial \eta}{\partial y}\end{pmatrix}\begin{pmatrix}\frac{\partial N_i}{\partial \xi}\\ \frac{\partial N_i}{\partial \eta}\end{pmatrix}. \tag{9.13}$$

The two by two matrix is traditionally called the inverse of the Jacobian matrix and denoted $\mathbf{J}^{-1}$. It contains the relation between the local and the global coordinates. The expression for J^{-1} is obtained by inverting the expression for J which is defined as:

$$J = \begin{pmatrix}\frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi}\\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta}\end{pmatrix}. \tag{9.14}$$

From equation (9.9), the formula for J can be derived as:

$$\begin{aligned}J &= \begin{pmatrix}\frac{\partial}{\partial \xi}(x,y)\\ \frac{\partial}{\partial \eta}(x,y)\end{pmatrix}\\ &= \begin{pmatrix}\frac{\partial}{\partial \xi}(\mathbf{N}(\mathbf{x}+\mathbf{L}_i b_i))\\ \frac{\partial}{\partial \eta}(\mathbf{N}(\mathbf{x}+\mathbf{L}_i b_i))\end{pmatrix}\\ &= \begin{pmatrix}\frac{\partial}{\partial \xi}\mathbf{N}\\ \frac{\partial}{\partial \eta}\mathbf{N}\end{pmatrix}(\mathbf{x}+\mathbf{L}_i b_i)\\ &= \nabla\mathbf{N}(\mathbf{x}+\mathbf{L}_i b_i),\end{aligned} \tag{9.15}$$

where $\mathbf{L}_i b_i$ uses the summation convention $(\sum_{i=1}^{3}\mathbf{L}_i b_i)$ and $\nabla\mathbf{N}$ is the matrix of the shape functions derivatives as shown below:

$$\nabla\mathbf{N} = \begin{pmatrix}\frac{\partial N_1}{\partial \xi} & \frac{\partial N_2}{\partial \xi} & \frac{\partial N_3}{\partial \xi} & \frac{\partial N_4}{\partial \xi} & \frac{\partial N_5}{\partial \xi} & \frac{\partial N_6}{\partial \xi} & \frac{\partial N_7}{\partial \xi} & \frac{\partial N_8}{\partial \xi}\\ \frac{\partial N_1}{\partial \eta} & \frac{\partial N_2}{\partial \eta} & \frac{\partial N_3}{\partial \eta} & \frac{\partial N_4}{\partial \eta} & \frac{\partial N_5}{\partial \eta} & \frac{\partial N_6}{\partial \eta} & \frac{\partial N_7}{\partial \eta} & \frac{\partial N_8}{\partial \eta}\end{pmatrix}. \tag{9.16}$$

Combining the equations (9.11), (9.13) and (9.15) the vector $\mathbf{B}$ can now be fully expressed in function of the discretised variables of the problem in a matrix form:

$$\mathbf{B} = (\mathbf{J}^{-1})(\nabla\mathbf{N})\mathbf{P}, \tag{9.17}$$

where $\mathbf{P}$ is the vector containing the ψ_i values as shown below:

$$\mathbf{P} = \begin{pmatrix} \psi_1 \\ \psi_2 \\ \psi_3 \\ \psi_4 \\ \psi_5 \\ \psi_6 \\ \psi_7 \\ \psi_8 \end{pmatrix}. \tag{9.18}$$

In the expression for Π_1 given in equation (9.4), the element domains Ω_i are expressed in terms of the global coordinates (x,y). This needs to be changed as the vector $\mathbf{B}$ is expressed in function of the local coordinates (ξ,η). The known formula related to transformation of coordinates in multidimensional integrals can be applied here:

$$d\Omega_i = \det\mathbf{J} d\xi d\eta, \tag{9.19}$$

where $\mathbf{J}$ is the Jacobian matrix as defined previously and (ξ,η) are the local coordinates related to element i. The volume term can therefore finally be expressed as:

$$\Pi_1 = \frac{1}{2}\int_{-1}^{1}\int_{-1}^{1} \mathbf{P}^t(\nabla\mathbf{N})^t(\mathbf{J}^{-1})^t\mathbf{P}(\nabla\mathbf{N})(\mathbf{J}^{-1})\det\mathbf{J} d\xi d\eta. \tag{9.20}$$

From the equation above, it is important to identify the dependence on the discretised variables of each term since the first order derivatives of Π_1 need to be evaluated. The main point is that the volume term is a linear function of the variable ψ and a nonlinear function of the variable n. In terms of discretised variables, Π_1 is a linear function of the ψ_i's and a nonlinear function of the b_i's. The nonlinearity comes from the term J^{-1}. The Jacobian matrix is a linear function of the b_i's as shown in equation (9.15). Therefore its inverse is a function of $1/b_i$ which is nonlinear. This implies that the calculation of the first order derivative of Π_1 with respect to n, or b_i in the discretised form, is not straightforward.

In the original work done by Peter and Jackie Bettess, the nonlinear term in b_i was developed in a Taylor series which was truncated at the second order for reasons we will discuss later. The novelty of the work done in this chapter is that the Computer Algebra system REDUCE has been used to obtain the exact analytical expression for the derivatives of the nonlinear term. The integration is still, however, carried out numerically because there is no exact analytical integrand to the differentiated nonlinear term, as it will be shown in a later section.

Even without considering the surface term of the functional Π_2, it appears that the system of equations which arises from the calculation of the stationary points of the functional is nonlinear. Indeed, the equation to be solved is:

$$\delta\Pi = 0, \tag{9.21}$$

which is equivalent in terms of the discretised variables to:

$$\begin{aligned} \frac{\partial \Pi}{\partial \psi_i} &= 0 \qquad i = 1, totnod \\ \frac{\partial \Pi}{\partial b_j} &= 0 \qquad j = 1, totsnd, \end{aligned} \tag{9.22}$$

where *totnod* is the total number of nodes in the mesh and *totsnd* is the total number of nodes on the surface of the flow. As Π is nonlinear in terms of b_j, it implies that the equations in (9.17) are nonlinear.

In the original work, the Newton-Raphson method was used to solve the nonlinear equations. This is why the Taylor series development was truncated after the second order term as the Newton method only requires the knowledge of the first order derivatives.

Before going any further in the derivation of the equations for the volume term, the discretised form of the surface term is obtained.

9.1.2 Discretised surface term

The equation for the surface term is recalled below:

$$\Pi_2 = -\frac{1}{2}\int_0^L z^2(x)dx. \tag*{(recall 8.18)}$$

A similar methodology to the one used for the volume term is followed here. This time, though, the discretisation is carried out along the line of the surface instead of in the two dimensional volume. This means that the interpolation functions used are different. They are the one dimensional shape function for three-noded elements, which directly correspond to the one dimensional Lagrange polynomials shown in chapter 3.

The line corresponding to the free surface of the flow is divided into one dimensional elements as shown in Figure 9.7.

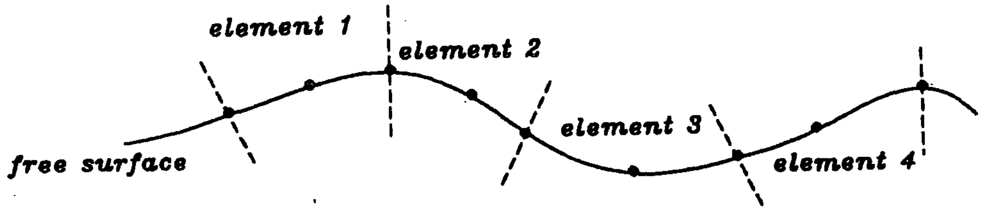


Figure 9.7: *Discretisation of the free surface*

The discretisation formulæ are therefore:

$$
\begin{aligned}
x &= \sum_{j=1}^{3} M_i x_i \\
z &= \sum_{j=1}^{3} M_i z_i,
\end{aligned}
\tag{9.23}
$$

where M_i denote the three-noded one dimensional shape functions. Although the equations are given for three-noded one dimensional elements, which correspond to the number of nodes in the x direction for the eight-noded two dimensional elements, they can be extended to any number of nodes by simply replacing the shape functions by the appropriate ones for the chosen number of nodes.

The movement of the nodes on the surface implies that the relations in equation (9.24) need to be altered. The alteration for the x coordinate is the same as the one stated in equation (9.9) where only the x direction is taken into account. Denoting the x component of the $\mathbf{L}_i$ matrices $\mathbf{R}_i$, and $\mathbf{M}$ the vector of the one dimensional shape functions, the following equation holds:

$$
x = \mathbf{M}(\mathbf{x} + \mathbf{R}_1 b_1 + \mathbf{R}_2 b_2 + \mathbf{R}_3 b_3). \tag{9.24}
$$

From the formula above, the relationship between the global x variable and the local ξ variable describing the position along the free surface can be derived for the discretised problem. As it is a one dimensional problem, x is a function of ξ only. Therefore the variation of x can be written as:

$$dx = \frac{\partial x}{\partial \xi} d\xi. \tag{9.25}$$

Using equation (9.25) where only $\mathbf{M}$ is a function of ξ, and adopting the summation convention leads to:

$$\begin{aligned} dx &= \frac{\partial}{\partial \xi}\left(\mathbf{M}(\mathbf{x} + \mathbf{R}_i b_i)\right)\xi \\ &= \nabla\mathbf{M}(\mathbf{x} + \mathbf{R}_i b_i)d\xi, \end{aligned} \tag{9.26}$$

where $\nabla\mathbf{M}$ is the vector of the shape function derivatives. The surface term can now be reformulated in terms of the local variable ξ and discretised variables as shown in the next equation :

$$\begin{aligned} \Pi_2 &= -\frac{1}{2}\int_0^L z^2(x)dx \\ &= -\frac{1}{2}\sum_{i=1}^{totsel}\int_{l_i}^{l_{i+1}} z^2(x)dx \\ &= -\frac{1}{2}\sum_{i=1}^{totsel}\int_{-1}^{1} z^2(x(\xi))\nabla\mathbf{M}(\mathbf{x} + \mathbf{R}_i b_i)d\xi, \end{aligned} \tag{9.27}$$

where *totsel* denotes the total number of elements along the surface, l_i and l_{i+1} are the bounds of element i. Taking into account the movement of the surface nodes z can be expressed in the discretised form as:

$$\begin{aligned} z(x(\xi)) &= \sum_{i=1}^{3} M_i z_i \\ &= \sum_{i=1}^{3} M_i(z_{0i} - \cos(\theta_i)b_i), \end{aligned} \tag{9.28}$$

where z_{0i} is the position of the surface before it moves and θ_i is the angle between the direction of movement and the vertical direction y. This is shown in Figure 9.8.

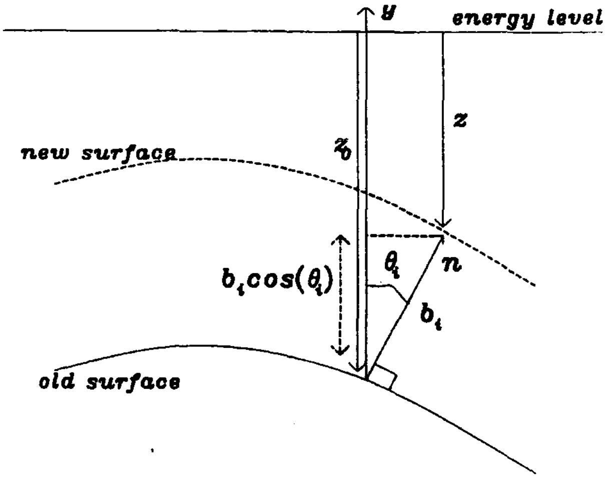


Figure 9.8: *Measurement of the movement of the surface*

The surface term can finally be expressed in matrix notation as:

$$\Pi_2 = -\frac{1}{2}g \sum_{j=1}^{totsel} \int_{-1}^{1} (\mathbf{M}(\mathbf{z}_0 - \cos(\theta_j)b_j)^2 \nabla \mathbf{M}(\mathbf{x} + \mathbf{R}_i b_i) d\xi, \tag{9.29}$$

where $\mathbf{z}_0$ is the vector of the free surface position before movement. Analysing the expression for Π_2 above leads to the conclusion that it is a function of the b_i's only and that the relationship is linear. Contrary to the volume term, the Jacobian rather than its inverse appears in Π_2 which is a linear function of the b_i's as shown in equation (9.24). The integrand in equation (9.29) is therefore a polynomial which can analytically be integrated. This has been carried out using REDUCE as explained in a later section.

9.2 Nonlinear solvers

As it has been shown in the previous section, the system of equations which arises from the discretisation of the flow's governing equations is nonlinear. A solver for nonlinear equations is used. Such solvers are based on a linearisation technique combined with an iteration scheme. This usually means that a set of iteration is carried out for which a system of linear equations has to be solved at each step.

The first method considered in this work is the Newton-Raphson method extended for a multivariable function. Further on, the problems encountered in the convergence of the algorithm has led us to consider both alternative linear and nonlinear solvers. The reason and nature of these investigations will be given in the next chapter concerned with testing. In the following sections the theory for the various nonlinear solvers is given.

9.2.1 Multidimensional Newton method

The one dimensional Newton-Raphson method, which will be called the Newton method, is concerned with finding the zeros of a function $f(x)$. It is an iterative method which requires the guess of a starting point. This starting point can in theory be chosen anywhere in the x space, but in practice the closer the starting point is from the solution the better the algorithm converges. Obviously, when there is more than one solution, the choice of a starting point will determine to which solution the algorithm converges. In one dimensional problems, the method normally works well providing the function f is continuous.

From a starting point x_0 the next point in the iteration is found by the following formula:

$$x_{k+1} = x_k - \Delta x, \tag{9.30}$$

where

$$\Delta x = \frac{f(x_k)}{\frac{df(x_k)}{dx}}. \tag{9.31}$$

This is illustrated in Figure 9.9.

One relevant application of this method to the problem considered in this chapter is that of finding the stationary points of a function. Although we are concerned with the stationary points of a multivariable function, it is simpler to explain the method in the one dimensional case first.

When looking for stationary points, the conditions is that the first derivative of the function is zero, which can be expressed as:

$$\frac{df}{dx} = 0. \tag{9.32}$$

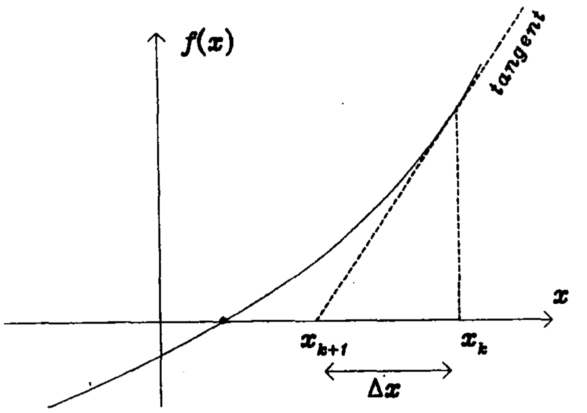


Figure 9.9: *One dimensional Newton method*

The conditions of validity of this equation include minimum, maximum and points of inflexions of the function f. This is better expressed by the phrase 'turning points'. In the problem of free surface flows, given the conditions of steady state to be achieved, the minimum is probably what is sought, although strictly speaking, there is nothing in the equations that suggests the condition for a minimum is present. The next chapter, related to testing, investigates in more detail this topic. In the remaining of this section the theory for finding turning points is presented.

Using the Newton method, equation (9.32) can be solved. Finding the turning points of the function f is equivalent to finding the zeros of its first derivative which can be calculated using the Newton iterative formula as follows. Starting from a guess of the solution x_0 the increment Δx is obtained as:

$$\begin{aligned}\Delta x &= \frac{\frac{df}{dx}(x_k)}{\frac{d^2f}{dx^2}(x_k)} \\ x_{k+1} &= x_k - \Delta x.\end{aligned} \tag{9.33}$$

The distinction between maximum, minimum and points of inflexion comes from the sign of the second order derivative. If d^2f/dx^2 is strictly positive, the turning point is a minimum, if it is strictly negative a maximum is found and if it is zero a point of inflexion is reached.

The conditions of convergence of this method are that either the value of the function df/dx is close enough to zero or that the series of the approximation to the solution x_k converges towards a constant value a. This can be expressed as:

$$\Delta x = x_{k+1} - x_k = a + \epsilon - a = \epsilon, \tag{9.34}$$

where ϵ is defined as the tolerance for convergence. This comes from the fact that only a finite number of iteration is practical to implement and therefore the calculations are stopped when the accuracy of the solution is sufficient.

The Newton method converges quadratically, which means that at each iteration the number of correct decimal points in x_k doubles. This comes from the fact that it is based on a Taylor's series expansion about the solution truncated at the second order. It also means that it is not very effective unless the current guess x_k is near to the solution when the truncation does not bring too much inaccuracy.

The drawback of the method, though, is that it is expensive as the evaluation of both the first and second order derivatives has to be carried out. This problem is mostly crucial for multivariable functions solved on a serial computer. With the advent of parallel processing, the cost of this method may decrease as the amount of calculation to perform is not so penalizing. The advantage of using simpler but less accurate methods may then be less obvious.

The extension of the formula to n dimensions is given[2] next. The function considered is denoted $f(x_1, x_2, \ldots, x_n)$ where x_i is the variable in the ith direction. The formulæ (9.31) and (9.32) can simply be extended to the n variable case as:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \Delta\mathbf{x}, \tag{9.35}$$

where

$$\begin{aligned} \Delta\mathbf{x} &= \frac{\frac{df(\mathbf{x}_k)}{dx}}{\frac{d^2 f(\mathbf{x}_k)}{d\mathbf{x}^2}} \\ \mathbf{x} &= (x_1, x_2, \ldots, x_n). \end{aligned} \tag{9.36}$$

The variable x is replaced by the vector of variables $\mathbf{x}$. The first and second order derivatives are the vector and matrix defined below:

$$\frac{df}{d\mathbf{x}} = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \\ \frac{\partial f}{\partial x_i} \\ \\ \frac{\partial f}{\partial x_n} \end{pmatrix} \quad \textit{and} \quad \frac{d^2 f}{d\mathbf{x}^2} = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \frac{\partial^2 f}{\partial x_1 \partial x_i} & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \frac{\partial^2 f}{\partial x_2 \partial x_i} & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \\ \frac{\partial^2 f}{\partial x_i \partial x_1} & \frac{\partial^2 f}{\partial x_i \partial x_2} & \frac{\partial^2 f}{\partial x_i^2} & \frac{\partial^2 f}{\partial x_i \partial x_n} \\ \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \frac{\partial^2 f}{\partial x_n \partial x_i} & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix}. \tag{9.37}$$

The vector of the first order derivatives is denoted $\mathbf{g}$ and the matrix of the second order derivatives is denoted $\mathbf{G}$. Equation (9.31) can be reformulated in term of these matrices as:

$$\Delta\mathbf{x} = \mathbf{G}^{-1}\mathbf{g}(\mathbf{x}_k). \tag{9.38}$$

In the above equation the matrix multiplication is implicitly denoted as an ordinary multiplication (no sign). Equation (9.38) is very useful in showing where the system of linear equations arises. The expression $\mathbf{G}^{-1}\mathbf{g}$ literally corresponds to solving a system of linear equation where the system matrix is $\mathbf{G}$ and the right hand side vector is $\mathbf{g}$ as shown below:

$$\mathbf{G}\Delta\mathbf{x} = \mathbf{g}. \tag{9.39}$$

The vector $\Delta\mathbf{x}$ can also be interpreted geometrically as it corresponds to a 'search direction' in which the solution is to be found. This aspect is discussed in greater detail in the two following sections.

The condition of convergence in the n-variable case is:

$$\Delta\mathbf{x} = \mathbf{x}_{k+1} - \mathbf{x}_k = \mathbf{a} + \epsilon - \mathbf{a} = \epsilon, \tag{9.40}$$

where ϵ is the vector of the errors on the first derivatives of the function f and $\mathbf{a}$ is the vector solution.

The distinction between minimum, maximum and points of inflexions for the n-variable case is made by considering the properties of the matrix $\mathbf{G}$. If the matrix $\mathbf{G}$ is positive definite then the solution found is a minimum. If it is negative definite the solution found is a maximum and if it is indefinite a point of inflexion is obtained, which is called in n dimensions a saddle point.

9.2.2 Line of steepest descent method

The line of steepest descent is based on a similar principle to the Newton method. From a starting point, an increment $\Delta\mathbf{x}$ is calculated and the method iterates until the solution is reached. The increment represents the direction of search which is different from the Newton one. It is defined as:

$$\Delta\mathbf{x} = \mathbf{g}(\mathbf{x}_k). \tag{9.41}$$

Geometrically, it can be interpreted as the vector normal to the hyperplane defined by $f(\mathbf{x}_k)$=*constant* at the point k. It is difficult to visualize this in the general case of n dimensions, so the example of a two variable function f is shown in Figure 9.10.

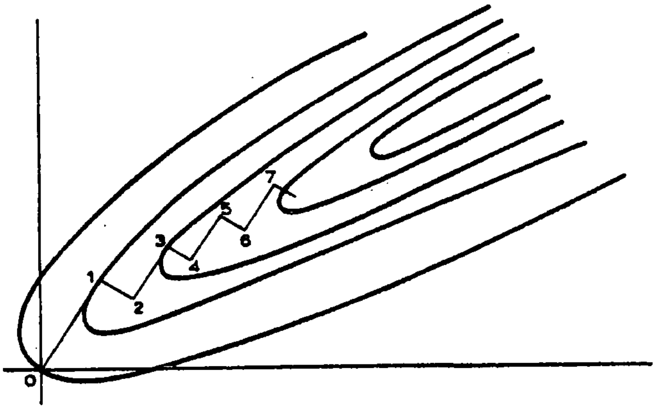


Figure 9.10: *Two dimensional steepest descent method*

This method ensures that the vector of search is always pointing in the direction of a minimum. Intuitively, the direction of search is always pointing downhill, hence the name of the method. This means that this method cannot find a saddle point. A maximum can easily be obtained by multiplying all the equations by -1. However, if there are local minimum on the way to the absolute minimum, the method will find these local minimum and stop there. This is dependent on the starting point chosen and, in this respect, the steepest descent method behaves similarly to the Newton method.

The condition of convergence of the method is similar to Newton in that the solution is reached when either the length of the search vector becomes smaller than a chosen tolerance or the approximation to the solution $\mathbf{x}_k$ converges within a certain tolerance.

The method is less costly than the Newton method in that at each step it only requires the evaluation of the first order derivatives of the function f. On the other hand it takes more iterations to converge and is less accurate.

The principle differences in the behaviour of the two methods is that the steepest descent method works well far from the solution and that the Newton method converges faster and with better accuracy when near to the solution. For this reason, the steepest descent method can be used to obtain the first guess to the Newton method. Practically, the iterations start with the search direction given by the steepest descent vector, which is the vector of the first order derivatives, and then switches to the direction given by the Newton method when close enough to the solution. The difficulty then is to find criteria to characterize the point when to switch from one method to the other.

A way of finding such criteria is to measure the angle between the directions of search predicted by each method. If this angle is bigger than a given value one method is used, otherwise the other method is used. This can be formally expressed as:

```
if $\theta > \frac{\pi}{2} - K$ then
    use Steepest descent
else
    use Newton
end if
```

Figure 9.11: *Algorithm for switching from steepest descent to Newton*

where the value of K has to be determined. Depending on the problem treated, K will vary, therefore its value is found by experimentation for each particular problem. An intuitive interpretation of this can be that far from the solution the steepest descent method gives a more realistic search direction whereas the Newton method might point out in a completely wrong direction. When nearer to the solution the two vectors would be pointing roughly in the same direction as

Newton becomes more accurate. This means that far from the solution the angle between the two vectors is greater than when close to it.

9.2.3 Line search improvement

A way of speeding up the convergence of both methods described above is to use a line search. The principle of the line search algorithm is to search for a minimum along the predicted direction. When using the steepest descent or the Newton method the distance of travel from the previous approximation of the solution x_k to the next approximation x_{k+1} is fixed and equal to the norm of the vector defining the search direction. This is illustrated in Figure 9.12.

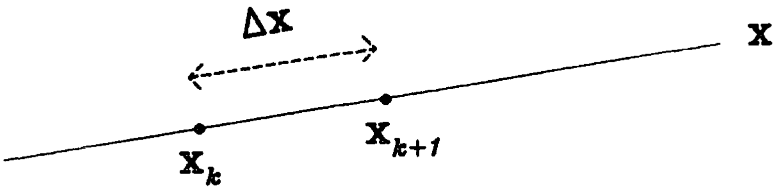


Figure 9.12: *Fixed distance along the search direction*

When performing a line search the next approximation for the solution x_{k+1} is allowed to be placed anywhere along the search direction, even in the opposite direction. This is shown in Figure 9.13.

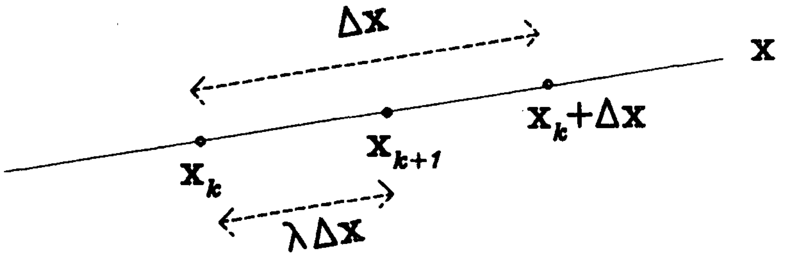


Figure 9.13: *Line search method*

This can formally be expressed as minimising the function $f(\mathbf{x} + \lambda\Delta\mathbf{x})$ where λ is allowed to take any positive or negative value. The steepest descent and Newton methods correspond to the case when λ=1. The problem is that of a one-dimensional minimisation where λ is the variable.

Various methods can be used to solve this problem and consideration must be given to the amount of time spent to carry out the minimisation. The aim of the line search method is to reduce the number of iterations, therefore as little time as possible should be spent on executing it. The simplest technique is to use the success and failure algorithm. Other methods include the Bisection method, the method of linear interpolation and the one dimensional Newton method[3]. The advantage of the success and failure method is that is only involves the evaluation of the function f at each step.

The idea behind the line search is to ensure that between each step of the method chosen (steepest descent or Newton) an improvement has been made. This improvement corresponds to a monotonic decrease or increase of the function whose turning points are to be found. The improvement is chosen to be increasing for maximum and decreasing for minimum. Since the two schemes are incompatible, a prior knowledge of the nature of the turning point sought (maximum or minimum) is needed in order to use a line search improvement.

For the free surface flow problem, such a knowledge is not available, although physically a minimum is more likely than a maximum. For reasons explained in the next chapter, a minimum was expected and the method for the line search in this case is given next, although its amendment for the maximum search would be straightforward.

The algorithm is iterative and needs a guessed starting point. A possible starting point would be to set $\lambda = 1$, which means that the search would start around the originally predicted next approximation to the solution x_{k+1}. The success and failure algorithm explores by a quantity $d\lambda$ the direction of search from the starting point. If the value of the function f at that point has decreased, the search continue in the same direction with the value $\lambda + 2d\lambda$. If not the search goes back on step and start again in the other direction with the value $\lambda - d\lambda/4$. This is shown in Figure 9.14.

As the main reason for using the line search is to speed up the corresponding iterative method, there is no need for having a high accuracy of the minimum of the function f along the direction of search. Moreover, unless the iterative algorithm is already near the solution, the direction of search is probably incorrect and a

```
if $f(\mathbf{x} + (\lambda + d\lambda)\Delta\mathbf{x}) < f(\mathbf{x} + \lambda\Delta\mathbf{x})$ then
    $d\lambda = 2d\lambda$
    $\lambda = \lambda + d\lambda$
else
    $d\lambda = -d\lambda/4$
end if
```

Figure 9.14: *Line search technique for minimum*

minimum on that line might be quite remote from the solution sought. This is illustrated in Figure 9.15.

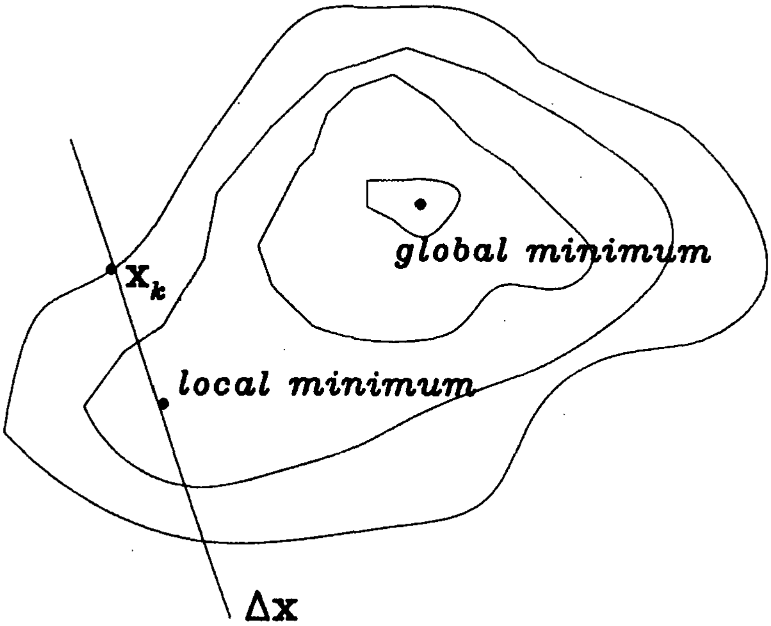


Figure 9.15: *Local improvement v. global improvement*

There are several ways for restricting the number of iterations for the line search. The simplest is to set a high tolerance when testing if the value of $d\lambda$ is changing from one step to another. More complicated schemes consist of setting up bounds within which the iterations are stopped. Although these techniques have been considered, they have not been implemented so no description is given here apart from their name: the Golstein and Wolfe conditions[4].

9.3 Formation of the element matrices: Use of REDUCE

When using the Newton method for solving the nonlinear equations arising from the term Π_1 of the functional, the vector $\mathbf{g}$ and the matrix $\mathbf{G}$ have to be evaluated at each step. This section concentrates on giving the equations for $\mathbf{g}$ and $\mathbf{G}$ and explaining how this has been obtained using REDUCE.

9.3.1 Equations for the Newton method

The governing equation for the free surface flow has been expressed in previous sections as the turning points of the functional Π. Applying the multidimensional Newton method to Π means that the vector $\mathbf{g}$ and the matrix $\mathbf{G}$ have to be derived. The discretised form of Π is used and the vector $\mathbf{x}$ of the Newton method is defined as follows:

$$\mathbf{x} = (\psi_1, \psi_2, \ldots, \psi_{totnod}, b_1, b_2, \ldots, b_{totsnd}), \tag{9.42}$$

where *totnod* is the total number of nodes in the mesh and *totsnd* is the total number of nodes on the free surface. The functional is expressed as the sum on all the elements of the local functionals as:

$$\begin{aligned} \Pi &= \sum_{k=1}^{totels} \Pi_k \\ \Pi_k &= \int_{-1}^{1}\int_{-1}^{1} \mathbf{P}^t(\nabla\mathbf{J}^{-1})^t(\mathbf{J}^{-1})^t(\mathbf{J}^{-1})(\nabla\mathbf{N})\mathbf{P} det\mathbf{J} d\xi d\eta \\ &\quad - \frac{1}{2}\int_{-1}^{1} (\mathbf{M}(\mathbf{z}_0 - \cos(\theta_i)b_i))^2 \, \nabla\mathbf{M}(\mathbf{x} + \mathbf{R}_i b_i) d\xi. \end{aligned} \tag{9.43}$$

This decomposition by element is possible because Π is composed of integrals which can be expressed as the sum of their local values in each element. In order to unify the unknowns ψ_i's and b_i's a 'modified' eight-noded serendipity element is used, where three extra nodes are added at the top of each element as shown in Figure 9.16.

These extra nodes have the same (x,y) coordinates as the associated surface nodes but in place of having ψ as unknown, they have b as unknown. All elements in a column have the same extra three nodes. When forming the $\mathbf{G}_k$ matrix and the $\mathbf{g}_k$ vector for each element, the dimension of these is then 11 rather than 8.

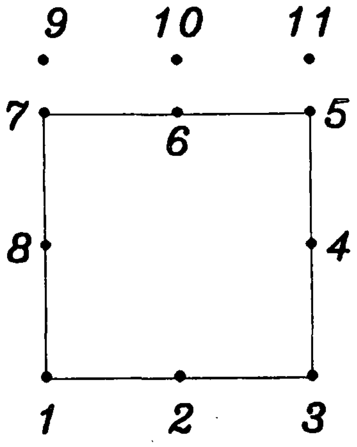


Figure 9.16: *extra top nodes for elements*

The outline of the structure of the Newton matrix and the Newton right hand side vector is given below:

$$\mathbf{G}_k = \begin{pmatrix} \frac{\partial^2 \Pi_k}{\partial \psi_i \partial \psi_j} & \frac{\partial^2 \Pi_k}{\partial \psi_i \partial b_j} \\ & \\ \frac{\partial^2 \Pi_k}{\partial b_i \partial \psi_j} & \frac{\partial^2 \Pi_k}{\partial b_i \partial b_j} \end{pmatrix} \qquad \mathbf{g}_k = \begin{pmatrix} \frac{\partial \Pi_k}{\partial \psi_i} \\ \\ \frac{\partial \Pi_k}{\partial b_i} \end{pmatrix}. \tag{9.44}$$

Using standard procedures from the finite element method, the matrix $\mathbf{G}_k$, called the element matrix, is assembled into the system matrix $\mathbf{G}$, which involves all the unknowns of the problem including the extra surface nodes. Similarly, the element right hand side vector $\mathbf{g}_k$ is assembled into the right hand side vector $\mathbf{g}$. The system $\mathbf{G}\Delta\mathbf{x}=\mathbf{g}$ is then solved for $\Delta\mathbf{x}$ which gives the direction of search where to find the new approximation to the solution $\mathbf{x}_{k+1}$. The calculation of Π_k and subsequently $\partial\Pi_k/\partial\psi_i$, $\partial\Pi_k/\partial b_i$, $\partial^2\Pi_k/\partial\psi_i\partial\psi_j$, $\partial^2\Pi_k/\partial\psi_i\partial b_j$ and $\partial^2\Pi_k/\partial b_i\partial b_j$ has been attempted using the Computer Algebra system REDUCE. The idea is to obtain the analytical form of the two integrands in Π_k and to carry out both integrations analytically.

9.3.2 Direct approaches

In the first place, an attempt was made to directly generate Π_k according to formula (9.43). Since the difficulty comes from the term Π_1 because it is not linear, this is considered first.

A simple REDUCE program has been written for calculating the expression $\mathbf{P}(\nabla\mathbf{N})^t(\mathbf{J}^{-1})^t(\mathbf{J}^{-1})(\nabla\mathbf{N})\mathbf{P}$. In the first place a very simple case was considered where $\mathbf{J}$ was calculated without taking into account the movement of the surface nodes. The outline of the REDUCE program is given below.

```
NB:=8;
MATRIX N(1,NB),DN(2,NB),XX(NB,2),PPSI(NB,1);
...  initialise N with the analytical expressions for the 8-noded
       serendipity element
...  initialise XX with (X1, X2, ...  XNB)
...  initialise PPSI with (PSI1, PSI2, ...  PSINB)
FOR I:=1:NB DO
<< DN(1,I) = DF(N(I),XI)
   DN(2,I) = DF(N(I),ET) >>;
J := DN*XX;
JINV := J**(-1);
JND := JINV*ND;
PI = TP(PPSI)*TP(JND)*JND*PPSI;
```

Figure 9.17: *Direct REDUCE code*

where XI and ET represent the local variables ξ and η, DN contains the shape function derivatives $\nabla\mathbf{N}$, J is the Jacobian matrix, JINV is the inverse of the Jacobian matrix and PI corresponds to Π. This code can easily be adapted for different type of elements by altering the parameter NB and changing the expressions for the shape functions. It was tested for linear triangular elements for which NB=3, the four and nine noded Lagrange elements and the eight-noded serendipity element.

The program managed to produce an expression for Π which, unfortunately, extended to more than 6 pages. When it came to find out the derivatives of Π with respect to ψ_i REDUCE failed to produce any results. Similarly when the integration with respect to ξ and η was attempted REDUCE ran out of space. The main problem was that in order to calculate these quantities a number of large intermediate expressions were generated by REDUCE which eventually ran out of space. REDUCE was run on the University of Durham mainframe computer with 1MByte of space available.

It appeared impossible to tackle the problem directly, especially in the view that the simple code given in Figure 9.17 did not take into account the nonlinear part of the problem.

A different approach has then be attempted where the calculations are decomposed in order to avoid the huge expressions obtained with the direct approach.

The idea is to keep each term of Π separate. The first and second order derivatives are then obtained from the derivatives of the various parts. As shown before, the volume term Π_1 is a linear function of ψ_i and a nonlinear function of b_i. The derivatives with respect to ψ_i should therefore be straightforward whereas the ones for b_i require more effort. The expression for Π_{1k} before integration is:

$$\Pi_{1k} = \mathbf{P}^t(\nabla\mathbf{J}^{-1})^t(\mathbf{J}^{-1})^t(\mathbf{J}^{-1})(\nabla\mathbf{N})\mathbf{P}\,det\mathbf{J}. \qquad (9.45)$$

The expression for the inverse of the Jacobian matrix can be formulated using the adjacent matrix **AJ** defined as follows:

$$\mathbf{AJ} = \begin{pmatrix} J_{22} & -J_{12} \\ -J_{21} & J_{11} \end{pmatrix} \quad \text{for} \quad \mathbf{J} = \begin{pmatrix} J_{11} & J_{12} \\ J_{21} & J_{22} \end{pmatrix}, \qquad (9.46)$$

and

$$\mathbf{J}^{-1} = \frac{\mathbf{AJ}}{det\mathbf{J}}, \qquad (9.47)$$

where $det\mathbf{J}$ is the determinant of the Jacobian matrix:

$$det\mathbf{J} = J_{11}J_{22} - J_{21}J_{12}. \qquad (9.48)$$

Therefore the term $(\mathbf{J}^{-1})^t(\mathbf{J}^{-1})det\mathbf{J}$, denoted **MD** (for MidDle term) can be expressed as:

$$\begin{aligned} \mathbf{MD} &= (\mathbf{J}^{-1})^t(\mathbf{J}^{-1})det\mathbf{J} \\ &= \left(\frac{\mathbf{AJ}}{det\mathbf{J}}\right)^t \left(\frac{\mathbf{AJ}}{det\mathbf{J}}\right) det\mathbf{J} \\ &= \frac{\mathbf{AJ}^t\mathbf{AJ}}{det\mathbf{J}}. \end{aligned} \qquad (9.49)$$

Π_{1k} is then formulated as:

$$\Pi_{1k} = \mathbf{P}^t(\nabla\mathbf{N})^t\mathbf{MD}(\nabla\mathbf{N})\mathbf{P}, \qquad (9.50)$$

where **MD** is a function of b_i only and $\mathbf{P}^t(\nabla\mathbf{N})^t$ $(\nabla\mathbf{N})\mathbf{P}$ is a function of ψ_i only. The formulæ for the derivatives with respect to b_i are thus:

$$\begin{aligned}\frac{\partial\Pi_{1k}}{\partial b_i} &= \mathbf{P}^t(\nabla\mathbf{N})^t\frac{\partial\mathbf{MD}}{\partial b_i}(\nabla\mathbf{N})\mathbf{P}\\ \frac{\partial^2\Pi_{1k}}{\partial b_i\partial b_j} &= \mathbf{P}^t(\nabla\mathbf{N})^t\frac{\partial^2\mathbf{MD}}{\partial b_i\partial b_j}(\nabla\mathbf{N})\mathbf{P}.\end{aligned} \tag{9.51}$$

The derivative with respect to ψ_i needs more attention. Denoting $(\nabla\mathbf{N})^t\mathbf{MD}(\nabla\mathbf{N})$ = **NMDN**, which is a 8×8 matrix, the first order derivatives are expressed as follows:

$$\frac{\partial\Pi_{1k}}{\partial\psi_i} = \frac{\partial\mathbf{P}^t}{\partial\psi_i}\mathbf{NMDN}(\mathbf{P}) + (\mathbf{P}^t)\mathbf{NMDN}\frac{\partial\mathbf{P}}{\partial\psi_i}. \tag{9.52}$$

The multiplication of $\partial\mathbf{P}^t/\partial\psi_i$ by **NMDN** corresponds to taking the ith row of the matrix **NMDN**, since the first order derivative of **P** is a vector composed of zero except at the ith position where there is a 1. This is illustrated in Figure 9.20.

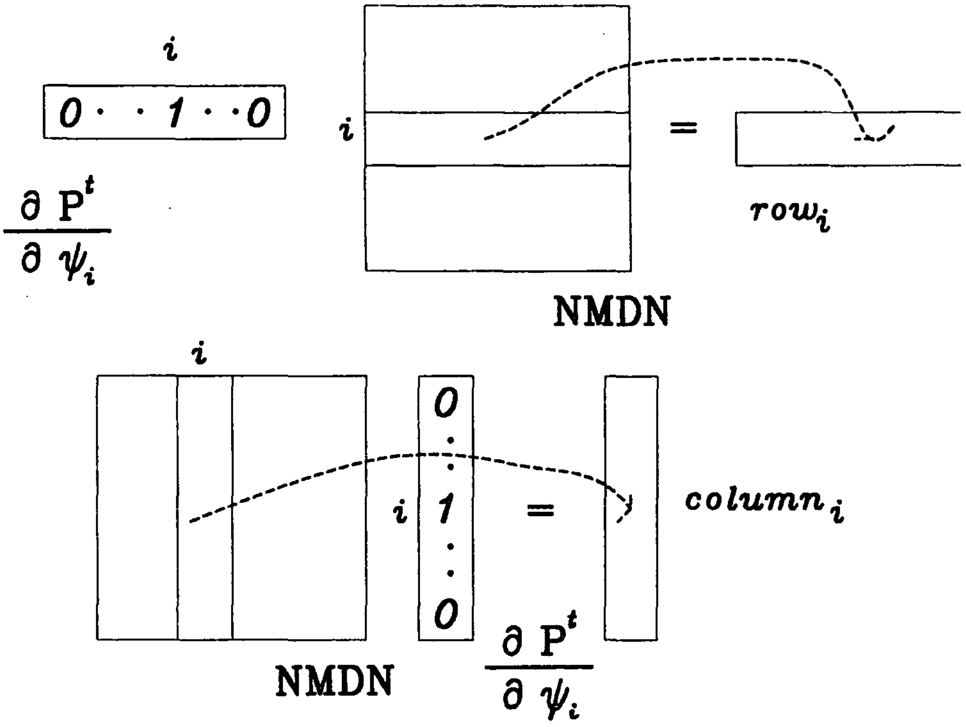


Figure 9.20: *Graphics interpretation of the first order derivatives*

Similarly, multiplying **NMDN** by $\partial\mathbf{P}/\partial\psi_i$ is equivalent to taking the ith column of **NMDN**, as shown in Figure 9.20. Therefore the expression for the first order derivative with respect to ψ_i becomes:

$$\frac{\partial\Pi_{1k}}{\partial\psi_i} = (\text{row}_i)\mathbf{P} + \mathbf{P}^t(\text{column}_i). \tag{9.53}$$

The second order derivatives are now obtained.

$$\begin{aligned}\frac{\partial^2\Pi_{1k}}{\partial\psi_i\partial\psi_j} = {}& \frac{\partial^2\mathbf{P}^t}{\partial\psi_i\partial\psi_j}\mathbf{NMDNP} + \frac{\partial\mathbf{P}^t}{\partial\psi_j}\mathbf{NMDN}\frac{\partial\mathbf{P}}{\partial\psi_i} + \\ & \frac{\partial\mathbf{P}^t}{\partial\psi_i}\mathbf{NMDN}\frac{\partial\mathbf{P}}{\partial\psi_j} + \mathbf{P}^t\mathbf{NMDN}\frac{\partial^2\mathbf{P}}{\partial\psi_i\partial\psi_j}.\end{aligned} \tag{9.54}$$

The second order derivatives of **P** are zero since **P** is a linear function of ψ_i. The cross terms can be found using a graphics interpretation similar to that used for the first order derivatives. This is shown in Figure 9.21.

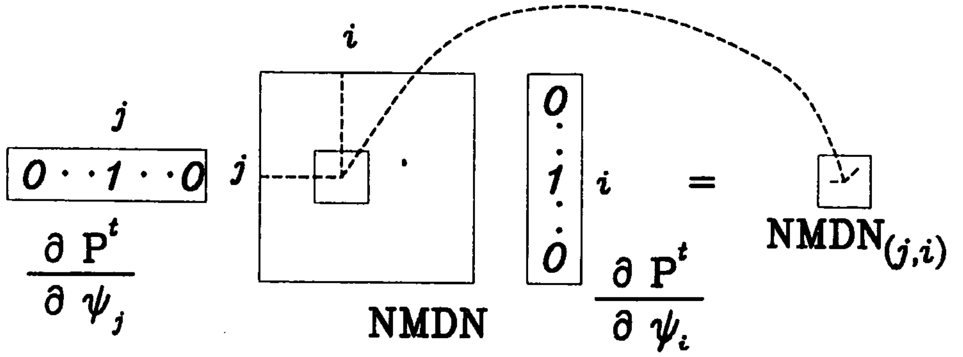


Figure 9.21: *Graphics interpretation of the second order derivatives*

Therefore the second order derivatives are:

$$\frac{\partial^2\Pi_{1k}}{\partial\psi_i\partial\psi_j} = \mathbf{NMDN}_{ij} + \mathbf{NMDN}_{ji}. \tag{9.55}$$

The advantages of this approach compared to the previous one is that the derivatives are easily obtained before the expressions become large. We have experimented with this method for three-noded triangular elements and the REDUCE program for it is given in Appendix G. REDUCE was able to produce expressions for the derivatives but each one was taking more than one page. An example is given in Appendix H.

An inspection of the analytical expressions obtained showed that it was a rational function of the b_i's with a polynomial of degree n at the numerator and a polynomial of degree d at the denominator, where typically n and d had the values 9 and 7. Such functions do not generally possess an analytical integrand.

The expression for **G** being too large for practical use, an alternative method has been devised based on the same principles as the second approach described here where the simple algebra is carried out by hand and only the complicated nonlinear part is obtained with REDUCE.

9.3.3 The refined method

In the refined method, the separate formulation of equation 9.50 is retained but the evaluation of the term **MD** is decomposed further. The numerator and the denominator are kept separate:

$$\mathbf{MD} = \frac{\mathbf{N}}{D} = \frac{\mathbf{AJ}^t\mathbf{AJ}}{det\mathbf{J}}, \tag{9.56}$$

where **N** is a 2×2 matrix and D is a scalar. **J** is expressed as a function of the b_i's as follows:

$$\begin{aligned} \mathbf{J} &= \nabla\mathbf{N}(\mathbf{x} + \mathbf{L}_i b_i) \\ &= \mathbf{K} + \mathbf{S}_i b_i, \end{aligned} \tag{9.57}$$

where **K** and $\mathbf{S}_i$ are 2×2 matrices. **K** correspond to the Jacobian matrix when the surface is fixed and $\mathbf{S}_i$ expresses the movement of the surface. The idea is that **N** and D being polynomial functions of the b_i's their derivatives can easily be obtained using REDUCE.

Next, the formula for the derivatives of $\mathbf{N}/D$, which can be derived by hand, can be expressed as a function of the the derivatives of **N** and D. The advantage of using REDUCE for obtaining this formula is that it can be directly translated into FORTRAN code, therefore avoiding errors in the calculations. The feature of REDUCE concerning the formal definition of formulæ is used here. In REDUCE, it is possible to declare a function as depending on a variable without explicitly giving the relationship. An example is given below:

```
OPERATOR N,D;
F := N(X,Y)/D(X,Y)  ;
DF(F,X);

             D(X,Y)*DF(N(X,Y),X)-N(X,Y)*DF(D(X,Y),X)
             ---------------------------------------
                            D(X,Y)**2
```

Figure 9.20: *Formal calculation of the derivative of a fraction*

This type of calculation is used to formally derive the expression of the first and second derivatives of **MD** in function of the first and second derivatives of **N** and D.

The REDUCE code to obtain the analytical expression for **MD** is given in Appendix J. The program includes the translation into FORTRAN of the results obtained in REDUCE form. The calculation of **G** and **g** from **MD** is carried out numerically according to the formulæ (9.51), (9.53), and (9.55) derived in the previous section. The integration is also carried out numerically using the Gaussian Quadrature procedure[5] with three integration points. The program has the option of increasing the number of integration points if necessary.

The evaluation of the surface term Π_2 is easier because it can be totally derived analytically, including the integration, since all the functions involved are simple polynomial functions. The REDUCE code and the corresponding generated FORTRAN code are given in Appendix J.

9.4 The complete finite element code for the free surface flow

In this section an outline of the structure of the whole program for the determination of the free surface is given, Some detail about the nature of the data input in the program and the automatic generation of the finite element mesh for spillways are explained.

9.4.1 Input data

Most of the data input in the program is concerned with the definition of the spillway shape and the type of mesh associated with the domain of study Ω. The parameters are now listed, in the order they should appear in the input file.

The gravity g is first defined. Although it varies very little around Europe, the necessity for it to be a variable parameter comes from some of the tests which have been devised to check the element matrices as explained in the next chapter.

The number of Gauss integration points is specified next. The program caters for up to ten Gauss points. A flag indicating whether full diagnostic is wanted or only partial information is needed has been defined to enable the examination of the results during the development of the program.

Next comes the geometrical definition of the spillway. It is divided in two parts: the upstream section lying before the crest of the spillway and the downstream section lying after the crest. The origin of the cartesian coordinate system Oxy is taken at the crest of the spillway. Two parameters nu and nd hold the number of points defining the geometry of the spillway in each part. The (x,y) coordinates of these points are stored in an array. An example of a spillway is given in Figure 9.21.

The program has been designed so that spillways coupled with obstructive gates can also be computed. The geometry of these gates is based on an arc of circle. They can pivot around the origin of the circle so that the flow over the spillway is variably obstructed by the gate. The geometry of such gates is defined by the radius of the gate r, the angle θ between the x axis and the bottom of the gate and the (x,y) coordinates of the origin of the circle defining the gate. This is shown in Figure 9.21.

Next comes the definition of the flow itself. This comprises the initial guess for the discharge and the level of the water upstream of the mesh, at the edge of the finite element mesh. This latter parameter enables the calculation of the energy level E. The discharge can either be explicitly specified or be evaluated by the program using an empirical formula based on a coefficient of discharge, which is also input.

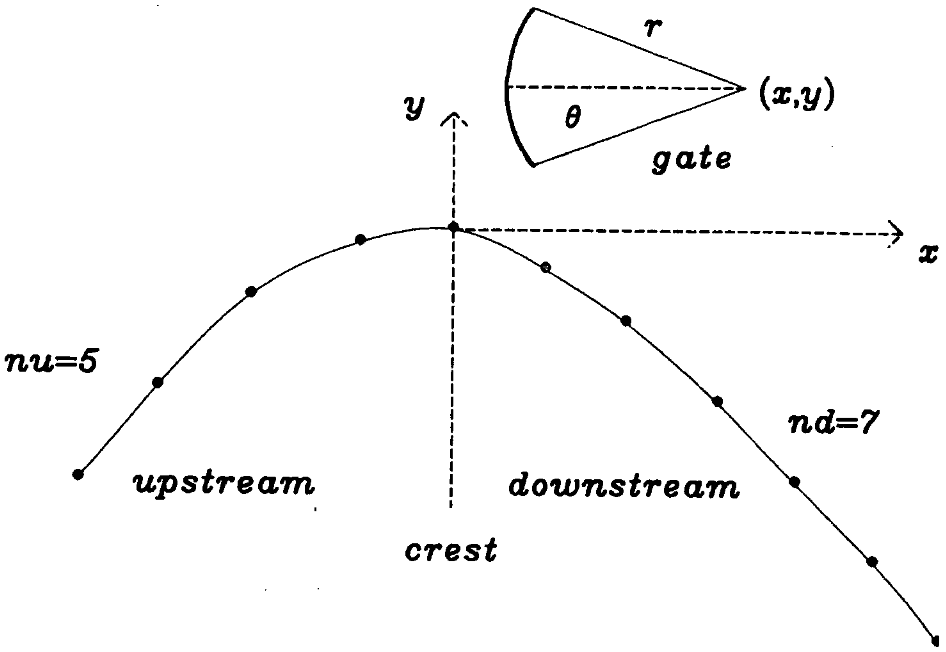
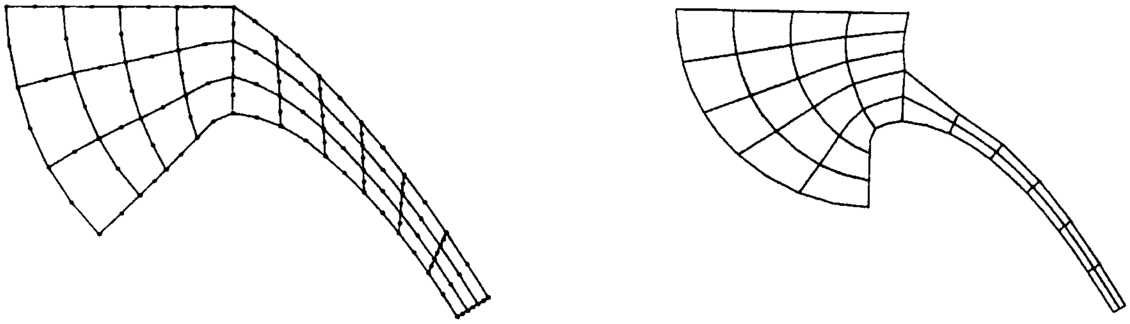


Figure 9.21: *Spillway and gate definition*

The definition of the finite element mesh comes next. The program contains an automatic mesh generator which defines both the extent of the mesh and splits the mesh up into elements. Depending on whether a gate is present or not, the domain of study Ω varies. This is shown in Figure 9.22.



a: without a gate *b: with a gate*

Figure 9.22: *Extent of the mesh (domain Ω)*

The upstream part of the mesh is constructed by swinging an arc of circle between the bottom of the spillway upstream to the level of the water upstream. The origin of the arc is placed at the intersection of the tangent at the bottom of the spillway and the upstream level (see Figure 9.22).

The mesh is then divided into elements. The division is notionally carried out in a straight edges mesh, as shown in Figure 9.23, and is then distorted to fit the real domain Ω. LX1, LX2, LY1 and LY2 represent the number of elements in each part of the mesh, in the x and the y directions. The mesh shown is L-shaped which corresponds to the case when a gate is present. If there is not a gate, the mesh is reduced to the lower rectangular part, which corresponds to LY2=0.

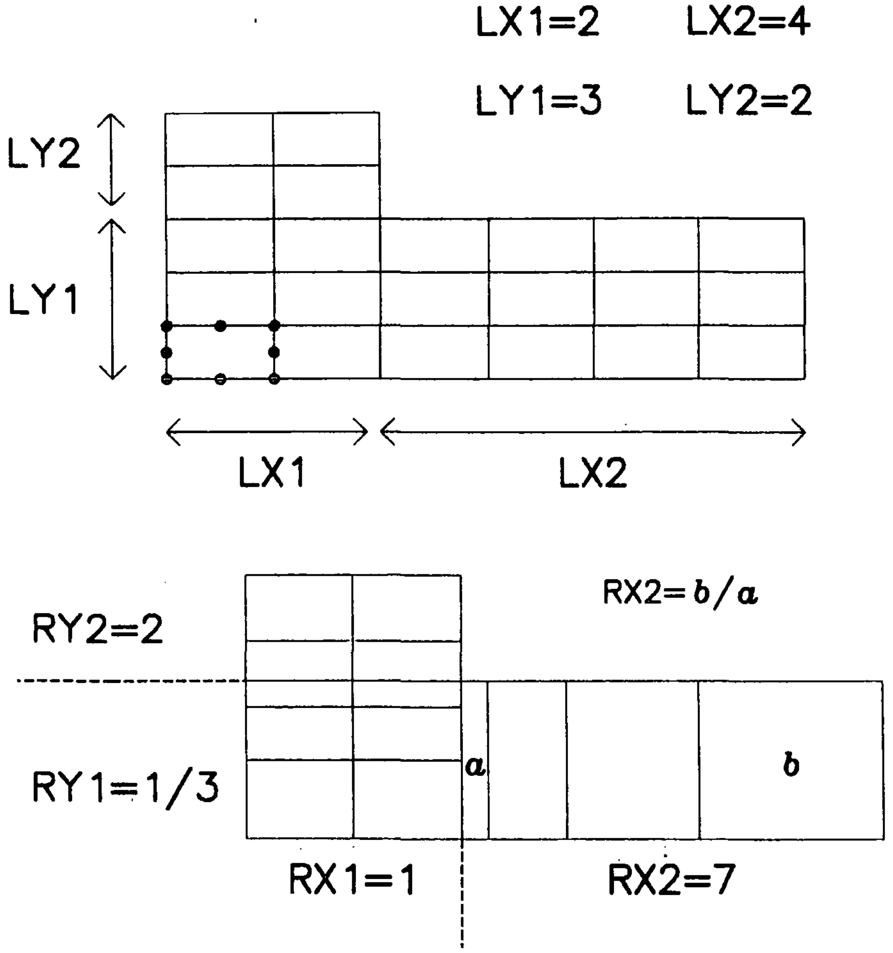


Figure 9.23: *Creation and grading of the mesh*

The grading of the mesh consists of specifying four ratios, corresponding to the four areas of the mesh. Each ratio denoted the size of the element furthest from the upstream section in terms of the element closest to the upstream section. An example is shown in Figure 9.23.

Finally various parameters, including the tolerance for convergence of the loop which finds the free surface shape, are input.

From the input data the attributes of the finite element mesh and the flow are generated. The topology and geometry of the mesh are calculated and stored as several FORTRAN arrays. The elements and the nodes of the mesh are numbered as shown in Figure 9.24. The initial values of the streamfunction are also generated from the value of the discharge. All the nodes on the bed have ψ values equal to zero and the nodes on the free surface have ψ values equal to the discharge Q. This corresponds to the boundary conditions expressed in equation (8.17) in the previous chapter. The values of ψ for intermediate nodes are deduced using a linear distribution between zero and Q.

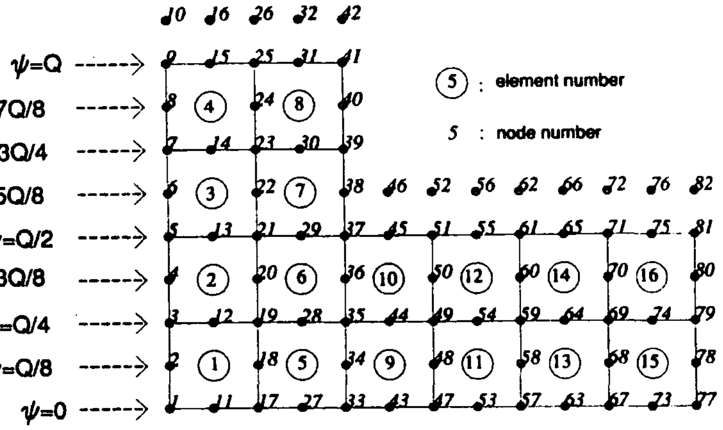


Figure 9.24: *Numbering of nodes and elements; Initial values of* ψ

The boundary conditions on the bed and the surface apply throughout the calculations, therefore a means of constraining the corresponding nodes must be included. This can be done by the use of a **fix** vector similar to the one presented in the chapter on the solvers. The matrix **G** and the right hand side vector **g** for the Newton method are at first calculated as if all nodes were free of movement and could take any ψ value. Before solving the system **G**Δ**x**=**g** the values of ψ for

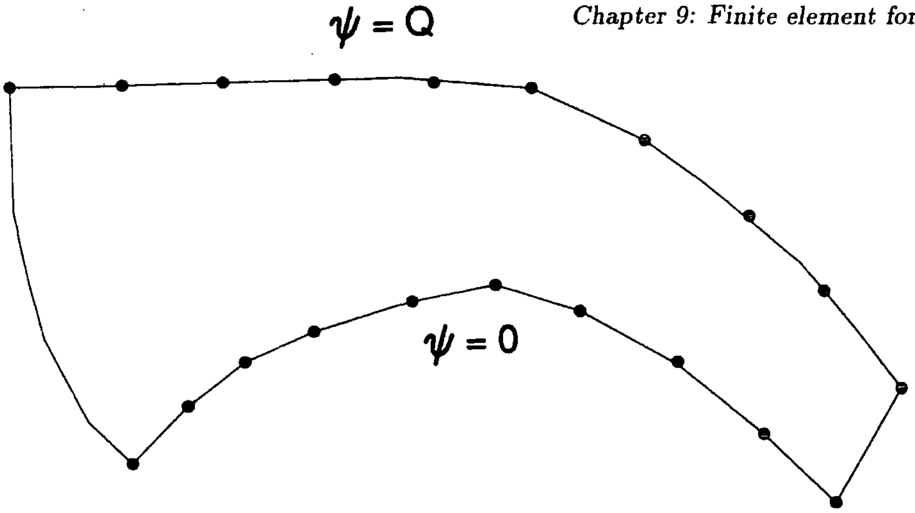


Figure 9.25: *Application of constraints*

the nodes on the bed are reset to zero and those for the nodes on the free surface are reset to Q. This is illustrated in Figure 9.25.

Other constraints can also be applied and their usefulness will be explain in the next chapter. They include fixing the ψ values of the nodes at the inlet and/or at the outlet of the domain, and fixing the surface node at the inlet and/or at the outlet so that they cannot move (b is fixed rather than ψ). The extra constraints are shown in Figure 9.25.

The program assumes that the surface always moves in the normal direction to the bed in the downstream section and vertically (parallel to Oy) in the upstream section. Other type of movements are possible since the angle between the vertical direction Oy and the direction of movement are stored in a FORTRAN array and can therefore be modified. This is illustrated in Figure 9.26.

Routines for inputting special shape of spillways have also been written. A mesh can, if desired, be fully specified by hand including the definition of all the nodes coordinates, the initial values of the streamfunctions ... etc. The special case of the flat channel is also catered for, for which a simple mesh generator has been developed.

9.4.2 General structure

To complete the picture of the program, the outline of the structure is given next, where the inner loop iterates towards a free surface shape and the outer loop

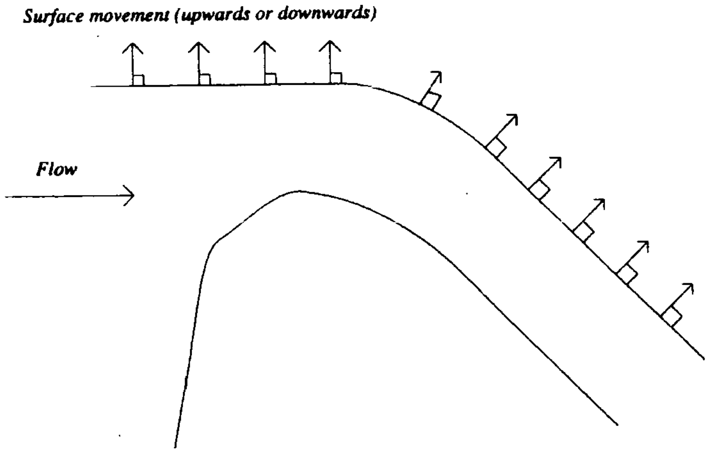


Figure 9.26: *Direction of movement of the surface*

iterates towards the associated discharge. The original program written by Peter and Jackie Bettess contained routines to automatically iterate towards the correct value of the discharge. These routines have been translated into FORTRAN but they have not been tested because of lack of time. The outer iteration is therefore carried out by hand for testing purposes, although an automatisation is outlined in the next chapter.

```
initialisations
input data and generate mesh attributes and flow specifications
display mesh
outer loop
  inner loop
    for all elements in the mesh do
      get the element data
      form the element matrix and right hand side vector
      assemble into system matrix and right hand side
    end for
    solve for $\Delta\mathbf{x}$ of the Newton method
    choose between Newton and steepest descent directions
    if chosen, carry out the line search
    find Bernoulli errors
    display mesh
    test inner convergence
  end inner
  calculate new mesh
  input new $Q$ value
end outer
```

Figure 9.27: *Algorithm of the free surface program*

The Bernoulli errors, which are found at the end of each iteration in the inner loop correspond to the difference between the predicted water level given by the Bernoulli equation and the real level found by the program. The velocity is obtained from the values of the streamfunction and the error is calculated as:

$$error = E - (y + \frac{V^2}{2g}). \tag{9.58}$$

Another parameter, *rat*, is calculated to help decide on whether the inner loop has converged or not. It measures the movement of the surface from one iteration to the other. Denoting $\mathbf{x}=(\psi_1, \psi_2, \ldots \psi_{totnod}, b_1, b_2, \ldots b_{totsnd})$ *rat* is defined as:

$$rat = \frac{\| \mathbf{x}_k \|}{\| \mathbf{x}_{k-1} \|}, \tag{9.59}$$

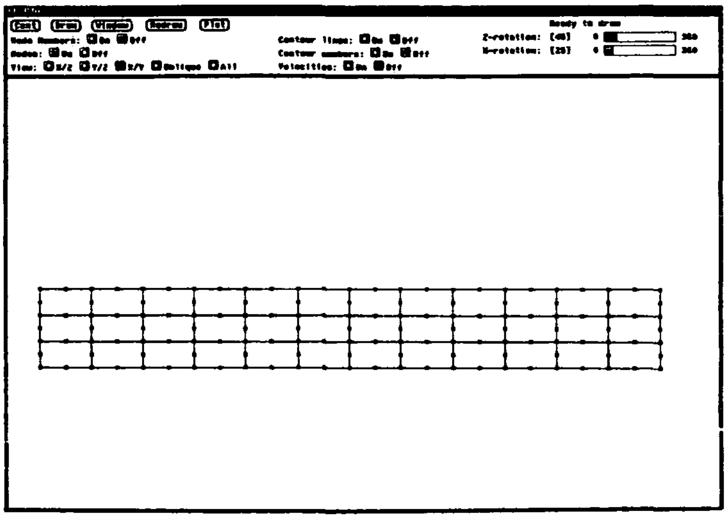
where k denotes the current iteration and $k-1$ the previous iteration. The norm used $\| \ \|$ is the ordinary norm.

The inner iteration convergence test is carried out by hand through the examination of the Bernoulli errors, the *rat* parameter, the length of the vectors $\mathbf{g}$ and $\Delta\mathbf{x}$ and the angle between these two vectors. Automatisating this procedure is straightforward but for test purposes it was easier to enable full control of the convergence.

Another powerful tool to enable decision is the graphical display at each step of the mesh including the shape of the free surface and the plotting of the contours of equal ψ values. An example of such graphics output is given in Figure 9.28.

A number of 'buttons' enable the optional display of the node numbers, the values on the contour lines, the zoom on part of the mesh and the selective display of the mesh and/or the contours.

Because of lack of time the program has not been tidied up to be fully user friendly and most of the efforts have concentrated on testing the program and obtaining a notion of the problems and their cause.



Node Numbers: ☐ On ☒ Off
Nodes: ☒ On ☐ Off
View: ☐ X/Z ☐ Y/Z ☒ X/Y ☐ Oblique ☐ All
Contour lines: ☐ On ☒ Off
Contour numbers: ☐ On ☒ Off
Velocities: ☐ On ☒ Off
Z-rotation: [45] 0 360
X-rotation: [25] 0 360

Figure 9.28: *Graphics display of mesh*

9.5 Parallelisation of the formation of the element matrices

9.5.1 Survey

The parallelisation of finite element codes has been a subject of great interest due to the number-crunching aspect of the method. This method is composed of three steps, namely the element formation and equation assembly, the solution for generalised displacements and determination of general stresses[6]. In classical linear analysis the solution is what takes most of the time and early studies have concentrated on the investigation of parallel solvers. More recently, the use of the method in nonlinear analysis has lead to the development of parallel algorithms for the element formation as this becomes the longest task to perform.

General references on linear parallel solvers can be found in chapter seven. A number of authors have investigated linear and nonlinear solvers in the context of the finite element method. Ideas that have emerged concern substructuring methods and domain decomposition.

Farhat[6] has developed an automatic finite element domain decomposer which has been implemented in FORTRAN on both shared and distributed memory machines. His idea is to help researchers develop parallel algorithms by providing an automatic way of dividing any regular of irregular two or three dimensional finite element mesh into a number of subdomains directly mapped onto the processors.

The algorithm produces a balanced division where either the number of elements or the number of degrees of freedom is balanced between the processors and the number of interface nodes is minimised. This allows a good load balancing and a minimisation of communications.

The domain decomposition technique has been used by Mandel [7] in the p-version of the finite element method. The p-version consists of considering elements with an increased number of nodes and an increased size. The number of unknowns involved is then very high, especially for three dimensions, direct methods for obtaining the solution of linear equations cannot be used due to the fill-in effect of such techniques.

Mandel has used the preconditioned Conjugate Gradient method to solve such systems. It is an iterative scheme based on the Conjugate Gradient method where an approximate solver, the preconditioner, is invoked at each step. The method is applied to the three-dimensional p-version finite element method where each element is treated as a subdomain. The condition number of the matrices is evaluated and the influence of the element ratios studied.

Mandel also did some work on iterative solvers by substructuring for the p-version finite element method[8] which is similar to the work describe above except that the algorithms are written in a slightly different form.

Schäfer has investigated parallel algorithms for the numerical solution of incompressible finite element elasticity problems[9]. These problems are related to nonlinear elasticity theory whose discretisation by finite element technique laeds to very large systems of nonlinear of equations solved with an augmented Lagrangian technique. This enables the nonlinear system to be transformed into a set of highly parallelised subproblems.

At each step three large systems of linear symmetric positive definite systems and many small nonlinear systems have to be solved. The large systems are such that the matrices remain the same during the whole iteration. A pre-conditioned parallel Conjugate Gradient method is used. The small systems are formed completely independently for each element and the solutions can therefore be fully parallelised using a Newton-Raphson method for each system. The authors comments on the fact that a 'direct numerical solution of this system, for example with the Newton-Raphson method, cannot be carried out efficiently, because in practical problems the number of unknowns is very large and the sparse linear systems, which have to be solved in each step, are very poorly conditioned'.

Miles and Havard[10] have experimented with parallel implementation of multifrontal technique in solving fluid mechanical finite element systems. The frontal scheme is a modified Gauss elimination for solving systems of linear equations. Traditionally, the assembly of the system matrix for finite element models is fully carried out before the solver is invoked. In the frontal scheme, the elimination of the unknowns start as soon as one row of the system becomes available. The backward substitution is subsequently performed. In order to maintain the accuracy a pivoting scheme is also used where the elimination on the equations happens after several rows have been assembled and the largest pivot for these rows is found.

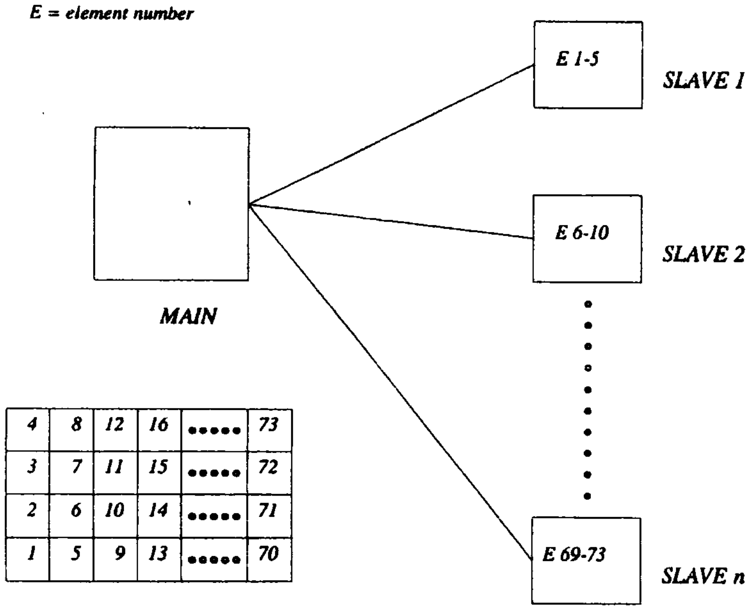
The multifrontal technique is similar to the frontal scheme except for the assembly of the system matrix and the simultaneous elimination starting at various places of the mesh rather than only one. The special case of the fusion of two or more fronts has then to be considered. In the parallel implementation, the mesh is split up into substructures assigned to processors and each processor carries out a frontal solution. The implementation is carried out in OCCAM as at the time the work was carried out only OCCAM had parallel constructs available.

9.5.2 Implementation

There are two main tasks in the program which take most of the running time: the formation of the element matrices and the solution of the system of linear equations. Rough timing indicates that the formation of the element matrices takes at least 80% of the total time of the inner loop, which means that parallelising this part of the program should yield good speed-ups.

The parallelisation of the formation of the element matrices is simple as it involves executing a loop in parallel, which is the simplest possible parallelisation. The idea is that each element matrix is formed in parallel on a slave process and that a main process is in charge of distributing the necessary data at the beginning, of collecting the element matrices as they are calculated and of assembling them in the system matrix. This is shown in Figure 9.29.

In order to simplify and unify the procedure, both the element matrix and the element right hand side vector have been concatenated in one global element matrix of size 11×12 where the last column contains the right hand side vector for that element. It means that both quantities can easily be communicated together.



4	8	12	16	●●●●●	73
3	7	11	15	●●●●●	72
2	6	10	14	●●●●●	71
1	5	9	13	●●●●●	70

Figure 9.29: *Parallel evaluation of the element matrices*

The processes are arranged in a processor farm configuration where all the slave processes execute the same instructions on different data.

The communication is simple. It involves, for p processes, p main-slave process communications at the beginning and *totels* slave-main process communications during the calculations.

The data concerning the mesh attributes and the flow specifications has to be distributed. Each slave process is assigned with the calculation of k consecutive elements where $k = totels/p$. An example is shown in Figure 9.29.

The evaluation of the efficiency of the parallelisation of the formation of the element matrices consists of comparing the time it takes in the serial approach and the time when executed in parallel. A 'theoretical' efficiency, which would consist of comparing the time it takes for the parallel program to run on one and then on p processes, is of little interested here since it would not show the real benefit of running the program in parallel. The efficiency is thus:

$$Eff = \frac{T_{serial}}{T_p}\frac{1}{p}. \tag{9.60}$$

This correspond to the efficiency Eff_2 defined in the chapter on parallel solvers.

9.5.3 Tests and conclusions

The efficiencies have been obtained for three types of spillways as shown in Figure 9.30.

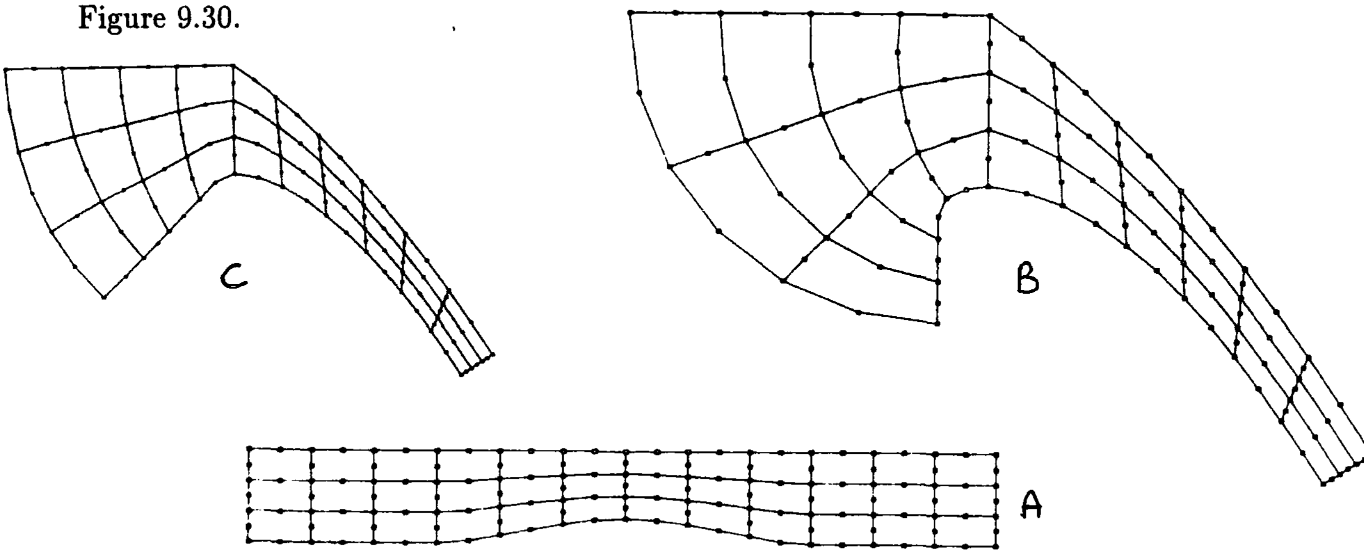


Figure 9.30: *Spillways used for test of the parallel implementation*

The efficiencies in percent (%) for these three spillways are given in Table 9.1. The values for the efficiencies have been obtained for 4, 8, 12 and 16 processors. The results are excellent. This comes from the fact that all the calculations are totally independent from one another, which implies a very small proportion of communications compared to calculations.

A number of lower efficiencies (92%, 87% and 75%) appear in Table 9.1 for spillways B and C and 12 and 16 processors. When examining how the elements are distributed amongst the processors, these lower efficiencies correspond to the case when the number of elements does not divide exactly by the number of processors.

Number of processors	Spillway type		
	A	B	C
4	99.76 (9)	99.61 (12)	99.48 (14)
8	90.21 (4.5)	98.25 (6)	98.38 (7)
12	96.98 (3)	97.13 (4)	92.51 (4.66)
16	75.15 (2.25)	95.24 (3)	87.02 (3.5)

Table 9.1: *Efficiencies (in %) and ratios (number of elements/number of processors)*

Therefore, some of the processors have more work to do than others (bad load balancing). Even in these cases, the efficiencies are very high.

The two schemes (*a*) and (*b*) described in chapter 7 on the solvers and shown in Figure 7.32 have been tested for 4 processors on the spillway problems. This is when the main process is either mapped on its own on a processor or mapped together with one of the slave process on a processor. The results are shown in Table 9.2. Similar conclusions to that drawn from the solver program can be deducted here. The scheme *b* is more efficient. The results shown in Table 9.1 were obtained using the scheme *b*.

Spillway type	Scheme a	Scheme b
A	80.35	99.76
B	80.27	99.61
C	80.25	99.48

Table 9.2: *Comparative Efficiencies (in %) of scheme a and b*

The next chapter concentrates on the testing the program and discusses results.

References

1. Bettess P. and Bettess J.A., 'Analysis of Free Surface Flows using Isoparametric Finite Elements', *International Journal for Numerical Methods in Engineering*, **19**, pp 1675–1689, 1983.
2. Dixon L.C.W., *Nonlinear Optimisation*, The English Universities press, 1972.
3. Burden R.L and Faires J.D., *Numerical Analysis*, third edition, pp 27–78, 1985.
4. Fletcher R., *Practical Methods of Optimization*, Second edition, John Wiley & Sons, 1987.
5. Cheney W. and Kincaid D., *Numerical Mathematics and Computing*, second edition, Brooks/ Cole Publishing Company, pp 192–200, 1985.
6. Farhat C., 'A Simple and Efficient Automatic FEM Domain Decomposer', *Computers and Structures*, **28**, Iss 5, pp 579–602, 1988.
7. Mandel J., *A Domain Decomposition Method for the p-version Finite Elements in Three Dimensions*, available from the author at the Computational Mathematics Group, University of Colorado at Denver, 1200 Larimer Street, Denver, CO 80204, USA.
8. Mandel J., *Iterative Solvers by Substructuring for the p-version Finite Element method*, available from the author at the Computational Mathematics Group, University of Colorado at Denver, 1200 Larimer Street, Denver, CO 80204, USA.
9. Schäfer M., 'Parallel Algorithms for the Numerical Solution of Incompressible Finite Elasticity Problems', *SIAM Journal of Statistical Computations*, **12**, Iss 2, pp247–259, 1991.
10. Miles R.G. and Havard S.P., 'Multifronts and Transputer Networks for Solving Fluid Mechanical Finite Element Systems', *International Journal for Numerical Methods in Fluids*, **9**, pp 731–740, 1989.

Chapter X

Tests and conclusions

This chapter is concerned with the testing of the free surface program described in the previous section. A number of difficulties have arisen due to the nonlinear nature of the problem. The main point is that there is no literature to which reference can be made concerning the correctness of the various elements of the program and that when testing the results their values are not known *a priori.* The major difficulty is in testing the element matrices since they are obtained in a novel way. Another problem is related to the solution of the system of linear and nonlinear equations in the context of the finite element method.

In the light of the results obtained from the tests for the correctness of the element matrices and of the solvers some further investigations may also be necessary at theory level to check that the correct boundary conditions are applied and the finite element approximation does not introduce too high an inaccuracy.

This chapter is organised in three main sections. Firstly, the tests carried out to check the element matrices are described. Secondly, the evaluation of the accuracy of the solvers used is presented. Lastly, some indications on further investigations and developments are indicated.

1 Tests of the element matrices

The tests on the element matrices can be divided in two categories: the direct and indirect tests. The direct tests are carried out on the numerical values held in the element matrices whereas the indirect tests consist of running the program on simple test spillways for which the answer is known in advance.

.1 Direct tests

The first test carried out is related to the existence of an alternative method for generating the element matrices. The original ALGOL program written by Peter and Jackie Bettess[1] has been run and the numerical values for the element matrices have been compared to those obtained from the FORTRAN code automatically generated by the REDUCE program. The test consists of two steps. Considering a one element mesh as shown in Figure 10.1, the contribution to the element matrix of the volume term Π_{k1} and of the surface term Π_{k2} have been compared between the two methods.

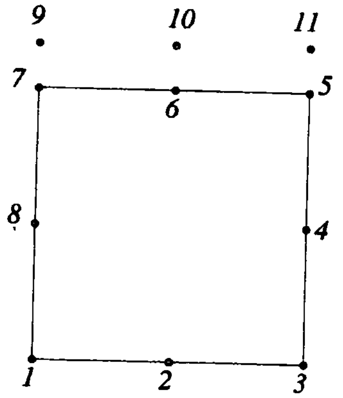


Figure 10.1: *Check on a one-element mesh*

The method used in the original ALGOL program is based on a Taylor expansion series of the nonlinear terms of the functional and the evaluation of the derivatives from the series. All the calculations have been carried out by hand by the authors and checked several times, also by hand. The code had to be run on the University of Bath's mainframe since the ALGOL compiler was not available on the mainframe of the University of Durham. While testing a few mistakes were found in the ALGOL which were corrected.

The two methods give the same numerical answers, within the tolerance due to rounding errors. Since these are two independent ways of obtaining the element matrices, the chances that the element matrices may be correct are reasonably high. Nevertheless, this is not sufficient to conclude absolute correctness.

The element matrix can be decomposed in two parts: the 8 × 8 section which corresponds to the ψ unknowns only and the rest which involves the ψ and b unknowns (see equation 9.44). The problem of solving the governing equations can be transformed in an ordinary potential solving problem if the surface is fixed.

Indeed, the governing equations are reduced to the Laplace equation with only the boundary conditions that the nodes on the bed and on the free surface are fixed. This means that the domain of study Ω is not variable any more and that all the b values are fixed and equal to zero. The solution of the Laplace equation is easily obtained since it corresponds to solving a system of linear equations.

The element matrix has been formed for this test problem and the solution obtained for the ψ values has been found correct. The element, the boundary conditions and the solution obtained are shown in Figure 10.2.

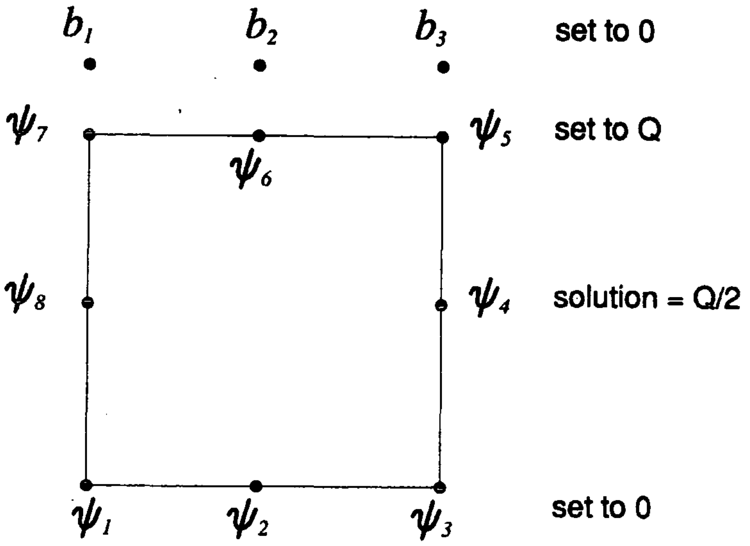


Figure 10.2: *test of the element matrix in a potential formulation*

Following similar testing method, a one-element mesh as shown in Figure 10.1 can be set up so that the two outside surface nodes, number 9 and 11, are fixed. The entry in the system matrix for the term (10,10) which corresponds to the movement of the middle surface node should then be zero since the surface is not expected to move, as shown in Figure 10.3.

Another test on this one-element mesh consists of singling out symmetries in the element matrix from the symmetries in the physical element. This only gives a test of consistency since the actual values are not checked. For a square element, denoting the element matrix ELK the following symmetries should appear:

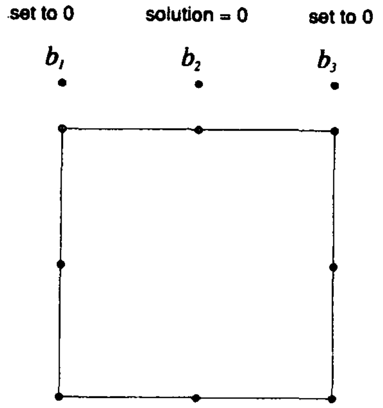


Figure 10.3: *Check of the term (10,10) when surface should be horizontal*

$$\begin{aligned}
\mathrm{ELK}(9,9) &= \mathrm{ELK}(11,11)\\
\mathrm{ELK}(9,5) &= \mathrm{ELK}(11,7)\\
\mathrm{ELK}(9,7) &= \mathrm{ELK}(11,5)\\
\mathrm{ELK}(9,6) &= \mathrm{ELK}(11,6)\\
\mathrm{ELK}(10,7) &= \mathrm{ELK}(10,5)\\
\mathrm{ELK}(10,9) &= \mathrm{ELK}(10,11)
\end{aligned}$$

This has been checked successfully. A last test on the element matrix has been carried out which concerns the conservation of the element matrices when the element is scaled up. The idea is that the fluid in the simple square one-element mesh is dominated by the Froude number. This number is defined as follows:

$$\text{Froude number} = \frac{V}{\sqrt{gd}}, \tag{10.1}$$

where V is the velocity of the fluid, g is the acceleration due to the gravity and d is the depth of water. When scaling up the element, for example from dimensions 1:1 enlarging it to dimensions 2:2, if the Froude number is kept constant by altering the gravity the new element matrix obtained should be identical to the old one. The velocity of the fluid is kept constant, so that the properties of the fluid are not altered. For the Froude number to remain constant the gravity has to be halved. Indeed, denoting the original element attributes with the index 1 and the new enlarged element attributes with the index 2, as shown in Figure 10.4, the following equations hold:

$$\frac{V_1}{\sqrt{g_1 d_1}} = \frac{V_2}{\sqrt{g_2 d_2}}, \tag{10.2}$$

where

$$V_1 = V_2 \Longrightarrow \sqrt{\frac{g_2}{g_1}} = \sqrt{\frac{d_1}{d_2}} = \sqrt{\frac{1}{2}}. \qquad (10.3)$$

For the velocities to remain the same, the ψ values in the element must be scaled up too. This can be seen from the definition of the velocity for a straight edges element:

$$Q = Vd, \qquad (10.4)$$

where Q is the discharge of the flow and is also the values of the streamfunction on the surface, since $Q = \psi$ on the surface is a boundary condition of the problem. If d is multiplied by two, V being constant, Q has to be multiplied by two. This means that the ψ values in the scaled up element are doubled. This is shown in Figure 10.4.

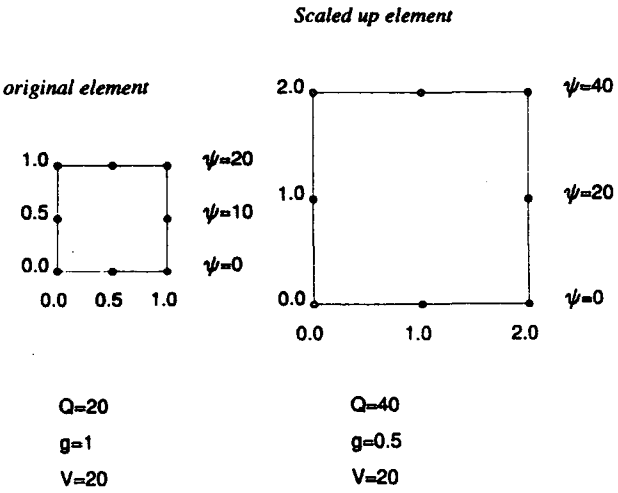


Figure 10.4: *Scaled up element and parameters*

The test has been carried out for g_1=1 and g_2=0.5 and the matrices were found to be identical.

This collection of tests carried out on the element matrices indicates that there is a reasonable probability for them to be correct. Nevertheless, this is not sufficient and more tests have been developed.

10.1.2 Indirect tests

All the indirect tests have been carried out on a special case of spillway: the flat channel, as shown in Figure 10.5.

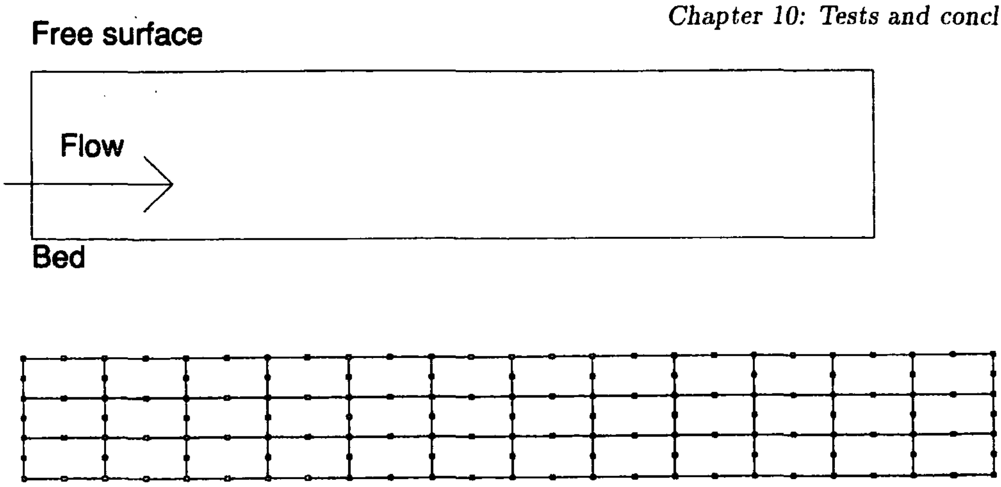


Figure 10.5: *Flat channel geometry*

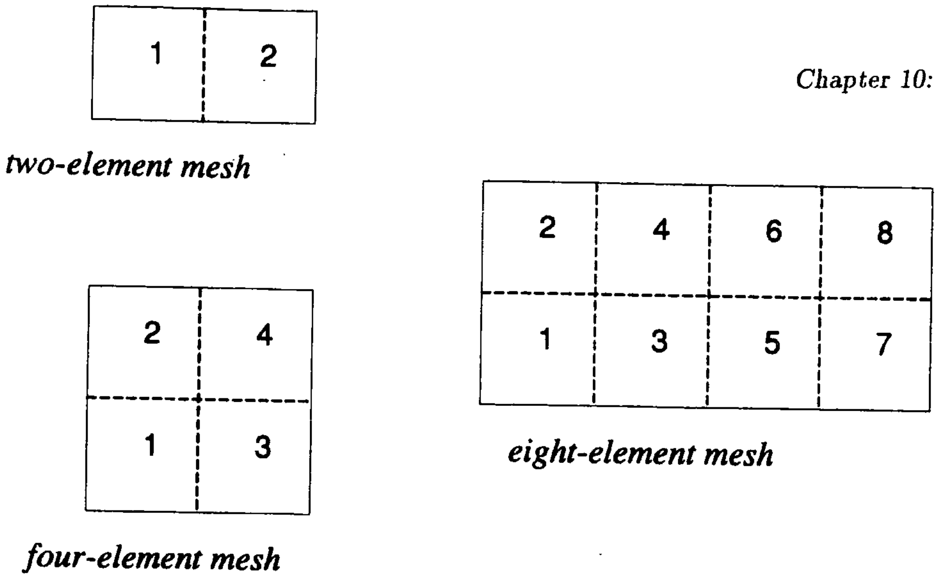
The element matrices have been formed and assembled into the system matrix. The solution of the nonlinear equations has been found using the Newton method. The graphics routines have been used to display the resulting shape of the free surface and the streamlines of equal ψ values.

The first obvious test is to check that the flat channel remains unchanged if no disturbance is applied. This means that the free surface should not move and that the streamfunction values should also be unchanged. This result should be obtained at the first step in the Newton iteration and further steps should not bring any further changes. The algorithm for this test is shown below:

```
Initialise $b$=0 and linearly distribute $\psi$
for $k$=1 to totels do
    form $\mathbf{g}_k$ and $\mathbf{G}_k$
    assemble into system matrix and right hand side vector
apply constraints
solve for $\Delta\mathbf{x}= (\delta\psi_1, \delta\psi_2, \ldots, \delta b_1, \delta b_2 \ldots)$
Check that $\Delta\mathbf{x}=\mathbf{0}$ within rounding errors tolerance
```

The second test is an extension of the test carried out on the element matrix where the problem is reduced to a potential problem by constraining the free surface. Three types of mesh have been tested: two, four and eight-element meshes as shown in Figure 10.6.

The b variables have been constrained so that the surface cannot move. The



two-element mesh

eight-element mesh

four-element mesh

Figure 10.6:*Meshes for potential problems*

ψ variables have been constrained on the bed and on the surface and left free everywhere else. The unknown ψ values should therefore distribute themselves linearly between the fixed value on the bed and the fixed value on the surface. An example of such a test for the eight-element mesh is shown in Figure 10.7.

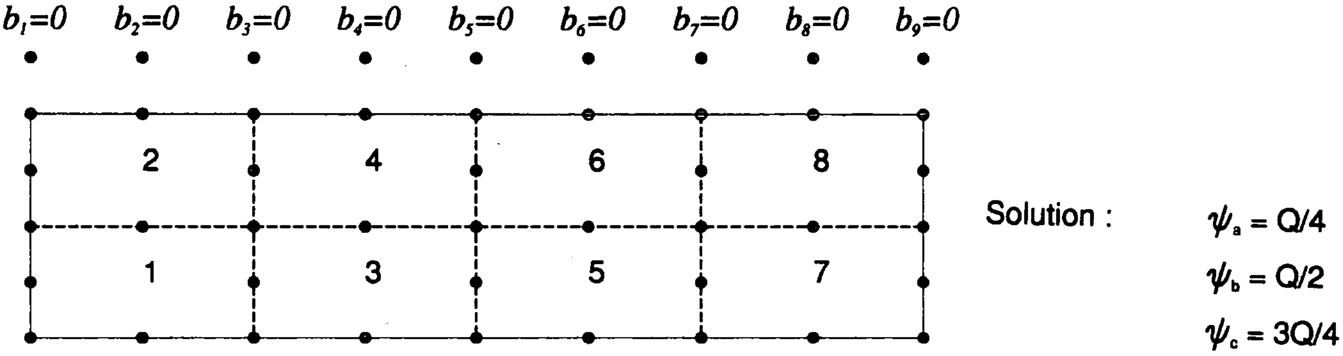


Figure 10.7: *Constraints and results for the 8-element mesh*

Less trivial tests have subsequently been carried out. The first series consists of disturbing the free surface in a way that its behaviour can be predicted. If, as a starting point, the surface is mostly flat except a few nodes which are either pulled above the flat level or pushed below the flat level, the surface is expected to converge back to the flat level. This is illustrated in Figure 10.8.

A series of tests has been carried out where a simple six-element mesh is used (see Figure 10.8) and the nodes on the surface are disturbed above and below the

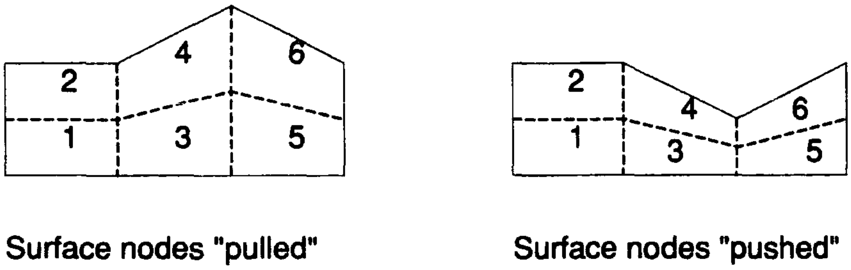


Figure 10.8: *Disturbances of the free surface in a flat channel*

original level. Plots of the convergence pattern have been obtained for both slow and fast fluids. The distinction between the two types of fluid is important as the behaviour is different in each case.

A slow fluid is a fluid for which the Froude number is smaller than one and a fast fluid has a Froude number greater than one. A fluid with a Froude number of one is called critical fluid. The plots for both fluids can be seen in Figure 10.9 and 10.10.

The divergence zones in Figures 10.9 and 10.10 correspond to values of d for which the program does not converge back to the flat level. The program either diverges, which means that the mesh is distorted so much that after a few iterations the finite element model is not valid anymore (flipped over elements, see Figure 9.4) and the mesh collapses towards the infinity.

The main conclusions that can be drawn from the plots is that the fast flow, called supercritical flow, is more stable and that the slow flow, called subcritical, is more sensitive to disturbances. Both flows converge within a few iterations, even with a small tolerance of 10^{-6} (coefficient rat from equation 9.59).

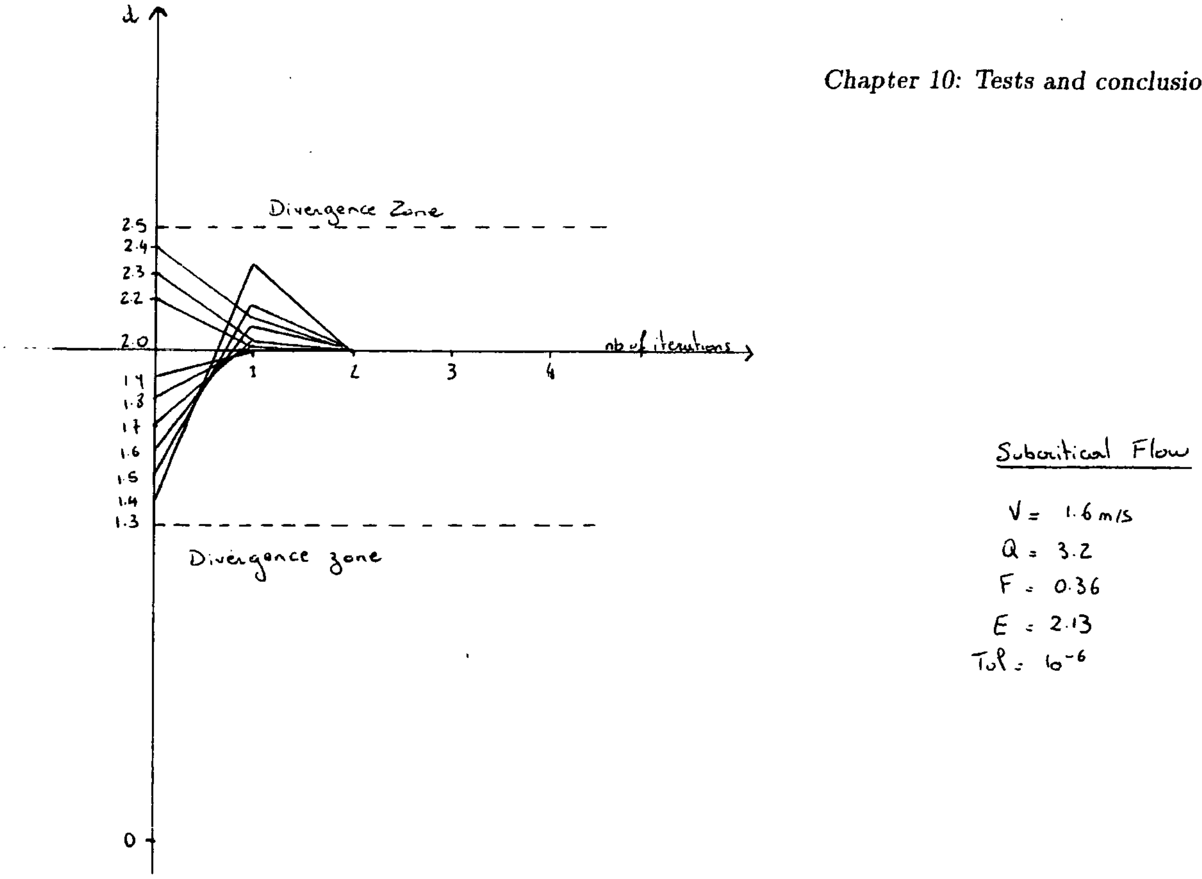


Figure 10.9: *Plot of the convergence pattern for a slow fluid modelled with a 6-element mesh*

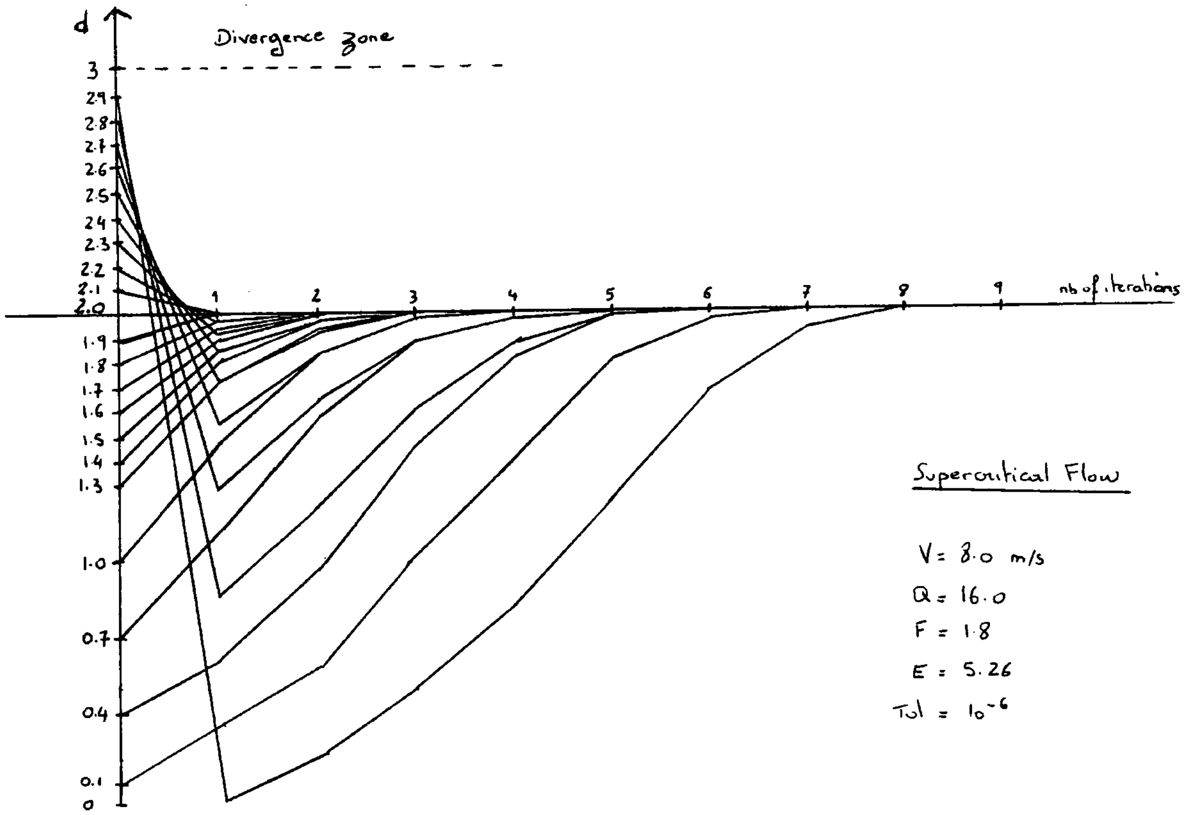


Figure 10.10: *Plot of the convergence pattern for a fast fluid modelled with a 6-element mesh*

The supercritical flow converges for values of d down to zero which means it can recover from a situation where the surface nodes are pushed right down to the level of the nodes on the bed. The divergence zone for values of d greater than 2.9 can be explained by noting that when the nodes on the surface are pulled above the flat level, the path of the convergence goes via a position where the nodes are pushed back down towards the bed. When the top node is pulled by a value 2.9, the first step in the convergence pushed it down to the level of the bed. Therefore for values greater than 2.9 the top node would be pushed below the level of the bed, which would cause some of the elements to flip over and the finite element model to become invalid. The results for the supercritical flow suggest that convergence is obtained for all cases when the integrity of the finite element model is retained.

The subcritical flow is more instable and the convergence domain is much smaller. Although the plot shows values of d greater than 2.13, which corresponds the energy level, it is unrealistic to try to force the flow above this level. This has been done for the purpose of studying the convergence pattern only.

Another simple test consists of pushing or pulling a node at one end of the mesh and checking that the surface level follows down or up to that new level. This is illustrated in Figure 10.11.

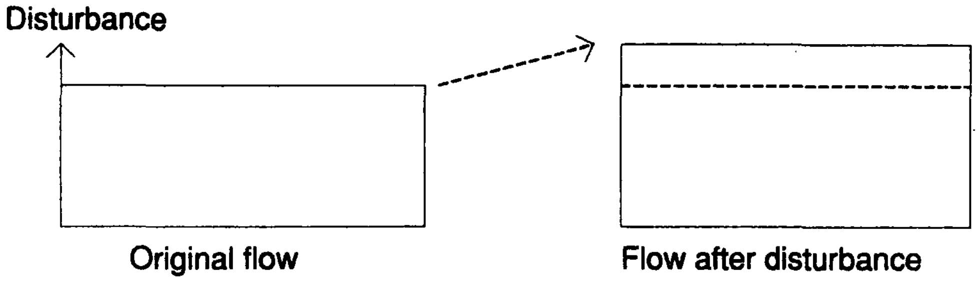


Figure 10.11: *Disturbance of the edge of the mesh in flat channel*

The flow behaves as expected for values of d ranging from -0.4 to 0.8. Outside these values divergence occurs.

Lastly, a special case of spillway has been tested which consists of a flat channel with a symmetric bump of variable size introduced in the middle of the bed, as shown in Figure 10.12.

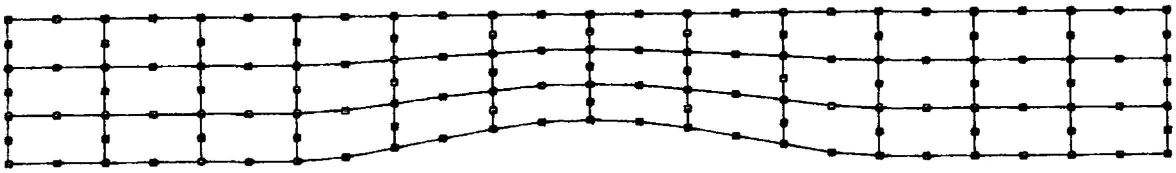


Figure 10.12: *Bump on the bed in a flat channel*

The advantage of testing this type of spillway is that the flow remains either mainly subcritical or supercritical since the bump only introduces a local disturbance. Because of the simple geometry of the problem, the theoretical value for the water level above the top of the bump can also be worked out which is an effective way of checking the results. In the remainder of this section the theoretical equations are derived and the results obtained presented.

In order to avoid any problems with the shape of the bump a smooth rounded bump was chosen. This should ensure that no singularities are introduced in the problem and that a steady state solution exists. The shape of the bump has been generated using a cubic polynomial interpolation for half of the bump, which has then been extended to the other half using a symmetry. This is shown in Figure 10.13.

The basic equations for the half bump are:

$$\begin{aligned} ax^3 + bx^2 + cx + d &= y \\ 3ax^2 + 2bx + c &= y', \end{aligned} \qquad (10.5)$$

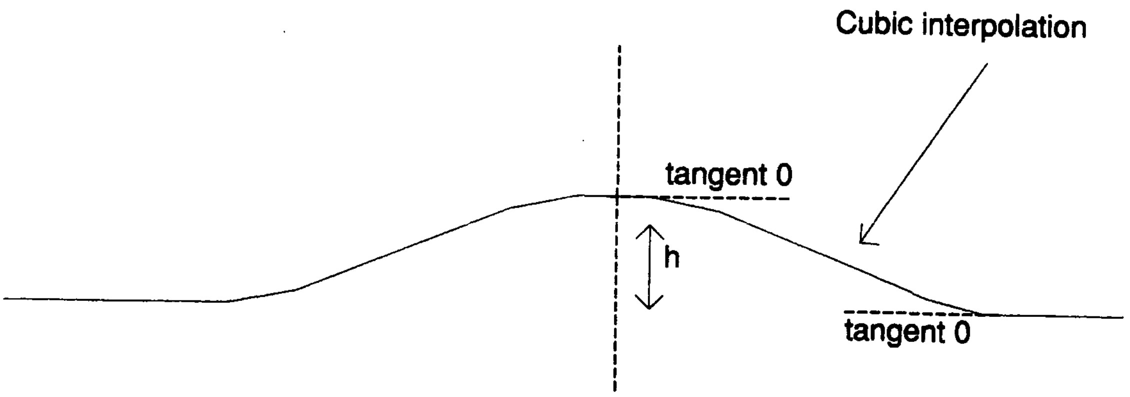


Figure 10.13: *Geometry of the bump*

with the conditions that:

$$\begin{array}{llll} \text{For } x = 0 & y = h & \text{and} & y' = 0 \\ \text{For } x = 6 & y = 0 & \text{and} & y' = 0. \end{array} \qquad (10.6)$$

This gives the following equation:

$$y = (\frac{1}{108}x^3 - \frac{1}{12}x^2 + 1)h. \qquad (10.7)$$

A number of meshes and flows have been tested. At first, the meshes attempted were too short and the results obtained incorrect because the surface could not move to its correct final position, being restricted on either sides by the fixed nodes. This is illustrated in Figure 10.14.

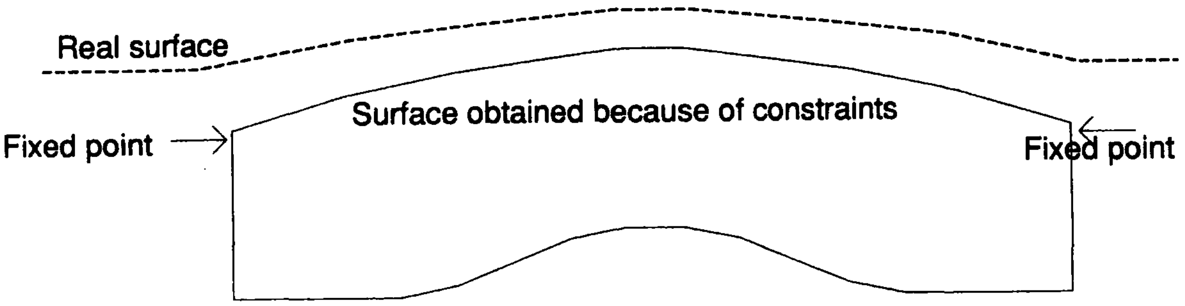


Figure 10.14: *Influence of the length of the mesh*

After a few attempts, it was found that a length of 48 units was long enough to correctly model the flow around the bump. This mesh has then been divided into elements in three different ways. Square and rectangular elements of dimensions ratios of 1:2 and 1:4 have been attempted. This is shown in Figure 10.15.

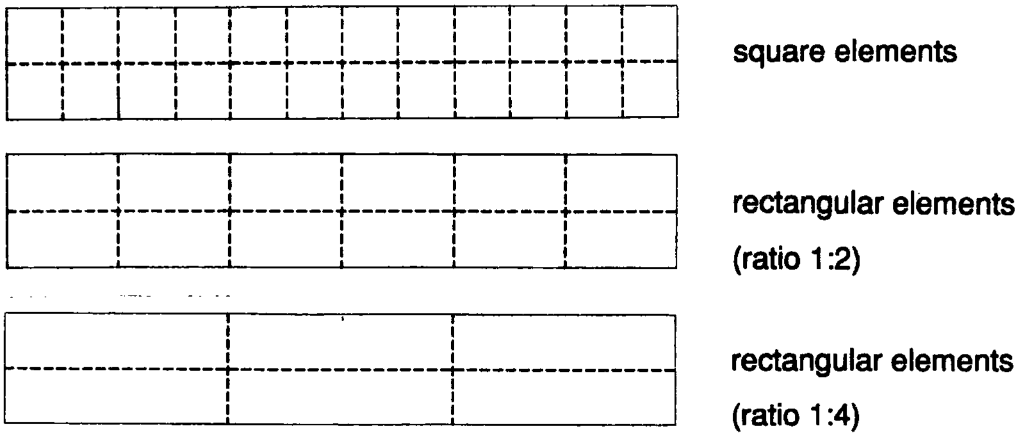


Figure 10.15: *Finite element meshes of the flow around the bump*

Both subcritical and supercritical flows have been considered.

The mean of checking the results relies on the use of the Bernoulli equation. As explained in the previous section, the Bernoulli equation holds along a streamline. Considering the streamline formed by the free surface the Bernoulli equation can be applied so that:

$$\frac{V_1^2}{2g} + d_1 = \frac{V_2^2}{2g} + d_2 = constant = E, \tag{10.8}$$

where V_1 and d_1 are the velocity and depth of the flow on the surface at the inlet and V_2 and d_2 are the velocity and the depth of the flow above the top of the bump. This is shown in Figure 10.16.

The velocity and the distance d_1 at the inlet are known. The velocity on the surface above the crest of the bump can be calculated from the values of the streamfunction around that point. The velocity is not necessarily tangent to the surface therefore the norm of the velocity vector is used in the formula. The distance d_2 is given by the shape of the surface above the bump when convergence is reached. The error can therefore be defined as the difference between the value

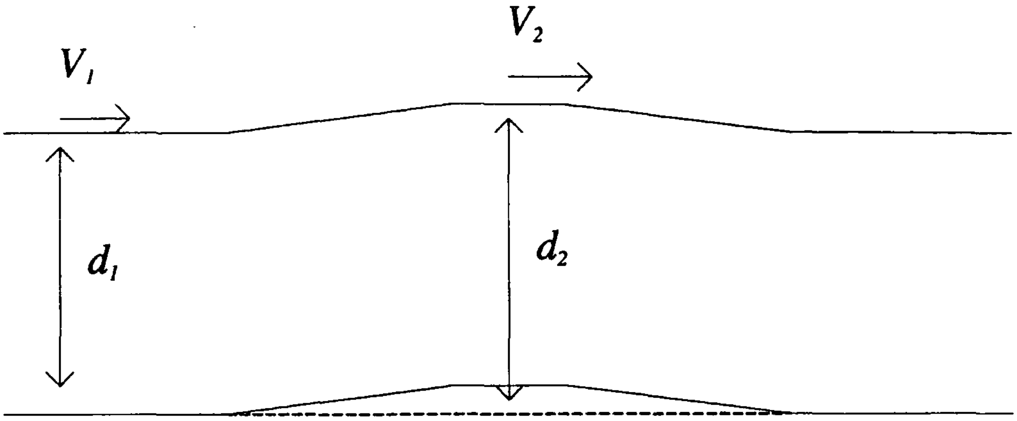


Figure 10.16: *Velocities and depths definition*

of E, the energy level, calculated at the inlet and the value obtained on the surface above the top of the bump.

Knowledge about this type of spillway[2] indicates that the subcritical fluid has its surface curving inwards (towards the bed) when going over the bump and that the supercritical flow has its surface curving outward, forming a bump similar to that of the bed. This is shown in Figure 10.17

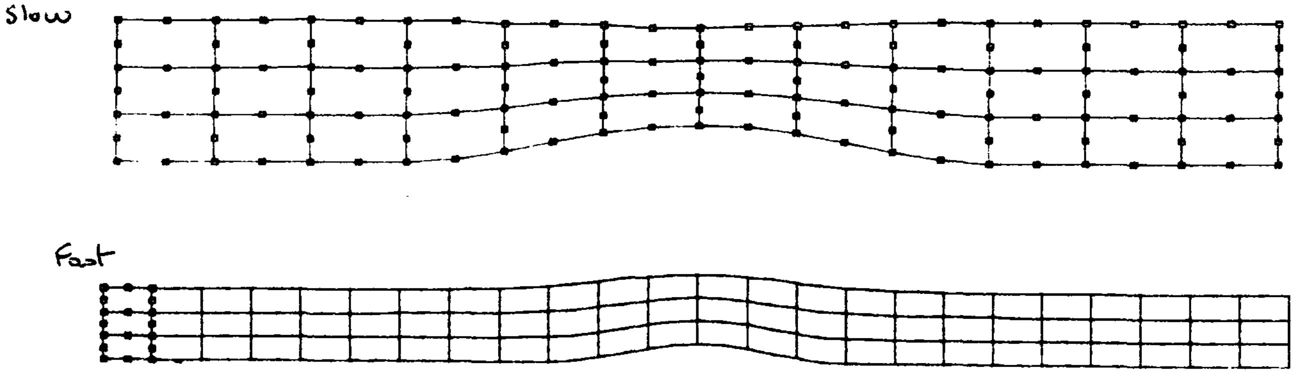


Figure 10.17: *Behaviour of slow and flat fluids above a bump*

The results for a supercritical flow, with $V_1 = 12$, are shown in Figure 10.18.

The results agree well with the theory. For bumps of size up to 2.0 convergence was observed and good agreement was also found. Such bumps are large compared to the depth of the fluid and some non steady states solutions have developed for

Bump size number of elements	0.4	0.8	1.2
12 elements	10.3333 (0.059%	10.3288 (0.10%)	10.3261 (0.12%)
24 elements	10.3252 (0.13%)	10.3145 (0.24%)	10.3074 (0.30%)
48 elements	10.3199 (0.18%)	10.2956 (0.42%)	10.2965 (0.41%)

Theoretical energy level $E = 12^2/2g + 3 = 10.3394$

Figure 10.18: *Calculated energy level and errors in %*

larger size of bumps like waves on the surface. Divergence has also been observed for large bumps. This can either be a problem linked to the convergence pattern of the Newton method or to a physical instability such as a hydraulic jump. To remain within the assumptions of the model developed the size of the bumps should be kept small in relation to the depth of the water at the inlet.

A fast fluid with $V_1 = 8$ has also been successfully checked. Similar tests have been carried out on a subcritical flow too. The results in this case, however, are more difficult to check since the actual movement of the surface is hardly visible. For example, for a bump of 0.6 the surface curves inwards by a quantity 0.01 which is very small. The results have been checked in a similar way to those for the fast fluid but the imprecision is higher since the displacements are so small.

The next step in the testing process has been to try a real spillway shape. This is discussed in the next section along with the additional tests carried out.

10.2 Tests on the spillways

Two types of spillway have been considered: gated and ungated spillways. The details of the data related to these spillways can be found in the previous chapter. In testing real spillways, the problem is complicated by the fact that the discharge Q is not known in advance. It is also important to test realistic spillways so that the approximation used models the fluid correctly.

The spillway shape tested here have been taken from previous studies on the original ALGOL program[3] based on standard spillway shapes recommended by the U.S. Army Engineer Water-Ways Experiment Station. Because of lack of time, no attempt has been made to investigate other spillways, although this is something which should be done. The process of testing many spillways would take time considering the amount of information that there is to examine: shape of the free surface, conditions of convergence, values of Q bringing convergence, accuracy... etc. In this work we have restricted ourselves to the test of two shapes: one spillway with a vertical upstream slope and the other one with a 45^o upstream slope. The spillway shapes and the finite element meshes associated when no gate is present are shown in Figures 10.19 and 10.20.

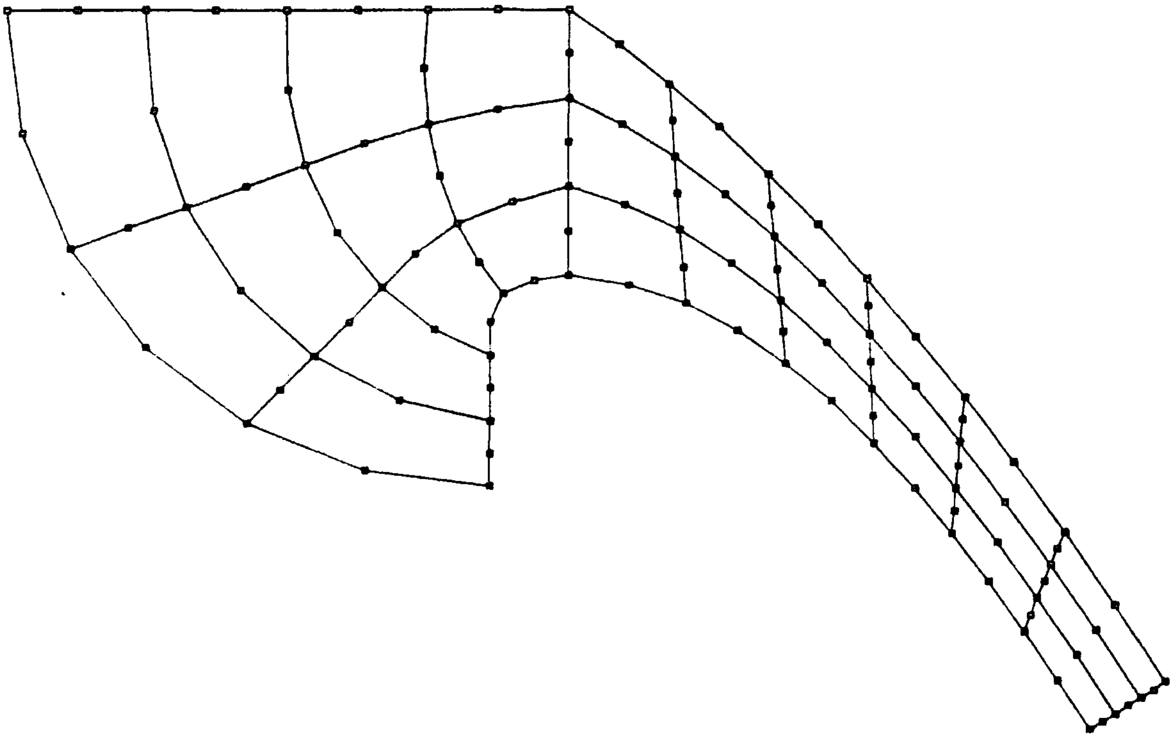


Figure 10.19: *Mesh for a spillway with a vertical upstream slope*

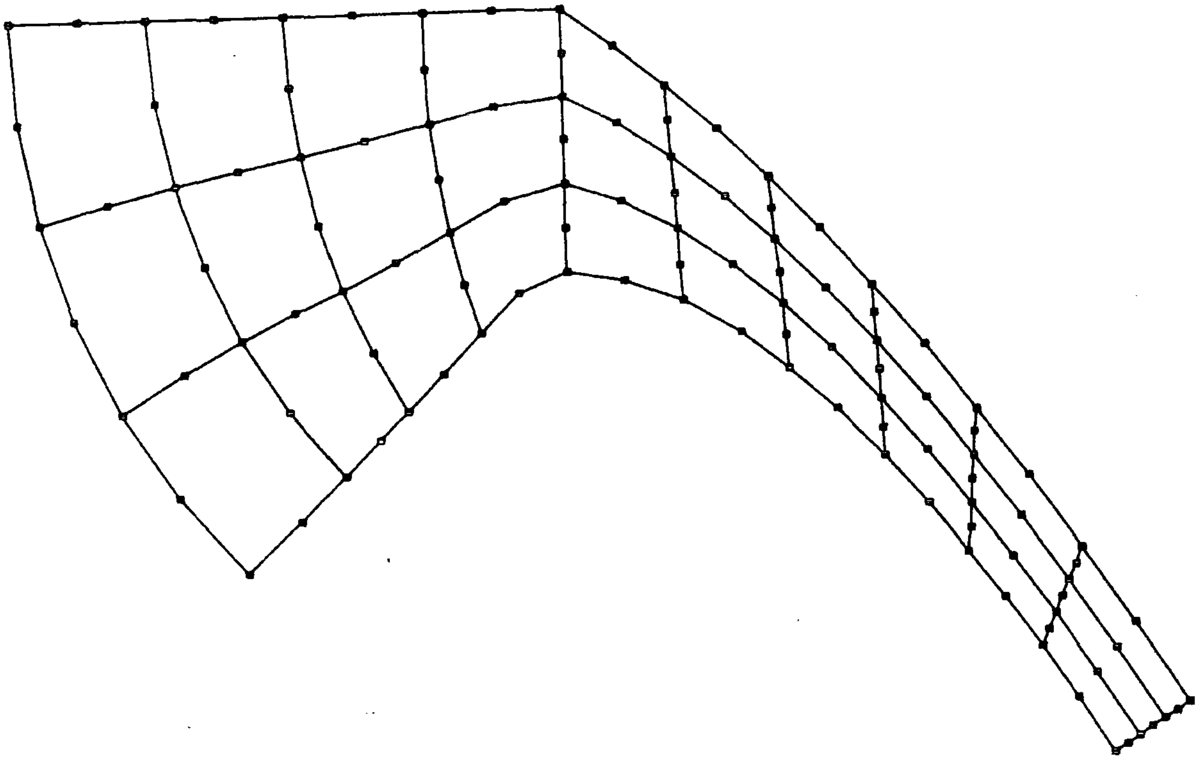


Figure 10.20: *Mesh for a spillway with a 45° upstream slope*

The choice of the initial value of Q is made around the predicted value obtained from the computation using the coefficient of discharge. The iterations on Q have been carried out by hand. The free surface shape is graphically displayed and the convergence ratio *rat* is also examined.

The chance of finding the correct value of Q the first time is small therefore most of the time convergence is sought with an incorrect value of Q. Previous studies[1] have shown that waves may develop on the free surface to account for the discrepancy in the value of Q. Such waves were observed. Convergence of this type can be seen in Figures 10.21, 10.22 and 10.23.

The two first Figures show the convergence for values of Q above and below the value of Q giving smooth surface, which is itself shown in the last Figure.

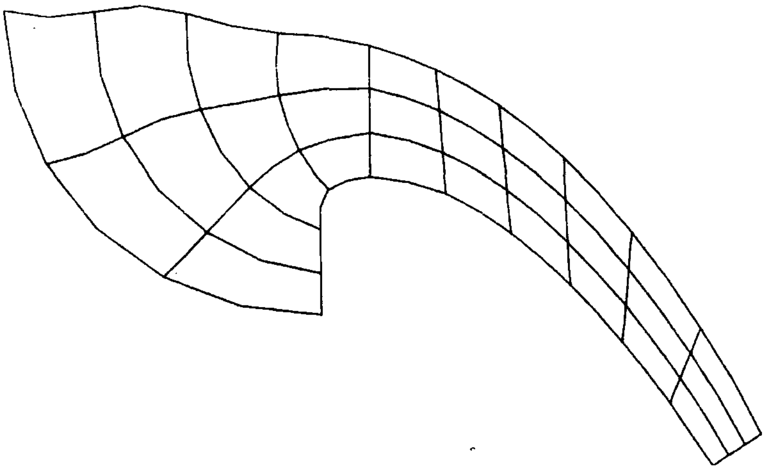


Figure 10.21: *Convergence for Q=12.5*

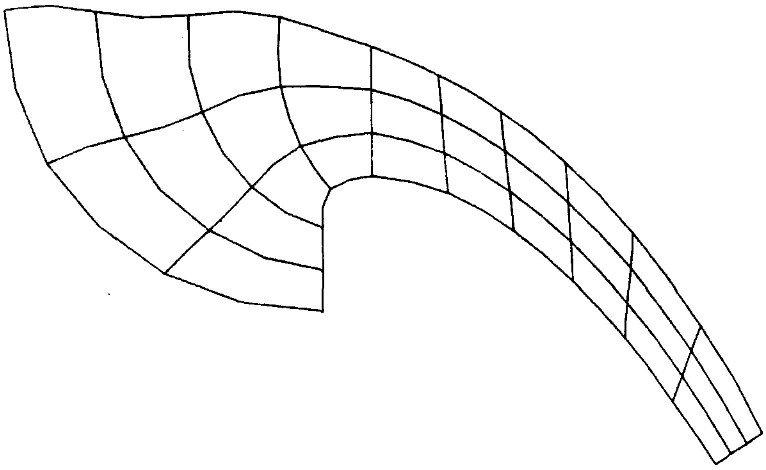


Figure 10.22: *Convergence for Q=12.0*

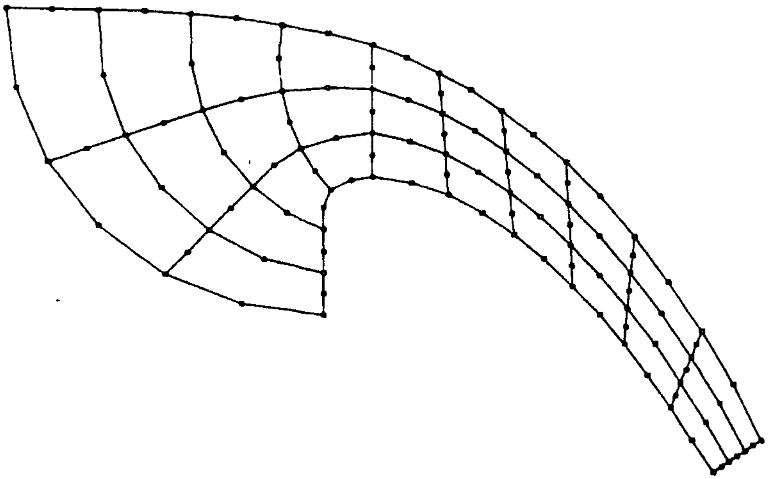


Figure 10.23: *Convergence for Q=12.28*

Good convergence has been obtained for the gated spillways. Some results are shown in Figures 10.24 and 10.25. The gate seems to help stabilise the model by bringing some extra constraints. The gate divide the flow in two parts: subcritical and supercritical regions. The critical region is reduced to the section of the water flowing beneath the gate. This region, which is the difficult part of the flow to model, is therefore limited which may explain the good results obtained in comparison to the non-gated spillways.

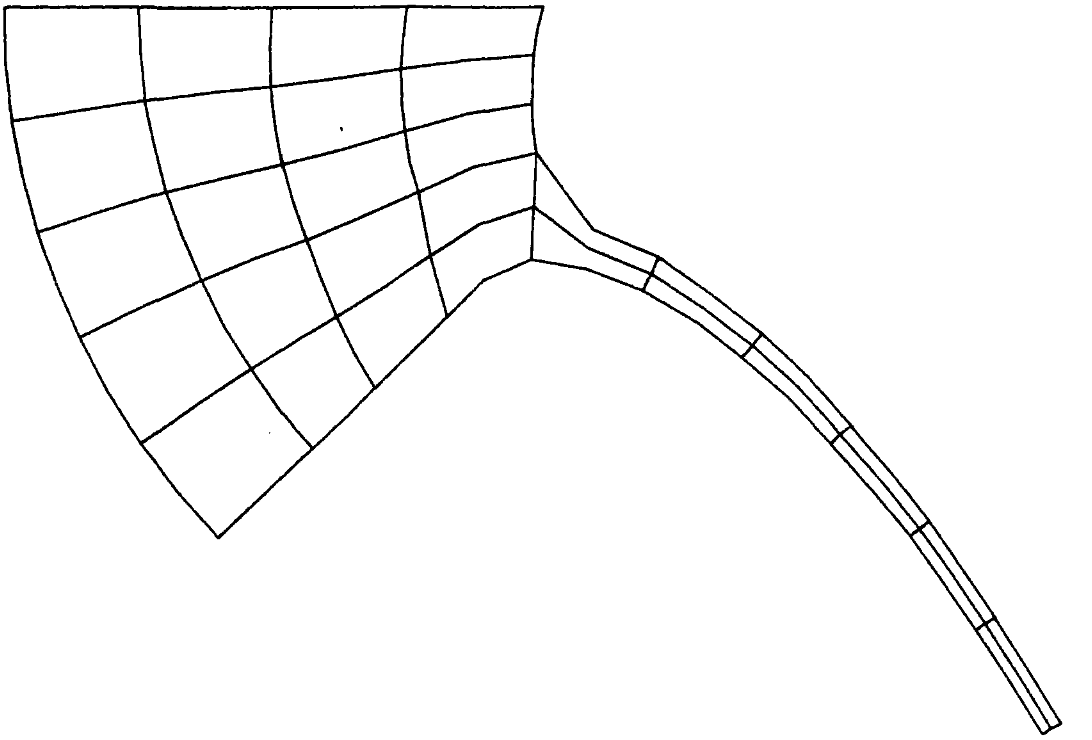


Figure 10.24: *Convergence for Q=3.2*

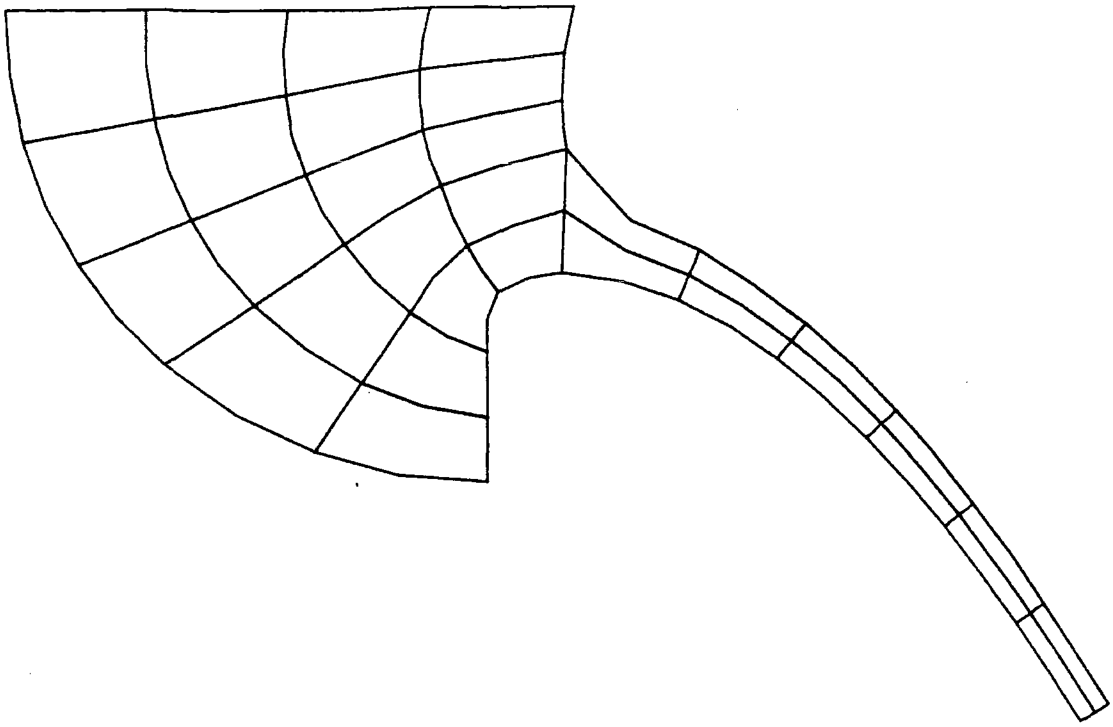


Figure 10.25: *Convergence for Q=4.5*

For non-gated problems convergence has been obtained but some problems have also appeared. An interesting observation in this case is that the instabilities come from the upstream part of the mesh where the flow is slow. When divergence occurs it is mostly induced by the nodes on the surface lying before and above the crest. The downstream part of the mesh is always very stable and of the expected shape. This is a similar behaviour to that observed in the simple tests on the element matrices. In the test with a mesh of 6 elements where nodes were pulled or pushed around the flat stable level, the subcritical flows had a small range of convergence whereas the supercritical flows were very stable. Some examples of convergence and divergence for non-gated spillways are shown in Figures 10.26, 10.27 and 10.28.

Another type of behaviour has been observed in which the model does not converge or diverge but oscillates between two or several free surface shapes. These

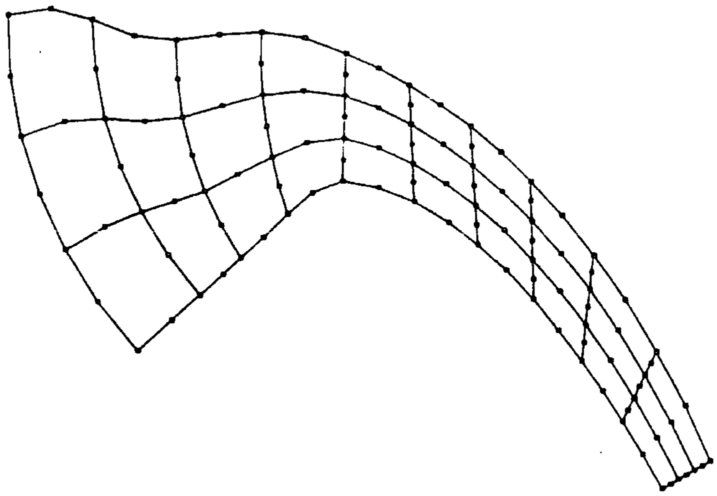


Figure 10.26: *Surface after 16 iterations, for Q=12.9*

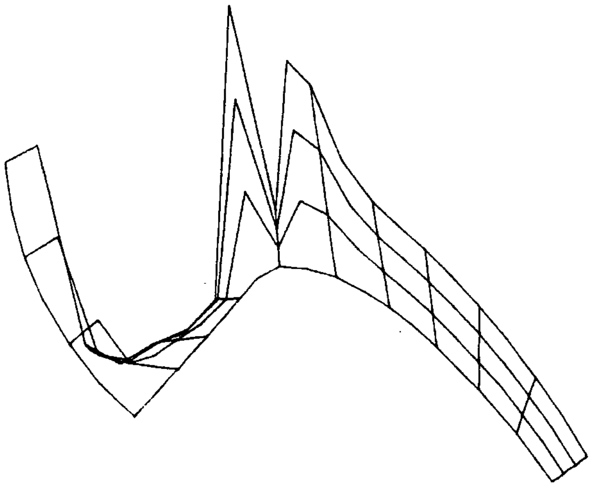


Figure 10.27: *Same as Figure 10.26, after 18 iterations*

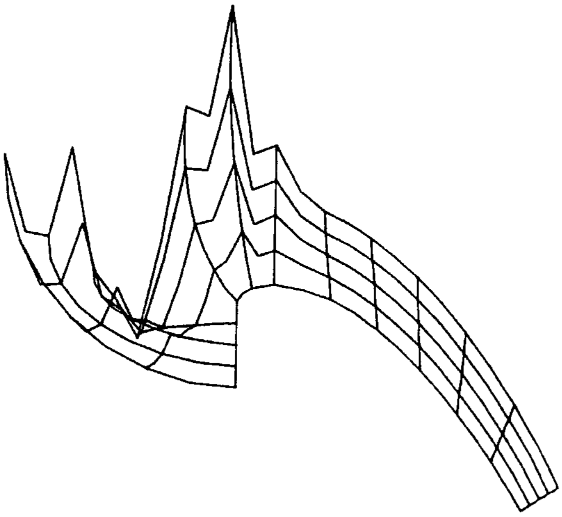


Figure 10.28: *Surface for Q=12.28, at the divergence point*

shapes tend to exhibit waves which may indicate a non-steady state solution.

In view of these problems, the question of testing the solvers used arose. This is described in the next section.

10.2.1 Tests on the nonlinear solver

The tests on the nonlinear solver have consisted of enhancing the Newton method by introducing the steepest descent technique as a means of obtaining the initial guess for Newton (see previous chapter) and refining Newton with a line search algorithm. First the tests have been carried out on flat channel problems to obtain experimentally the parameters for switching from the steepest descent method to the Newton method. Better convergence has been observed on these problems where the steepest descent method enabled convergence to occur when the Newton method on its own diverged, especially in the case when surface nodes were pulled far away above the flat level.

When the line search was introduced a peculiar behaviour appeared. As explained in the previous chapter, the line search method is capable of finding a minimum along the direction of search chosen, even if this minimum lies at infinity. While testing the line search algorithm some erratic behaviour was observed. Subsequently, a plot of the values of the functional along the direction of search which induced this behaviour was obtained. An outline of such a plot is given in Figure 10.29.

This plot shows the values of the n-dimensional (n=*totnods*+*totsnd*) functional Π taken along a one dimensional line defined by the n-dimensional vector $\Delta\mathbf{x}$ of the direction of search.

The plot shows a central region, of parabolic shape for which a local minimum exists, and a series of outer regions of tangent shape (tangent=sin/cos). Investigations have shown that the further from the solution the plot is obtained the sharper the central parabola zone is and the line search algorithms tends to go out of the central zone and converged towards minus infinity. When closer to the solution the central zone becomes very flat and the line search converged towards the minimum of the parabola.

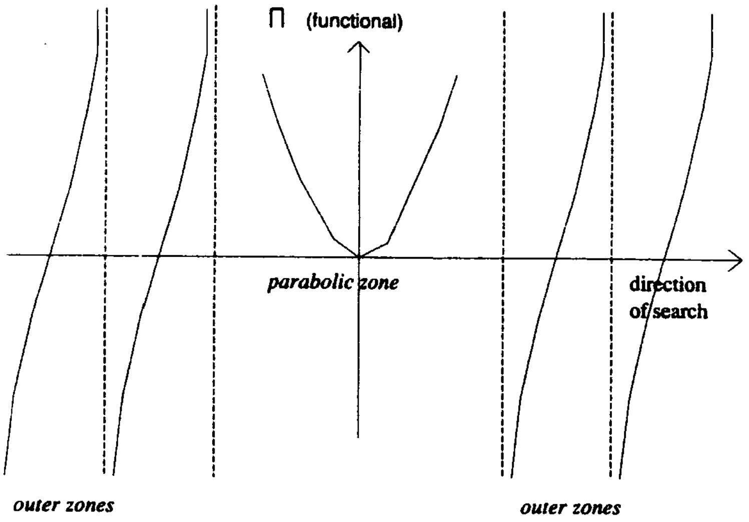


Figure 10.29: *Plot of the functional in the direction of search*

This may indicate that a minimum of the functional was not necessarily a solution of the problem. Furthermore, the repetition of the same shape in the outer region could come from the finite element approximation. Indeed, the discretised equation for the functional contains the inverse of the jacobian matrix. The jacobian matrix being a linear function of the b, the inverse is therefore a function of $1/P(b)$ where P is a polynomial function of b. This is a function singular around the zeros of the polynomial P which could correspond on the plot to the singular points between the tangent shapes of the outer regions.

No definite conclusions have been drawn from this result apart from the fact that the finite element method might incur some singularities which have to be avoided as they do not correspond to feasible solutions.

10.2.2 Tests on the linear solver

Some simple tests for assessing the accuracy of the linear solver have been carried out. They mainly focus on obtaining the condition number which indicates how sensitive the system is to small variations. This can formally be expressed as:

$$\frac{\mid \delta x \mid}{\mid x \mid} = K \frac{\mid \delta b \mid}{\mid b \mid}, \tag{10.9}$$

where K is the condition number of the system matrix A. There are various ways of evaluating the condition number. Whichever method is used the value of K

will be obtained with an error since the calculation involves at some stage the computation of Ax which is likely to be inexact if the matrix is ill-conditioned. It is therefore necessary to find some error bounds on the condition number to obtain an accurate evaluation.

The method chosen for obtaining K is based on the use of the eigenvalues of A[4,5]. A simple algorithm enables us to find K together with an error bound on the lowest eigenvalue which is the critical value. Indeed, the condition number can be defined in terms of the eigenvalues as:

$$K = \frac{\mid \lambda_L \mid}{\mid \lambda_S \mid}, \tag{10.10}$$

where λ_L is the largest eigenvalue of the matrix A and λ_S is the smallest eigenvalue. The condition number can take very large values when λ_S is in the vicinity of zero. Large condition numbers indicate high sensitivity to small variations in b when calculating x, which means that in extreme cases the solution to the system can be totally wrong.

The calculation of λ_L and λ_s can be carried out using two simple iterative algorithms: the power method and the inverse power method. Starting from an initial guessed value λ_0 the smallest eigenvalue is obtained by solving $A\lambda_1 = \lambda_0$ for λ_1, normalising λ_1 (λ_1=$\lambda_1/|\lambda_1|$), solving $A\lambda_2 = \lambda_1$ for $\lambda_2 \ldots$ etc. The value λ_n tends to $1/\lambda_S$. Similarly, the largest eigenvalue is obtained by iterative calculation of $A\lambda_i = \lambda_{i+1}$ for λ_{i+1}, and normalisation of λ_{i+1}. When the algorithm converges, λ_L is obtained.

The error on the eigenvalues can be calculated as follows[4,5]. Let λ and $\mathbf{q}$ be the approximations to the eigenvalue and the eigenvector. If these values are exactly the eigenvalue and eigenvector the following relation holds:

$$A\mathbf{q} - \lambda\mathbf{q} = 0. \tag{10.11}$$

When λ and $\mathbf{q}$ are approximations to the real values, a residual r is appears in equation (10.11) such as:

$$A\mathbf{q} - \lambda\mathbf{q} = r. \tag{10.12}$$

It can be shown that this residual gives a measure of the error bounds on the eigenvalues. This is expressed as:

$$min(\lambda_i - \lambda) \leq \parallel r \parallel, \tag{10.13}$$

where λ_i is the i^{th} eigenvalue.

The smallest condition numbers obtained for spillways when convergence occurred are in the region of 1000. They can increase well above this value up to 12000 and more when divergence happens. A typical error on λ_L would be 1.0 for λ_L=100 which is reasonable. More worrying is the error for λ_S which could be 1.5 for λ_S=0.9. This indicates that λ_S could be zero, which would make the condition number infinite. In order to assess whether this had an effect on the accuracy of the solution two tests have been carried out.

The residual r on the solution has been calculated. When the solution is obtained, the expression $Ax - b = r$ is evaluated. If $\mid r \mid$ is zero or very small it is unlikely that the result is incorrect. Otherwise the solution can be corrected by subtracting from x the value x' obtained as the solution of the system $Ax' = r$. The values of the residual found were in the range 10^{-10} to 10^{-14}, which are very small and indicate that the solution is likely to be accurate.

In order to ensure that the system of linear equations is properly solved a different solver has also been used which implements a pivoting scheme using partial pivoting around the largest pivot in a column. The results from both this solver and our original solver have been compared. If the matrix was ill-conditioned a discrepancy in the solution should have be detected, but the results were found to be identical.

10.3 Conclusions

The program developed converges well in cases when the discharge of the flow is known in advance, such as in flat channel problems. Good convergence is also observed for gated spillways where the critical part of the flow is short and constrained.

For non-gated spillway cases where the discharge is unknown divergence has been observed. Subsequently tests on the validity of the formation of the element matrices using REDUCE and the accuracy of the solvers used have been carried out. The results of these tests indicate good confidence in the code developed although no definite conclusions were drawn for the element matrices.

It is suspected that the source of the instabilities in the non-gated problems partly comes from the fact that the discharge is unknown. Further developments should therefore include the design of an automatic scheme for obtaining Q. Such a technique could be based on the introduction of Q as a variable in the functional formulation, expressing any dependency relation between the ψ's, the b's and Q, reformulating the element matrices in terms of the new independent variables and solving simultaneously for the free surface shape and the discharge. This should be easy and quick to implement since REDUCE would help with the analytical calculations.

Several facts indicate that the previous scheme could solve some the convergence problems. Firstly, most of the recent papers published on this topic include schemes where Q is found as part of the solution and the results obtained are good. Secondly, when the flat channel problems were tested, it was observed that if the value of Q was not given with enough accuracy, typically less than three figures after the decimal point, it affected the convergence pattern. This may indicate that the flow is very sensitive to the value of Q. Therefore an iterative scheme to find Q, as implemented here, may not work.

The speed-ups obtained when parallelising the formation of the element matrices are very promising. This implies that much finer meshes may be considered without incurring unreasonable delays. The user should therefore be able to visualise on the screen the free surface shape at each step in the calculations with a minimal delay. The program could be used advantageously by engineers investigating spillway shapes for a particular flow problem.

References

1. Bettess P. and Bettess J.A., 'Analysis of Free Surface Flows using Isoparametric Finite Elements', *International Journal for Numerical Methods in Engineering*, **19**, pp 1675–1689, 1983.
2. Sellin R.H.J, *Flow in channels*, Macmillan St. Martins's press, 1969.
3. Doon M.K.B.M., 'Determination of flow profile over spillway using Finite Elements', *Thesis BSC*, University of Wales, Swansea, 1983.
4. Jennings A., *Matrix Computation for Engineers and Scientists*, John Wiley & Sons, 1977.
5. Bathe K., *Finite Element procedures in Engineering Analysis*, Prentice-Hall Inc., 1982.

Appendix A

Reduce code to generate two dimensional Serendipity shape functions and mapping functions.

```
OFF ECHO ;OFF OUTPUT ;
PROCEDURE MFS2 (MIN , UPL);
COMMENT
Procedure Mapping Function Serendipity 2 dimensional
-------------------------------------------------------------------------------
PURPOSE : Calculates the 2 dimensional Serendipity Mapping Functions and
          outputs them and their derivatives as a FORTRAN program.

ARGUMENTS IN :
 Explicit MIN     : Lowest degree wanted for the mapping functions.(min=2)
          UPL     : Highest degree wanted for the mapping functions.(max=4)
 Implicit LGSF    : contains the 1 dimensional Lagrange polynomials.
          LGMF    : contains the 1 dimensional mapped Lagrange polynomials.
          LC2D    : Contains the node numbers for 2D Serendipity elements.
          INFI    : Contains the infinite directions (=0 if finite, =1 else)
          M       : Will hold the mapping functions (For nodes at infinity,
                    mapping functions are set to zero).
          WCOORD  : Will hold the coordinate system.
******************************************************************************;
BEGIN
  ARRAY COORD(2),BILIN(2),ORD(2),NUME(2,2);
  COMMENT
         Set parameters for FORTRAN output
         ---------------------------------
  ;
  NUME (1,0) := 1 ;    NUME (0,1) := 2 ;    NUME (1,1) := 4 ;
  NUMERO := NUME (INFI(1),INFI(2)) ;
  TAG1 := NUMERO ;     TAG2 := S ;
  WCOORD(0) := 2 ;     WCOORD(1) := XI ;    WCOORD(2) := ET ;
  NDER := 2 ;
  COMMENT
         Calculates 2D serendipity mapping functions
         -------------------------------------------
  ;
  FOR NB := MIN : UPL DO
  <<  NMF := NB - 1 ;
      NODES := 4*NMF ;
      INDJ := 1 ;
      COMMENT
         For each corner of the square
      ;
      FOR J := 1 : 2  DO
      << INDI := 1 ;
         FOR I := 1 : 2 DO
         << BILIN(1) := I ;      BILIN(2) := J ;
            COORD(1) := INDI ;   COORD(2) := INDJ ;
            NBNODE    := LC2D(NMF,INDI,INDJ) ;
            M(NBNODE) := 1 ;
            FOR IND := 1 : 2 DO
            << IF INFI(IND) = 0 THEN   M(NBNODE) := M(NBNODE)
                                    *SUB(VAR1=WCOORD(IND),LGSF(1,BILIN(IND)))
               ELSE
               <<IF COORD(IND) = NB THEN  M(NBNODE) := 0
                                    ELSE  M(NBNODE) := M(NBNODE)*
                             SUB(VAR1=WCOORD(IND),LGMF(1,BILIN(IND)))  >> >>;
          INDI := NB  >> ;  INDJ := NB  >> ;
      COMMENT
             If nodes between corners
      ;
      IF NB > 2 THEN
      << INDJ := 1 ;
         FOR J := 1 : 2 DO
```

```
   <<
    COMMENT
       For each node between corners, first along the a and d edges,
       second along the c and b edges where :
             xccccccccccx
             d    2     b       x=corner
             d 1      2 b       a,b,c,d=edges
             d    1     b       1,2=1st and 2nd time the loop is run
             xaaaaaaaaa x
    ;
    FOR I := 2 : NMF DO
    << ORD(1) := NMF ;   ORD(2) := 1 ;
       BILIN(1) := I ;   BILIN(2) := J ;
       COORD(1) := I ;   COORD(2) := INDJ ;
       IND1 := 1 ;
       NBNODE    := LC2D(NMF,I,INDJ) ;
       M(NBNODE) := 1 ;
       FOR IND := 1 : 2 DO
       << IF INFI(IND) = 0 THEN  M(NBNODE) := M(NBNODE)*
                      SUB(VAR1=WCOORD(IND),LGSF(ORD(IND),BILIN(IND)))
          ELSE
          <<IF COORD(IND) = NB THEN  M(NBNODE) := 0
                               ELSE  M(NBNODE) := M(NBNODE)*
                  SUB(VAR1=WCOORD(IND),LGMF(ORD(IND),BILIN(IND))) >> >>;
      COMMENT
         For the current node (edge a or c) alteration of the corner
         nodes
      ;
      FOR L := 1 STEP NMF UNTIL NB DO
      << NVERT := LC2D(NMF,L,INDJ) ;
         IF M(NVERT) NEQ 0 THEN
         <<IF INFI(IND1) = 0 THEN  SCALE := 1-ABS(L-I)/NMF
                             ELSE  SCALE := 1/(1-ABS(L-I)/NMF)   >>;
           M(NVERT) := M(NVERT) - SCALE*M(NBNODE)   >> ;
      ORD(1) := 1;         ORD(2) := NMF ;
      COORD(1) := INDJ ;   COORD(2) := I ;
      BILIN(1) := J ;      BILIN(2) := I ;
      IND1 := 2 ;
      NBNODE    := LC2D(NMF,INDJ,I) ;
      M(NBNODE) := 1 ;
      FOR IND := 1 : 2 DO
      << IF INFI(IND) = 0 THEN M(NBNODE) := M(NBNODE)*
                        SUB(VAR1=WCOORD(IND),LGSF(ORD(IND),BILIN(IND)))
         ELSE
         <<IF COORD(IND) = NB THEN  M(NBNODE) := 0
                              ELSE  M(NBNODE) := M(NBNODE)*
                  SUB(VAR1=WCOORD(IND),LGMF(ORD(IND),BILIN(IND))) >> >>;
      COMMENT
         For the current node (edge b or d) alteration of the corner
         nodes
      ;
      FOR L := 1 STEP NMF UNTIL NB DO
      << NVERT := LC2D(NMF,INDJ,L) ;
         IF M(NVERT) NEQ 0 THEN
         <<IF INFI(IND1) = 0 THEN  SCALE := 1-ABS(L-I)/NMF
                             ELSE  SCALE := 1/(1-ABS(L-I)/NMF) >> ;
           M(NVERT) := M(NVERT) - SCALE*M(NBNODE)  >>  >>   ;
    INDJ := NB >>  >> ;
WRTMF (TAG1,NDER,TAG2,NODES,NB)  >>;  END ;
```

Appendix B

Reduce code to generate FORTRAN code from the REDUCE expressions, using GENTRAN.

```
OFF ECHO ; OFF OUTPUT ;
LOAD GENTRAN ; ON GENDECS ; ON  PERIOD ;
PROCEDURE WRTMF (TAG1,NDER,TAG2,NODES,NB) ;
COMMENT
Procedure WRiTe Mapping Function
-----------------------------------------------------------------------------
PURPOSE : Outputs the Mapping Functions and Mapping Function Derivatives
          as a FORTRAN  program. It  uses the translator GENTRAN (from
          REDUCE to  FORTRAN). The Mapping function derivatives are
          calculated and translated all at once.

ARGUMENTS IN :
 Explicit TAG1   : Second digit of the subroutine name. Denotes the infinite
                   directions (1=XI, 2=ET, 3=ZE, 4=XI-ET, 5=ET-ZE, 6=XI-ZE,
                   7=XI-ET-ZE)
          NDER   : Third digit of the subroutine name. Number of spacial
                   directions.
          TAG2   : Fourth digit of the subroutine name. Tag which indicates
                   the kind of function, L for Lagrange or S for Serendipity.
          NODES  : Total number of nodes in an element.
          NB     : Fifth digit of the subroutine name.  Number of nodes per 1
                   dimensional direction.
 Implicit WCOORD : Contains the coordinate system.
          M      : Contains the mapping functions.
*****************************************************************************
;
BEGIN
  COMMENT
         Write the SUBROUTINE statment
         -----------------------------
  ;
  GENTRAN  << LITERAL  TAB!*,"SUBROUTINE M", EVAL(TAG1),EVAL(NDER),EVAL(TAG2),
                       EVAL(NB)," (",EVAL(WCOORD(1)), ", ",EVAL(WCOORD(2)) >>;
  IF WCOORD(0) = 3 THEN
    GENTRAN   << LITERAL   ", ",EVAL(WCOORD(3)) >> ;
  GENTRAN   << LITERAL ", MF, MFDL, IMFDL)",CR!*  >>;
  COMMENT
         Write the comments
         ------------------
  ;
  IF TAG2 = L THEN
    TYPENAME := "Lagrangian" ;
  IF TAG2 = S THEN
    TYPENAME := "Serendipity" ;
  GENTRAN << LITERAL
           "C *** Subroutine Mapping function ",EVAL(TAG1)," ,",
           EVAL(NDER)," dimensional ",EVAL(TYPENAME)," ",
           EVAL(NB)," nodes",CR!*,
           "C -----------------------------",
           "--------------------------------------------" , CR!* ,
           "C PURPOSE : ", CR!* ,
           "C            Forms element mapping function and derivative",
           CR!* ,"C" , CR!* ,"C ARGUMENTS IN : " , CR!*,"C",CR!*,
           "C           ",EVAL(WCOORD(1)),"     : First co-ordinate.",CR!*,
           "C           ",EVAL(WCOORD(2)),"     : Second co-ordinate.",CR!*>>;
  IF WCOORD(0) = 3 THEN
    GENTRAN   << LITERAL "C            ",EVAL(WCOORD(3)),
                          "     : Third co-ordinate.",CR!*  >> ;
  GENTRAN << LITERAL
               "C",TAB!*,"      ",
               "IMFDL  : 1st dimension of mapping function derivative array.",
               CR!*,"C",CR!*,
                "C ARGUMENTS OUT : ",CR!*,"C",CR!*,
                "C           MF      : Mapping  function array.",CR!*,
                "C           ",
                "MFDL   : Array of mapping function derivatives with respect ",
                CR!*,"C                      to local co-ordinates.",CR!*,
                "C",CR!*,"C ****************************",
                "****************************************",CR!*,"C" ,CR!*  >>;
  COMMENT
         Write the type declarations
         ---------------------------
  ;
  IF WCOORD(0) = 2 THEN
    << GENTRAN
      << DECLARE
           << MF(!*),MFDL(IMFDL,!*)                    : DIMENSION ;
              IMFDL                                    : INTEGER  ;
              MF,MFDL,EVAL(WCOORD(1)),EVAL(WCOORD(2)) : DOUBLE! PRECISION >>;
           LITERAL "C",CR!*,"C*** Define the local variables",CR!*,"C",
                   CR!* >> >>;
  IF WCOORD(0) = 3 THEN
    << GENTRAN
      << DECLARE
           << MF(!*),MFDL(IMFDL,!*)                    : DIMENSION ;
              IMFDL                                    : INTEGER ;
              MF,MFDL,EVAL(WCOORD(1)),EVAL(WCOORD(2)) : DOUBLE! PRECISION ;
              EVAL(WCOORD(3))                          : DOUBLE! PRECISION >>;
           LITERAL "C",CR!*,"C*** Define the local variables",CR!*,"C",
                   CR!* >> >>;
  COMMENT
       Call optimization algorithm, outputs the arrays VAR of temporary
       variables and R of common sub-expressions.
  ;
  OPTMF (MAX);
  FOR IND := 1 : MAX DO
  << GENTRAN  <<   EVAL(VAR(IND))  :=:  R(IND)   >> >> ;
  COMMENT
       Write the mapping functions
       ---------------------------
  ;
  GENTRAN  << LITERAL "C",CR!*,"C*** Form the element mapping functions",
                      CR!*,"C",CR!*  >> ;
  FOR IND := 1 : NODES DO
  << GENTRAN   <<  MF(IND) ::=: M (IND)  >> >> ;
  COMMENT
       Write the mapping functions derivatives
       ---------------------------------------
  ;
  GENTRAN  << LITERAL "C",CR!*,"C***Form the mapping function derivatives",
                      CR!*,"C",CR!*   >> ;
  FOR IND1 := 1 : NDER DO
  << FOR IND2 := 1 : NODES DO
       << GENTRAN  << MFDL (IND1,IND2) ::=: DF( M(IND2) , WCOORD(IND1) )
  >> >>;
  COMMENT
       Write the end statements
       ------------------------
  ;
  GENTRAN  << LITERAL  "C",CR!*,TAB!*,"RETURN",CR!*,TAB!*,"END",CR!*  >>
END ; END ;
```

Appendix C

Generated FORTRAN code for two and three dimensional quadratic mapping functions of type 1.

```
      SUBROUTINE M12S3 (XI, ET, MF, MFDL, IMFDL)
C *** Subroutine Mapping function 1 ,2 dimensional Serendipity 3 nodes
C ---------------------------------------------------------------------
C PURPOSE :
C          Forms element mapping function and derivative
C
C ARGUMENTS IN :
C
C          XI     : First co-ordinate.
C          ET     : Second co-ordinate.
C          IMFDL  : 1st dimension of mapping function derivative array.
C
C ARGUMENTS OUT :
C
C          MF     : Mapping  function array.
C          MFDL   : Array of mapping function derivatives with respect
C                   to local co-ordinates.
C
C *********************************************************************
C
      INTEGER IMFDL
      DOUBLE PRECISION MF(*),MFDL(IMFDL,*),XI,ET
C
C*** Define the local variables
C
      DOUBLE PRECISION T1,T2,T3,T4,T5,T6,T7,T8,T9,T10
      T1=ET+1.000000E0
      T2=ET+1.000000E0+XI
      T3=ET-1.000000E0
      T4=XI-1.000000E0
      T5=XI+1.000000E0
      T6=XI+1.000000E0-ET
      T7=ET+2.000000E0
      T8=ET-2.000000E0
      T9=2.000000E0*ET+XI
      T10=2.000000E0*ET-XI
C
C*** Form the element mapping functions
C
      MF(1)=-(T3*T2)/(T4)
      MF(2)=T3*T5/(2.000000E0*T4)
      MF(3)=0
      MF(4)=0
      MF(5)=0
      MF(6)=-(T5*T1)/(2.000000E0*T4)
      MF(7)=T6*T1/(T4)
      MF(8)=2.000000E0*T3*T1/(T4)
C
C***Form the mapping function derivatives
C
      MFDL(1,1)=T3*T7/(T4**2)
      MFDL(1,2)=-T3/(T4**2)
      MFDL(1,3)=0
      MFDL(1,4)=0
      MFDL(1,5)=0
      MFDL(1,6)=T1/(T4**2)
      MFDL(1,7)=T1*T8/(T4**2)
      MFDL(1,8)=-(2.000000E0*T3*T1)/(T4**2)
      MFDL(2,1)=-T9/(T4)
      MFDL(2,2)=T5/(2.000000E0*T4)
      MFDL(2,3)=0
      MFDL(2,4)=0
      MFDL(2,5)=0
      MFDL(2,6)=-T5/(2.000000E0*T4)
      MFDL(2,7)=-T10/(T4)
      MFDL(2,8)=4.000000E0*ET/(T4)
C
      RETURN
      END
      SUBROUTINE M13S3 (XI, ET, ZE, MF, MFDL, IMFDL)
C *** Subroutine Mapping function 1 ,3 dimensional Serendipity 3 nodes
C ---------------------------------------------------------------------
C PURPOSE :
C          Forms element mapping function and derivative
C
C ARGUMENTS IN :
C
C          XI     : First co-ordinate.
C          ET     : Second co-ordinate.
C          ZE     : Third co-ordinate.
C          IMFDL  : 1st dimension of mapping function derivative array.
C
C ARGUMENTS OUT :
C
C          MF     : Mapping  function array.
C          MFDL   : Array of mapping function derivatives with respect
C                   to local co-ordinates.
C
C *********************************************************************
C
      INTEGER IMFDL
      DOUBLE PRECISION MF(*),MFDL(IMFDL,*),XI,ET,ZE
C
C*** Define the local variables
C
      DOUBLE PRECISION T1,T3,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16,T17,
     . T18,T19,T20,T21,T5,T22,T2,T4,T6
      T1=ET+XI+ZE+2.000000E0
      T2=ET-1.000000E0
      T3=ZE-1.000000E0
      T4=XI-1.000000E0
      T5=XI+1.000000E0
      T6=ET+1.000000E0
      T7=ET-XI-ZE-2.000000E0
      T8=ZE+1.000000E0
      T9=ET+XI-ZE+2.000000E0
      T10=ET-XI+ZE-2.000000E0
      T11=ET+ZE+3.000000E0
      T12=ET-ZE-3.000000E0
      T13=ET-ZE+3.000000E0
      T14=ET+ZE-3.000000E0
      T15=2.000000E0*ET+XI+ZE+1.000000E0
      T16=2.000000E0*ET-XI-ZE-1.000000E0
      T17=2.000000E0*ET+XI-ZE+1.000000E0
      T18=2.000000E0*ET-XI+ZE-1.000000E0
      T19=ET+XI+2.000000E0*ZE+1.000000E0
      T20=ET-XI-(2.000000E0*ZE)-1.000000E0
      T21=ET+XI-(2.000000E0*ZE)+1.000000E0
      T22=ET-XI+2.000000E0*ZE-1.000000E0
C
C*** Form the element mapping functions
C
      MF(1)=T3*T2*T1/(2.000000E0*T4)
      MF(2)=-(T3*T2*T5)/(4.000000E0*T4)
      MF(3)=0
      MF(4)=0
      MF(5)=0
      MF(6)=T3*T6*T5/(4.000000E0*T4)
      MF(7)=T3*T6*T7/(2.000000E0*T4)
      MF(8)=-(T3*T6*T2)/(T4)
      MF(9)=-(T3*T2*T8)/(T4)
      MF(10)=0
      MF(11)=0
      MF(12)=T3*T6*T8/(T4)
      MF(13)=-(T2*T9*T8)/(2.000000E0*T4)
      MF(14)=T2*T5*T8/(4.000000E0*T4)
      MF(15)=0
      MF(16)=0
      MF(17)=0
      MF(18)=-(T6*T5*T8)/(4.000000E0*T4)
      MF(19)=-(T6*T10*T8)/(2.000000E0*T4)
      MF(20)=T6*T2*T8/(T4)
C
C***Form the mapping function derivatives
C
      MFDL(1,1)=-(T3*T2*T11)/(2.000000E0*T4**2)
      MFDL(1,2)=T3*T2/(2.000000E0*T4**2)
      MFDL(1,3)=0
      MFDL(1,4)=0
      MFDL(1,5)=0
      MFDL(1,6)=-(T3*T6)/(2.000000E0*T4**2)
      MFDL(1,7)=-(T3*T6*T12)/(2.000000E0*T4**2)
      MFDL(1,8)=T3*T6*T2/(T4**2)
      MFDL(1,9)=T3*T2*T8/(T4**2)
      MFDL(1,10)=0
      MFDL(1,11)=0
      MFDL(1,12)=-(T3*T6*T8)/(T4**2)
      MFDL(1,13)=T2*T13*T8/(2.000000E0*T4**2)
      MFDL(1,14)=-(T2*T8)/(2.000000E0*T4**2)
      MFDL(1,15)=0
      MFDL(1,16)=0
      MFDL(1,17)=0
      MFDL(1,18)=T6*T8/(2.000000E0*T4**2)
      MFDL(1,19)=T6*T14*T8/(2.000000E0*T4**2)
      MFDL(1,20)=-(T6*T2*T8)/(T4**2)
      MFDL(2,1)=T3*T15/(2.000000E0*T4)
      MFDL(2,2)=-(T3*T5)/(4.000000E0*T4)
      MFDL(2,3)=0
      MFDL(2,4)=0
      MFDL(2,5)=0
      MFDL(2,6)=T3*T5/(4.000000E0*T4)
      MFDL(2,7)=T3*T16/(2.000000E0*T4)
      MFDL(2,8)=-(2.000000E0*T3*ET)/(T4)
      MFDL(2,9)=-(T3*T8)/(T4)
      MFDL(2,10)=0
      MFDL(2,11)=0
      MFDL(2,12)=T3*T8/(T4)
      MFDL(2,13)=-(T8*T17)/(2.000000E0*T4)
      MFDL(2,14)=T5*T8/(4.000000E0*T4)
      MFDL(2,15)=0
      MFDL(2,16)=0
      MFDL(2,17)=0
      MFDL(2,18)=-(T5*T8)/(4.000000E0*T4)
      MFDL(2,19)=-(T18*T8)/(2.000000E0*T4)
      MFDL(2,20)=2.000000E0*ET*T8/(T4)
      MFDL(3,1)=T19*T2/(2.000000E0*T4)
      MFDL(3,2)=-(T2*T5)/(4.000000E0*T4)
      MFDL(3,3)=0
      MFDL(3,4)=0
      MFDL(3,5)=0
      MFDL(3,6)=T6*T5/(4.000000E0*T4)
      MFDL(3,7)=T6*T20/(2.000000E0*T4)
      MFDL(3,8)=-(T6*T2)/(T4)
      MFDL(3,9)=-(2.000000E0*T2*ZE)/(T4)
      MFDL(3,10)=0
      MFDL(3,11)=0
      MFDL(3,12)=2.000000E0*T6*ZE/(T4)
      MFDL(3,13)=-(T2*T21)/(2.000000E0*T4)
      MFDL(3,14)=T2*T5/(4.000000E0*T4)
      MFDL(3,15)=0
      MFDL(3,16)=0
      MFDL(3,17)=0
      MFDL(3,18)=-(T6*T5)/(4.000000E0*T4)
      MFDL(3,19)=-(T6*T22)/(2.000000E0*T4)
      MFDL(3,20)=T6*T2/(T4)
C
      RETURN
      END
```

Appendix D

REDUCE program to generate bending element matrices.

```
OFF OUTPUT;OFF ECHO; OUT "HERM.OUT";
COMMENT  ***  Definition of the procedure to write out the subroutine
         ***  statement and the subroutine header of comments 1D CASE;
PROCEDURE WRITEEL1(TYPE,NAME,ELNAME,N1DIM);
BEGIN
       IF TYPE=1 THEN <<WRITE "      SUBROUTINE CALELM (A,ELM,IELM,RHO)">>
  ELSE IF TYPE=2 THEN <<WRITE "      SUBROUTINE CALELG (A,ELG,IELG,SIGMA)">>
  ELSE IF TYPE=3 THEN <<WRITE "      SUBROUTINE CALELK (A,EI,ELK,IELK)">>;
  WRITE "C";
  WRITE "C PURPOSE :";
  WRITE "C           Forms the ",NAME," matrix for the ";
  WRITE "C           ",N1DIM,"-noded one dimensional elements.";
  WRITE "C";
  WRITE "C **********************************************************";
  WRITE "C";
  WRITE "      INTEGER             I",ELNAME;
       IF TYPE=1 THEN <<WRITE "      DOUBLE PRECISION    A,ELM,RHO">>
  ELSE IF TYPE=2 THEN <<WRITE "      DOUBLE PRECISION    A,ELG,SIGMA">>
  ELSE IF TYPE=3 THEN <<WRITE "      DOUBLE PRECISION    A,EI,ELK">>;
  WRITE "      DIMENSION          ",ELNAME,"(I",ELNAME,",*)";
  WRITE "C";
  WRITEEL3(TYPE,2*N);
  WRITE "C";  WRITE "      RETURN";  WRITE "      END"
END;
COMMENT  ***  Definition of the procedure to write out the subroutine
         ***  statement and the subroutine header of comments 2D CASE;
PROCEDURE WRITEEL2 (TYPE,NAME,ELNAME,N);
BEGIN
       IF TYPE=1 THEN <<WRITE "      SUBROUTINE CALELM (A,B,ELM,IELM,RHO)">>
  ELSE IF TYPE=2 THEN <<WRITE "      SUBROUTINE CALELG (A,B,ELG,IELG,",
                                                  "SIGMAX,SIGMAY,",
                                                  "THOXY)" >>
  ELSE IF TYPE=3 THEN <<WRITE "      SUBROUTINE CALELK (A,B,DX,DY,DXY,D1,",
                                                  "ELK,IELK)">>;
  WRITE "C";
  WRITE "C PURPOSE : ";
  WRITE "C           Forms the ",NAME," matrix for the ";
  WRITE "C           ",N,"-noded two dimensional elements.";
  WRITE "C";
  WRITE "C **********************************************************";
  WRITE "C";
  WRITE "      INTEGER            I",ELNAME;
       IF TYPE=1 THEN <<WRITE "      DOUBLE PRECISION   A,B,ELM,RHO" >>
  ELSE IF TYPE=2 THEN <<WRITE "      DOUBLE PRECISION   A,B,ELG,SIGMAX,",
                                                  "SIGMAY,THOXY" >>
  ELSE IF TYPE=3 THEN <<WRITE "      DOUBLE PRECISION   A,B,DX,DY,DXY,D1,",
                                                  "ELK">>;
  WRITE "      DIMENSION          ",ELNAME,"(I",ELNAME,",*)";
  WRITE "C";
  WRITEEL3(TYPE,4*N);
  WRITE "C";   WRITE "      RETURN";   WRITE "      END";
END;
COMMENT  ***  Definition of the procedure to write out the element matrices;
PROCEDURE WRITEEL3 (TYPE,LIMIT);
BEGIN
       IF TYPE=1 THEN <<FOR  I:=1:LIMIT DO
                        <<FOR  J:=1:I  DO
                          <<WRITE "      ELM(",I,",",J,") = ",EELM(I,J) >>
                      >> >>
  ELSE IF TYPE=2 THEN <<FOR  I:=1:LIMIT DO
                        <<FOR  J:=1:I  DO
                          <<WRITE "      ELG(",I,",",J,") = ",EELG(I,J) >>
                      >> >>
  ELSE IF TYPE=3 THEN <<FOR  I:=1:LIMIT DO
                        <<FOR  J:=1:I  DO
                          <<WRITE "      ELK(",I,",",J,") = ",EELK(I,J) >>
                      >> >>;
END;
COMMENT  ***  Definition of the elements sizes.
         ***  N     is the total number of nodes
         ***  N1DIM is the number of nodes in each dimension
         ***  DIM   is the number of dimensions;
N:=4; N1DIM:=2; DIM:=2;
COMMENT  ***  Variables declarations;
ARRAY  XX(N),COORD(2);
MATRIX HERM(1,2*N1DIM),DHERM(1,2*N1DIM),D2HERM(1,2*N1DIM);
MATRIX HERMXY(1,4*N),DHERMXY(2,4*N),D2HERMXY(3,4*N);
MATRIX D(3,3),G(2,2);
OPERATOR INT1,INT2;
COMMENT  ***  Definition of the integration operator;
FOR ALL F,U,L LET  INT1(F,U)    = INT(F,U),
                   INT2(F,U,L) = SUB(U=L,INT1(F,U))-SUB(U=0,INT1(F,U));
COMMENT  ***  Initialisations of the nodes co-ordinates in one dimension;
FOR I:=1:N1DIM DO <<XX(I) := L*(I-1)/(N1DIM-1) >>;
COMMENT  ***  Initialisation of the constants of the problem;
IF DIM=2 THEN << COORD(1):=X;     COORD(2):=Y;
                 G(1,1):= SIGMAX; G(1,2):= THOXY;
                 G(2,1):= THOXY;  G(2,2):= SIGMAY;
                 D(1,1):= DX;     D(1,2):= D1;       D(1,3):= 0 ;
                 D(2,1):= D1;     D(2,2):= DY;       D(2,3):= 0 ;
                 D(3,1):= 0 ;     D(3,2):= 0 ;       D(3,3):= DXY >> ;
COMMENT  ***  Calculation of the one dimensional Hermite polynomials;
FOR  I:=1:N1DIM  DO
<<LL :=1;    AA :=0;
  FOR  J:=1:N1DIM  DO
  <<IF  I NEQ J THEN
    <<LL := LL*(X-XX(J))/(XX(I)-XX(J)) ;
      AA := AA + 1/(XX(I)-XX(J))  >>
  >>;
  AA := -2* AA ;  BB := 1-AA*XX(I);
  HERM   (1,2*I-1) := (AA*X+BB) * LL * LL ;
  HERM   (1,2*I)   := (X-XX(I)) * LL * LL ;
  IF DIM=1 THEN <<  DHERM  (1,2*I-1) := DF (HERM (1,2*I-1),X);
                    DHERM  (1,2*I)   := DF (HERM (1,2*I),X);
                    D2HERM (1,2*I-1) := DF (DHERM(1,2*I-1),X);
                    D2HERM (1,2*I)   := DF (DHERM(1,2*I),X) >>
>> ;
COMMENT  ***  Calculation of the two dimensional Hermite polynomials;
IF DIM=2 THEN << K1:=0;
                 FOR II:=0:N1DIM-1 DO
                 <<ADI:=2*II;
                   FOR JJ:=0:N1DIM-1 DO
                   <<ADJ := 2*JJ;
                     FOR I:=1:2 DO
                     <<K3:= I+ADI;
                       FOR J:=1:2 DO
                       <<K2 := J+ADJ;
                         K1 := K1+1;
                         HERMXY(1,K1) := SUB(L=A,HERM(1,K2))*
                                 SUB(X=Y,SUB(L=B,HERM(1,K3)));
                         FOR K:=1:2 DO
                         <<DHERMXY(K,K1)  := DF (HERMXY(1,K1),COORD(K));
                           D2HERMXY(K,K1) := DF (DHERMXY(K,K1),COORD(K)) >> ;
                           D2HERMXY(3,K1) := DF (HERMXY(1,K1),COORD(1),
                                                              COORD(2))
>> >> >> >> >>;
COMMENT  ***  Switch to FORTRAN mode;
ON FORT;OFF PERIOD;
COMMENT  ***  Calculate and output as FORTRAN code all the element matrices;
FOR TYPE:=1:3 DO
<<      IF TYPE=1 THEN
   <<ELNAME:=ELM; NAME="mass" ;
     IF DIM=1 THEN << EELM := TP(HERM)  *HERM ;
                      FOR I:=1:2*N DO  <<FOR J:=1:I DO
                        <<EELM(I,J):=RHO * SUB(L=A,INT2(EELM(I,J),X,L) )
                   >> >>;
                      WRITEEL1(TYPE,NAME,ELNAME,N1DIM) >>
     ELSE
                   <<EELM := RHO * TP(HERMXY)  *HERMXY ;
                     FOR I:=1:4*N DO <<FOR J:=1:I DO
                       <<EELM(I,J):=INT2( INT2(EELM(I,J),Y,B) , X,A) >> >>;
                     WRITEEL2(TYPE,NAME,ELNAME,N) >> >>
   ELSE IF TYPE=2 THEN
   << ELNAME:=ELG; NAME="geometric stiffness" ;
      IF DIM=1 THEN <<EELG := TP(DHERM) *DHERM ;
                      FOR I:=1:2*N DO <<FOR J:=1:I DO
                        << EELG(I,J):=SIGMA*SUB(L=A,INT2(EELG(I,J),X,L) )
                    >> >>;
                      WRITEEL1(TYPE,NAME,ELNAME,N1DIM) >>
      ELSE          <<EELG := TP(DHERMXY)* G *DHERMXY ;
                      FOR I:=1:4*N DO  <<FOR J:=1:I DO
                        <<EELG(I,J):=INT2( INT2(EELG(I,J),Y,B) , X,A) >> >>;
                      WRITEEL2(TYPE,NAME,ELNAME,N) >> >>
   ELSE IF TYPE=3 THEN
   << ELNAME:=ELK; NAME="stiffness" ;
      IF DIM=1 THEN <<EELK := TP(D2HERM) *D2HERM ;
                      FOR I:=1:2*N DO <<FOR J:=1:I DO
                        << EELK(I,J):=EI   *SUB(L=A,INT2(EELK(I,J),X,L) )
                    >> >>;
                      WRITEEL1(TYPE,NAME,ELNAME,N1DIM) >>
      ELSE          <<EELK := TP(D2HERMXY)* D *D2HERMXY ;
                      FOR I:=1:4*N DO <<FOR J:=1:I DO
                        <<EELK(I,J):=INT2( INT2(EELK(I,J),Y,B) , X,A) >> >>;
                      WRITEEL2(TYPE,NAME,ELNAME,N) >>
>> >> ;
END;
```

Appendix E

Modified REDUCE program to generate bending element matrices.

```
OFF OUTPUT;OFF ECHO;OUT "HERM.OUT";
COMMENT  ***  Definition of the procedure to write out the subroutine
         ***  statement and the subroutine header of comments -- 1D CASE;
PROCEDURE WELEM1(TYPE,NAME,ELNAME,N1DIM);
BEGIN
       IF TYPE=1 THEN <<WRITE "      SUBROUTINE CALELM (A,ELM,IELM,RHO)">>
  ELSE IF TYPE=2 THEN <<WRITE "      SUBROUTINE CALELG (A,ELG,IELG,SIGMA)">>
  ELSE IF TYPE=3 THEN <<WRITE "      SUBROUTINE CALELK (A,EI,ELK,IELK)">>;
  WRITE "C";
  WRITE "C PURPOSE :";
  WRITE "C           Forms the ",NAME," matrix for the ";
  WRITE "C           ",N1DIM,"-noded one dimensional elements.";
  WRITE "C";
  WRITE "C *********************************************************";
  WRITE "C";
  WRITE "      INTEGER             I",ELNAME;
       IF TYPE=1 THEN <<WRITE "      DOUBLE PRECISION    A,ELM,RHO">>
  ELSE IF TYPE=2 THEN <<WRITE "      DOUBLE PRECISION    A,ELG,SIGMA">>
  ELSE IF TYPE=3 THEN <<WRITE "      DOUBLE PRECISION    A,EI,ELK">>;
  WRITE "      DIMENSION           ",ELNAME,"(I",ELNAME,",*)";
  WRITE "C";
END;
COMMENT  ***  Definition of the procedure to write out the subroutine
         ***  statement and the subroutine header of comments 2D CASE;
PROCEDURE WELEM2 (TYPE,NAME,ELNAME,N);
BEGIN
       IF TYPE=1 THEN <<WRITE "      SUBROUTINE CALELM (A,B,ELM,IELM,RHO)">>
  ELSE IF TYPE=2 THEN <<WRITE "      SUBROUTINE CALELG (A,B,ELG,IELG,",
                                                    "SIGMAX,SIGMAY,",
                                                    "THOXY)" >>
  ELSE IF TYPE=3 THEN <<WRITE "      SUBROUTINE CALELK (A,B,DX,DY,DXY,D1,",
                                                    "ELK,IELK)">>;
  WRITE "C";
  WRITE "C PURPOSE : ";
  WRITE "C           Forms the ",NAME," matrix for the ";
  WRITE "C           ",N,"-noded two dimensional elements.";
  WRITE "C";
  WRITE "C *********************************************************";
  WRITE "C";
  WRITE "      INTEGER            I",ELNAME;
       IF TYPE=1 THEN <<WRITE "      DOUBLE PRECISION   A,B,ELM,RHO" >>
  ELSE IF TYPE=2 THEN <<WRITE "      DOUBLE PRECISION   A,B,ELG,SIGMAX,",
                                                    "SIGMAY,THOXY" >>
  ELSE IF TYPE=3 THEN <<WRITE "      DOUBLE PRECISION   A,B,DX,DY,DXY,D1,",
                                                    "ELK">>;
  WRITE "      DIMENSION          ",ELNAME,"(I",ELNAME,",*)";
  WRITE "C";
END;
COMMENT  ***  Definition of the procedure to calculate the two dimensional
         ***  Hermite Polynomials;
PROCEDURE CHEMXY  (TYPE,N1DIM);
BEGIN
  IND := 0;
  FOR J:=1:N1DIM DO
  <<FOR I:=1:N1DIM DO
    <<IND := IND+1;
      INDI(1) := 2*I-1;  INDI(2) := 2*I;
      INDJ(1) := 2*J-1;  INDJ(2) := 2*J;
      FOR JJ:=1:2 DO
      <<FOR II:=1:2 DO
        <<      IF TYPE=1 THEN
           <<HERMXY(IND,2*JJ+II-2)    := SUB(L=A,HERM(1,INDI(II)))*
```

```
                                SUB(X=Y,SUB(L=B,HERM(1,INDJ(JJ)))) >>
          ELSE IF TYPE=2 THEN
           <<HERMXY(IND,2*JJ+II-2)   := SUB(L=A,HERM(2,INDI(II)))*
                                SUB(X=Y,SUB(L=B,HERM(1,INDJ(JJ))));
             HERMXY(IND,2*JJ+II-2+4) := SUB(L=A,HERM(1,INDI(II)))*
                                SUB(X=Y,SUB(L=B,HERM(2,INDJ(JJ)))) >>
          ELSE IF TYPE=3 THEN
           <<HERMXY(IND,2*JJ+II-2)   := SUB(L=A,HERM(3,INDI(II)))*
                                SUB(X=Y,SUB(L=B,HERM(1,INDJ(JJ))));
             HERMXY(IND,2*JJ+II-2+4) := SUB(L=A,HERM(1,INDI(II)))*
                                SUB(X=Y,SUB(L=B,HERM(3,INDJ(JJ))));
             HERMXY(IND,2*JJ+II-2+8) := SUB(L=A,HERM(2,INDI(II)))*
                                SUB(X=Y,SUB(L=B,HERM(2,INDJ(JJ)))) >>
>> >> >> >>;
END;
COMMENT  ***  Definition of the procedure which calculates the element
         ***  matrices, integrates them and output them as FORTRAN routines
         ***  (only half of the matrices are dealt with as they are
         ***   symmetrical) 1D CASE ;
PROCEDURE CWEL1 (TYPE,N);
BEGIN
  FOR I:=1:2*N DO
  <<FOR J:=I:2*N DO
    <<IF TYPE=1 THEN
      <<EL := RHO*SUB(L=A,HERM(1,I))*SUB(L=A,HERM(1,J));
        EL := INT2(EL,X,A);
        WRITE "       ELM(",I,",",J,") = ",EL >>
      ELSE IF TYPE=2 THEN
      <<EL := SIGMA*SUB(L=A,HERM(2,I))*SUB(L=A,HERM(2,J));
        EL := INT2(EL,X,A);
        WRITE "       ELG(",I,",",J,") = ",EL >>
      ELSE IF TYPE=3 THEN
      <<EL := EI*SUB(L=A,HERM(3,I))*SUB(L=A,HERM(3,J));
        EL := INT2(EL,X,A);
        WRITE "       ELK(",I,",",J,") = ",EL >>
  >> >>;
END;
COMMENT  ***  Definition of the procedure which calculates the element
         ***  matrices, integrates them and output them as FORTRAN routines
         ***  (only half of the matrices are dealt with as they are
         ***   symmetrical) 2D CASE;
PROCEDURE CWEL2 (TYPE,N);
BEGIN
  FOR I:=1:N DO
  <<FOR J:=I:N DO
    <<FOR II:=1:4 DO
      <<IF  I=J  THEN LIMIT:=II ELSE LIMIT:=1;
        FOR JJ:=LIMIT:4 DO
        <<IF TYPE=1 THEN
          <<EL := RHO*HERMXY(I,II)*HERMXY(J,JJ);
            EL := INT2(INT2(EL,Y,B),X,A);
            WRITE "       ELM(",II+4*(I-1),",",JJ+4*(J-1),") = ",EL >>
          ELSE IF TYPE=2 THEN
          <<EL := SIGMAX * HERMXY(I,II)    * HERMXY(J,JJ)+
                  SIGMAY * HERMXY(I,II+4)  * HERMXY(J,JJ+4)+
                  TAUXY  * HERMXY(I,II)    * HERMXY(J,JJ+4)+
                  TAUXY  * HERMXY(I,II+4)  * HERMXY(J,JJ);
            EL := INT2(INT2(EL,Y,B),X,A);
            WRITE "       ELG(",II+4*(I-1),",",JJ+4*(J-1),") = ",EL >>
          ELSE IF TYPE=3 THEN
          <<EL := DX  * HERMXY(I,II)   * HERMXY(J,JJ)+
                  DY  * HERMXY(I,II+4) * HERMXY(J,JJ+4)+
                  DXY * HERMXY(I,II+8) * HERMXY(J,JJ+8)+
                  D1  * HERMXY(I,II+4) * HERMXY(J,JJ)+
                  D1  * HERMXY(I,II)   * HERMXY(J,JJ+4);
            EL := INT2(INT2(EL,Y,B),X,A);
            EL := SUB(DX=D,EL);
            EL := SUB(DY=D,EL);
            EL := SUB(D1=NU*D,EL);
            EL := SUB(DXY=2*(1-NU)*D,EL);
            WRITE "       ELK(",II+4*(I-1),",",JJ+4*(J-1),") = ",EL >>
  >> >> >> >>;
END;
COMMENT  ***  Definition of the element sizes.
         ***  N     is the total number of nodes
         ***  N1DIM is the number of nodes in each dimension
         ***  DIM   is the number of dimensions (1 or 2);
N:=4;  N1DIM:=2;  DIM:=2;
COMMENT  ***  Variables declarations;
ARRAY  XX(N),INDI(2),INDJ(2);
MATRIX HERM(3,2*N1DIM),HERMXY(N,12);
OPERATOR INT1,INT2;
COMMENT  ***  Definition of the integration operator;
FOR ALL F,U,L LET  INT1(F,U)   = INT(F,U),
                   INT2(F,U,L) = SUB(U=L,INT1(F,U))-SUB(U=0,INT1(F,U));
COMMENT  ***  Initialisations of the nodes co-ordinates in one dimension;
FOR I:=1:N1DIM DO <<XX(I):=L*(I-1)/(N1DIM-1)>>;
COMMENT  ***  Calculation of the one dimensional Hermite polynomials;
FOR  I:=1:N1DIM DO
<<LL :=1; AA :=0;
  FOR  J:=1:N1DIM  DO
  IF  I NEQ J THEN
    <<LL := LL*(X-XX(J))/(XX(I)-XX(J)) ; AA := AA + 1/(XX(I)-XX(J)) >>;
  AA := -2* AA ; BB := 1-AA*XX(I);
  HERM(1,2*I-1) := ( AA*X+BB ) * LL * LL ;
  HERM(1,2*I)   := ( X-XX(I) ) * LL * LL ;
  HERM(2,2*I-1) := DF( HERM(1,2*I-1),X);HERM(3,2*I-1) := DF(HERM(2,2*I-1),X);
  HERM(2,2*I)   := DF( HERM(1,2*I),X);  HERM(3,2*I)   := DF(HERM(2,2*I),X)
>>;
COMMENT  ***  Switch to FORTRAN mode ;
ON  FORT; OFF PERIOD;
COMMENT  ***  Calculate and output as FORTRAN code all the element matrices;
FOR TYPE:=1:3 DO
<<     IF TYPE=1 THEN << ELNAME:=ELM; NAME:="mass" >>
  ELSE IF TYPE=2 THEN << ELNAME:=ELG; NAME:="geometric stiffness" >>
  ELSE IF TYPE=3 THEN << ELNAME:=ELK; NAME:="stiffness" >>;
  IF DIM=1 THEN << WELEM1(TYPE,NAME,ELNAME,N1DIM);
                   CWEL1(TYPE,N1DIM) >>
  ELSE
                << WELEM2(TYPE,NAME,ELNAME,N);
                   CHEMXY(TYPE,N1DIM) ;
                   CWEL2(TYPE,N) >>;
  WRITE "      RETURN";
  WRITE "      END   "
>>;
END;
```

Appendix F

Generated FORTRAN code for the stiffness matrix in one and two dimensions.

```
      SUBROUTINE CALELK (A,EI,ELK,IELK)
C
C PURPOSE :
C          Forms the stiffness matrix for the
C          2-noded one dimensional elements.
C
C **********************************************************
C
      INTEGER              IELK
      DOUBLE PRECISION     A,EI,ELK
      DIMENSION            ELK(IELK,*)
C
      ELK(1,1) = (12*EI)/A**3
      ELK(2,1) = (6*EI)/A**2
      ELK(2,2) = (4*EI)/A
      ELK(3,1) = -(12*EI)/A**3
      ELK(3,2) = -(6*EI)/A**2
      ELK(3,3) = (12*EI)/A**3
      ELK(4,1) = (6*EI)/A**2
      ELK(4,2) = (2*EI)/A
      ELK(4,3) = -(6*EI)/A**2
      ELK(4,4) = (4*EI)/A
C
      RETURN
      END
      SUBROUTINE CALELK (A,B,DX,DY,DXY,D1,ELK,IELK)
C
C PURPOSE :
C          Forms the stiffness matrix for the
C          2-noded two dimensional elements.
C
C **********************************************************
C
      INTEGER              IELK
      DOUBLE PRECISION     A,B,DX,DY,DXY,D1,ELK,TEMP1,TEMP2,TEMP3,TEMP4,
     .                     TEMP5,TEMP6,TEMP7,TEMP8,TEMP9,TEMP10,TEMP11,
     .                     TEMP12,TEMP13
      DIMENSION            ELK(IELK,*)
C
C***  Definition of the temporary variables
C
      TEMP1 = A**4*DY
      TEMP2 = A**2*B**2*DXY
      TEMP3 = A**2*B**2*D1
      TEMP4 = B**4*DX
      TEMP5 = A*B
      TEMP6 = TEMP5*A
      TEMP7 = TEMP5*B
      TEMP8 = TEMP5*A*B
      TEMP9 = TEMP5*A**2
      TEMP10 = TEMP8*A
      TEMP11 = TEMP5*B**2
      TEMP12 = TEMP8*B
      TEMP13 = TEMP8*A*B
C
      ELK(1,1) = (12*(65*TEMP1+21*TEMP2+42*TEMP3+65*TEMP4))/(175*TEMP13)
      ELK(2,1) = (110*TEMP1+21*TEMP2+252*TEMP3+390*TEMP4)/(175*TEMP12)
      ELK(2,2) = (4*(5*TEMP1+7*TEMP2+14*TEMP3+65*TEMP4))/(175*TEMP11)
      ELK(3,1) = (390*TEMP1+21*TEMP2+252*TEMP3+110*TEMP4)/(175*TEMP10)
      ELK(3,2) = (220*TEMP1+7*TEMP2+854*TEMP3+220*TEMP4)/(700*TEMP8)
      ELK(3,3) = (4*(65*TEMP1+7*TEMP2+14*TEMP3+5*TEMP4))/(175*TEMP9)
      ELK(4,1) = (220*TEMP1+7*TEMP2+154*TEMP3+220*TEMP4)/(700*TEMP8)
ELK(4,2) = (30*TEMP1+7*TEMP2+84*TEMP3+110*TEMP4)/(525*TEMP7)
ELK(4,3) = (110*TEMP1+7*TEMP2+84*TEMP3+30*TEMP4)/(525*TEMP6)
ELK(4,4) = (4*(15*TEMP1+7*TEMP2+14*TEMP3+15*TEMP4))/(1575*TEMP5)
ELK(5,1) = (6*(45*TEMP1-42*TEMP2-84*TEMP3-130*TEMP4))/(175*TEMP13)
ELK(5,2) = (65*TEMP1-21*TEMP2-42*TEMP3-390*TEMP4)/(175*TEMP12)
ELK(5,3) = (135*TEMP1-21*TEMP2-252*TEMP3-110*TEMP4)/(175*TEMP10)
ELK(5,4) = (130*TEMP1-7*TEMP2-84*TEMP3-220*TEMP4)/(700*TEMP8)
ELK(5,5) = (12*(65*TEMP1+21*TEMP2+42*TEMP3+65*TEMP4))/(175*TEMP13)
ELK(6,1) = -(65*TEMP1-21*TEMP2-42*TEMP3-390*TEMP4)/(175*TEMP12)
ELK(6,2) = -(15*TEMP1+7*TEMP2+14*TEMP3-130*TEMP4)/(175*TEMP11)
ELK(6,3) = -(130*TEMP1-7*TEMP2-84*TEMP3-220*TEMP4)/(700*TEMP8)
ELK(6,4) = -(90*TEMP1+7*TEMP2+84*TEMP3-220*TEMP4)/(2100*TEMP7)
ELK(6,5) = -(110*TEMP1+21*TEMP2+252*TEMP3+390*TEMP4)/(175*TEMP12)
ELK(6,6) = (4*(5*TEMP1+7*TEMP2+14*TEMP3+65*TEMP4))/(175*TEMP11)
ELK(7,1) = (135*TEMP1-21*TEMP2-252*TEMP3-110*TEMP4)/(175*TEMP10)
ELK(7,2) = (130*TEMP1-7*TEMP2-84*TEMP3-220*TEMP4)/(700*TEMP8)
ELK(7,3) = (2*(45*TEMP1-14*TEMP2-28*TEMP3-10*TEMP4))/(175*TEMP9)
ELK(7,4) = (65*TEMP1-7*TEMP2-14*TEMP3-30*TEMP4)/(525*TEMP6)
ELK(7,5) = (390*TEMP1+21*TEMP2+252*TEMP3+110*TEMP4)/(175*TEMP10)
ELK(7,6) = -(220*TEMP1+7*TEMP2+854*TEMP3+220*TEMP4)/(700*TEMP8)
ELK(7,7) = (4*(65*TEMP1+7*TEMP2+14*TEMP3+5*TEMP4))/(175*TEMP9)
ELK(8,1) = -(130*TEMP1-7*TEMP2-84*TEMP3-220*TEMP4)/(700*TEMP8)
ELK(8,2) = -(90*TEMP1+7*TEMP2+84*TEMP3-220*TEMP4)/(2100*TEMP7)
ELK(8,3) = -(65*TEMP1-7*TEMP2-14*TEMP3-30*TEMP4)/(525*TEMP6)
ELK(8,4) = -(45*TEMP1+7*TEMP2+14*TEMP3-30*TEMP4)/(1575*TEMP5)
ELK(8,5) = -(220*TEMP1+7*TEMP2+154*TEMP3+220*TEMP4)/(700*TEMP8)
ELK(8,6) = (30*TEMP1+7*TEMP2+84*TEMP3+110*TEMP4)/(525*TEMP7)
ELK(8,7) = -(110*TEMP1+7*TEMP2+84*TEMP3+30*TEMP4)/(525*TEMP6)
ELK(8,8) = (4*(15*TEMP1+7*TEMP2+14*TEMP3+15*TEMP4))/(1575*TEMP5)
ELK(9,1) = -(6*(130*TEMP1+42*TEMP2+84*TEMP3-45*TEMP4))/(175*
.               TEMP13)
ELK(9,2) = -(110*TEMP1+21*TEMP2+252*TEMP3-135*TEMP4)/(175*TEMP12)
ELK(9,3) = -(390*TEMP1+21*TEMP2+42*TEMP3-65*TEMP4)/(175*TEMP10)
ELK(9,4) = -(220*TEMP1+7*TEMP2+84*TEMP3-130*TEMP4)/(700*TEMP8)
ELK(9,5) = -(18*(15*TEMP1-14*TEMP2-28*TEMP3+15*TEMP4))/(175*
.               TEMP13)
ELK(9,6) = (65*TEMP1-21*TEMP2-42*TEMP3+135*TEMP4)/(175*TEMP12)
ELK(9,7) = -(135*TEMP1-21*TEMP2-42*TEMP3+65*TEMP4)/(175*TEMP10)
ELK(9,8) = (130*TEMP1-7*TEMP2-14*TEMP3+130*TEMP4)/(700*TEMP8)
ELK(9,9) = (12*(65*TEMP1+21*TEMP2+42*TEMP3+65*TEMP4))/(175*TEMP13)
ELK(10,1) = -(110*TEMP1+21*TEMP2+252*TEMP3-135*TEMP4)/(175*TEMP12)
ELK(10,2) = -(2*(10*TEMP1+14*TEMP2+28*TEMP3-45*TEMP4))/(175*
.                TEMP11)
ELK(10,3) = -(220*TEMP1+7*TEMP2+84*TEMP3-130*TEMP4)/(700*TEMP8)
ELK(10,4) = -(30*TEMP1+7*TEMP2+14*TEMP3-65*TEMP4)/(525*TEMP7)
ELK(10,5) = -(65*TEMP1-21*TEMP2-42*TEMP3+135*TEMP4)/(175*TEMP12)
ELK(10,6) = (15*TEMP1+7*TEMP2+14*TEMP3+45*TEMP4)/(175*TEMP11)
ELK(10,7) = -(130*TEMP1-7*TEMP2-14*TEMP3+130*TEMP4)/(700*TEMP8)
ELK(10,8) = (90*TEMP1+7*TEMP2+14*TEMP3+130*TEMP4)/(2100*TEMP7)
ELK(10,9) = (110*TEMP1+21*TEMP2+252*TEMP3+390*TEMP4)/(175*TEMP12)
ELK(10,10) = (4*(5*TEMP1+7*TEMP2+14*TEMP3+65*TEMP4))/(175*TEMP11)
ELK(11,1) = (390*TEMP1+21*TEMP2+42*TEMP3-65*TEMP4)/(175*TEMP10)
ELK(11,2) = (220*TEMP1+7*TEMP2+84*TEMP3-130*TEMP4)/(700*TEMP8)
ELK(11,3) = (130*TEMP1-7*TEMP2-14*TEMP3-15*TEMP4)/(175*TEMP9)
ELK(11,4) = (220*TEMP1-7*TEMP2-84*TEMP3-90*TEMP4)/(2100*TEMP6)
ELK(11,5) = (135*TEMP1-21*TEMP2-42*TEMP3+65*TEMP4)/(175*TEMP10)
ELK(11,6) = -(130*TEMP1-7*TEMP2-14*TEMP3+130*TEMP4)/(700*TEMP8)
ELK(11,7) = (45*TEMP1+7*TEMP2+14*TEMP3+15*TEMP4)/(175*TEMP9)
ELK(11,8) = -(130*TEMP1+7*TEMP2+14*TEMP3+90*TEMP4)/(2100*TEMP6)
ELK(11,9) = -(390*TEMP1+21*TEMP2+252*TEMP3+110*TEMP4)/(175*TEMP10)
ELK(11,10) = -(220*TEMP1+7*TEMP2+854*TEMP3+220*TEMP4)/(700*TEMP8)
ELK(11,11) = (4*(65*TEMP1+7*TEMP2+14*TEMP3+5*TEMP4))/(175*TEMP9)
```

```
      ELK(12,1) = (220*TEMP1+7*TEMP2+84*TEMP3-130*TEMP4)/(700*TEMP8)
      ELK(12,2) = (30*TEMP1+7*TEMP2+14*TEMP3-65*TEMP4)/(525*TEMP7)
      ELK(12,3) = (220*TEMP1-7*TEMP2-84*TEMP3-90*TEMP4)/(2100*TEMP6)
      ELK(12,4) = (30*TEMP1-7*TEMP2-14*TEMP3-45*TEMP4)/(1575*TEMP5)
      ELK(12,5) = (130*TEMP1-7*TEMP2-14*TEMP3+130*TEMP4)/(700*TEMP8)
      ELK(12,6) = -(90*TEMP1+7*TEMP2+14*TEMP3+130*TEMP4)/(2100*TEMP7)
      ELK(12,7) = (130*TEMP1+7*TEMP2+14*TEMP3+90*TEMP4)/(2100*TEMP6)
      ELK(12,8) = -(90*TEMP1-7*TEMP2-14*TEMP3+90*TEMP4)/(6300*TEMP5)
      ELK(12,9) = -(220*TEMP1+7*TEMP2+154*TEMP3+220*TEMP4)/(700*TEMP8)
      ELK(12,10) = -(30*TEMP1+7*TEMP2+84*TEMP3+110*TEMP4)/(525*TEMP7)
      ELK(12,11) = (110*TEMP1+7*TEMP2+84*TEMP3+30*TEMP4)/(525*TEMP6)
      ELK(12,12) = (4*(15*TEMP1+7*TEMP2+14*TEMP3+15*TEMP4))/(1575*TEMP5)
      ELK(13,1) = -(18*(15*TEMP1-14*TEMP2-28*TEMP3+15*TEMP4))/(175*
     .             TEMP13)
      ELK(13,2) = -(65*TEMP1-21*TEMP2-42*TEMP3+135*TEMP4)/(175*TEMP12)
      ELK(13,3) = -(135*TEMP1-21*TEMP2-42*TEMP3+65*TEMP4)/(175*TEMP10)
      ELK(13,4) = -(130*TEMP1-7*TEMP2-14*TEMP3+130*TEMP4)/(700*TEMP8)
      ELK(13,5) = -(6*(130*TEMP1+42*TEMP2+84*TEMP3-45*TEMP4))/(175*
     .             TEMP13)
      ELK(13,6) = (110*TEMP1+21*TEMP2+252*TEMP3-135*TEMP4)/(175*TEMP12)
      ELK(13,7) = -(390*TEMP1+21*TEMP2+42*TEMP3-65*TEMP4)/(175*TEMP10)
      ELK(13,8) = (220*TEMP1+7*TEMP2+84*TEMP3-130*TEMP4)/(700*TEMP8)
      ELK(13,9) = (6*(45*TEMP1-42*TEMP2-84*TEMP3-130*TEMP4))/(175*
     .            TEMP13)
      ELK(13,10) = (65*TEMP1-21*TEMP2-42*TEMP3-390*TEMP4)/(175*TEMP12)
      ELK(13,11) = -(135*TEMP1-21*TEMP2-252*TEMP3-110*TEMP4)/(175*
     .               TEMP10)
      ELK(13,12) = -(130*TEMP1-7*TEMP2-84*TEMP3-220*TEMP4)/(700*TEMP8)
      ELK(13,13) = (12*(65*TEMP1+21*TEMP2+42*TEMP3+65*TEMP4))/(175*
     .             TEMP13)
      ELK(14,1) = (65*TEMP1-21*TEMP2-42*TEMP3+135*TEMP4)/(175*TEMP12)
      ELK(14,2) = (15*TEMP1+7*TEMP2+14*TEMP3+45*TEMP4)/(175*TEMP11)
      ELK(14,3) = (130*TEMP1-7*TEMP2-14*TEMP3+130*TEMP4)/(700*TEMP8)
      ELK(14,4) = (90*TEMP1+7*TEMP2+14*TEMP3+130*TEMP4)/(2100*TEMP7)
      ELK(14,5) = (110*TEMP1+21*TEMP2+252*TEMP3-135*TEMP4)/(175*TEMP12)
      ELK(14,6) = -(2*(10*TEMP1+14*TEMP2+28*TEMP3-45*TEMP4))/(175*
     .             TEMP11)
      ELK(14,7) = (220*TEMP1+7*TEMP2+84*TEMP3-130*TEMP4)/(700*TEMP8)
      ELK(14,8) = -(30*TEMP1+7*TEMP2+14*TEMP3-65*TEMP4)/(525*TEMP7)
      ELK(14,9) = -(65*TEMP1-21*TEMP2-42*TEMP3-390*TEMP4)/(175*TEMP12)
      ELK(14,10) = -(15*TEMP1+7*TEMP2+14*TEMP3-130*TEMP4)/(175*TEMP11)
      ELK(14,11) = (130*TEMP1-7*TEMP2-84*TEMP3-220*TEMP4)/(700*TEMP8)
      ELK(14,12) = (90*TEMP1+7*TEMP2+84*TEMP3-220*TEMP4)/(2100*TEMP7)
      ELK(14,13) = -(110*TEMP1+21*TEMP2+252*TEMP3+390*TEMP4)/(175*
     .              TEMP12)
      ELK(14,14) = (4*(5*TEMP1+7*TEMP2+14*TEMP3+65*TEMP4))/(175*TEMP11)
      ELK(15,1) = (135*TEMP1-21*TEMP2-42*TEMP3+65*TEMP4)/(175*TEMP10)
      ELK(15,2) = (130*TEMP1-7*TEMP2-14*TEMP3+130*TEMP4)/(700*TEMP8)
      ELK(15,3) = (45*TEMP1+7*TEMP2+14*TEMP3+15*TEMP4)/(175*TEMP9)
      ELK(15,4) = (130*TEMP1+7*TEMP2+14*TEMP3+90*TEMP4)/(2100*TEMP6)
      ELK(15,5) = (390*TEMP1+21*TEMP2+42*TEMP3-65*TEMP4)/(175*TEMP10)
      ELK(15,6) = -(220*TEMP1+7*TEMP2+84*TEMP3-130*TEMP4)/(700*TEMP8)
      ELK(15,7) = (130*TEMP1-7*TEMP2-14*TEMP3-15*TEMP4)/(175*TEMP9)
      ELK(15,8) = -(220*TEMP1-7*TEMP2-84*TEMP3-90*TEMP4)/(2100*TEMP6)
      ELK(15,9) = -(135*TEMP1-21*TEMP2-252*TEMP3-110*TEMP4)/(175*TEMP10)
      ELK(15,10) = -(130*TEMP1-7*TEMP2-84*TEMP3-220*TEMP4)/(700*TEMP8)
      ELK(15,11) = (2*(45*TEMP1-14*TEMP2-28*TEMP3-10*TEMP4))/(175*TEMP9)
      ELK(15,12) = (65*TEMP1-7*TEMP2-14*TEMP3-30*TEMP4)/(525*TEMP6)
      ELK(15,13) = -(390*TEMP1+21*TEMP2+252*TEMP3+110*TEMP4)/(175*
     .              TEMP10)
      ELK(15,14) = (220*TEMP1+7*TEMP2+854*TEMP3+220*TEMP4)/(700*TEMP8)
      ELK(15,15) = (4*(65*TEMP1+7*TEMP2+14*TEMP3+5*TEMP4))/(175*TEMP9)
      ELK(16,1) = -(130*TEMP1-7*TEMP2-14*TEMP3+130*TEMP4)/(700*TEMP8)
      ELK(16,2) = -(90*TEMP1+7*TEMP2+14*TEMP3+130*TEMP4)/(2100*TEMP7)
      ELK(16,3) = -(130*TEMP1+7*TEMP2+14*TEMP3+90*TEMP4)/(2100*TEMP6)
      ELK(16,4) = -(90*TEMP1-7*TEMP2-14*TEMP3+90*TEMP4)/(6300*TEMP5)
      ELK(16,5) = -(220*TEMP1+7*TEMP2+84*TEMP3-130*TEMP4)/(700*TEMP8)
      ELK(16,6) = (30*TEMP1+7*TEMP2+14*TEMP3-65*TEMP4)/(525*TEMP7)
      ELK(16,7) = -(220*TEMP1-7*TEMP2-84*TEMP3-90*TEMP4)/(2100*TEMP6)
      ELK(16,8) = (30*TEMP1-7*TEMP2-14*TEMP3-45*TEMP4)/(1575*TEMP5)
      ELK(16,9) = (130*TEMP1-7*TEMP2-84*TEMP3-220*TEMP4)/(700*TEMP8)
      ELK(16,10) = (90*TEMP1+7*TEMP2+84*TEMP3-220*TEMP4)/(2100*TEMP7)
      ELK(16,11) = -(65*TEMP1-7*TEMP2-14*TEMP3-30*TEMP4)/(525*TEMP6)
      ELK(16,12) = -(45*TEMP1+7*TEMP2+14*TEMP3-30*TEMP4)/(1575*TEMP5)
      ELK(16,13) = (220*TEMP1+7*TEMP2+154*TEMP3+220*TEMP4)/(700*TEMP8)
      ELK(16,14) = -(30*TEMP1+7*TEMP2+84*TEMP3+110*TEMP4)/(525*TEMP7)
      ELK(16,15) = -(110*TEMP1+7*TEMP2+84*TEMP3+30*TEMP4)/(525*TEMP6)
      ELK(16,16) = (4*(15*TEMP1+7*TEMP2+14*TEMP3+15*TEMP4))/(1575*TEMP5)
C
      RETURN
      END
```

References

Abate T., 'Engines power business towards the chequered flag', *Computing*, pp 22–23, 13 december 1990.

Aho, Hopcroft and Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

Aitchison J.M., 'A variable finite element method for the calculation of flow over a weir', *RL-79-069*, Rutherford Laboratory, 1979.

Akin J.E., *Finite Element Analysis for Undergraduates*, Academic press, 1986.

Almasi G.S. and Gottlieb A., *Highly Parallel Computing*, The Benjamin/Cummings Publishing Company Inc, 1989.

Amano H., Boku T. and Kudoh T., '$(SM)^2$-II: A Large-Scale Multiprocessor for Sparse Matrix Calculations', *IEEE Transactions on Computers*, **39**, Iss 7, pp 889–905, July 1990.

Anderson D.L. and Ungless R.L., 'Infinite finite elements', *Int. Symp. Innovative Num. Anal. Appl. Eng. Sci.*, France, 1977.

* Applegarth I. and Barbier C., 'A Parallel Equation Solver for Unsymmetric Systems of Linear Equations', to be submitted for publication.

* Barbier C., 'Automatic generation of bending element matrices for finite element method using REDUCE', to appear in *Engineering Computation*

* Barbier C., Bettess P. and Bettess J.A, 'Automatic Generation of Mapping Functions for Infinite Elements using REDUCE', submitted for publication in the *Journal of Symbolic Computation*

* Barbier C., Clark P.J, Bettess P. and Bettess J.A, 'Automatic generation of shape functions for finite element analysis using REDUCE', *Engineering Computation*, **7**, pp 349–358, Dec. 1990.

Bardnell N.S., 'The application of Symbolic Computing to the Hierarchical Finite Element Method', *International Journal for Numerical Methods in Engineering*, **28**, Iss 5, pp 1181–1204, 1989.

Bathe K., *Finite Element procedures in Engineering Analysis*, Prentice-Hall Inc., 1982.

Beer G. and Meek J.L., 'Infinite domain elements', *International Journal for Numerical Methods in Engineering*, **17(1)**, pp 43–52, 1981.

Bertsekas D.P. and Tsitsiklis J.N., 'Some Aspects of Parallel and Distributed Iterative Algorithms — A Survey', *Automatica*, **27**, Iss 1, pp 3–21, 1991.

Bettess P. and Bettess J.A, 'Analysis of free surface flows using isoparametric finite elements', *International Journal for Numerical Methods in Engineering*, **19**, pp 1675–1689, 1983.

Bettess P. and Bettess J.A., 'Automatic Generation of Shape Function Routines', Paper S20 in *Numerical Techniques for Engineering and Design*, Proceedings of the International Conference on Numerical Methods in Engineering: Theory and Applications, NUMETA '87, Swansea, Vol. II, Matinus Nijhoff, Dordrecht, 1987.

Bettess P. and Bettess J.A., 'A profile matrix solver with built-in constraint facility', *Engineering Computations*, **3**, Iss 3, pp 209–216, September 1986.

Betts P.L., 'A variational principle in terms of stream function for free-surface flows and its application to the finite element method', *Computers and fluids*, **7**, pp 145-153, 1979.

Bogner F.K, Fox R.L and Schmit L.A., 'The generation of interelement– compatible stiffness and mass matrices by the use of interpolation formulæ', *Proceedings of the Conference on Matrix Methods in Structural Mechanics*, Air Force Institute of Technology, Wright Patterson A.F Base, Ohio, USA, October 1965.

Bond E. *et al*, 'FORMAC an Experimental Formula Manipulation Compiler', *Proceedings of the 19th ACM Conference*, pp K2.1-1–K2.1-18, 1964.

Brown W.S., 'The ALPAK System for Nonnumerical Algebra on a Digital Computer — I: Polynomials in Several Variables and Truncated Power Series with Polynomial Coefficients', *Bell Systems Truncated Journal*, **42**, pp 2081–2119, 1963.

Buchberger B., Collins G.E. and Loos R., *Computer Algebra : Symbolic and Algebraic Computation, Computing supplementum 4*, Springer-Verlag, 1982.

Burden R.L. and Faires J.D., *Numerical Analysis*, third edition, Prindle, Weber and Schmidt publishers, 1985.

Calmet J., 'Computer Algebra Applications', *Computer Algebra : Symbolic and Algebraic Computation, Computing supplementum 4*, edited by Buchberger B., Collins G.E. and Loos R., pp 245–258, Springer-Verlag, 1982.

Cassidy J.J., 'Irrotational flow over spillways of finite height', *J. Engrg. Mech. Div., ASCE*, **91**, Iss 6, pp 155–173, 1965.

Cheney W. and Kincaid D., *Numerical Mathematics and Computing*, second edition, Brooks/ Cole Publishing Company, 1985.

Clark. P.J., 'The Vibration of a Stiffened Panel: Analysis by means of an Orthotropic Plate Conforming Finite Element', *MSc Marine Technology*, The University of Newcastle-upon-Tyne, September 1982.

Cohen J., 'Symbolic and Numerical Computer Analysis of the Combined Local and Overall Buckling of Rectangular Thin-Walled Columns', *Computer Methods in Applied Mechanics and Engineering*, **7**, Jan. 1976.

Computer Algebra Support Officer, Computing Laboratory, University of Liverpool, P.O. Box 147, Liverpool, L69 3BX. Tel: 051–794 3755. Email: Algebra@ uk.ac.liverpool.

Computer Weekly, weekly publication, Quadrant House, The Quadrant, Sutton, Surrey, SM2 5AS, UK.

Computing, weekly publication, VNU House, 32–34 Broadwick street, London, W1A 2HG, UK.

Computing Surface, CS Tools for SunOS documentation, edition 83 - 009 A00 - 02.02, two volumes, available from Meiko Limited.

Courant R. and Hilbert D., *Methods of Mathematical Physics*, Vol. 1, Wiley-Interscience, New-York, 1953.

Courant R., 'Variational methods for the solution of problems of equilibrium and vibration', *Bulletin of American Methematical Society*, **49**, pp 1–23, 1943.

Craeger W.P., *Engineering of masonary dams*, John Wiley publisher, New York, 1929.

Crespo Da Silva and Marcela R.M., 'The Role of Computerized Symbolic Manipulation in Rotorcraft Dynamics Analysis', *Computer and Mathematics with Applications*, **12A**, Iss 1., pp 161–172, 1986.

CS Tools, A technical overview and *A programmer's introduction to SUN - CS Tools*, Meiko Limited (Ref 30).

Davenport J.H., Siret Y. and Tournier E., *Computer Algebra : Systems and Algorithms for Algebraic Computation*, Academic Press, 1988.

Davis C.V., *Handbook of applied hydraulics*, McGraw-Hill, New York, second edition, 1952.

Derive is available from Soft Warehouse Inc., 3615 Harding Avenue, Suite 505, Honolulu, Hawaii 96816, USA.

Desai C.S and Abel J.F, *Introduction to the Finite Element Method (a numerical method for engineering analysis)*, Van Nostrand Reinhold Company, 1972.

Dias F., Keller J.B. and Vanden-Broeck J., 'Flows over rectangular weirs', *Physics of Fluids*, **31**, Iss 8, pp 2071-2076, 1988.

Diersch H, Schirmer A. and Busch K., 'Analysis of Flows with Initially Unknown Discharge', *Journal of the Hydraulic Division*, **103**, pp 213–232, 1977.

Dixon L.C.W., *Nonlinear Optimisation*, The English Universities press, 1972.

Doon M.K.B.M., 'Determination of flow profile over spillway using Finite Elements', *Thesis BSC*, University of Wales, Swansea, 1983.

Dubbey J.M., *The mathematical work of Charles Babbage*, Cambridge University Press, 1978.

Edinburgh Parallel Computing Centre, the King's Building, Mayfield road, Edinburgh, EH9 3JZ, UK.

Encore Computer Corporation, PO Box 409148, Fort Lauderdale, Florida, 33340–9148, USA.

Farhat C., 'A Simple and Efficient Automatic FEM Domain Decomposer', *Computers and Structures*, **28**, Iss 5, pp 579–602, 1988.

Farhat C. and Wilson E., 'A Parallel Active Column Equation Solver', *Computers and Structures*, **28**, Iss 2, pp 289–304, 1988.

Filho J.S.R.A, 'The Use of Transputer Based Computers in Finite Element Calculations', *PhD thesis*, University of Wales, University College of Swansea, Department of Civil Engineering, September 1989.

Fletcher R., *Practical Methods of Optimization*, Second edition, John Wiley & Sons. 1987.

Flynn M.F, 'Some Computer organisations and their effectiveness', *IEEE Trans. Compt. C-21*, pp 948-960, 1972.

Forbes L.K., 'Critical free-surface flow over a semi-circular obstruction', *Journal of Engineering Mathematics*, **22**, pp 3–13, 1988.

Forbes L.K., 'Two-layer critical free-surface flow over a semi-circular obstruction', *Journal of Engineering Mathematics*, **23**, pp 325–342, 1989.

Forbes L.K., 'An Algorithm for 3-Dimensional Free-Surface Problems in Hydrodynamics', *Journal of Computational Physics*, **82**, pp330–347, 1989.

Foster I. and Taylor S., *STRAND: New Concepts in Parallel Programming*, Prentice Hall.

Gajski D.D. and Peir J., 'Comparison of five multiprocessor systems', *Parallel Computing 2*, pp 265–282, 1985.

Gallivan K.A., Plemmons R.J. and Sameh A.H., 'Parallel Algorithms for Dense Linear Algebra Computations', *SIAM Review*, **32**, Iss 1, pp 54–135, March 1990.

Gates B.L., *GENTRAN user's Manual, REDUCE version*, Information Sciences Department, The RAND Corporation, P.O box 2138, 1700 Main Street, Santa Monica, CA 90406-2138, U.S.A.

Gehani N., *ADA, Concurrent Programming*, Prentice Hall, 1984.

Geist G.A. and Romine C.H., 'LU Factorisation on Distributed-Memory Multiprocessors', *Proceedings of the third SIAM Conference on Parallel processing for Scientific Computing*, pp 15–18, Los Angeles, California, USA, December 1987.

Gerald C.F. and Wheatley P.O., *Applied Numerical Analysis*, third edition, Addison–Wesley Publishing Company, 1984.

Godden W.G., *Numerical Analysis of Beam and Column Structures*, Prentice-Hall Inc.,1965.

Grigorev F.N and Kistlerov V.L, 'Computer Algebra Methods used in Analysing the Stability of Linear Dynamic Systems', *Automation and Remote Control USSR*, **50**, Iss 7, pp 925–988, 1989.

Harp G., *Transputer Applications*, Pitman ed., p5, 1989.

Harper D., 'A Guide to Computer Algebra Systems', *Computer Algebra Support Project*, University of Liverpool, P.O Box 147, Liverpool, L69 3BX, fourth edition, March 1990.

Hearn A.C., *REDUCE User's Manual*, RAND Publication CP78, Rev. 7/87, July 1987.

Henderson H.C., Kok M. and De Koning W.L., 'Computer-aided spillway design using the boundary element method and non-linear programming', *International Journal of Numerical Methods in Fluids*, **13**, pp 625–641, 1991.

Hoare C.A.R., 'Communication Sequential Process', *Communication of ACM*, **21**, (8), August 1978.

Hockney R.W. and Jesshope C.R, *Parallel Computers 2: Architecture, programming and Algorithms*, ed. Adam Hilger, 1988.

Hodgkinson D., 'The use of Computer Algebra in teaching', *IUSC Workshop on Algebraic Computing*, University of Liverpool, 4–5 July 1989.

Hord R.M., *The Illiac-IV: The first Supercomputer*, Computer Science Press, 1982.

Hosack J.M, 'A guide to Computer Algebra Systems', *The College Mathematics Journal*, 17,5, pp 434–441, 1986.

van Hulzen J.A., 'Computer Algebra Systems', *Computer Algebra : Symbolic and Algebraic Computation, Computing supplementum 4*, edited by Buchberger B., Collins G.E. and Loos R., pp 221–243, Springer-Verlag, 1982.

INMOS Limited, *OCCAM 2 Reference Manual*, Prentice Hall, 1988.

Ikegawa M. and Washizu K., 'Finite element method applied to analysis of flow over a spillway crest', *International Journal for Numerical Methods in Engineering*, **6**, pp 179–189, 1973.

Ioakimidis N.I, 'Symbolic Computation — A Powerful Method for the Solution of Crack Problems in Fracture Mechanics', *International Journal of Fracture*, **43**, Iss 3, pp &39–&42, 1990.

Jennings A., *Matrix Computation for Engineers and Scientists*, John Wiley & Sons, 1977.

Jennings A., 'Solution of variable band-width partial differential equations', *Computer. J.*, **15**, p 446, 1971.

Jensen J. and Niordson F., 'Symbolic and Algebraic Manipulation Languages and their Applications in Mechanics', *Structural Mechanics Software Series*, Volume 1, Editors Perrone N. and Pilkey W., University Press of Virginia, Charlottesville.

Kahrimanian H.G. and Nolan J., *Analytic Differentiation by a Digital Computer*, MA thesis, Temple Univ. Phil., PA. and Math. Dept., M.I.T Cambridge, Mass., 1953.

Kidger D.J, 'The 14 node brick element', *Proceedings of the second annual Robert J. Melosh Medal Paper competition*, Duke University, N.C., March 1990, to be published in a special issue of *Finite Elements in Analysis and Design.*

King A.C. and Bloor M.I.G., 'Free-surface flow of a stream obstructed by an arbitrary bed topography', *Quaterly Journal of Mechanics and Applied Mathematics*, **43**, Iss 1., pp 87–106, 1990.

Kuck D.J, *The structure of Computers and Computations*, John Wiley publishing, 1978.

Kuhn R.H. and Padua D.A, *Tutorial on Parallel Processing*, IEEE Computer Society publication, 1981.

Kumar P., *International Journal for Numerical Methods in Engineering*, **20**, pp 1173–1174, 1984.

Ladefoged T., 'Triangular Ring Element with Analytic Expressions for Stiffness and Mass Matrix', *Computer Methods in Applied Mechanics and Engineering*, **67**, pp 171–187, North-Holland, 1988.

Lee X.G. and Dasgupta G., 'Analysis of Structural Variability with Computer Algebra', *Journal of Engineering Mechanics–ASCE*, **114**, Iss 1, pp 161–171, 1988.

Leler W., 'Linda meets UNIX', *Application of Transputers, Proceedings of the First International Conference on Application of Transputers*, Liverpool, UK. 23–25 August 1989.

Lengauer C. 'Systolic Design', *Edinburgh Parallel Computing Centre annual Seminar: Abstracts*, 23rd September 1991.

Levi I.M., 'Symbolic Algebra by Computer — Applications to Structural Mechanics', *AIAA, 12th Structure, Structural Dynamics and Materials Conference*, Anaheim, California, 19–21 April, 1971.

Li W., Xie Q. and Chen C.J., 'Finite Analytic Solution of Flow over Spillways', *Journal of Engineering Mechanics*, **115**, Iss 12, pp 2635–2648, 1989.

Lin A. and Zhang H., 'A new Parallel Algorithm for Linear Triangular Systems', *Proceedings of the third SIAM Conference on Parallel processing for Scientific Computing*, pp 36–39, Los Angeles, California, USA, December 1987.

Livesley R.K., *Matrix Methods of Structural Analysis*, Pergamon press, 1964.

Loos R., 'Introduction', *Computer Algebra : Symbolic and Algebraic Computation, Computing supplementum 4*, Springer-Verlag, pp 1–10, 1982.

Luke J.C, 'A variational principle for a fluid with a free surface', *Journal of Fluid Mechanics*, **27**, Iss 2, pp 395–397, 1967.

Lynn P.P. and Hadid H.A., 'Infinite elements with $1/r^n$ type decay', *International Journal for Numerical Methods in Engineering*, **17(3)**, pp 347–355, 1981.

Macsyma-Symbolics Ltd, St. John's Court, Easton St, High Wycombe, Bucks, HP11 1JX, UK.

Maeder R., *Programming for Mathematica*, Addison-Wesley, 1989.

Mandel J., *A Domain Decomposition Method for the p-version Finite Elements in Three Dimensions*, available from the author at the Computational Mathematics Group, University of Colorado at Denver, 1200 Larimer Street, Denver, CO 80204, USA.

Mandel J., *Iterative Solvers by Substructuring for the p-version Finite Element method*, available from the author at the Computational Mathematics Group, University of Colorado at Denver, 1200 Larimer Street, Denver, CO 80204, USA.

Markland E., 'Calculation of flow at a free overfall by relaxation method', *Proc. Inst. Civ. Engrs.*, **31**, pp 71–78, 1965.

Marques J.M.M.C. and Owen D.R.J., 'Infinite elements in quasi-static materially non-linear problems', *Computers and Structures*, to be published.

Martin H.C and Carey G.F, *Introduction to Finite Element Analysis, Theory and application*, McGraw-Hill Book Company, 1973.

Massey B.S., *Mechanics of Fluids*, Second edition, Van Nostrand Reinhold Company, London, 1970.

Mathematica is available from Wolfram Research Inc., PO Box 6059, Champaign, 61821, USA.

McMinn S.J., *Matrices for structural analysis*, 1962.

Meiko computing surface, available from Meiko Limited, 650 Aztec west, Bristol, BS12 4SD, UK.

Menabrea L.F, *Sketch of the analytical engine invented by Charles Babbage*, ESP. Bibliothèque Universelle de Genève, 82, 1842.

Miles R.G. and Havard S.P., 'Multifronts and Transputer Networks for Solving Fluid Mechanical Finite Element Systems', *International Journal for Numerical Methods in Fluids*, **9**, pp 731–740, 1989.

MuMath is available from Soft Warehouse Inc., 3615 Harding Avenue, Suite 505, Honolulu, Hawaii 96816, USA.

Neesham C., 'Hi-tech wheathercocks help to save the planet', *Computing*, pp 16–17, 19 july 1990.

Nishioka T. and Takemoto Y., 'Moving Finite Element Method Aided by Computerized Symbolic Manipulation and its Application to Dynamic Fracture Simulation', *JSME International Journal Series I-solid Mechanics Strength of Materials*, **32**, Iss 3, pp 403–410, 1989.

Noor A.K and Andersen C.M., 'Computerized Symbolic Manipulation in Nonlinear Finite Element Analysis', *Computers and Structures*, **13**, pp 379–403, June 1981.

Parallel FORTRAN user guide, 3L Ltd, Peel House, Ladywell, Livingston, EH54 6AG, UK, 1988.

Pease D., Ghafoor A., Ahmad I., Andrews D.L., Foudil-Bey K., Karpinski T.E., Mikki M.A. and Zerrouki M., 'PAWS: A Performance Evaluation Tool for Parallel Computing Systems', *Computer*, **24**, Iss 1, pp 18–29, January 1991.

Pedersen P. and Megahed M.M., 'Axisymmetric Element Analysis using Analytical Computing', *Computers and Structures*, **5**, pp 241–247, 1975.

Pedersen P., 'On Computer-Aided Analytic Element Analysis and the Similarities of Tetrahedron Elements', *International Journal for Numerical Methods in Engineering*, **11**, pp 61–622, 1977.

Pissanetzky S., 'A Simple Infinite Element', *COMPEL*, Boole Press, to be published.

Pissanetzky S., 'An infinite element and a formula for numerical quadrature over an infinite interval', *International Journal for Numerical Methods in Engineering*, **19**, pp 913–928, 1983.

Przemieniecki J.S., *Theory of matrix structural analysis*, McGraw-Hill Book Company, 1968.

REDUCE is available from the RAND Corporation, 1700 Main Street, Santa Monica, CA, 90406-2138, USA.

Rao S.S., *The Finite Element Method in Engineering*, Second Edition, Pergamon Press, 1989.

Rayna G., *REDUCE — Sòftware for Algebraic Computation*, Springer, 1987.

SCAFI'91, *Studies in Computer Algebra for Industry*, 10-11 December 1991, Computer Algebra Amsterdam (CAN), Amsterdam, The Netherlands.

SENAC - A Software Environment for Numeric and Algebraic Computation, developed by the Mathematical Software Team, University of Waikato, New Zealand, distributed in Europe by the University of London Computer Centre, 20 Guilford Street, London, WC1N 1DZ.

SERC/DTI Transputer Initiative Mailshot, *INMOS News Release*, p 17, May 1991.

SIGSAM–ACM, Special Interest Group in Symbolic and Algebraic Manipulation, 11 Wesr 42nd. St., NY 10036, U.S.A.

SMP-Inference Corporation, 5300 West Central Building, Los Angeles, CA 90045, USA.

Schäfer M., 'Parallel Algorithms for the Numerical Solution of Incompressible Finite Elasticity Problems', *SIAM Journal of Statistical Computations*, **12**, Iss 2, pp247–259, 1991.

Scratchpad is available from IBM Research Division, T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, Ny 10598, USA.

Sellin R.H.J, *Flow in channels*, Macmillan St. Martins's press, 1969.

Sharp J.A, *An introduction to Distributed and Parallel Processing*, Blackwell Scientific publications, 1987.

Shore J.E., 'Second thoughts on parallel processing', *Computers and Electrical Engineering*, **1**, (1), pp 95–109, 1973.

Sims C.S., *Abstract Algebra : A Computational Approach*, Wiley, 1984.

Smith I.M., 'Are there any new elements', *The finite element method in the 1990's*, Editors Oñate E., Periaux J. and Samuelsson A., Cimne Barcelona, pp 109–118, 1991.

Southwell R.V. and Vaisey G., 'Relaxation methods applied to engineering problems. XII. Fluid motions characterised by 'free' stream-lines', *Phil. Trans. Roy. Soc.*, Land, **240 A**, pp 117-160, 1946.

Stein D., *Ada, A Life and a Legacy*, The MIT Press, 1985.

TMB08 installation and user manual, available form Transtech Devices Limited.

Transputer – manufactured by INMOS Limited, member of the SGS–Thomson Microelectronics Group, 1000 Aztec West, Almondsbury, Bristol, BS12 4SQ, UK.

Transputer Developement System, second edition, INMOS Ltd, Prentice Hall, 1990.

Transtech Devices Limited, Unit 17, Wye industrial estate, London road, High Wycombe, Bucks., HP11 1LH, UK.

Treleaven P., 'Control-Driven Data-Driven and Demand-Driven Computer Architecture (abstract)', *Parallel Computing 2*, 1985.

Trew A. and Wilson G., *Past, Present, Parallel: A Survey of Available Parallel Computing Systems*, Springer-Verlag, 1991.

USBR, *Design of small dams*, US Government Printing Office, Washington, 1973.

Ungless R.L., *An infinite finite element*, M ASc Thesis, University of British Columbia, 1973.

Varoglu E. and Finn W.D.L, 'Variable domain finite element analysis of free surface gravity flow', *Computers and fluids*, **6**, pp 103–114, 1978.

van der Waerden B.L., *Modern Algebra*, Frederick Ungar, 1953.

Wang P.S., Chang T.Y.P and van Hulzen J.A., 'Code generation and optimization for finite element analysis', *Proc. EUROSAM '84, London*, pp 237–247, 9–11 July 1984.

Wang P.S., Tan H., Saleeb A. and Chang T.Y., 'Code generation for hybrid mixed mode formulation in finite element analysis', *ACM SYMSAC'86 Conference*, University of Waterloo, Canada, 21–23 July.

Wang P.S., 'FINGER : a symbolic system for automatic generation of numerical programs in finite element analysis', *Journal for Symbolic Computation*, **2**, pp 305–316, 1986.

Waterloo Maple Software Inc., 160 Columbia Street, W., Waterloo, Ontario, Canada, N2L 3L3.

Williams J.M., 'An integral equation method for the computation of progressive gravity waves of finite height', *HRS Report INT 136*, 1974.

Winston P.H. and Horn B.K.P., *LISP*, Addison-Wesley publishing Company, 1981.

Wooff C. and Hodgkinson D., *muMATH : A Microcomputer Algebra System*, Academic Press, 1987.

Yagawa G., Ye G.W and Yoshimura S., 'A Numerical Integration Scheme for Finite Element Method based on Symbolic Manipulation', *International Journal for Numerical Methods in Engineering*, **29**, Iss 7, pp 1539–1549, 1990.

Yu D.C. and Wang H., 'A New Approach to the Forward and Backward Substitutions of Parallel Solution of Sparse Linear Equations — Based on Dataflow Architecture', *IEEE Transactions on Power Systems*, **5**, Iss 2, pp 621–627, May 1990.

Yu D.C. and Wang H., 'A New Parallel LU Decomposition Method', *IEEE Transactions on Power Systems*, **5**, Iss 1, pp 303–310, February 1990.

Zienkiewicz O.C and Cheung Y.K, *The Finite Element Method in Structural and Continum Mechanics*, McGraw-Hill, London, 1967.

Zienkiewicz O.C, *The Finite Element Method*, 3rd edn, McGraw-Hill, 1977.

Zienkiewicz O.C., Bettess P., Chiam T.C. and Emson C., 'Numerical methods for unbounded field problems and a new infinite element formulation', *ASME, AMD*, **46**, pp 115–148, New York, 1981.

Zienkiewicz O.C., Emson C. and Bettess P., 'A novel boundary infinite element', *Int.J. Num. Meth. Eng.*, **19**, pp 393–404, 1983.

NIV DUNELM